university of groningen

faculty of science and engineering

Developing Deep Learning Approaches to Find and Classify Architectural Design Decisions in Issue Tracking Systems

Jesse Maarleveld & Arjan Dekker

First supervisor: dr. Mohamed Soliman
Second supervisor: prof. dr. ir. Paris Avgeriou

August 2023

Abstract

Architectural design decisions (ADDs) are considered to be an important part of the software architecture, but they are often not explicitly documented, making it challenging to understand the rationale behind a system's structure. This lack of documentation complicates software maintenance and evolution. Additionally, software architects frequently rely on existing ADDs as a basis for creating new ones. Often, they make use of their own experience from past decisions instead of documented ADDs. Recent studies show that ADDs do tend to be discussed implicitly in some places, such as issue trackers in open-source systems. However, identifying these discussions is difficult. In this work, we built upon previous efforts to leverage deep learning techniques for finding ADDs in issue tracking systems by Dekker and Maarleveld (2022). In this work, we extended the dataset used there from 2179 to 6225 issues. Moreover, we performed a more fine-grained classification of issues while also improving classifier performance to 0.67 F_1 score. We also investigated the abilities of classifiers to generalise to different projects and different domains. Our main finding is that deep learning models, and in particular large language models such as BERT, are a promising search tool to find ADDs in issue tracking systems since they are able to find ADDs with a precision ≥ 0.63 even when applied to different domains.

Contents

1 Introduction

Software architecture is the foundation of a software system. In the past, it primarily consisted of the arrangement and interactions of the components within the system. However, nowadays, the concept of software architecture has expanded to include the overarching design decisions that shape the system as well. These design decisions are of great variety, such as the decisions on programming languages, technologies, guidelines, protocols, as well as the overall organisation and structure of the system. By incorporating high-level design decisions into the software architecture, developers and architects can effectively capture the overarching vision and goals of the software system [46].

Using previously made design decisions, helps developers establish a solid foundation for software systems. This foundation should result in better maintainable systems, while also allowing the future evolution of the system to become more manageable. Making incorrect design decisions during the initial development of a system, can be very costly in a later phase of the development [46]. Software engineers therefore prefer to re-use previous design decisions (either be created by themselves [35] or by others [54]) that have been proven in prior projects, instead of experimenting with new solutions [54]. This is especially common in projects where time and budget are constraints [36].

However, writing documentation for the software architecture can be a tedious task. Hence, it is quite common to come across situations where only the high-level architecture of a system is documented, leaving out the design decisions that led to the architecture [9]. These design decisions often remain in the architect's head [46], or they are only accessible through informal discussions among the developers involved in the project. This is problematic, because this makes architecture re-use hard.

Design decisions often require a discussion among developers in order to find the most suitable solution for the system that is being discussed. For open source projects, issue tracking systems serve as a means of communication between developers, among other means such as email. An issue tracking system is a software tool specifically designed for tracking and managing issues, which can encompass bugs, tasks, and new features that need to be implemented. Each issue is assigned a unique key, allowing developers to easily reference and track issues. Studies have found that these systems are also used to discuss design decisions [4, 70].

Exact numbers of how many issues contain ADDs were not available previously, but the majority of issues have been found to be non-architectural[4, 68]. In this thesis, we have found that only 10-15% of the issues contain ADDs. Another problem with finding architectural issues is that currently, developers do not explicitly tag the issues containing ADDs and also not what type of ADD is discussed [4]. Given that only a small portion of the issues is architectural, and developers do not specify which issues are architectural, make tools for identifying architectural issues a must. Tools for identifying architectural issues helps researchers and practitioners to more efficiently find architectural issues, aiding them in doing architectural research and potentially recovering previously made ADDs.

Researchers have identified and developed various methods for capturing architectural issues. The first method is a keyword-based search. This approach searches using keywords that are commonly used to express ADDs to find potentially relevant architectural issues [28]. The second approach is a source code analysis. This method analyses source code changes between commits to identify architectural changes. Potential architectural issues can then be found by linking such architectural commits with their corresponding issues [70, 28]. The third method is Maven POM file analysis. This method is similar to the previous one, with the difference being that it analyses Maven POM file changes between commits. Similarly, by linking such commits with their corresponding issues, it can find potential architectural issues [17, 25]. The fourth method makes use of machine learning and deep learning approaches. While [4] only used traditional machine learning methods such as support vector machine (SVM), [18] used more complex deep learning methods. Additionally, [4] only classified existence decisions (a subtype of ADDs), while [18] classified all three subtypes identified by [45].

This thesis is a continuation of the effort from [18]. Specifically, we address multiple possible improvements. First, we found that the number of labelled issues for some of the ADD subtypes was small. Research has not experimented with applying machine learning or deep learning tools to find architectural issues before. As such, we have experimented with finding architectural issues using deep learning classifiers to extend the dataset, and compared them with existing methods, such as keyword search, Maven dependencies analysis and source code analysis. Additionally, we have experimented with BERT, a more advanced deep learning model, achieving state-of-the-art results for text classification tasks [21]. Besides, we have used deep learning models from [18] to be able to compare their performances with BERT. Moreover, we have improved the evaluation techniques, more focused towards evaluating classifier's performance in practical settings. Specifically, our research has lead to the following contributions:

- Information regarding the expected proportions of architectural design decision types one can expect to find in issue tracking systems in open source projects.
- An evaluation of deep learning as a technique to find architectural design decisions in issue tracking systems.
- The development of several deep learning classifiers for identifying and classifying architectural design decisions in issues in issue tracking systems. Pretrained versions of all classifiers can be downloaded.
- A dataset of 6108 manually classified issues, 3673 contain at least one architectural design decision, and 3552 without architectural design decisions. 3934 of these issues were newly added in this research.
- A coding book for manually classifying architectural design decisions in issues.
- A dataset of 1,345,784 issues from six different domains annotated using a deep learning classifier.
- Maestro, a tool for finding and exploring architectural design decisions in issue tracking using deep learning, keyword searches, and statistical analysis.

The remainder of this work is structured as follows: Section 2 introduces some notation. Next, Section 3 introduces relevant background material, including previous work by the two authors of this work on the subject, and other related work. Next, we introduce our research questions and explain our study design in Section 4. In Section 5, we discuss Maestro, a tool which was created based on techniques developed in this research. In Section 6, we discuss our results and answer our research questions. Next, we discuss the implications of our findings for researchers in practitioners and researchers in Section 7. We discuss threats to validity in Section 8. Finally, we end with a conclusion and outlook in future work in Section 9.

2 Notation

In this section, we briefly explain some of the mathematical notation used throughout this work.

First, we will use the convention that numbers are represented by normal, non-styled symbols (e.g. x, α). Vectors will be denoted in bold case (e.g. $\boldsymbol{x}$). $\mathbf{1}_n$ denotes the n-dimensional vector containing all ones. Matrices will be denoted in upper case (e.g. A).

When a square root operation is applied to a vector, or a division is performed with on both sides a vector, the operations are performed element-wise, unless stated differently. For element-wise multiplication of vectors, we use $\odot$.

With RMS($\boldsymbol{x}$), we denote the root mean square of the vector $\boldsymbol{x} \in \mathbb{R}^n$ with entries $x_1, \ldots, x_n$, which is defined as

$$\text{RMS}(\boldsymbol{x}) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$$

3 Background

In this section, we will cover background information relevant for understanding the remainder of this work. We will be discussing background on architectural design decisions, issue trackers, deep learning, and previous and related work.

3.1 Architectural Design Decisions

Classically, the architecture of a software system was seen as a high level decomposition describing the major components and how these components interact [78]. However, such a decomposition of the system does not necessarily make clear why a system is designed the way it is. Hence, it can be said that such a high level decomposition does not encompass the entirety of the *Architectural Knowledge* (AK) about the system. Instead, the *Architectural Design Decisions* (ADDs) made while designing the system are also an important part of the architectural knowledge: they help explain why the system is the way it is. As such, it is said that the entirety of the architectural knowledge about a system is formed by the design itself, and the design decisions [46].

In [45], Kruchten defined an ontology of different types of architectural design decisions. He identified three major types of design decisions: existence decisions, executive decisions, and property decisions:

- **Existence** design decisions relate to the existence of components and interactions between components in the system; existence decisions state the presence or absence of some component or behaviour in the system. Existence decisions can also be further subdivided as follows:
  - *Structural* design decisions lead to the creation of components in the system.
  - *Behavioural* design decisions are about connectors or interactions between components; specifically how these interactions fulfil some (non-functional) requirement.
  - *Ban* design decisions assert that some component or behaviour will not show up in the system. They can also be thought of as "non-existence" decisions.
- **Executive** design decisions are design decisions driven by the business environment of the system. Such decisions can affect the method by which the system is developed. These decisions can also be further subdivided into multiple categories:
  - *Process* decisions dictate aspects of the workflow of the development process.
  - *Technology* decisions dictate what technologies (e.g. programming languages) will be used to develop a system.
  - *Tool* decisions dictate what tools will be used to develop the system (e.g. some specific IDE). These are different from technology decisions, because they describe the actual tools that must be used to develop a system, while technology decisions describe the languages, libraries, and framework that will be used to develop the system.
- **Property** design decisions state enduring and overarching traits of the system. Usually, these are expressed in terms of quality attributes. Often, property decisions are stated implicitly; in issue tracking system in particular, they are often stated as "We will do X to improve quality attribute Y" [45].

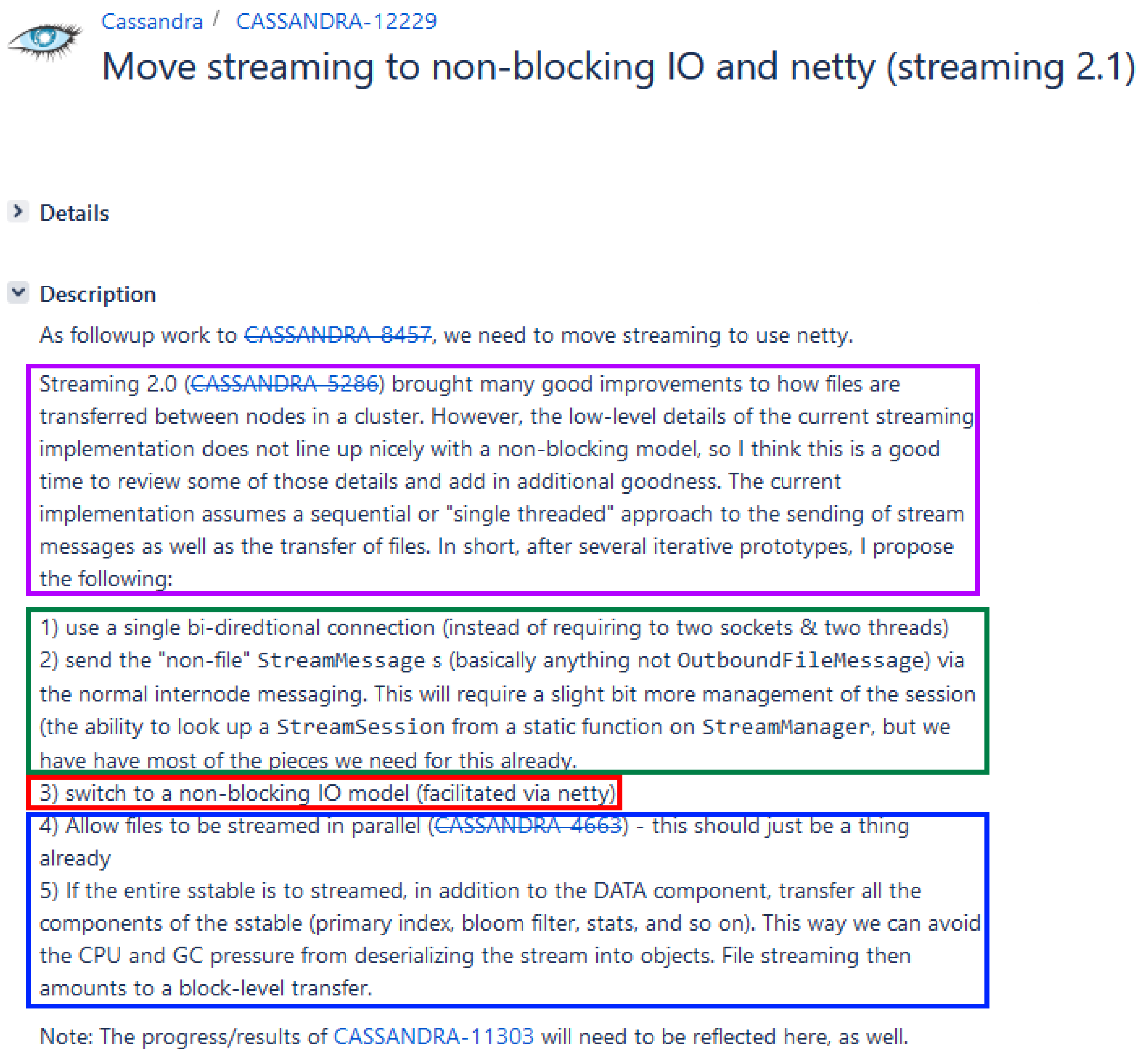
3.2 Issue Tracking Systems & Issues

Issue tracking systems allow software developers to coordinate the development of some project, by tracking bugs, feature requests, and other changes to a system. Each issue in an issue tracking system is generally used to discuss a single change to the system. Sometimes, issues also contain discussions on the architecture of a system. This means the issues are also a source of architectural knowledge [70]. From now on, we will refer to such issues as *architectural issues*. Figure 1 gives an example of such an architectural issue, alongside an explanation why it would be considered architectural.

In this work, we are specifically looking at issues from Jira issue trackers. The issues in Jira contain a variety of information. The most basic information is that what is necessary for a discussion between the developers: A short summary (similar to a title), a description of the issue, and a comment section. An example issue given in Figure 2. In that issue, we can see that Jira also stores a number of other *issue characteristics*:

- *Status*: the current status of an issue. This generally denotes the next steps required for an issue. Basic statuses include "Open" and "Closed".
- *Resolution*: the resolution shows how, in the end, an issue was handled. "Fixed" is a common resolution, but the resolution can also be something else. An example could be "Cannot Reproduce" for behaviour which cannot be reproduced by others.
- *Votes*: people may vote on particular issues they find interesting or important. Jira keeps track of the number of votes (and who voted) per issue.
- *Watches*: people may "watch" particular issues they find interesting or important. Jira keeps track of the people watching an issue, and thus also the amount of people watching the issue.
- *Priority*: an attribute denoting the importance of an issue. Some issues can wait, while others are critical and must be fixed as soon as possible.
- *Issue Type*: the type of the issue. Some example issue types are "Bug" and "New Feature".
- *Parent*: some issues are linked with other issues in a child/parent relationship; often the parent issue describes a large problem, and the child issues solve smaller parts of that issue. Many issues do not have a parent.

[2]`https://issues.apache.org/jira/browse/CASSANDRA-12229`
[4]`https://issues.apache.org/jira/browse/HADOOP-7119`


Cassandra / CASSANDRA-12229

Move streaming to non-blocking IO and netty (streaming 2.1)

Details

Description

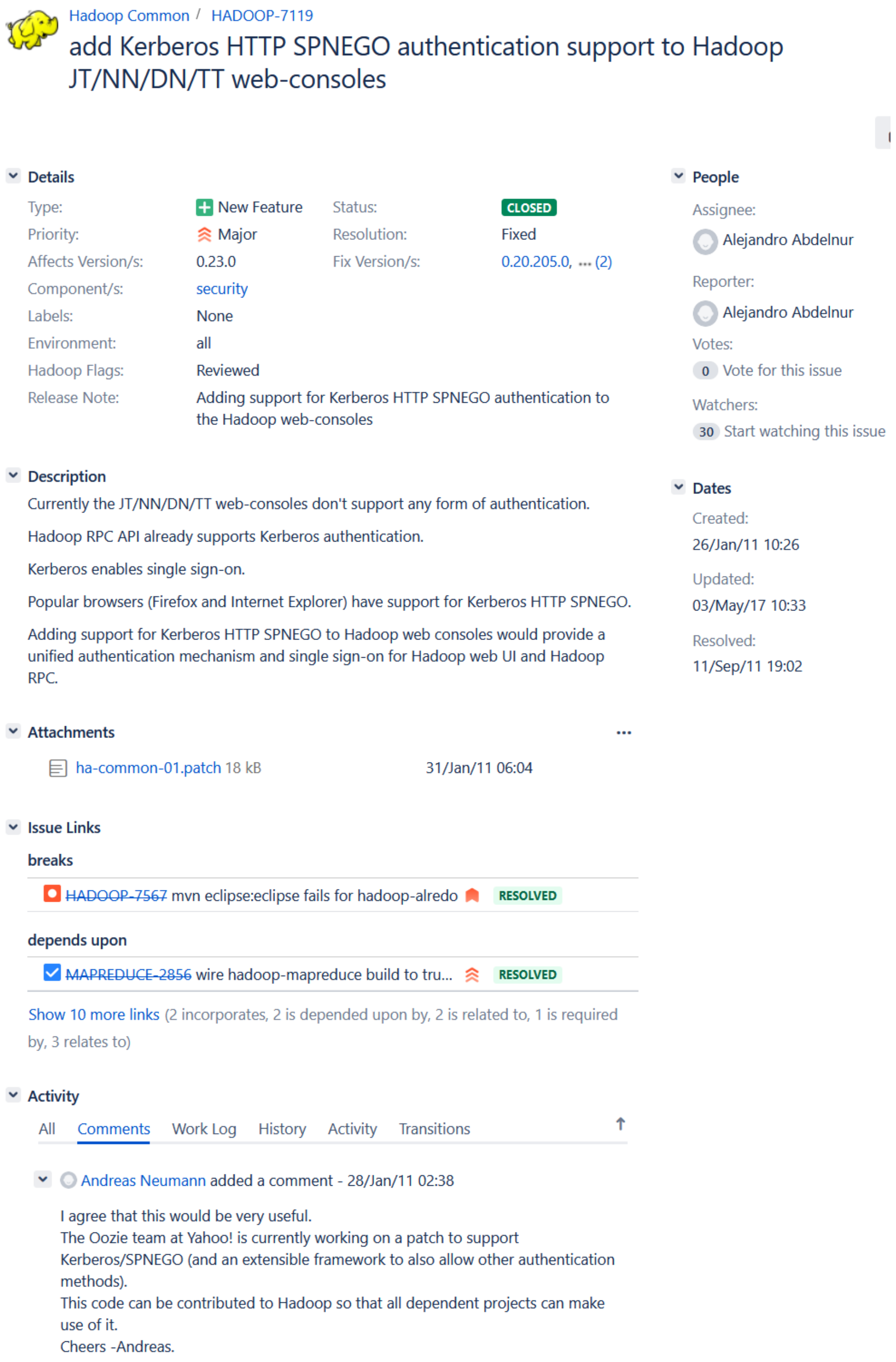
As followup work to ~~CASSANDRA-8457~~, we need to move streaming to use netty.

Streaming 2.0 (~~CASSANDRA-5286~~) brought many good improvements to how files are transferred between nodes in a cluster. However, the low-level details of the current streaming implementation does not line up nicely with a non-blocking model, so I think this is a good time to review some of those details and add in additional goodness. The current implementation assumes a sequential or "single threaded" approach to the sending of stream messages as well as the transfer of files. In short, after several iterative prototypes, I propose the following:

1) use a single bi-diredtional connection (instead of requiring to two sockets & two threads)
2) send the "non-file" `StreamMessage` s (basically anything not `OutboundFileMessage`) via the normal internode messaging. This will require a slight bit more management of the session (the ability to look up a `StreamSession` from a static function on `StreamManager`, but we have have most of the pieces we need for this already.
3) switch to a non-blocking IO model (facilitated via netty)
4) Allow files to be streamed in parallel (~~CASSANDRA-4663~~) - this should just be a thing already
5) If the entire sstable is to streamed, in addition to the DATA component, transfer all the components of the sstable (primary index, bloom filter, stats, and so on). This way we can avoid the CPU and GC pressure from deserializing the stream into objects. File streaming then amounts to a block-level transfer.

Note: The progress/results of CASSANDRA-11303 will need to be reflected here, as well.

Fig. 1. Example of an architectural issue in Apache Cassandra (CASSANDRA-12229[2]). The content in the purple box explain the current design of the system, and some of the shortcomings. Hence, it provides the basic reasoning why a change should be made. Next, the text in the green box explains what functionality and behaviour should be present, in place of what is currently there. This can be seen as a decision regarding interaction between components in the system. Because of this, this issue would be labelled as existence. The text in the red box explains that Netty should be used as a technology to implement non-blocking IO. This is a technology decision, making this issue also executive. The text in the blue box explains how making these changes allows for parallelization of communication, and alleviates CPU and garbage collection pressure. Hence, this issue improves the performance of the system. Hence, this issue would also be labelled as property. In the end, this issue would be labelled as being existence, executive, and property.



Hadoop Common / HADOOP-7119

add Kerberos HTTP SPNEGO authentication support to Hadoop JT/NN/DN/TT web-consoles

Details

Type:	New Feature	Status:	CLOSED
Priority:	Major	Resolution:	Fixed
Affects Version/s:	0.23.0	Fix Version/s:	0.20.205.0, ... (2)
Component/s:	security		
Labels:	None		
Environment:	all		
Hadoop Flags:	Reviewed		
Release Note:	Adding support for Kerberos HTTP SPNEGO authentication to the Hadoop web-consoles		

People

Assignee:
Alejandro Abdelnur

Reporter:
Alejandro Abdelnur

Votes:
0 Vote for this issue

Watchers:
30 Start watching this issue

Description

Currently the JT/NN/DN/TT web-consoles don't support any form of authentication.

Hadoop RPC API already supports Kerberos authentication.

Kerberos enables single sign-on.

Popular browsers (Firefox and Internet Explorer) have support for Kerberos HTTP SPNEGO.

Adding support for Kerberos HTTP SPNEGO to Hadoop web consoles would provide a unified authentication mechanism and single sign-on for Hadoop web UI and Hadoop RPC.

Dates

Created:
26/Jan/11 10:26

Updated:
03/May/17 10:33

Resolved:
11/Sep/11 19:02

Attachments

ha-common-01.patch 18 kB 31/Jan/11 06:04

Issue Links

breaks

~~HADOOP-7567~~ mvn eclipse:eclipse fails for hadoop-alredo RESOLVED

depends upon

~~MAPREDUCE-2856~~ wire hadoop-mapreduce build to tru... RESOLVED

Show 10 more links (2 incorporates, 2 is depended upon by, 2 is related to, 1 is required by, 3 relates to)

Activity

All Comments Work Log History Activity Transitions

Andreas Neumann added a comment - 28/Jan/11 02:38

I agree that this would be very useful.
The Oozie team at Yahoo! is currently working on a patch to support Kerberos/SPNEGO (and an extensible framework to also allow other authentication methods).
This code can be contributed to Hadoop so that all dependent projects can make use of it.
Cheers -Andreas.

Fig. 2. This is an example of an (architectural) issue from Apache Hadoop[4]. This issue contains most of the described issue characteristics. Note that we removed a couple of attachments and issue links from this image to make it fit. For the complete issue, refer to the Jira issue tracker of Apache Hadoop.

- *Sub-tasks*: some issues are linked with other issues in a parent/child relationship. This attribute represents the parent side of things: it is a list of all the child issues of the parent.
- *Issue-links*: links to other issues. Linked issues are related to the issue, but not in a child/parent or parent/child relationship. An example of an issue link could be a bug which has a link to the issue which accidentally introduced the bug.
- *Attachments*: a list of attachments added to the issue. Example attachments include design documents or explanatory graphs.
- *Components*: a list of software components of the system which are affected by the issue.
- *Labels*: labels added to the issue. These labels serve as tags which mark certain properties of the issue. For instance, the label "beginner issue" can be used to mark issues suitable for new contributors.
- *Dates*: each issue has a creation date. Additionally, an issue may have a last-updated date, and a resolved issue will have a resolution date.
- *Affected Versions*: versions of the software affected by the issue.
- *Fix Versions*: version of the software in which the particular issue should be fixed or resolved.

3.3 Deep Learning Background

In this section, we will provide an overview of background material on deep learning. Since deep learning is a very broad field, we will mostly restrict our discussion to those things which are also available in the software we will be using: Keras[5] and TensorFlow[6].

Deep learning is a sub-field of machine learning. The goal is to take labelled example points, turn these into numerical representations called feature vectors, train a model which learns a rule to map feature vectors to labels, and use the trained model to make predictions (we call this last part the *working phase* or *inference phase*). We will start with an overview of different types of classification problems. Next, we will cover feature generation. After that, we will cover basic neural network architectures. Next, we will discuss the basic method of training a neural network, and how to evaluate their performance. We will then discuss overfitting, dataset imbalance, and model selection.

3.3.1 Types of Classification Tasks

In machine learning, there are multiple types of problems we can solve with classifiers. Broadly speaking, there are regression and classification. Regression is the process of learning and predicting a continuous function, while classification is the process of predicting a discrete class label.

The most basic type of classification tasks, is binary classification. This can be seen as "yes/no" classification. The classical example of this is disease detection: a patient is either sick or not. In binary classification, there is usually one class called the positive class, and one class called the negative class. The positive class is the class of interest.

When we have more than two categories, we have a multi-class classification problem. An example of a multi-class classification problem, is categorising different types of fruits. In multi-class classification problems, it no longer makes sense to talk about a positive class and a negative class. It should also be noted that in multi-class classification, every training point should belong to exactly one class, and the classifier will predict a single class for every input.

If data points can have multiple labels, we have a multi-label classification problem. A multi-label classification problem could be assigning movies to different categories (e.g. action/comedy). A multi-label classification problem can be modelled as a set of binary classification problems: For every label, a data point should either have that label or not.

3.3.2 Feature Generation

Machine learning models generally accept numerical inputs, and often cannot deal with inputs in other forms directly. In this research, we will be dealing with text based inputs. These inputs have to be converted into numerical representations. In this section, we will explain a number of ways of obtaining numerical *feature vectors* from text data.

3.3.2.1 Bag of Words & TF-IDF

Bag of words, or term frequency, is one of the most basic feature generation methods for text. When using bag of words, we simply encode the word counts per document into a vector. We can also express this a bit more formally. Suppose that we have documents $D_1, \dots D_n$, and the corpus of all words present in the documents is $w_1, \dots, w_k$. Using a bag of words encoding, the i-th entry in the feature vector for document j will be the amount of occurrences of w_i in D_j [41].

A related variant is one where we divide each word frequency by the length of the document. We will refer to this as normalised bag of words.

TF-IDF (term frequency / inverse document frequency) is a refinement of bag of words. When using bag of words, common terms may become artificially important just because they occur frequently. The idea behind TF-IDF is that most likely, terms occurring in few documents will be important for classifiers [41]. Hence, TF-IDF weighs the word count (term frequency) for a word by the inverse document frequency, which is defined as

$$\text{idf}(w_i) = \log\left(\frac{n}{|\{D_j \mid w_i \in D_j\}|}\right) \qquad ([41])$$

3.3.2.2 Semantic Embeddings: Word2Vec & Doc2Vec

Bag of words and related models do not preserve the document structure, because all information is broken down into (weighted) word counts. Additionally, the features do not contain any information about the semantic meaning of words.

Semantic embeddings instead transform words or paragraphs into continuous vectors, such that similar words are mapped to similar vectors. The idea is that metric distance becomes some sort of measure of semantic distance [57].

Word2Vec is a family of algorithms for mapping words to semantic vectors. The idea with Word2Vec is that words that occur in similar contexts should be mapped to similar vectors, thus establishing the semantic closeness we described. The mapping is learned by training a neural network; the word vectors are obtained by taking the weights learnt by the network. The two original variants of Word2Vec are continuous bag of words and continuous skip-gram. In the continuous bag of words variant, the model aims to predict the current word w given a set of k preceding words, and k future words. On the other hand, the continuous skip-gram model does the inverse; it aims to predict the context, given the current word w [57].

Intuitively, it can be observed that the prediction task for continuous bag of words is easier than the one for continuous skip-gram; the latter requires a more thorough understanding of the semantic meaning of words. This is also reflected in the results obtained by Mikolov et al. when they introduced these

[5]`https://keras.io/`
[6]`https://www.tensorflow.org/`

two algorithms: continuous skip-gram took significantly longer to train (one versus three days for 3 epochs), but also achieved better accuracy (50% versus 15.5%) [57].

Doc2Vec is a technique which does not convert words, but entire paragraphs into vectors. It is based on the same techniques as Word2Vec, and also comes with two variants: PV-DM (Distributed Memory Model of Paragraph Vectors) and PV-DBOW (Distributed Bag of Words Model of Paragraph Vectors). The main change is that, besides the word input, a paragraph vector containing the context in the paragraph is also fed into the neural network. PV-DM is conceptually similar to continuous bag of words, in the sense that it has to predict the missing word, given the paragraph vector and the context. PV-DBOW is conceptually similar to continuous skip-gram; the model has to predict a small fragment of the paragraph, given the paragraph vector. PV-DM consistenly performs better than PV-DBOW [48].

3.3.3 Network Architectures

In this section, we will be covering different possible architectures for neural networks. We will be covering fully connected networks, convolutional networks, recurrent networks, activation functions, and the large language model "BERT".

3.3.3.1 Fully Connected Networks

One of the simplest and most well-known neural networks is the fully connected network. Such networks consist of layers of neurons, where each neuron in a layer is connected to every neuron in the previous layer. We call such layers *dense layers*. From now on, we will refer to networks consisting exclusively of dense layers as *fully connected neural networks*. An example of such a network is given in Figure 3.

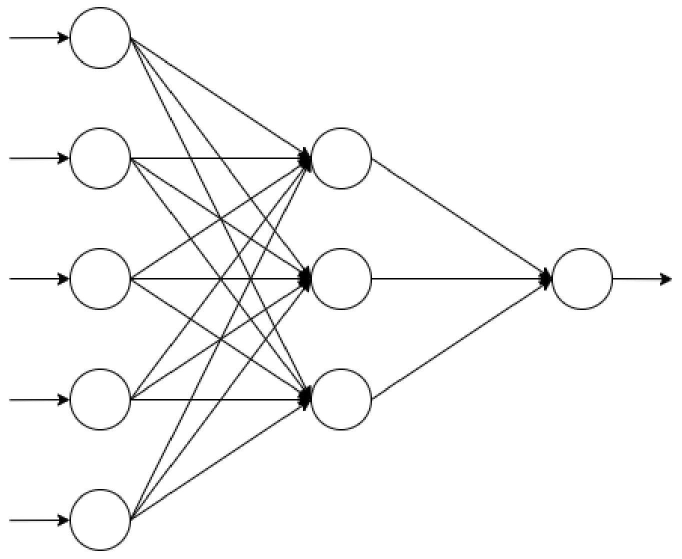


Fig. 3. Example of a fully connected neural network with an input layer of size 5, a hidden layer of size 3, and 1 output neuron.

Every neuron in a layer outputs a weighted sum of the outputs of the previous layer. Suppose that the neurons in the previous layer have outputs $v_1, \ldots, v_n$, and the neuron in consideration in the current layer has trained weights $w_1, \ldots, w_n$. The output of the neuron (also called the activity) is then given by

$$b + \sum_{i=1}^{n} w_i v_i$$

Here, b is a so-called bias term, which adds a constant factor to the output.

It can be seen that a fully connected neural network defines a function $f : \mathbb{R}^m \to \mathbb{R}^k$. A neural network is then trained to learn a function which maps given example data to given example targets.

However, with the scheme described above, neural networks would only be able to perform linear regression, since the activities of the neurons are just linear combinations of the activities of previous neurons. The real power of neural networks is unlocked by applying nonlinear transformations to those linear combinations. Hence, the output of a neuron would become

$$g\left(b + \sum_{i=1}^{n} w_i v_i\right)$$

where g is some nonlinear *activation function*. In Section 3.3.3.5, we will cover a number of common activation functions.

It is also common to arrange the weights of a single layer into a matrix W, and the bias terms in a vector $\boldsymbol{b}$. As a shorthand notation, the output of the complete layer is then denoted by $g(W\boldsymbol{v} + \boldsymbol{b})$

3.3.3.2 Convolutional Neural Networks

Convolution neural networks are a variant of neural networks which contain convolutions and so-called *pooling* layers. These types of networks can be designed to take in arbitrarily dimensional arrays as input. Hence, unlike fully connected networks, they can also take in matrices or tensors.

Every convolution layer contains one or more kernels which are applied to its input. In the case of a 2D input, like in Figure 4, this will result in a three-dimensional output. The trainable component in a convolutional neural network are the convolution kernels. A bias and activation function can also be applied to the output of the convolution layer.

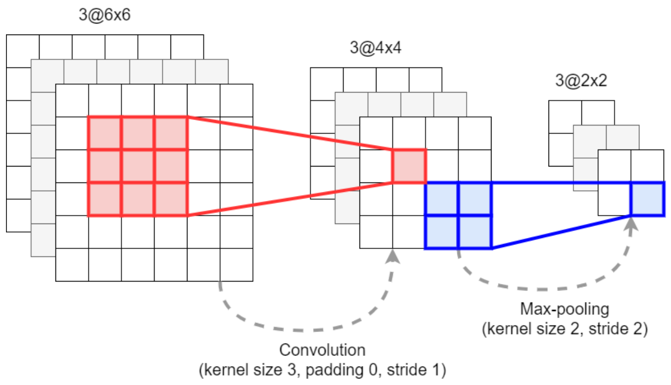

3@6x6
3@4x4
3@2x2
Max-pooling
(kernel size 2, stride 2)
Convolution
(kernel size 3, padding 0, stride 1)

Fig. 4. Example of how an input can be transformed by a convolutional neural network.

Pooling layers are layers which reduce or simplify the input. Somewhat similar to convolutions, these layers also apply a sliding window over their input. The inputs in the sliding window are then simplified (e.g. by taking the maximum or average). Through this process, the input is simplified or its dimension is reduced. Pooling layers contain no trainable parameters.

3.3.3.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are neural networks well-suited for handling sequential data, making them particularly effective for NLP tasks such as issue classification.

RNNs achieve this by retaining information from previous inputs. An RNN unit with a basic architecture, including its unfolded variant, is depicted in Figure 5. In this diagram, each word is processed one at a time by the RNN unit. The output of the next word depends on the current state of the RNN and the next word itself. The current state of the RNN, in turn, relies on the previous state of the RNN unit and the current word. Consequently, RNNs can propagate information along a sequence of words. As word meanings in our language depend on context, RNNs have the ability to consider the context (i.e. previous words) in order to determine word meanings.

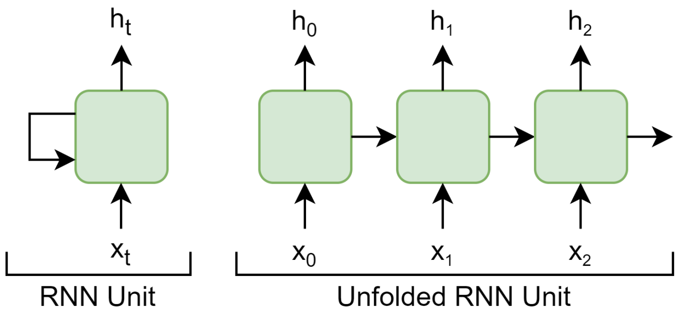


Fig. 5. This figure shows an RNN unit with a basic architecture on the left. On the right, the unfolded variant of this RNN unit is depicted. For the first input x_0 it cannot make use of previous inputs. The first output h_0 is therefore completely dependent on x_0. For the second input x_1, the output h_1 is both depending on the current input x_1 and the previous hidden state. This process goes on until the entire input is processed. When you think of each input x_t as a word, you can see that an RNN unit is able to take into account the context of the previous words.

However, the simple version of RNN described above suffers from difficulties during training. Either the error in the weight updates becomes large, leading to oscillating weights, or the updates become too small to retain any information from previous inputs [38].

To address this problem, Hochreiter and Schmidhuber introduced Long Short-Term Memory (LSTM) as a solution in [39] . LSTM utilizes gates to control the retention or forgetting of information. An LSTM unit's input consists of two components: the memory c_{t-1}, which the LSTM unit maintains over time, and the combination of the previous hidden state h_{t-1} and the current input x_t. It uses a forget gate to regulate the amount of existing memory to retain, an input gate to determine how much new memory (i.e., the previous hidden state and current input) is added to the existing memory, and an output gate to control the flow of existing memory to the output of the unit (h_t). Figure 6 provides a visual representation of an LSTM unit.

An initial limitation of the RNN was its ability to consider time dependencies in only one direction, typically from left to right. However, this is insufficient for many applications where, e.g. the meaning of a word depends on both preceding and subsequent words within a sentence.

To address this limitation, [66] introduced the concept of a bidirectional RNN. Bidirectional RNNs are trained in both forward and backward directions simultaneously, allowing them to consider the context of a word in both directions. [66] also demonstrated that bidirectional RNNs outperformed unidirectional RNNs in networks with similar computational requirements. Given this finding, we exclusively consider bidirectional RNNs in our research as well.

Another type of RNN unit is the gated recurrent unit (GRU) proposed by [10]. Unlike the LSTM unit, the GRU does not possess a dedicated memory unit, but it employs gates to control information flow. The reset gate determines the utilisation of information from the previous state, while the update gate governs the extent to which the current state can change. Figure 7 illustrates a graphical representation of the GRU.

3.3.3.4 BERT

A more recent and advanced model that has demonstrated state-of-the-art performance on NLP tasks is the Bidirectional Encoder Representations from Transformers (BERT) model [21]. Training BERT consists of two main stages: pre-training and fine-tuning. BERT undergoes pre-training on two tasks using unlabelled data, and the learned network parameters are then utilised to initialise the model for fine-tuning on a specific task of choice.

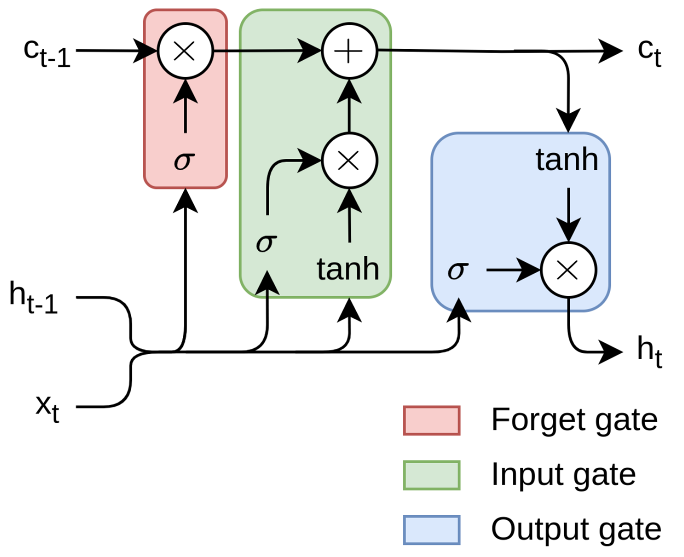


Fig. 6. A graphical illustration of an LSTM unit. The unit receives the memory state c_{t-1} and a concatenation of the previous hidden state h_{t-1} and the current input x_t (new memory). The forget gate applies a sigmoid σ function on the new memory. The forget gate essentially determines how much the memory state is updated with the new memory. The input gate determines how much of the new memory is added to the memory, again using a sigmoid function. Finally, the output gate determines how much of the new memory is used for the new hidden state.

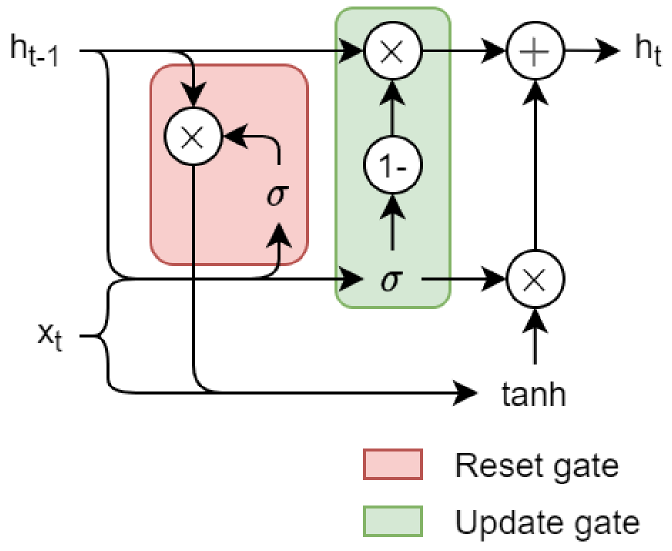


Fig. 7. A graphical illustration of a GRU. The input of the GRU consists of the previous hidden state h_{t-1} and the current input x_t. The unit itself consists of two gates. The first gate is the reset gate that determines how much the previous state h_{t-1} is used for computation. The update gate controls how much the hidden state can change with the new input.

BERT receives input in the form of token sequences, which include special tokens to indicate the beginning of a sequence ([CLS]) and to separate sentences ([SEP]). These input sequences are then processed through multiple transformer blocks. [21]

During pre-training, BERT is trained on two tasks. The first task is the masked language model (MLM), where a certain percentage (15%) of each input sequence is randomly masked, and the model is tasked with predicting the masked tokens. Since the masked tokens are not present during the fine-tuning phase, 10% of the time the masked token is substituted with a random token, while another 10% of the time it remains unchanged. The remaining 80% of the time, the token is replaced by the masked token. The second task in pre-training is next sentence prediction (NSP), which involves pairs of sentences. In half of the cases, the second sentence follows logically from the first sentence, while in the other half, a random sentence from the training corpus is used as the next sentence. BERT should then predict whether the second sentence is the follow-up sentence of the first. The first task enables BERT to model our language, while the second task helps the model understand the relationships between sentences. [21]

After completing the pre-training phase, BERT can be fine-tuned for specific tasks. In our case, this will involve issue classification. While pre-training requires substantial computational resources, fine-tuning is relatively cheap. This approach is known as transfer learning. During pre-training, BERT acquires a general understanding of language, and this knowledge can be transferred to specific tasks [21].

3.3.3.5 Activation Functions

As already explained previously, activation functions are what give neural networks their great power. In this section, we will cover the activation functions supported by Keras[7][8].

One popular class of activation functions, are the sigmoidal activation functions. Keras supports three different types of such activation functions: the sigmoid function, the hyperbolic tangent, and the softsign function. These functions can be seen in Figure 8a. The sigmoid function is the classical example of an activation function, and is defined as

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad ([26])$$

However, the sigmoid function is not zero-centred. In particular, this means that both negative and positive inputs are mapped to positive outputs. This can lead to slow or poor convergence [26, 1]. Because of this, it is generally advised to use zero-centred sigmoidal activation functions. The hyperbolic tangent is such a function, and has become the preferred alternative for sigmoidal activation [60]. The sigmoid function is mainly still used for cases where an output from 0 to 1 (e.g. a probability) is required [60].

However, both the sigmoid function and the hyperbolic tangent have very small derivatives for inputs with large absolute values. This leads to the so-called vanishing gradient problem: due to the small gradients, weights are hardly updated [26, 1, 60].

The softsign function is a sigmoidal activation function, which somewhat alleviates the problems with vanishing gradients encountered by other sigmoidal activation functions. It is defined as

$$\text{softsign}(x) = \frac{x}{1+|x|} \qquad ([1])$$

Another class of activation function, is that of the linear unit family. Keras supports 7 different types: ReLU, LeakyReLU, PReLU, ELU, SELU, GELU, and Swish. These are shown in Figures 8{b,c,d}. Originally, the Rectified Linear Unit (ReLU) activation function was designed as a means to overcome the vanishing gradient problem present in sigmoidal activation function, as well as overcoming their computational complexity [26, 60]. This is done by *rectifying* negative inputs to 0. ReLU is defined as follows:

$$\text{ReLU}(x) = \max\{0, x\} \qquad ([1])$$

ReLU has become a very popular activation function in the deep learning scene [26]. ReLU allows for faster learning, while also outperforming the sigmoid and hyperbolic tangent functions in terms of generalisability of the resulting models [1].

However, ReLU also has a number of drawbacks. It is more prone to overfitting, leading to a need to incorporate mechanisms to avoid this. Additionally, there is the dying neuron problem: neurons tend to become inactive because ReLU squishes many activities to zero. Many variants of ReLU have been developed, which avoid this problem by having a nonzero output for negative inputs [60].

The most simple such variants are the *Leaky Rectified Linear Unit (LeakyReLU)* and *Parametric Rectified Linear Unit (PReLU)*. The idea behind both is that for negative x, the activation should not be 0, but should instead be scaled linearly by some factor α. Both functions use the same basic formula

$$\text{leakyrelu}(x) = \text{prelu}(x) = \begin{cases} x & x \geq 0 \\ \alpha x & x < 0 \end{cases} \qquad ([26])$$

Here, $0 < \alpha < 1$ controls the scaling for negative x. The main difference is that for LeakyRelu, α is a hyperparameter, while it is a learnable parameter for PReLU. The latter was introduced because, in practice, it turned out to be difficult to find an appropriate value of α for use in LeakyReLU [26, 60].

There are also variants of ReLU that do not use linear decay for negative outputs. In Keras, these are the *Exponential Linear Unit (ELU)* and *Scaled Exponential Linear Unit (SELU)*. ELU is defined in the following way:

$$\text{elu}(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases} \qquad ([26])$$

Here, α is a hyperparameter. In contrast to LeakyRelu and PReLU, the idea behind ELU is that the mean activity converges towards zero. Additionally, for large negative x, the gradient of ELU is vanishing. It is claimed that this is not necessarily a bad thing, but instead introduces robustness to noise not present in LeakyRelu and PReLU [26, 1].

A variant of ELU is SELU. SELU is defined in the following way:

$$\text{selu}(x) = \begin{cases} \lambda x & x \geq 0 \\ \lambda\alpha(e^x - 1) & x < 0 \end{cases} \qquad ([60])$$

Here, $\lambda \approx 1.0507$ and $\alpha \approx 1.6733$ are fixed constants. The idea is that in a SELU network, activities convergence to have zero mean and unit variance [60, 26].

The *Gaussian Error Linear Unit (GELU)* was not designed to solve the vanishing gradient problem, but was designed with regularisation in mind. In particular, the design of GELU is strongly tied to dropout regularisation, in which random nodes in the network are "disabeled" each epoch. The idea behind GELU is that it simulates linear activation with dropout. The dropout is modelled according to $m \sim \text{Bernoulli}(\Phi(x))$, where $\Phi(x) = P(X \leq x)$ with $X \sim \mathcal{N}(0,1)$. Hence, the starting point for GELU is the identity function, multiplied by 0 with probability $1 - \Phi(x)$, and by 1 with probability $\Phi(x)$ [59, 37]. In order to obtain a deterministic activation function, the expectation is computed, which gives the following definition for GELU:

$$\text{gelu}(x) = x\Phi(x) = \frac{x}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right] \qquad ([37])$$

The final activation, in the linear unit family we will be discussing, is the Swish activation function. The Swish function is defined by

[7]`https://keras.io/api/layers/activations/`

[8]`https://www.tensorflow.org/api_docs/python/tf/keras/activations`

$$\text{swish} = x\sigma(x) \quad ([61])$$

Swish is also called the *Sigmoidal Linear Unit (SiLU)* [37]. Swish has been shown to outperform the ReLU function in many situations [62, 37]. However, GELU may perform even better than Swish [37]. It should be noted that some authors define Swish using a parameter β as follows:

$$\text{swish}_\beta = x\sigma(\beta x) \quad ([26])$$

However, we will adhere to the convention used by TensorFlow and Keras. It should be noted that for $\beta \approx 1.702$, this Swish function is an approximation of the GELU function [37].

The softplus function is not really a linear unit, but is still somewhat related to them. The softplus function is designed as follows:

$$\text{softplus}(x) = \log(e^x + 1) \quad [60]$$

For large negative x, $\log(e^x + 1) \approx \log(1) = 0$. For large positive x, $\log(e^x + 1) \approx \log(e^x) = x$. Hence, this function can be seen as a smooth approximation of the ReLU function [60]. The function is plotted in Figure 8e. Softplus has been shown to outperform ReLU and Swish in some cases [60].

Up until this point, we discussed activation functions which take as input the output of a single neuron, and transform this input independent of the outputs of other neurons in the layer. Now, we will discuss a somewhat different activation function: the softmax function. The softmax function is used for multi-class classification problems, and its output is a probability distribution [60]. Suppose that the current layer has neurons with untransformed outputs $y_1, \ldots, y_n$. Each output is then transformed as follows:

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}} \quad ([60])$$

It can easily be seen that this defined as probability distribution, where $\text{softmax}(y_i)$ defines the probability of the input sample belonging to class i [60].

All activation functions here can be used to construct universal approximators; networks with sufficiently many layers and neurons can approximate any function to arbitrary precision. This is even the case for a simple function such as ReLU, which is conceptually close to a linear function [7].

3.3.4 Training a Network

In this section, we will discuss the basic concepts required to train a network with some given architecture. We will discuss the optimisation goals, as well as the algorithms which can be used to train a neural network.

3.3.4.1 Loss Functions

At its core, training a neural network is optimising the difference between the predictions made by the network and the ground truths, where the difference is measured using a so-called *loss function*. Hence, training a neural network is a minimisation task, where we aim to minimise the loss [31].

First, suppose that we have c classes $1 \ldots c$. Additionally, we have n labelled data points $(\boldsymbol{x}_i, \boldsymbol{y}_i)$, where $\boldsymbol{x}_i \in \mathbb{R}^m$ is the measured data point, and $\boldsymbol{y}_i \in \mathbb{B}^c$ the corresponding ground truth; specifically, $\boldsymbol{y}_i$ is one-hot encoded, which means that exactly one entry is equal to 1, which is the index of the actual class the data point $\boldsymbol{x}_i$ belongs to. Finally, we have a classifier f, which maps feature vectors $\boldsymbol{x}_i$ to confidences p_{ij}, where p_{ij} denotes the confidences that x_i belongs to class j. Note that the range of possible values for the confidences is determined by the activation function used.

A loss is now a function L which maps $\boldsymbol{y}_i$ and $\boldsymbol{p}_i$ to some real number. The loss can then be used to define the empirical risk, which is the average loss over all samples:

$$R^{\text{emp}} = \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{y}_i, \boldsymbol{p}_i)$$

The empirical risk is the function which is eventually optimised after training the neural network.

Neural networks are trained using a gradient descent algorithm. As such, the loss function must be differentiable [31]. Hence, we cannot use a loss function such as the counting loss (which simply counts the number of misclassifications) as a loss function for classification tasks, even though this might look like an intuitive choice.

In the remainder of this section, we will cover the three main loss functions for classification offered by Keras: Cross entropy, Kullback-Leibler divergence, and the hinge loss [9].

The cross entropy and Kullback-Leibler divergence loss functions are both so-called *probabilistic loss functions*; they assume both the ground truths y_i and predictions $f(x_i)$ represent probabilities, and provide a measure of disagreement between the two. In fact, for this loss to be applicable, y_i does not have to be a binary vector, but can be a vector of probabilities [13]. The cross entropy loss function is based on the concept of entropy from information theory, and is defined as follows ([13]):

$$L_{\text{CE}}(y_i, p_i) = -\sum_{j=1}^{c} \left[y_{ij}\log(p_{ij}) + (1 - y_{ij})\log(1 - p_{ij})\right]$$

To contrast, the Kullback-Leibler divergence loss is defined as ([13])

$$L_{\text{KL}}(y_i, p_i) = \sum_{j=1}^{c} \left[y_{ij}\log\left(\frac{y_{ij}}{p_{ij}}\right) + (1 - y_{ij})\log\left(\frac{1 - y_{ij}}{1 - p_{ij}}\right)\right]$$

Note that, in general, we have $x\log(y/x) = x\log(y) - x\log(x)$. As such, we can rewrite the Kullback-Leibler divergence loss in terms of the cross entropy loss:

$$\begin{aligned} &L_{\text{KL}}(y_i, p_i) \\ &= L_{\text{CE}}(y_i, p_i) + \sum_{j=1}^{c} \left[y_{ij}\log(y_{ij}) + (1 - y_{ij})\log(1 - y_{ij})\right] \end{aligned}$$

The latter term is simply a constant factor. As such, the cross entropy loss and Kullback-Leibler divergence loss are equivalent [13]. In this work, we will restrict our attention to the cross entropy loss. As a final remark about cross entropy, we note that for binary classification problems, the cross entropy is simply given by

$$L_{\text{CE}}(y_i, p_i) = -\left[y_i\log(p_i) + (1 - y_i)\log(1 - p_i)\right]$$

Hence, in this scenario, we only count the term for the positive class.

The other major class of losses, is the class of margin losses. The Hinge loss is such a loss, and for binary classification problems it is defined as ([13])

$$L_H(y_i, p_i) = \max\{0, 1 - (2 * y_i - 1)p_i\}$$

Note that the awkward $2y_i - 1$ term stems from the fact that we assumed ground truths to be either 0 or 1, while the binary hinge requires the values -1 and 1 [13].

[9] `https://keras.io/api/losses/`

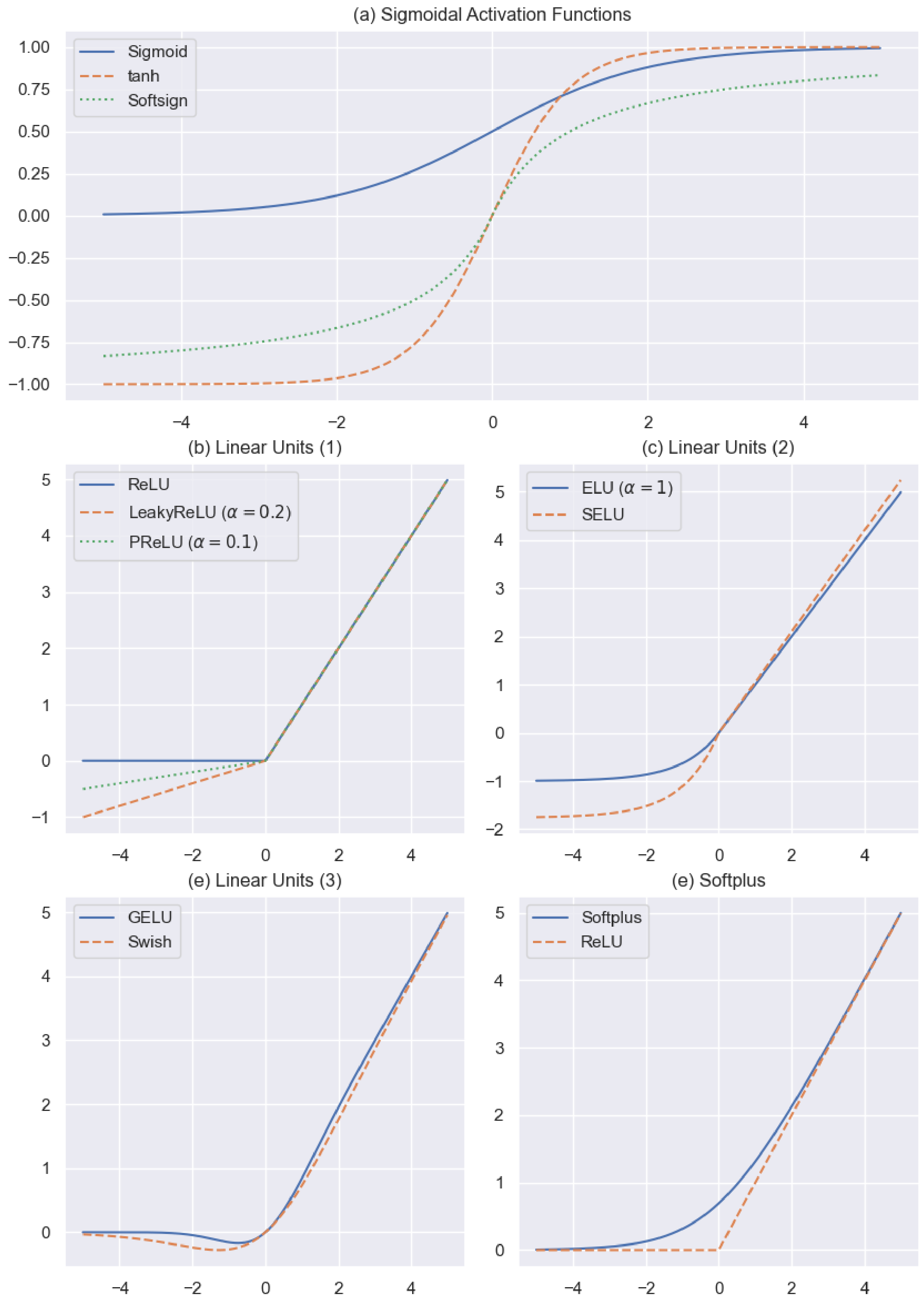


Fig. 8. Plots of different activation functions supported by Keras.

We will give a sketch for an intuition behind the definition of this loss function, in order to contrast it with cross entropy. The main idea behind margin losses, is that the classifier defines a decision boundary between the two classes. The quantity $y_i p_i$ defines a measure of how close the point x_i is to this decision boundary.

For misclassified points, the signs of $2y_i - 1$ and p_i differ, and $L_H(y_i, p_i) > 1$. For points close to the decision boundary but correctly classified, $0 < L_H(y_i, p_i) < 1$. If the point is far away (i.e. the classification is correct, and the model is pretty certain), $y_i p_i \geq 1$, and thus the loss will be 0; only for points close to the decision boundary (the model is not certain) will the loss be nonzero. The idea behind this is that the classifier pays attention to the "hardest" examples in the dataset by 1) assigning the highest loss for misclassified samples, 2) having the loss decay linearly as the distance to the decision boundary decreases, and 3) having the loss be 0 if the distance is sufficiently far [7]. This is in contrast with cross-entropy, where a penalty is always applied regardless of distance to the decision boundary; the cross entropy loss is more aimed towards *exactly* matching the ground truths [7, 13]. In particular, it may occur that a greater penalty is applied for a correctly classified sample than for a misclassified one.

The multi-class generalisation of the hinge loss used by Keras[10] is defined as follows:

$$L(y_i, p_i) = \max\left\{ \max_{1 \leq j \leq c} (1 - y_{ij}) p_{ij} - \sum_{j=1}^{c} y_{ij} p_{ij} + 1, 0 \right\}$$

In order to understand this definition, it is important to note that y_i is one-hot encoded for multi-class problems. In this case, the first term (the maximum) computes the maximum confidence given by the classifier for any negative class. Next, the second term (the sum) computes the confidence given by the classifier for the positive class. This categorical hinge loss is now zero if the confidence given for the (true) positive class is at least one higher than the next highest confidence for any of the negative classes; otherwise, the loss is linearly proportional to the difference between the two [16].

A variant on the Hinge loss is the squared Hinge loss. This loss is defined as the hinge loss raised to the power of two. This loss is more sensitive to outliers (high loss values), and less sensitive to low loss samples; the function $x \to x^2$ maps values < 1 to smaller values, and values > 1 to larger values [83].

3.3.4.2 Regularisation

Since deep learning models are very powerful, they have a tendency to overfit. This means that the models become too specific to the training data, and especially to the noise in the training data. This causes them to generalise poorly to novel data. One way to avoid this is through the use of regularisation: the addition of penalties to the loss function [31].

The idea behind regularisation is that generally, the weights learned by the network should be small; very large weights are generally a sign of overfitting. To avoid this, a penalty for large weights is added to the empirical risk. The new empirical risk then becomes

$$\overline{R^{\text{emp}}} = R^{\text{emp}} + \sum_{j=1}^{m} \left(\alpha_j L^1(\boldsymbol{w}_j) + \beta_j L^2(\boldsymbol{w}_j) \right)$$

Here, the sum is over all weight vectors/matrices in the network, and L^p denotes the p-norm. α_j and β_j are parameters for specifying the degree of regularisation. When $\alpha_j \neq 0$, we have L^1-regularisation. Similarly, when $\beta_j \neq 0$, we have L^2-regularisation [31].

[10] `https://www.tensorflow.org/api_docs/python/tf/keras/losses/CategoricalHinge`

Instead of penalising large weights, we can also decrease the size of the weights each time we update them in the training algorithm, by subtracting the weights multiplied by some number λ from the weights themselves. This is called weight decay [31].

3.3.4.3 The Basic Algorithm

Now that we have discusses what a neural network is actually optimising, we can explain how to optimise the loss. Neural networks are trained using a gradient descent algorithm. Given a loss function L and labelled dataset (x_i, y_i), $1 \leq i \leq n$, we want to train the network to learn a function $f(x; \boldsymbol{w})$, parameterised by the weights w of the neural network, which minimises the *empirical risk*

$$\frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x; \boldsymbol{w}))$$

The idea behind gradient descent, is that the gradient $\nabla_{\boldsymbol{w}}$ of some function points in the direction of the steepest increase of that function, while $-\nabla_{\boldsymbol{w}}$ thus points in the direction of the steepest descent. The idea is to update the weights according to the rule

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla_{\boldsymbol{w}} \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(x; \boldsymbol{w}))$$

We thus update the weights in the direction of the steepest descent of the empirical risk (w.r.t. $\boldsymbol{w}$), and "move" the weights in this direction in order to arrive at a minimum where the gradient is zero as fast as possible. Here, η is a so-called learning rate, which is used to control the speed of the descent [31, 7].

However, computing the gradient using all samples in the training set is expensive. Hence, it is common to instead approximate this gradient by using mini-batches of size b, sampled from the full dataset. This variant is called stochastic gradient descent – whereas the variant using the entire dataset for every update is called batch gradient descent [31, 7].

Because we approximate the gradient in stochastic gradient descent (SGD), the estimates we obtain when we are at a local minimum are not exactly zero; the values for the gradient are just small. In order to guarantee convergence, the learning rate must be decreased over time. We thus do not have a singular learning rate η, but a sequence of learning rates $\eta_1, \eta_2, \ldots$ [31]. In order to guarantee convergence, these learning rates must satisfy

$$\sum_{k=1}^{\infty} \eta_k \to \infty, \qquad \sum_{k=1}^{\infty} \eta_k^2 < \infty$$

3.3.4.4 Optimisers

The basic algorithm described in the previous section can be formalised using the pseudocode in Algorithm 1. An *optimiser* is an algorithm which takes as an argument the gradient estimate g, and computes the weight update Δw. Hence, in Algorithm 1, line 6 contains the actual optimiser.

One typical variation on Algorithm 1, is that often the model is trained for a set number of epochs. Every epoch, each item in the dataset is presented once as a training sample, possible contained in some mini-batch. This may also mean that in practice, the mini-batches are not sampled uniformly with replacement [7]. For example, if there are 1000 training samples, the batch size is 50, and the model is trained for 20 epochs, then there will be a total of 1000 / 50 * 20 = 400 individual weight updates.

In the remainder of this section, we will discuss a number of different optimisers. We will present each optimiser in the same style as Algorithm 1, but the considerations regarding epochs and mini-batches outlined above may apply to any optimiser.

One possible improvement to stochastic gradient descent, is stochastic gradient descent with momentum. In this scheme, the update step v (also called the velocity in this case) is an exponentially decaying average of the successive negative gradients; this means that the velocity is updated to equal α times the old velocity, and $1 - \alpha$ times the current negative gradient. This makes the training process more resilient to noisy gradients (i.e. the gradients differ a lot per mini-batch). There is one single hyperparameter α (the momentum parameter) which dictates the rate of decay [31]. The formal pseudocode for the optimiser is given in Algorithm 2.

An alternative to SGD with momentum, is SGD with Nesterov momentum. With Nesterov momentum, first temporary weights $\overline{w}$ are computed with the current velocity, then the gradient is computed with these weights, the velocity is updated, and then the actual weights are updated again with the new velocity. This can be interpreted as adding some sort of correction to regular momentum [31]. The algorithm can be found in Algorithm 3.

Another algorithm is AdaGrad. The idea behind AdaGrad is to scale the learning rate of every model parameter w_i inversely proportional to the square root of the sum of all historical squared values of the gradient [31]. Concretely, this means that if for parameter w_i, the historical gradients are $0.1, 0.2,$ and 0.15, then the learning rate will be scaled inversely proportional to $\sqrt{0.1^2 + 0.2^2 + 0.15^2}$. The effect is that the learning rate is reduced more quickly for parameters with large corresponding partial derivatives. The effect of this process is greater in the directions with less extreme slopes. The AdaGrad algorithm has no need for a decreasing learning rate, and has no real hyperparameters (aside from a small constant δ used for numerical stability) [31]. The AdaGrad algorithm in pseudocode is given in Algorithm 4.

RMSProp is a modified version of AdaGrad, which performs better for nonconvex optimisation problems. This makes it more widely applicable. The mechanism behind RMSProp is similar to momentum; the accumulated squared gradients from the past are made less important by employing an exponentially decaying average [31]. The implementation of RMSProp in pseudocode is given in Algorithm 5. Experience with AdaGrad and RMSProp has shown that the latter is effective in practice, while AdaGrad is more likely to perform poorly.

AdaDelta is also an improved version of AdaGrad. Like RMSProp, it also solves the problem that in AdaGrad, gradients will eventually vanish because of the infinitely stored history [82, 31]. AdaDelta achieves this through the same mechanism as RMSProp: by using an exponentially decaying average. However, Adedelta is different from RMSProp in that it also does something else: AdaDelta uses the squared gradients in order to estimate the diagonal of the Hessian matrix. The idea is that, under the correct circumstances, this will also lead to second order corrections in the update step. A final benefit of AdaDelta is that it does not require the specification of an initial learning rate [82]. The AdaDelta algorithm is shown in Algorithm 6.

Another optimiser is Adam. Adam makes use of both an exponentially decaying average of the gradient ("first moment"; this also implements the concept of momentum), and an exponentially decaying average of the squares of the gradient ("second moment"). Compared to RMSProp, Adam suffers from less bias at the start of the training. Similar to RMSProp, Adam is a popular choice for use in deep learning [31]. The pseudocode for Adam is given in Algorithm 7. Note that nowadays, the parameter δ which was initially meant for numerical stability, is sometimes also considered a hyperparameter which should be optimised [11].

AdaMax is a variant of Adam, based on the observation that Adam scales the gradients of individual weights according to the L^2 norm of the current and past gradients. Another option is to use the infinity norm (max function) instead. This leads to the algorithm given in Algorithm 8 [44]. The performance of AdaMax is comparable with that of Adam and RMSProp [79].

AdaFactor is a variant of Adam which was designed to reduce memory usage. Adam stores two moving averages: the first and second moment. For weights stored in a matrix $W \in \mathbb{R}^{m \times n}$, this requires $\mathcal{O}(mn)$ space. The core idea behind AdaFactor is to instead approximate the moving average V of the squared gradients with two matrices $R \in \mathbb{R}^{m \times 1}$, $S \in \mathbb{R}^{1 \times n}$ such that $RS \approx V$. This reduces the memory requirement to $\mathcal{O}(m+n)$. At the same time, AdaFactor ignores the first moment. Hence, AdaFactor is essentially Adam with $\beta_1 = 0$, and using an approximation of the moving average of squared gradients. Finally, AdaFactor suffers from too large updates when β_2 is too large. To avoid this, it performs clipping; if the update step would be too large, it is limited [69]. The pseudocode for AdaFactor can be found in Algorithm 9.

AdamW is a variant of Adam which includes weight decay. Here, weight decay means that on every update step, a quantity λw is subtracted from the weight w, effectively decreasing their magnitude [51]. The idea behind this is to prevent overfitting and improve generalisation, since large weights are generally associated with overfitting [31].

The final algorithm we will consider is Nadam. Nadam incorporates the observations that both Adam and stochastic gradient descent with Nesterov momentum are powerful algorithms, and combines the two into one [24]. This leads to the algorithm in Algorithm 10. As an extension to this, in TensorFlow, it is also possible to use Nadam in combination with weight decay (similar to how AdamW is Adam with weight decay) [11].

Among all the optimiser we discussed up until this point, no single one can be seen as the best [31]. However, testing has shown that, given enough fine-tuning of the parameters, a more specific algorithm will never perform worse than a more general algorithm [11]. We have the following generality relations between optimisers ([11]):

$$\text{SGD} \subseteq \text{SGD w/ Momentum} \subseteq \text{Adam} \subseteq \text{AdamW}$$
$$\text{SGD} \subseteq \text{SGD w/ Momentum} \subseteq \text{RMSProp}$$
$$\text{SGD} \subseteq \text{SGD w/ Nesterov} \subseteq \text{Nadam}$$

Furthermore, research has shown that Adam en Nadam will generally outperform RMSProp [11]. In particular, this also means that AdamW will never underperform RMSProp, because AdamW reduces to Adam when the weight decay is set to zero [51].

In practice, Adam is one of the most widely adopted optimisers [31]. However, recent insights suggest that SGD combined with (Nesterov) momentum leads to more generalisable models than Adam [33, 51]. In order to avoid this, good regularisation is required, which was done with the introduction of AdamW [51].

It should be noted that all optimisers above, which use a global learning rate, can also be augmented to use a scheduled (decaying) learning rate (multiplier). In fact, it has been shown that doing so can lead to better performance in the case of Adam [51].

3.3.4.5 Training, Validation, and Test Sets

When training neural networks, we usually use three different data sets, obtained by splitting our full dataset into three disjoint subsets. The training set, which usually contains the majority of the data, is used for actually updating the weights of the network [31].

The validation set is used to "monitor" the training process; after every update step, the performance of the network on the validation set is recorded. This is done in order to estimate the performance of the network on data foreign to the training set. The validation set plays a more important role in model selection and early stopping, which we will discuss later [31].

The testing set is used to determine the actual performance of the network on novel data, and is used to obtain the actual performance numbers for a network [7, 31]. The validation set

[11]`https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/experimental/Nadam`

Algorithm 1: The basic algorithm for stochastic gradient descent

```
Input: Labelled data points $D = (x_i, y_i)$
Parameters: batch size $m$, learning rate schedule $\eta_k$
Result: Weight vector $\boldsymbol{w}$ which minimises the empirical risk
1  $\boldsymbol{w} \leftarrow$ initialise;
2  $k \leftarrow 1$;
3  while not converged do
4      $b \leftarrow$ sample $m$ items $\{(x'_1, y'_1), \ldots, (x'_m, y'_m)\}$ from $D$;
5      $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{w}} \sum_{i=1}^{m} L(y'_i, f(x'_i; \boldsymbol{w}))$;
6      $\Delta\boldsymbol{w} = -\eta_k \boldsymbol{g}$;
7      $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w}$;
8      $k \leftarrow k + 1$;
9  end
```

Algorithm 2: SGD with momentum

```
Input: Labelled data points $D = (x_i, y_i)$
Parameters: batch size $m$, initial velocity $\boldsymbol{v}$, learning rate schedule $\eta_k$, momentum parameter $\alpha$
Result: Weight vector $\boldsymbol{w}$ which minimises the empirical risk
1  $\boldsymbol{w} \leftarrow$ initialise;
2  $k \leftarrow 1$;
3  while not converged do
4      $b \leftarrow$ sample $m$ items $\{(x'_1, y'_1), \ldots, (x'_m, y'_m)\}$ from $D$;
5      $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{w}} \sum_{i=1}^{m} L(y'_i, f(x'_i; \boldsymbol{w}))$;
6      $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \eta_k \boldsymbol{g}$;
7      $\Delta\boldsymbol{w} \leftarrow \boldsymbol{v}$;
8      $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w}$;
9      $k \leftarrow k + 1$;
10 end
```

Algorithm 3: SGD with Nesterov momentum

```
Input: Labelled data points $D = (x_i, y_i)$
Parameters: batch size $m$, initial velocity $\boldsymbol{v}$, learning rate schedule $\eta_k$, momentum parameter $\alpha$
Result: Weight vector $\boldsymbol{w}$ which minimises the empirical risk
1  $w \leftarrow$ initialise;
2  $k \leftarrow 1$;
3  while not converged do
4      $b \leftarrow$ sample $m$ items $\{(x'_1, y'_1), \ldots, (x'_m, y'_m)\}$ from $D$;
5      $\overline{\boldsymbol{w}} \leftarrow \boldsymbol{w} + \alpha\boldsymbol{v}$;
6      $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\overline{\boldsymbol{w}}} \sum_{i=1}^{m} L(y'_i, f(x'_i; \overline{\boldsymbol{w}}))$;
7      $\boldsymbol{v} \leftarrow \alpha\boldsymbol{v} - \eta_k \boldsymbol{g}$;
8      $\Delta\boldsymbol{w} \leftarrow \boldsymbol{v}$;
9      $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w}$;
10     $k \leftarrow k + 1$;
11 end
```

Algorithm 4: AdaGrad

```
Input: Labelled data points $D = (x_i, y_i)$
Parameters: batch size $m$, Small parameter $\delta$ for numerical stability, Global learning rate $\eta$
Result: Weight vector $\boldsymbol{w}$ which minimises the empirical risk
1  $\boldsymbol{w} \leftarrow$ initialise;
2  $\boldsymbol{r} \leftarrow 0$;
3  while not converged do
4      $b \leftarrow$ sample $m$ items $\{(x'_1, y'_1), \ldots, (x'_m, y'_m)\}$ from $D$;
5      $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{w}} \sum_{i=1}^{m} L(y'_i, f(x'_i; \boldsymbol{w}))$;
6      $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$;
7      $\Delta\boldsymbol{w} \leftarrow -\frac{\eta}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$;
8      $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w}$;
9  end
```

Algorithm 5: RMSProp

```
Input: Labelled data points $D = (x_i, y_i)$
Parameters: batch size $m$, Small parameter $\delta$ for numerical stability, Global learning rate $\eta$
Result: Weight vector $\boldsymbol{w}$ which minimises the empirical risk
1  $\boldsymbol{w} \leftarrow$ initialise;
2  while not converged do
3      $b \leftarrow$ sample $m$ items $\{(x'_1, y'_1), \ldots, (x'_m, y'_m)\}$ from $D$;
4      $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{w}} \sum_{i=1}^{m} L(y'_i, f(x'_i; \boldsymbol{w}))$;
5      $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1 - \rho)\boldsymbol{g} \odot \boldsymbol{g}$;
6      $\Delta\boldsymbol{w} \leftarrow -\frac{\eta}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$;
7      $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w}$;
8  end
```

Algorithm 6: AdaDelta

```
Input: Labelled data points $D = (x_i, y_i)$
Parameters: batch size $m$, Small parameter $\delta$ for numerical stability, parameter $\rho \in (0, 1)$
Result: Weight vector $\boldsymbol{w}$ which minimises the empirical risk
1  $\boldsymbol{w} \leftarrow$ initialise;
2  $\boldsymbol{r} \leftarrow 0$;
3  $\boldsymbol{v} \leftarrow 0$;
4  while not converged do
5      $b \leftarrow$ sample $m$ items $\{(x'_1, y'_1), \ldots, (x'_m, y'_m)\}$ from $D$;
6      $g \leftarrow \frac{1}{m} \nabla_{\boldsymbol{w}} \sum_{i=1}^{m} L(y'_i, f(x'_i; \boldsymbol{w}))$;
7      $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1 - \rho)\boldsymbol{g} \odot \boldsymbol{g}$;
8      $\Delta\boldsymbol{w} \leftarrow -\frac{\sqrt{\boldsymbol{v} + \delta}}{\sqrt{\boldsymbol{r} + \delta}} \boldsymbol{g}$;
9      $\boldsymbol{v} \leftarrow \rho\boldsymbol{v} + (1 - \rho)\Delta\boldsymbol{w} \odot \Delta\boldsymbol{w}$;
10     $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w}$;
11 end
```

Algorithm 7: Adam

```
Input: Labelled data points $D = (x_i, y_i)$
Parameters: batch size $m$, step size $\eta$, parameters
            $\beta_1, \beta_2 \in [0, 1)$, parameter $\delta$
Result: Weight vector $\boldsymbol{w}$ which minimises the empirical
        risk
1  $\boldsymbol{w} \leftarrow$ initialise;
2  $\boldsymbol{s} \leftarrow 0$;
3  $\boldsymbol{r} \leftarrow 0$;
4  while not converged do
5      $b \leftarrow$ sample $m$ items $\{(x'_1, y'_1), \dots, (x'_m, y'_m)\}$ from $D$;
6      $g \leftarrow \frac{1}{m} \nabla_{\boldsymbol{w}} \sum_{i=1}^{m} L(y'_i, f(x'_i; \boldsymbol{w}))$;
7      $\boldsymbol{s} \leftarrow \beta_1 \boldsymbol{s} + (1 - \beta_1)\boldsymbol{g}$;
8      $\boldsymbol{r} \leftarrow \beta_2 \boldsymbol{r} + (1 - \beta_2)\boldsymbol{g} \odot \boldsymbol{g}$;
9      $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\beta_1^t}$;
10     $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\beta_2^t}$;
11     $\Delta\boldsymbol{w} \leftarrow -\eta \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}+\delta}}$;
12     $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w}$;
13 end
```

Algorithm 8: AdaMax

```
Input: Labelled data points $D = (x_i, y_i)$
Parameters: batch size $m$, step size $\eta$, parameters
            $\rho_1, \rho_2 \in [0, 1)$, parameter $\delta$
Result: Weight vector $\boldsymbol{w}$ which minimises the empirical
        risk
1  $\boldsymbol{w} \leftarrow$ initialise;
2  $\boldsymbol{s} \leftarrow 0$;
3  $\boldsymbol{r} \leftarrow 0$;
4  $k \leftarrow 1$;
5  while not converged do
6      $b \leftarrow$ sample $m$ items $\{(x'_1, y'_1), \dots, (x'_m, y'_m)\}$ from $D$;
7      $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{w}} \sum_{i=1}^{m} L(y'_i, f(x'_i; \boldsymbol{w}))$;
8      $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$;
9      $\boldsymbol{r} \leftarrow \max\{\rho_2 \boldsymbol{r}, |\boldsymbol{g}|\}$;
10     $\Delta\boldsymbol{w} \leftarrow -\frac{\rho_1}{1-\rho_2^k} \frac{\boldsymbol{s}}{\boldsymbol{r}}$;
11     $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w}$;
12     $k \leftarrow k + 1$;
13 end
```

Algorithm 9: AdaFactor. Note that we only provide the matrix variant. AdaFactor also has a vector variant [69].

```
Input: Labelled data points $D = (x_i, y_i)$
Parameters: batch size $m$, step size schedule $\eta_k$,
            regularisation parameters $\epsilon_1, \epsilon_2, \beta_2 \in (0, 1)$
Result: Weight vector $\boldsymbol{w}$ which minimises the empirical
        risk
1  $\boldsymbol{w} \leftarrow$ initialise;
2  $\boldsymbol{r} \leftarrow 0$;
3  while not converged do
4      $b \leftarrow$ sample $m$ items $\{(x'_1, y'_1), \dots, (x'_m, y'_m)\}$ from $D$;
5      $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{w}} \sum_{i=1}^{m} L(y'_i, f(x'_i; \boldsymbol{w}))$;
6      $\alpha \leftarrow \max\{\epsilon_2, \mathrm{RMS}(\boldsymbol{w})\}\eta_k$;
7      $\boldsymbol{r} \leftarrow \beta_2 \boldsymbol{r} + (1 - \beta_2)(\boldsymbol{g} \odot \boldsymbol{g} + \epsilon_1 \mathbf{1}_n \mathbf{1}_m^T)\mathbf{1}_m$;
8      $\boldsymbol{c} \leftarrow \beta_2 \boldsymbol{c} + (1 - \beta_2)\mathbf{1}_n^T(\boldsymbol{g} \odot \boldsymbol{g} + \epsilon_1 \mathbf{1}_n \mathbf{1}_m^T)$;
9      $V \leftarrow \boldsymbol{rc}/(\mathbf{1}_n^T \boldsymbol{r})$;
10     $\boldsymbol{u} \leftarrow \boldsymbol{g}/\sqrt{V}$;
11     $\hat{\boldsymbol{u}} \leftarrow \boldsymbol{u}/\max\{1, \mathrm{RMS}(\boldsymbol{u})/d\}$;
12     $\Delta\boldsymbol{w} \leftarrow -\alpha\hat{\boldsymbol{u}}$;
13     $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w}$;
14     $k \leftarrow k + 1$;
15 end
```

Algorithm 10: Nadam

```
Input: Labelled data points $D = (x_i, y_i)$
Parameters: batch size $m$, step size $\eta$, parameters
            $\rho_1, \rho_2 \in [0, 1)$, parameter $\delta$
Result: Weight vector $\boldsymbol{w}$ which minimises the empirical
        risk
1  $\boldsymbol{w} \leftarrow$ initialise;
2  $\boldsymbol{s} \leftarrow 0$;
3  $\boldsymbol{r} \leftarrow 0$;
4  $k \leftarrow 1$;
5  while not converged do
6      $b \leftarrow$ sample $m$ items $\{(x'_1, y'_1), \dots, (x'_m, y'_m)\}$ from $D$;
7      $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{w}} \sum_{i=1}^{m} L(y'_i, f(x'_i; \boldsymbol{w}))$;
8      $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$;
9      $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$;
10     $\hat{\boldsymbol{s}} \leftarrow \frac{\rho_1 \boldsymbol{s}}{1-\rho_1^k} + \frac{(1-\rho_1)\boldsymbol{g}}{1-\rho_1^{k-1}}$;
11     $\hat{\boldsymbol{r}} \leftarrow \frac{\rho_2 \boldsymbol{r}}{1-\rho_2^k}$;
12     $\Delta\boldsymbol{w} \leftarrow -\eta \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}+\delta}}$;
13     $\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta\boldsymbol{w}$;
14     $k \leftarrow k + 1$;
15 end
```

is insufficient for this purpose, because it is already used for model selection; it may happen that we select a model that is too specifically tuned towards the validation set [7].

3.3.5 Evaluating Classifiers

In this section, we will describe a number of ways in which we can evaluate the effectiveness of classifiers. Of course, we can use the loss as such a measure. However, minimising the loss is a regression task, and it is difficult to relate the loss to the performance of the trained neural network in a classification task.

3.3.5.1 Confusion Matrices

The most basic way of evaluating a classifier, is a confusion matrix. In an n-class classification problem, the confusion matrix is an $n \times n$ matrix whose rows correspond to ground truths and whose columns correspond to predicted labels. The confusion matrix shows how the classifier labelled samples with different ground truths [7]. An example of a confusion matrix is given in Figure 9.

3.3.5.2 More Advanced Metrics

From the confusion matrix, we can derive various other metrics. First, we introduce the true positive count (TP), false positive count (FP), true negative count (TN), and false negative count (FN).

For a given class c, the true positive count is the amount of samples from c correctly predicted to be of class c. The false positive count is the amount of samples not from class c predicted to be from class c. The false negative count is the amount of samples from c not predicted to be from class c. The true negative count is the amount of samples not from class c not predicted as class c [7]. We can also express this mathematically. Let M be the confusion matrix and denote its entries by M_{ij}. We then have

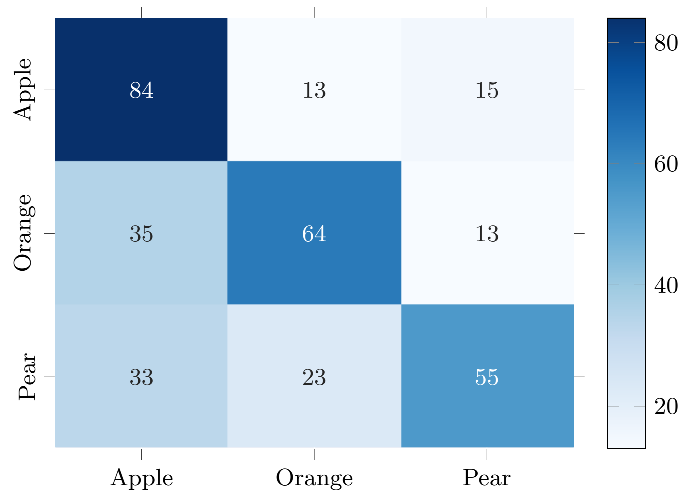


Fig. 9. Example of a 3x3 confusion matrix. Rows correspond to ground truths, while the columns correspond to predicted labels. For example: 13 samples originally labelled as "Apple" were predicted as "Orange" by the classifier.

$$
\begin{aligned}
\mathrm{TP}_c &= M_{cc} \\
\mathrm{FP}_c &= \sum_{i \neq c} M_{ic} \\
\mathrm{FN}_c &= \sum_{i \neq c} M_{ci} \\
\mathrm{TN}_c &= \sum_{i \neq c} \sum_{j \neq c} M_{ij}
\end{aligned}
$$

We can now define other metrics which are more commonly used to evaluate classifiers. Once such metric is precision. The precision for a class c is the fraction of samples predicted as class c, that also has class c as ground truth [7]. Hence, it is defined as

$$\mathrm{precision}_c = \frac{\mathrm{TP}_c}{\mathrm{TP}_c + \mathrm{FP}_c}$$

Another metric is recall. Recall is a measure of how many positive samples were "missed" by the classifier; it is the fraction of samples of class c in the dataset that were correctly predicted as class c [7]. It is defined as

$$\mathrm{recall}_c = \frac{\mathrm{TP}_c}{\mathrm{TP}_c + \mathrm{FN}_c}$$

Often, we want both high precision and high recall; this implies that our classifier does not miss many samples, while also not having many false positives. In order to capture this, we use the F_1 score, which is defined as the harmonic mean of precision and recall [7]:

$$F_1^c = \frac{2 * \mathrm{precision}_c * \mathrm{recall}_c}{\mathrm{precision}_c + \mathrm{recall}_c}$$

Up until this point, all metrics discussed are class-specific. We can also combine these metrics together into so-called macro averages. A macro average for some metric is the arithmetic mean of all class specific metrics. For example, the macro precision is the arithmetic mean of the precision for all the classes.

In literature, there are two macro F_1 scores. One is computed as the harmonic mean of the macro precision and macro recall, while the other is computed as the arithmetic mean of all class specific F_1 scores [7]. We use the latter, i.e.

$$F_1^{\mathrm{macro}} = \frac{1}{C} \sum_{i=1}^{C} F_1^i$$

3.3.6 Preventing Overfitting

The complexity of a model determines the complexity of the rules (functions) it is able to learn. Very simple models have high bias, and are unable to learn complex underlying rules from the data. This is what we call underfitting: the model is insufficiently complex to capture the patterns in the dataset. On the other hand, we have complex models, which have high variance: they are able to capture the details in the training data to such a great degree of accuracy, that their performance on novel test data is poor. This is what we call overfitting. Both under- and overfitting lead to poorly generalisable models. As such, it is important to find models with the right degree of complexity [7].

As already discussed in Section 3.3.4.5, the test set is used to evaluate the performance of a model; it is used to measure the generalisation error of the model. In the remainder of this section, we will discuss a number of techniques which can help in improving the generalisation error. Note that we already discussed weight decay and regularisation in Section 3.3.4.2, because those techniques are integral part of some optimisers.

3.3.6.1 Early Stopping

One possible technique to prevent overfitting, is early stopping. The idea is to monitor the loss on the validation set. Generally, we should expect both the training and validation loss to decrease in the beginning; at this point, the model is still learning a generalisable rule. However, as the models begins to over-fit, the loss on the validation set will no longer decrease, or may start to increase. At this point, training should be stopped [31].

3.3.6.2 Dropout

Dropout is a regularisation technique in which neurons in the network are randomly turned off during training. For each layer in the network, a probability of a neuron being turned off can be specified. For each update step, this probability is used to randomly turn off a number of neurons in the network [75].

The result of dropout is that the network effectively becomes smaller. This also limits the amount of information which can be stored in the network, reducing the "ability" of the network to overfit [75, 31].

At the same time, dropout also provides an approximation of ensemble functionality. The final trained network of n units can be seen as an ensemble of the 2^n subnetworks, which can be obtained by disabling one or more neurons. For each connection in the network, a weight w_{ij} is learned during training. By multiplying the learned weights by the dropout probability p in the working phase, we can approximate an ensemble which would average the predictions of all 2^n subnetworks [75].

Empirical observations have shown that dropout leads to better results (i.e. less overfitting and better overall performance) than weight decay and regularisation [31, 75]. Dropout has been shown to work effectively in fully connected neural networks and recurrent neural networks [31].

3.3.6.3 Batch Normalisation

Batch normalisation is another technique to avoid overfitting. The idea is that the activities of all layers are normalised to have 0 mean and unit variance (before applying the activation function). Hence, when a mini-batch $B = (b_1, \ldots, b_k)$ is fed as input into the network in the training phase, the outputs $W\boldsymbol{x}+b$ of each layer are normalised before being passed through the activation function of the layer [40]. Hence, the output of the layer will become $g(\mathrm{BN}(Wx + b))$ instead of $g(W\boldsymbol{x} + b)$.

The entries a_i of the vector $\boldsymbol{a} = W\boldsymbol{x} + \boldsymbol{b}$ are then transformed according to

$$\overline{a_i} = \gamma \frac{a_i - \text{mean}(\boldsymbol{a})}{\sqrt{\text{Var}(\boldsymbol{a}) + \epsilon}} + \beta \qquad ([40])$$

Here, ϵ is a small constant for numerical stability, γ is a learnable scaling factor, and β is a learnable shift [40].

At the same time, the batch normalisation layer also attempts to learn the true population mean $\overline{x}$ and variance $\overline{\sigma^2}$ for use in the inference phase, in which the mean and variance of the mini-batch are replaced by these estimates [8]. This is done by using a moving average with a configurable parameter α as follows:

$$\overline{x} \leftarrow \alpha\overline{x} + (1-\alpha)\,\text{mean}(\boldsymbol{a})$$
$$\overline{\sigma^2} \leftarrow \alpha\overline{\sigma^2} + (1-\alpha)\,\text{Var}(\boldsymbol{a})$$

Empirical observations have shown that batch normalisation has a regularisation effect, and improves generalisation performance. This has especially been observed in deep neural networks. However, the mechanism why this is the case has not been fully understood [52]. Additionally, batch normalisation allows the use of higher learning rates [40].

Batch normalisation is more effective than dropout for convolutional neural networks. For fully connected neural networks, there is little difference. However, dropout is significantly faster when training [30]. Batch normalisation as explained above is not possible for recurrent neural networks due to their time-dependent structure [47, 15].

3.3.7 Model Selection

Some parameters of a model, such as the weights, are learned during training. However, other parameters of the model, such as the architecture (e.g. number of layers, type of activation function, size of the layers), the optimiser, or loss function, need to be configured beforehand. We call such parameters *hyperparameters*. Hyperparameters of machine learning models have great influence on their performance. Identifying good hyperparameters is therefore an essential step in the machine learning process.

The most basic and straight forward method for selecting the best hyperparameters is by using a grid search. With this method, all possible combinations of hyperparameters are tested. Given the number of hyperparameters and values we want to optimise (see Section 4) and given the fact that our deep learning models take quite some time to train, this approach is infeasible.

In our previous work [18], we applied a variation of grid search, which we will call *iterative grid search*. With this approach, we optimise each hyperparameter sequentially. However, it fails to consider the potential performance of specific combinations of hyperparameters, and it requires substantial manual effort. This is because after tuning a single parameter, we must analyse the results and prepare for the next round of optimisation.

Given that we are using Keras, we will focus on hyperparameter optimisation algorithms supported by Keras Tuner [12]. The first algorithm is a random search, which selects random values for each hyperparameter in each trial. This algorithm stops once it has exceeded its allocated resource budget.

Random search is a rather naive approach and it may repeatedly try unpromising values. A more advanced method is Bayesian optimisation, which uses a surrogate function to sample values based on previous performance data. This improves the likelihood of selecting values that have shown promising results in previous runs [29].

Another algorithm is Hyperband. Hyperband is comparable to random search in the sense that it selects the values it wants to try randomly. However, Hyperband combines random search with successive halving and early stopping. This works as follows. Initially, it selects n combinations of hyperparameter values and trains each model for a minimal number of epochs, i.e. allocating a very low resource budget to each model. Then, after each model is trained, it selects the $n/2$ most promising models for the next round. In the next round it continues training the selected models for a couple more epochs than in the first round (i.e. their resource budget is increased a bit) and it again selects the most promising $n/4$ models. It continues this process of successive halving until one model is left. This model is then trained for any number of epochs that is required (i.e. unlimited budget). According to experiments, Hyperband can achieve speed-ups of up to 10 times compared to Bayesian optimisation [49].

All of these methods require a way to measure the performance of each set of hyperparameters. We use a validation set for this. A validation set allows us to monitor the performance of a model on unseen data, i.e. data not used for training the model. The performance on the validation set is more important than the performance on the training set, because it provides a more reliable estimation of the model's real-world performance.

While the validation set might be a good estimate of the real-world performance of a classifier, model selection itself is a potential source of overfitting. For that reason, we keep apart a test set, which is not used at all during this model selection phase. This test set is only used for evaluating the final optimised models.

3.4 Previous Work

We are building heavily on the work done by Dekker and Maarleveld in [18], Dekker et al. in [20], and Faroghi in [28]. To a somewhat lesser extent, we are also building on the work of Soliman, Galster, and Avgeriou in [70], Dekker in [17] and Druyts in [25]. As such, we will provide a thorough explanation of the work done there in order to explain the starting point and goals of this research. Related work by other researchers will be discussed later in Section 3.6.

In [28], [17], [70], and [18] a number of approaches for finding architectural design decisions in issue tracking systems were developed. The work done in [70] and [28] was evaluated in [28]; the work done in [17] was evaluated in [25], and the work done in [18] was evaluated in [18]. All this research is part of a combined effort to develop and evaluate different ways of finding architectural design decisions in issue tracking systems. All approaches were evaluated on the following Apache projects: Hadoop, Yarn, Tajo, HDFS, MapReduce, and Cassandra. In total, across all work, four different approaches were evaluated:

- *Keyword-based Search*

  One of the approaches is using a keyword search done using Apache Lucene, developed and evaluated in [28]. Four different sets of keywords were tested, falling into the following categories: *Components and Connectors*, *Decision Factors*, *Rationale*, and *Reusable Solutions*. For each group of keywords, a separate search was performed and evaluated. Every search yielded a ranked list of issues, resulting in a total of four ranked lists of issues potentially containing architectural design decisions.

- *Static Source Code Analysis*

  in [70], architectural design decisions were identified by 1) identifying architectural changes in commits, and 2) mapping these commits to issues in issue trackers. For every project, all commits were analysed in order to obtain a dependency graph. Graphs from successive commits were then analysed to compute the so-called a2a metric. The a2a metric was used as a measure of the amount of architectural changes in commits. Commits were then ranked based on

[12]`https://keras.io/keras_tuner/`

the a2a values, and the corresponding issue IDs were extracted from the commit messages. This way, a ranked list of issues potentially containing architectural design decisions was obtained. This ranking was subsequently labelled and evaluated in [28].

- *Maven Dependency Analysis*

  Another approach is the analysis of dependencies in Maven POM files in [17]. Successive commits in projects were once again analysed, but this time only the amount of dependency changes per commit was analysed. The issues corresponding to these commits were once again obtained through the commit messages. By ranking issues according to the number of Maven dependency changes in the corresponding commits, once again a ranked list of issues possibly containing architectural design decisions was obtained. In [25], this technique was used to exhaustively mine all ADDs which could be found using this method from the six projects of interest; the found issues were subsequently labelled to properly evaluate this method.

- *Neural Networks*

  The final approach we discuss is different types of neural networks for detection and classification of issues containing architectural design decisions. This approach was developed and evaluated in [18] (with errata in [19]). These classifiers were trained and evaluated using data obtained with the other three approaches.

Since deep learning has not been tested in the same setting as the other approaches (i.e. it was only evaluated on the dataset created using the other approaches), we will first discuss the effectiveness of keyword searches, static source code analysis, and Maven POM file analysis. Faroghi ([28]) found that keyword searches are the best way to find existence and property design decisions (precision@50 equal to 0.8 and 0.4, respectively). Maven dependency analysis was the most effective way of finding executive design decisions (precision@50 equal to 0.5). Static source code analysis performed poorly for both property and executive design decisions (precision@50 < 0.2), and performed mediocre for finding existence design decisions (around 0.4 precision@k up until $k = 400$). Maven dependency analysis scores poorly for finding property design decisions, and scores also worse for existence design decisions. One interesting thing to note was that the fraction of existence decisions found using Maven analysis actually increased further down the list. Keyword searches only performed poorly for finding executive design decisions, with the precision@k always lower than 0.2 [28, 25].

What remains to be discussed is the work done on neural networks in [18]. This is work done by the same authors as this research. In [18], we experimented with a number of different neural-network based classifiers for the detection and multi-class classification of architectural design decisions in issues. By detection, we mean that the classifier must discriminate between issues containing design decisions, and those that do not; it is thus a binary classification problem. The issues in the dataset we used had multiple labels, naturally lending themselves to a multi-label problem for classification. However, we opted to instead perform a simplified multi-class problem. Here, we used the "most important" label of an issue as the label to be predicted by the classifier. In order from most to least important, we had Executive > Property > Existence. This was because executive decisions tend to drive property and existence decisions, while property decisions also tend to drive existence decisions.

For feature generation, we experimented with Bag-of-words features (both word frequencies and normalised frequencies), TF-IDF, Doc2Vec (PV-DM), and Word2Vec (continuous bag of words). For Word2Vec, we experimented with a variant trained on all issues from Hadoop, Yarn, Tajo, HDFS, MapReduce, and Cassandra, and a pretrained variant trained on posts from StackOverflow (SO) [27]. We also experimented with a classifier which uses issue characteristics. Finally, we had a list of ontology classes (words which denote the same general concept; e.g. a list of components). We also experimented with a BOW variant which counted the amount of words from every ontology class. We combined all these features with fully connected neural networks, convolutional neural networks, and recurrent neural networks. Specifically, we experimented with the combinations listed in Table 1.

Feature Type	**Network Type(s)**
BOW (frequency)	Fully Connected
BOW (normalised)	Fully Connected
TF-IDF	Fully Connected
Doc2Vec	Fully Connected
Word2Vec	Convolutional, Recurrent
Word2Vec (SO)	Convolutional, Recurrent
Issue Characteristics	Fully Connected
Ontology BOW	Fully Connected

Table 1. Feature/Classifiers combinations experimented with in [18].

The models were all trained with the Adam optimiser (with default settings $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\delta = 10^{-7}$), a linearly decaying learning rate decaying from 0.005 to 0.0005 in 470 steps, and cross-entropy loss [19]. The detection models were trained with class weights, while the data for the (multi-class) classification task was balanced by limiting the amount of items from any given class (existence, executive, property, non-architectural) to 237 items [18].

Each type of model (fully connected, convolutional, recurrent) had a specific base architecture. The output neurons were determined by the task the models were trained for; a single unit with sigmoidal activation for detection, and a layer with four softmax units for multi-class classification. The fully connected models all consisted of an input layer, an output layer, and one or more hidden layers in between. This means that they follow the same architecture as depicted in Figure 3.

The convolutional neural networks were based on the work of Ren et al. in [65]. With Word2Vec, texts are represented as matrices $T \in \mathbb{R}^{m \times n}$. Here, m is the (maximum) number of words (where issues which are too short are further zero-padded). n is the size of the vectors used to represent the words. The idea in [65] is to have multiple convolutional layers in parallel, with kernels of size $k \times n$, where k is varied per convolution layer. Each convolutional layer can then be thought of as assigning a score to all k-grams in the text (and for convenience, we will refer to this as a convolution of size k). Then, a global max-pooling operation is applied to the output of every convolutional layer, meaning only the highest output per kernel from each convolutional layer is passed though. Next, the outputs of these max-pooling layers are flattened and combined into a single layer ("concatenated"). Finally, this layer also feeds into the output neuron. This network architecture is also depicted in Figure 10.

Finally, the RNN models consisted of a bidirectional LSTM layer, followed by zero or mode fully connected layers [18]. The exact architectures we used are given in Table 2.

We also experimented with ensembles of these models. The final results we obtained for detection are the ones listed in Table 3. The results for the (simplified) classification task are given in Table 4.

Our work in [18] also lead to the development of a deep learning utility tool[13]. This is a tool which allows users to train, test, and predict with deep learning models. The tool provides a command line interface for mixing and matching different types of feature generation and models, as well as specifying the exact (hyper)-parameters (e.g. the model architecture).

[13]`https://github.com/mining-design-decisions/mining-design-decisions/releases/tag/v1.0.0`

Task	**Model**	**Architecture**
Detection	BOW (Frequency)	One hidden layer of size 2
	BOW (Normalised)	Two hidden layers, both of size 32
	TF-IDF	Two hidden layers of size 64 and 2, respectively.
	CNN	Word2Vec vectors of size 25. One convolutional layer with 32 kernels of size 75.
	CNN (SO)	Word2Vec vectors of size 200. One convolutional layer with 32 kernels of size 75.
	RNN	Word2Vec vectors of size 25. One bidirectional layer of size 64, followed by a fully connected layer of size 4
	RNN (SO)	Word2Vec vectors of size 200. One bidirectional layer of size 64, followed by a fully connected layer of size 4
	Issue Characteristics	No hidden layers
	Ontology BOW	Two hidden layers of size 128 and 16, respectively.
Multi-class classification	BOW (Frequency)	Two hidden layers of size 64
	BOW (Normalised)	Two hidden layers of size 32 and 16, respectively
	TF-IDF	Two hidden layers of size 256 and 128, respectively
	CNN	Word2Vec vectors of size 10. One convolutional layer with 64 kernels of size 50.
	CNN (SO)	Word2Vec vectors of size 10. One convolutional layer with 64 kernels of size 50.
	RNN	Word2Vec vectors of size 300. One bidirectional layer of size 128.
	RNN (SO)	Word2Vec vectors of size 300. One bidirectional layer of size 128.
	Issue Characteristics	One hidden layer of size 8
	Ontology BOW	Two hidden layers of size 64 and 32, respectively.

Table 2. Model architectures used in [18].

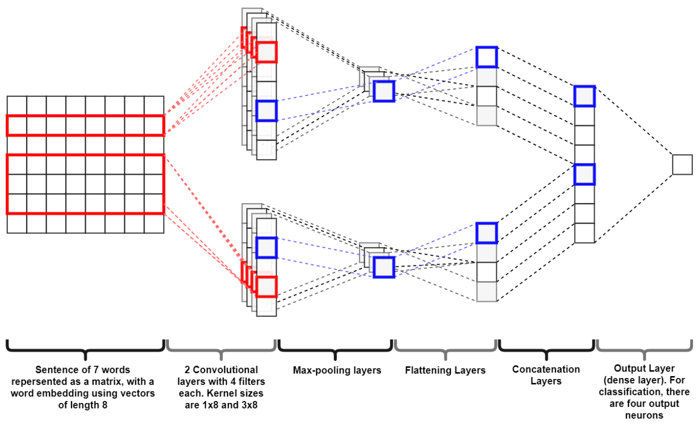

Sentence of 7 words repersented as a matrix, with a word embedding using vectors of length 8
2 Convolutional layers with 4 filters each. Kernel sizes are 1x8 and 3x8
Max-pooling layers
Flattening Layers
Concatenation Layers
Output Layer (dense layer). For classification, there are four output neurons

Fig. 10. Simplified depiction of the CNN architecture used in [18]. Image source: [18]

Model	F_1	**P**	**R**
BOW (Frequency)	0.827	0.770	0.897
BOW (Normalised)	0.816	0.764	0.876
TF-IDF	0.826	0.764	0.901
CNN	0.833	0.777	0.899
CNN (SO)	0.827	0.723	0.905
RNN	0.824	0.792	0.869
RNN (SO)	0.827	0.773	0.891
Doc2Vec	0.746	0.837	0.676
Issue Characteristics	0.808	0.750	0.877
Ontology BOW	0.823	0.757	0.906
BOW (Frequency) + CNN + RNN (stacking)	0.833	0.766	0.916

Table 3. Results for identifying architectural issues obtained in [18]. Columns: model, F_1 score, Precision, Recall. Results were obtained using 10-fold cross validation. We only listed the best ensemble model.

Model	F_1	**P**	**R**
BOW (Frequency)	0.529	0.539	0.530
BOW (Normalised)	0.514	0.534	0.423
TF-IDF	0.500	0.526	0.518
CNN	0.482	0.484	0.485
CNN (SO)	0.493	0.515	0.508
RNN	0.566	0.568	0.569
RNN (SO)	0.574	0.580	0.575
Doc2Vec	0.489	0.493	0.492
Issue Characteristics	0.343	0.353	0.352
Ontology BOW	0.413	0.430	0.426

Table 4. Results for classifying architectural issues obtained in [18]. Columns: model, macro F_1 score, macro Precision, macro Recall. Results were obtained using 10-fold cross validation. We omitted ensemble models because those all performed worse than RNN.

A small continuation of the work done in [18] was performed in [20] by Dekker et al. In this work, we performed the same classification tasks, but 1) used traditional machine learning classifiers (decision tree, random forest, and Naive Bayes), 2) only used issue characteristics, and 3) performed more elaborate preprocessing on the issue characteristics. Additionally, an assessment of feature importance was performing by manually inspecting the learned decision tree classifier, and by performing feature shuffling with the random forest classifier. The best performing model for detection was random forest with an F_1 score of 0.691, thus under-performing neural networks. For classification, random forest was once again the best classifier, with an F_1 score of 0.432. This is not better than text based features for neural networks, but outperforms the use of issue characteristics with neural networks. However, this would come at the cost of very elaborate preprocessing, which would require elaborate manual work and domain knowledge to generalise well to different projects or domains. Finally, it was found that the only really important feature was the type of an issue; Issues of type bug were considered considerably less likely to be architectural [20].

3.5 Cohen's Kappa

Part of this work will involve labelling issues into the categories defined by Kruchten in [45]. In order to evaluate the quality of the labelling, we will evaluate the agreement between reviewers. We will do this using a metric called Cohen's Kappa.

Suppose that we have two annotators A and B, who are annotating documents. We will assume that every document is either annotated with "Yes" or "No". We can then use documents annotated by both A and B in order to check their agreement. This is done by first computing the matrix in Table 5.

		A	
		Yes	**No**
B	**Yes**	a	b
	No	c	d

Table 5. Example of a table displaying number of annotated documents by two annotators.

A naive metric that we can obtain from Table 5, is the agreement. The agreement can be defined as

$$p_o = \frac{a+d}{a+b+c+d}$$

Hence, the agreement is the fraction of all documents on which both annotators agree. However, this does not take into account agreement by random chance. Annotator A assigns "Yes" with probability $(a+c)/(a+b+c+d)$, and "No" with probability $(b+d)/(a+b+c+d)$. Similarly, B assigned "Yes" with probability $(a+b)/(a+b+c+d)$ and "No" with probability $(c+d)/(a+b+c+d)$. The probability of random agreement is now given by

$$p_c = \frac{a+c}{a+b+c+d}\frac{a+b}{a+b+c+d} + \frac{b+d}{a+b+c+d}\frac{c+d}{a+b+c+d}$$

Cohen's Kappa is now given by

$$\kappa = \frac{p_o - p_c}{1 - p_c} \qquad [14]$$

The above illustrates κ for a 2-class classification task. However, κ can be defined for an n-class classification task with corresponding matrix

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ a_{n1} & \dots & \dots & a_{nn} \end{bmatrix}$$

First, define

$$X = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij}$$

Then, we have

$$p_o = \frac{1}{X}\sum_{i=1}^{n} a_{ii}$$

and

$$p_c = \sum_{k=1}^{n}\left[\left(\frac{1}{X}\sum_{i=}^{n} a_{kj}\right) \times \left(\frac{1}{X}\sum_{j=1}^{n} a_{jk}\right)\right]$$

And κ remains defined as $\kappa = \frac{p_o - p_c}{1 - p_c}$.

κ can be seen as the ratio of the observed non-chance agreement to the potential non-chance agreement. $\kappa = 1$ corresponds to perfect agreement, while $\kappa = 0$ means fully random agreement [14]. $0.6 \leq \kappa < 0.8$ would be considered moderate agreement, although many studies advise $\kappa \geq 0.8$ (strong agreement) [55].

3.6 Related Work

In this section, we will cover related work performed by other researchers. We will cover some other ways to automatically mine for design decisions, and we will also cover other sources of architectural knowledge.

3.6.1 Machine Learning Approaches in Issue Tracking Systems

Bhat et al. also tried classifying design decisions in issue trackers using machine learning in [4]. They randomly selected issues from the Apache projects Hadoop and Spark (excluding issues marked as bug, issues with a minor priority, or unresolved issues). Because Miesbauer and Weinreich found in [56] that 65% of the design decisions are existence decisions, they decided that, in order to save effort in finding property and executive decisions, to only consider existence decisions. In total, they obtained a dataset with 2139 issues. 781 issues were found to be design decisions and 1358 were found to be non-architectural. Furthermore, they considered the subtypes structural (226 issues), behavioural (389 issues) and ban decisions (166 issues).

Bhat et al. used this dataset to train machine learning models, specifically support vector machine (SVM), decision tree, logistic regression, one-vs-rest and Naive Bayes. They applied a two-step approach. In the first step, the model was trained to detect design decisions; i.e. the model had to mark issues as either "design" or "not design". To make sure that the models were not biased towards non-architectural issues, they applied a straight forward class limiting approach. They used all the 781 issues that contained design decisions and randomly selected 790 from the 1358 issues that did not contain design decisions. SVM was found to be the best performing classifier for this task, with a F_1 score of 91.29%. Their second step was to classify the design decisions into one of the subtypes. Since the idea was to use the detection classifier as a filter before this classifier, this classifier only needed to be trained on data containing design decisions. Bhat et al. randomly selected 160 issues from each of the three subclasses. After training the classifiers on this data, SVM was again found to be the best performing classifier with a F_1 score of 82.79%.

The classifiers of Bhat et al. were later incorporated in ADeX ([3]), a tool for exploring architectural design decisions. ADeX allows users to automatically find design decisions, find similar past decisions, get technology suggestions, and in general explore the design history of a project in terms of architectural elements and design decisions.

In [68], Shahbazian et al. identify design decisions in issue tracking systems, using static source code analysis. Their tool, RecovAr, compares different versions of software (in contrast to Soliman, Galster, and Avgeriou in [70], who directly compare successive commits). They use the ACDC (Algorithm for Comprehension Driven Clustering) and ARC (Architecture Recovery using Concerns) algorithms in order to recover the architecture of software. They then perform an analysis to detect *architectural changes*, which are sets of changed *architectural entities.* At the same time, they map commits to issues (restricting themselves to issues that were resolved with resolution "Fixed" or "Closed", or something similar), and map commits to software versions (i.e. what commit was merged in what version). They then perform source code analysis to determine what architectural entities were affected by what issues. This results in what they call the *architectural impact list.* At this point, they have 1) the architectural changes, each of which contains a set of affected architectural entities, and 2) the architectural impact list, which maps issues to architectural entities. They now combine these two and map issues to architectural changes by looking for non-empty intersections of architectural entities. They claim their approach is able to achieve a precision ranging from 76% to 78%. They also measured recall, and achieved an average recall of 73%. However, that did require the removal of architectural changes resulting from "external factors", such as changes in dependencies [68].

In [67], Shahbazian, Nam, and Medvidovic continued with this work. They developed a machine learning model in order to predict architecturally significant issues based on the issue text. They used their existing techniques in order to identify architecturally significant issues. They then trained a Naive Bayes classifier on the issues, and attempted to predict whether issues were architecturally significant or not. They obtained an average precision of 0.81, and an average recall of 0.583 [67].

3.6.2 Other Sources of Architectural Knowledge

In [22], Dieu et al. performed a systematic mapping study on different ways to mine architectural information from various sources. They extracted data from 87 different studies on the subject. They found that the extracted information can be categorised into the following categories: architectural descriptions (models, views, rationale, concerns), architectural design decisions, architectural solutions (patterns and tactics), system requirements, architectural changes, design relationships, architectural technical debt, and general architectural information.

From all 21 studies that focus on architectural design decisions, only two focus on a more fine-grained classification of decision than architectural vs. non-architectural. These two studies are the works of Bhat et al. (i.e. [4, 3]).

Dieu et al. also found that the most common sources mined for architectural knowledge are version control systems (including GitHub), online Q&A sites (e.g. StackOverflow), and Wikis. When looking at tools specifically designed for the extraction of design decisions, version control systems and issue trackers are most frequently used as a source of architectural knowledge [22].

In the remainder of this section, we will explore some sources of architectural knowledge in more detail. In particular, we will discuss sources of discussions between developers which can be mined for architectural knowledge. For sources on mining architectural knowledge from source code and documentation, we refer to the systematic mapping study of Dieu et al. ([22]).

3.6.2.1 Architectural Knowledge in StackOverflow

Several researchers have studied Q&A sites, and in particular StackOverflow, as sources of architectural knowledge. In [72], Soliman et al. investigated *architecturally relevant* posts on StackOverflow. Posts from the *middleware* topic were selected based on certain criteria (at least one answer, questions with a score > 7), and the posts were classified as either pure programming posts (PPP), architecturally relevant posts (ARP), or cross architecture/cross programming posts (CAPP). The first type only asks questions about programming and has no architectural relevance, while the second type is for questions related to performing some sort of architectural design activity. The third type is for questions that fall in between the two categories and may be relevant for both programmers and architects. Out of 2561 posts, 1659 (65.8%) were classified as PPP, 769 (30.6%) as ARP, and 89 (3.5%) as CAPP.

ARP posts were further classified along the *purpose dimension* and the *solution type dimension.* In the purpose dimension, posts could be classified as *solution synthesis* or *solution evaluation.* The former deals with searching for a suitable technology solutions based on certain characteristics (e.g. features or quality attributes). The latter type deals with assessing one or more proposed technology solutions. Some questions could be considered multipurpose: they fall into both categories. In the solution type dimension, ARPs could be classified as *technology feature*, *technology bundle*, *architecture configuration*, or *combined solution.* Technology feature refers to solutions focusing on specific technology features. Contrarily, technology bundle ARPs consider technologies as single architectural solutions without direct consideration for the individual features. Architectural configuration ARPs are concerned with the design configurations of components and connectors. Components and connectors could be related to technology features or technology bundles, but could also be purely conceptual (although this was found to be rare). Finally, posts could also contain a mixture of solution types. By combining the purpose and solution type dimensions, twelve different types of ARPs can be identified.

Building on this, in [74], Soliman et al. developed a tool to help architects to search for architectural knowledge in posts on StackOverflow. The tool uses a keyword search to find relevant posts, with a few augmentations. They used a voting ensemble consisting of a Bayesian network, a logistic model tree, and a Naive Bayes classifier to classify posts into the categories *programming post*, *technology identification*, *technology evaluation*, *features and configuration.* The classifiers were trained on the dataset created in [72]. The output of the ensemble was used to re-rank the posts retrieved from the keyword search; programming posts are ranked below the others; other posts are ranked based on the step of the design process for which information is being sought (entered by the user): identification of design concepts, selecting design concepts, or instantiating architecture elements.

In [6], Bi et al. experimented with machine learning to semi-automatically mine architectural tactics and quality attribute knowledge from StackOverflow posts. They use a two-step approach: first, a classifier is used to find potentially relevant posts. Next, these posts are manually examined to extract architectural knowledge regarding tactics and quality attributes.

In order to obtain posts for training, a keyword search was used. Specifically, the researchers searched for posts containing a number of common tactics *and* quality attributes. These posts were then manually annotated to create a set of training data (labelled as "true" when the post contains information on both tactics and quality attributes). By leveraging Word2Vec in combination with an SVM classifier, they were able to identify relevant posts with an F_1 score of 0.865. These classifiers were then used to mine additional posts about tactics and quality attributes. Through manual analysis, the researchers 1) identified relationships between quality attributes and tactics not commonly discussed in literature, and 2) analysed the design considerations discussed in posts discussing both tactics and quality attributes.

In [76], Tian et al. investigated the automatic extraction of discussions on architectural smells from StackOverflow. By using a keyword search, they obtained a ranked list of posts and labelled the top 400 according to whether they contained any sentence mentioning architectural smells or not [77]. In the end, they used a dataset containing 208 posts containing information on architectural smells, and 187 not containing such information. They found Word2Vec in combination with an SVM classifier to be most effective, yielding an F_1 score of 0.731 [76].

3.6.2.2 Architectural Knowledge in Mailing Lists

Bi et al. investigated architectural information communication in the mailing lists of two open source projects: ArgoUML and Hibernate. They annotated 26,647 posts (i.e. individual emails) from ArgoUML, and 22,888 from Hibernate. In the end, they found 316 architectural posts from ArgoUML, and 256 from Hibernate. They found that most emails were about discussing solutions, somewhat closely followed by questions about clarifications about the current architecture. The remaining emails could be classified as either suggestions or notifications on updates to the architecture. The most frequently discussed topics were architecture rationale and architecture models. Other topics were concerns, stakeholders, and systems of interest. It was found that different architecture topics (e.g. "new component", "adaptive design principle", "alternative") were discussed in both mailing lists. Additionally, in both mailing lists and in discussions on the different topics, different quality attributes were used. This was attributed to the different requirements of the two projects.

The reason for architectural changes can also be investigated from mailing lists. This was investigated by Ding et al. in [23]. Once again, emails from ArgoUML (26,439) and Hibernate (20,413) were studied. It was found that most architectural changes are either perfective (change system to better fit user needs) or preventitive (improve maintainability). In Hibernate, many changes were also adaptive (support new environments, platforms, or standards). A very small proportion was corrective (in response to defects).

Li, Liang, and Liu investigated decision-making (not restricted to architectural decisions) in emails from the Hibernate mailing list in [50]. They analysed 9006 emails and extracted 980 design decisions. 42.6% of decisions were found to be about design.

3.6.2.3 Architectural Knowledge in Pull Requests

Viviani et al. investigated the automatic extraction of design information from pull requests on GitHub in [81]. They did this on the paragraph level. They classified a total of 10,790 paragraphs from 34 commits from 3 different projects. Each paragraph was labelled according to whether it contained design information or not. In the end, 2475 paragraphs were annotated as containing design information.

They then trained a machine learning classifier on this dataset. As features, they used information regarding the process (information about the other), position (e.g. position of the comment in the thread), text (e.g. a Boolean indicating whether the paragraph contains the world "should"), and the content. The content features were generated by training different types of classifiers (naive Bayes, multinomial naive Bayes, complement naive Bayes, and random forest) in a manner which prevents bias, and use their outputs as features. In the end, random forest was the best classifier, with an ORC AUC score of 0.87. The classifier was also evaluated for generalisability on 250 paragraphs from five different projects. This results in an AUC score of 0.81.

3.6.2.4 Architectural Knowledge in the Web

In [73], Soliman et al. explored the effectiveness of Google search as a means for finding architectural knowledge. To do this, 53 software engineers were asked to complete three different search tasks (identify design concepts, select design concepts, instantiate architecture elements). Participants marked the degree of relevance of search results found using Google. A total of 2623 unique web pages were examined.

The most common sources of architectural knowledge were found to be blogs, tutorials, vendor documentation, and (to a somewhat lesser extent) scientific sources. Source code repositories and knowledge repositories were less common. Results from issue tracking systems were never encountered.

Solution descriptions were the most commonly encountered design concepts. Additionally, information on benefits, drawbacks, and alternatives was common, and usually some combinations of these seem to co-occur. The least common form of knowledge was actual design decisions.

The relevance of results was the best for the selection of design concepts. This showed that it is easier to select a solution from alternatives than it is to find the actual alternatives, or to instantiate a solution.

Blogs were found to be the most useful source of architectural knowledge. Vendor technology documentation was found to be useful mostly for identification of design concepts and instantiating of architecture elements. Knowledge repositories were only found to be useful for the selection of design concepts. Scientific sources are once again more useful for instantiating architecture elements. Forums and source code repositories were found to be less useful.

Architectural knowledge in blogs was studied in more detail in [71] by Soliman, Gericke, and Avgeriou. This study builds upon [73], and uses 718 web pages identified as blogs posts as a basis for the study. Using open coding, the blogs were categorised based on their type, and the topics inside the blogs were categorised. Additionally, latent Dirichlet allocation (LDA) was used to identify topics in blogs. It was found that most of the architectural knowledge comes in the form of lists, evaluation, or comparisons of architectural solutions. Many blogs also discuss architectural patterns and component design and principles. Finally, technologies are often discussed.

4 Study Design

4.1 Research Questions

Our main goals are to further improve upon the work we initially did in [18]. In particular, we are interested in improving classifier performance and generalisability. The following research questions build up to these goals, and motivate various steps we will need to achieve these goals.

RQ1 *What is the expected distribution of architectural design decision types found in issue trackers in open source systems when taking a random sample?*

As far as the authors know of, there is currently no research into what proportions of architectural design decisions in issues fall into the different categories defined by Kruchten. Knowledge about the amounts of existence, executive, property, and non-architectural issues one could expect to find within a system could be useful in evaluating search tools in a practical setting with some simple rules. For instance, if a machine learning classifier predicts an excessive amount of issues to be architectural compared to expected baselines, it is likely that the classifier is not suitable for real world applications. This can serve as an additional "sanity check", to verify whether models which perform well in the training/testing phase, also perform reasonably in the working phase, without requiring labour-intensive labelling of a random sample of issues labelled by the classifier.

This check is motivated by some observations we made while attempting to apply our classifiers designed in [18] in a more practical setting. We found that in general, the detection classifiers predicted too many issues as architectural. At one point, we trained a bag of words (frequency) classifier with particularly poor generalisability, which predicted over 50% of issues as being architectural. This did not seem plausible to us. As such, we decided to introduce a check for such behaviour in this research.

The work of Bhat et al. already provides a rough upper bound on the amount of architectural issues one would expect when taking a random sample of issues; in their work, they took an almost random sample, with some mild selections based on some issue characteristics. In their sample, 36.5% of issues contained architectural design decisions [4].

We are interested in a totally random sample, thus eliminating the potential bias from issue characteristics. Additionally, we are interested in knowing the proportions of existence, executive, and property issues. Bhat et al. only considered existence issues. Furthermore, in [18], we found that we do not fully agree with their labelling of issues.

RQ2 *How effective are deep learning models for finding architectural design decisions in issue tracking systems?*

Looking back at the work we did in [18], we determined that we were using very little data to train the models for the classification task (only 237 samples per class, because we limited the amount of data from certain classes so that we had an equal amount of samples from every class); we deemed it plausible that we were lacking sufficient data to train good performing classification models. As such, for this work, we intend to collect more data.

Additionally, collecting more data might help in alleviating the problem with classifiers predicting too many issues as architectural. Although there exists methods to deal with imbalanced classes (such as limiting the amount of data, or assigning weights to different classes in the loss function), the most optimal way to dealing with such biases is by training on a balanced dataset [42]. This means that by collecting more issues from the types we have few from, we hope to obtain classifiers which are more useful for practical purposes (i.e. prediction).

Besides considering some of the existing methods (i.e. static source code analysis, Maven POM file analysis, and keyword search) to find architectural issues, we will also be investigating the effectiveness of using deep learning classifiers to search for architectural issues in issue tracking systems. We will be investigating whether this method is a good addition to the other methods.

RQ3 *How well do classifiers perform for a multi-label classification of issues?*

In Section 3.4, we explained that for our previous research, we experimented with detection and a simplified multi-class classification scheme. One severe limitation of such a classification approach is that in the working phase, a model can only label an issue as either non-architectural, or existence, or executive, or property: it is not possible to assign multiple labels to a single issue. In practice, we want to find all types of decisions present in an issue.

In this research, the goal was to use a non-simplified, multi-label classification scheme in the hopes of obtaining a more useful classifier. In order to answer this research question, we want to evaluate the neural network architectures outlined in [18], and use these for the multi-label classification task.

RQ4 *How much can the performance of the classifiers be improved using additional data?*

In this research, we significantly expanded the existing dataset of architectural issues. We wanted to determine whether there would be any value in collecting even more issues in future research. Hence, we evaluated how the classifiers are performing with different amounts of training data.

RQ5 *How well do multi-label classification models generalise to different projects in the same software domain?*

In [18], we investigated the generalisability of the detection and simplified classification models to projects they were not trained on. We did the same in this research for multi-label models. This way, we can estimate the usefulness of the machine learning models when applied to projects not present in the dataset, but which belong to the same domain.

RQ6 *How well does multi-label classification generalise to different software domains?*

Deep learning models should ideally be as generalisable as possible. One basic type of generalisability is the one described in **RQ5**. There we look at generalisability to projects foreign to the dataset, but still belonging to the same domain. We can also investigate generalisability in a broader scope, where we investigate how well the trained models generalise to projects from different domains.

4.2 High-level Study Design

We first introduce our study design on a high level. A high-level depiction of our study design is given in Figure 11. On a high level, our research consisted of six main steps:

1. **Update & Improve Deep Learning Code**

   The deep learning code we initially developed while working on [18] was difficult to work with and lacked features which would allow the trained models to be used in practice. As such, a number of adjustment were made to make the code more usable. In the end, this lead to the development of a system consisting of a refactored version of the deep learning code, which makes use of a database to retrieve and store data. Additionally, a number of new features were added. A more complete motivation for and description of the changes will be given in Section 4.4.

2. **Collect More Data & Evaluate Deep Learning as a Search Tool**

   One of the things we suspected after our work on [18], was that there was too little data to train a sufficiently performant classifier for classifying architectural issues. As such, part of our research was spent finding more architectural issues to add to the existing dataset. Part of the issues were collected through random sampling to answer **RQ1**. We also tried to collect more issues using keyword searches, but most issues were collected by using deep learning classifiers. Hence, this step also serves to answer **RQ2**.

3. **Optimise and Evaluate Multi-label Deep Learning Models**

   With more data collected, we moved to the development of new deep learning models. We mainly experimented with the same type of models as our previous work in [18], but with a number of changes; the most noteworthy change being that we switched from detection and multi-class classification to multi-label classification. We also experimented with BERT because in [18], we did not experiment with state-of-the-art transformer based large language models. In this step, we performed hyperparameter optimisation to find the best performing multi-label models, and evaluated this model. The results of this evaluation were used to answer **RQ3**.

4. **Evaluate Need for Additional Data**

   Like mentioned before, we wanted to test whether there would be a need to collect more issues in future research. To test this, we trained the best performing classifiers from step 3 with different amounts of data to observe how performance improves. This allowed us to answer **RQ4**.

5. **Evaluate Generalisability of Multi-label Classifiers to Projects in the Same Domain**

   The next step in our research was to evaluate the generalisability to projects in the same domain. In our case, that is to projects in the data storage & processing domain. In this step, we trained classifiers on a number of projects, and evaluated the classifier performance on projects not present in the training and validation sets. The tests done in this step serve to answer **RQ5**.

6. **Evaluate Generalisability of Multi-label Classifiers to Projects in Different Domains**

   The final step in our research was to evaluate the performance of one of the classifiers to different domains. Specifically, we tested the classifier which showed the best generalisability to different project, which in our case was BERT. We evaluated both the capabilities of BERT as a classifier, and as a search tool. The work done in this step was meant to answer **RQ6**.

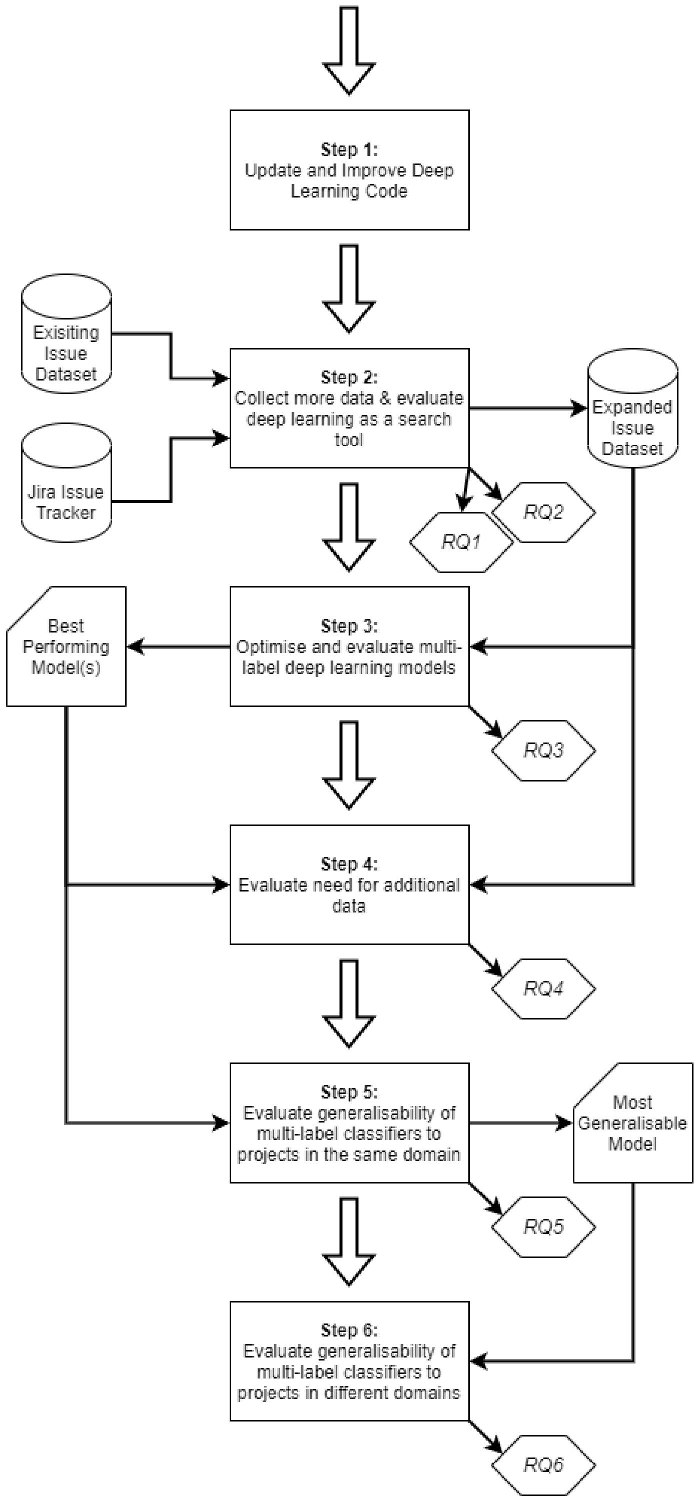

Step 1:
Update and Improve Deep Learning Code
Exisiting Issue Dataset
Jira Issue Tracker
Step 2:
Collect more data & evaluate deep learning as a search tool
Expanded Issue Dataset
RQ1
RQ2
Best Performing Model(s)
Step 3:
Optimise and evaluate multi-label deep learning models
RQ3
Step 4:
Evaluate need for additional data
RQ4
Step 5:
Evaluate generalisability of multi-label classifiers to projects in the same domain
Most Generalisable Model
RQ5
Step 6:
Evaluate generalisability of multi-label classifiers to projects in different domains
RQ6

Fig. 11. Graphical depiction of our study design.

4.3 Initial Dataset Description

Here, we describe the initial dataset we had, which is the dataset we used in [18], and whose acquisition is described in [18, 28, 70, 25].

The dataset consists of 2179 labelled issues, annotated based on the architectural decisions made in the summary and descriptions of those issues. 1431 of the issues are architectural, and 748 are non-architectural. The issues come from six different Apache projects: Hadoop, Yarn, Tajo, MapReduce, HDFS, and Cassandra. The exact breakdown of issue labels per project is given in Table 6.

4.4 Step 1: Changes to the Existing Deep Learning Code & Addition of a Database

In this section, we will describe some of the changes we made to our existing deep learning tool. We will focus mainly on the changes necessary to understand the remainder of the study design. We will have a separate chapter (Section 5) which goes into more detail in the changes not directly related to the design of our study.

4.4.1 Step 1.1: Augmenting The Deep Learning Tool With Multi-label Prediction

In order to obtain a more widely applicable classifier, we implemented a third classification task alongside the existing detection and multi-class classification tasks, that we will henceforth call *multi-label classification*. In multi-label classification, models have 3 sigmoidal outputs; one for every type of architectural design decisions. We apply a threshold of 0.5. Hence, if the neuron for the existence label outputs a value > 0.5, the issue is considered existence. Non-architectural issues should have all three neurons output a value ≤ 0.5.

4.4.2 Step 1.2: Adding BERT to the available models

Since BERT has shown state-of-the-art results for NLP tasks [21], we want to include BERT in our deep learning tool as well. Pre-trained versions of BERT are readily available online due to libraries such as Huggingface[14]. Huggingface in particular makes working with BERT easy for programmers. Therefore, we opted to use the "bert-base-uncased" model from their library. The BERT model can be used like any other model that our tool has to offer.

4.5 Step 2: Collect More Data & Evaluate Deep Learning as a Search Tool

In this section, we will be describing how we collected more issues for our dataset, and how we evaluated deep learning as a search tool, allowing us to answer **RQ1** and **RQ2**. The detailed design for this step is outlined in Figure 12. We collected issues through random sampling, keyword search, and by using classifiers as search tools. We then labelled these issues, and performed relabelling to correct systematic labelling errors.

Aside from a random sample of issues from the web development domain, we are always searching for issues from the six projects from which the issues of the original dataset came: Hadoop, Tajo, Yarn, HDFS, MapReduce, and Cassandra.

4.5.1 Step 2.1: Selecting Candidate Issues for Labelling

In this section, we will describe how we selected issues to be annotated.

Initially, our dataset contained 1112 existence, 265 property, 295 executive and 748 non-architectural issues (including issues with multiple labels; see Table 6 for more details). The number of property and executive issues were low compared to existence and non-architectural. In [70, 18, 28, 25], several ways of findings architectural issues were discussed. For property issues, the search engine was found to be the most effective approach. For executive issues, analysing Maven POM files was found to be the most effective approach. However, this type of analysis was mostly exhausted for our main target projects (Hadoop, Cassandra, HDFS, Tajo, Yarn, and MapReduce). In the end, to increase the number of samples in our dataset, we tried two approaches. The first is the keyword search. The second is an experimental approach. With this approach, we train a classifier on the issues that are currently in our dataset and select the issues for which the classifier predicted the issue to be of the desired class with a high confidence.

We also took random samples of issues and classified these in order to answer **RQ1**. This increased the size of our dataset, although not many architectural issues were found with this approach.

In the remainder of this section, we will describe each selection method in more detail.

4.5.1.1 Step 2.1.1: Establish Baseline for ADD Occurrences

One of the steps in our research was to establish a baseline estimate for the proportions of different ADD types in issues in issue tracking systems. In order to obtain this baseline, we took two random samples of 400 issues from different Apache Projects. This amount of 400 was chosen because a sample size of 385 is statistically significant (in the case of a 95% confidence interval) in case of an infinite population [80]. We rounded this number up to 400.

The first 400 issues were taken from the following Apache Projects: Solr, JSPWiki, CloudStack, Brooklyn, and TomEE. The second set of 400 issues was taken from the Apache projects Hadoop, Tajo, Yarn, HDFS, MapReduce, and Cassandra. Note that for the latter sample, we also included issues already contained in our dataset in order to avoid introducing any bias.

4.5.1.2 Step 2.1.2: Finding ADDs using a Keyword Search Engine

One way we attempted to find more issues for the dataset, was by using the search engine used by Faroghi[15] in [28]. At this point in our research, we mostly needed property issues, and the search engine approach had shown somewhat promising results – precision@k generally exceeded 0.35, while this could not be said for the source code analysis or Maven dependency analysis.

We re-used the keywords used by Faroghi in [28]. Because we were only interested in property issues, we only used the keywords for decision factors and reusable solutions – these gave the best precision for finding property issues. The exact keywords we used are shown in Table 7.

We searched for issues from the following Apache projects: Hadoop, Tajo, Yarn, HDFS, MapReduce, and Cassandra. We downloaded all the issues from these projects on February 22, 2023. We applied both keywords searches individually. For the results of both queries, we took all issues from the first 1050 search results that were not yet contained in our dataset. 241 issues were found using the decision factor keywords; 34 were found using the reusable solution keywords.

4.5.1.3 Step 2.1.3: Finding ADDs using a Multi-Label Classifier

4.5.1.3.1 Finding Property Issues

Since we were finding relatively few property issues using keyword search (precision@241 = 0.22), we decided to attempt to use deep learning techniques to find additional property issues.

[14] `https://huggingface.co/docs/transformers/model_doc/bert`

[15] Available from `https://github.com/Shadania/Jira_Arch`

Label	Cassandra	Hadoop	HDFS	MapReduce	Tajo	Yarn	Total
Existence	249	171	164	44	69	202	899
Executive	112	48	17	7	27	4	215
Property	31	19	14	6	5	8	83
Executive/Existence	22	10	6	2	9	3	52
Executive/Property	12	4	3	1	1	0	21
Existence/Property	54	32	32	11	8	17	154
Executive/Existence/Property	3	2	1	0	1	0	7
Architectural	483	286	237	71	120	234	1431
Non-Architectural	276	135	95	53	98	91	748
Total	759	421	332	124	218	325	2179

Table 6. Description of the dataset used in [18].

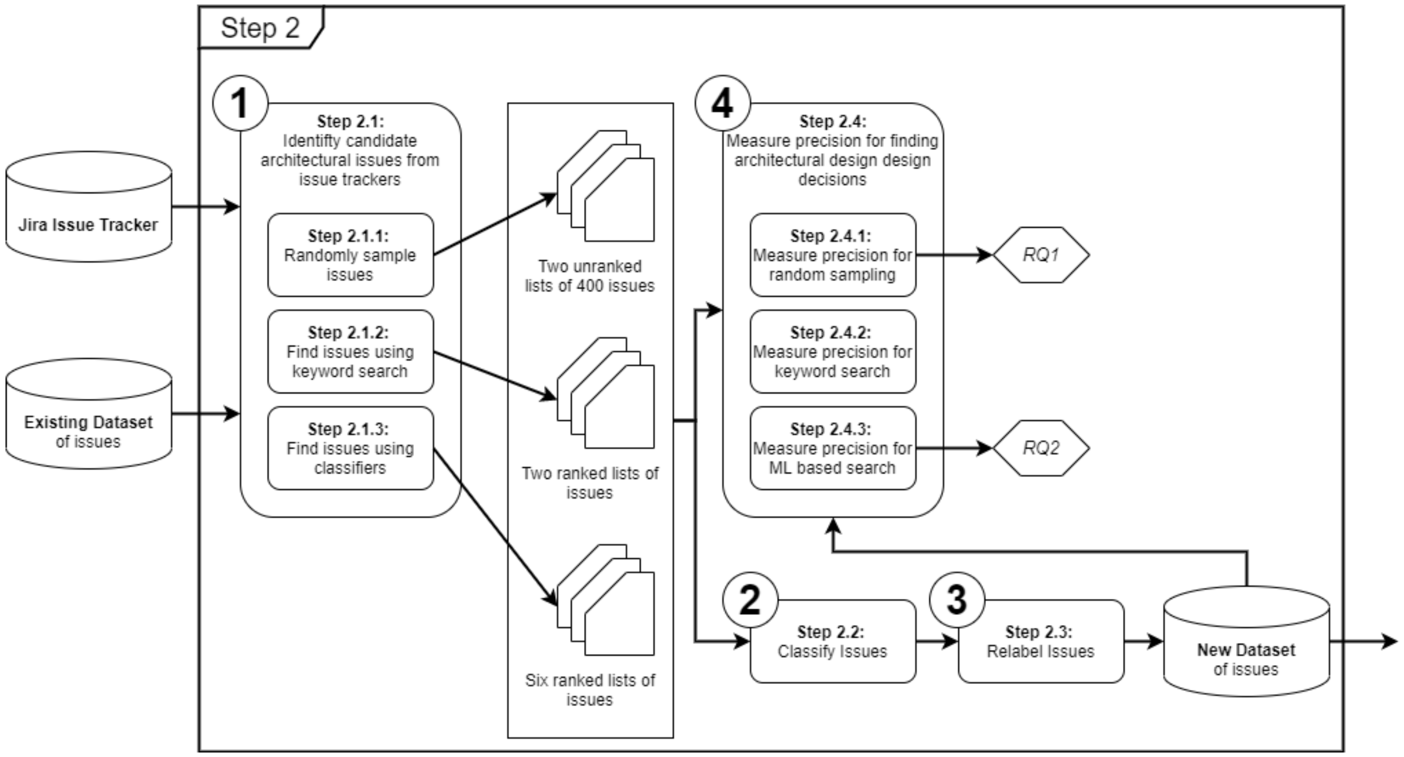

Step 2
Jira Issue Tracker
Existing Dataset of issues
1
Step 2.1: Identifty candidate architectural issues from issue trackers
Step 2.1.1: Randomly sample issues
Step 2.1.2: Find issues using keyword search
Step 2.1.3: Find issues using classifiers
Two unranked lists of 400 issues
Two ranked lists of issues
Six ranked lists of issues
4
Step 2.4: Measure precision for finding architectural design design decisions
Step 2.4.1: Measure precision for random sampling
Step 2.4.2: Measure precision for keyword search
Step 2.4.3: Measure precision for ML based search
RQ1
RQ2
2
Step 2.2: Classify Issues
3
Step 2.3: Relabel Issues
New Dataset of issues

Fig. 12. Detailed study design of step 2: collect more data & evaluate deep learning as a search tool. The new issue dataset is the main result of this step, and will be used in subsequent steps. Besides, this step helps us to answer our first two research questions.

Category	Keywords
Decision Factors	actor* availab* budget* business case* client* concern* conform* consisten* constraint* context* cost* coupl* customer* domain* driver* effort* enterprise* environment* experience* factor* force* function* goal* integrity interop* issue* latenc* maintain* manage* market* modifiab* objective* organization* performance* portab* problem* purpose* qualit* reliab* requirement* reus* safe* scal* scenario* secur* stakeholder* testab* throughput* usab* user* variability limit* time cohesion efficien* bandwidth speed* need* compat* complex* condition* criteria* resource* accura* complet* suitab* complianc* operabl* employabl* modular* analyz* readab* chang* encapsulat* transport* transfer* migrat* mova* replac* adapt* resilienc* irresponsib* stab* toleran* responsib* matur* accountab* vulnerab* trustworth* verif* protect* certificat* law* flexib* configur* convent* accessib* useful* learn understand*
Reusable Solutions	action* adapt* alloc* alternativ* approach* asynch* audit* authentic* authoriz* balanc* ballot* beat bridg* broker* cach* capabilit* certificat* chain* challeng* characteristic* checkpoint* choice* cloud composite concrete concurren* confident* connect* credential* decorat* deliver* detect* dual* echo encapsulat* encrypt* esb event* expos* facade factor* FIFO filter* flyweight* framework* function* handl* heartbeat* intermedia* layer* layoff* lazy load lock* mandator* measure* mechanism* memento middleware minut* monitor* mvc observ* offer* opinion* option* orchestrat* outbound* parallel passwords pattern* peer* period* piggybacking ping pipe* platform* point* pool principle* priorit* processor* profil* protect* protocol* prototyp* provid* proxy publish* recover* redundan* refactor* removal replicat* resist* restart restraint* revok* rollback* routine* runtime sanity* schedul* sensor* separat* session* shadow* singleton soa solution* spare* sparrow* specification* stamp* standard* state stor* strap strateg* subscrib* suppl* support* synch* tactic* task* technique* technolog* tier* timer* timestamp* tool* trail transaction* uml unoccupied* view* visit* vot* wizard* worker*

Table 7. Keywords used for the search engine

Hyperparameter	Value
model	bert-base-uncased
learning rate	5e-5
batch size	1
epochs	3
frozen layers	first 10 encoder layers
padding	true
max tokens	512
truncation	true

Table 8. Hyperparameters used for all search rounds with BERT. Note that we did not use any text preprocessing for these search rounds.

Instead of using one of the previous models we developed in [18], we experiment with BERT in combination with multi-label classification. The BERT model was chosen as the classifier due to its demonstrated effectiveness across various text classification tasks [21]. For all rounds of BERT, we used the hyperparameters as described in Table 8. Although these parameters were not as suggested by Devlin et al. ([21]), these allowed us to train the model on our own machines; using a larger batch sizes exceeded our VRAM. We froze the first 10 layers, because otherwise the performance was poor with such a small batch size. Normally, we would first strip all special formatted text (e.g. code blocks) from the text before feeding the text into the classifier (see section 4.6.1.1). However, due to a bug in our initial code, none of the search rounds with BERT are performed using text preprocessing. The results in Section 6 show that this should not have harmed the performance a lot, if at all.

We collected issues using an iterative approach. After training BERT on all the labelled issues in the initial dataset, BERT classified all issues from the projects Hadoop, Cassandra, Tajo, HDFS, MapReduce and Yarn that did not have a manual label yet. We then selected the issues that had the highest predicted confidence for the property class, and labelled these issues until we found that the precision had become too low. We then expanded the dataset using these new issues and began the process all over again. This process is also depicted graphically in Figure 13. This way, we can continuously improve the classifier with new feedback (in the form of new issues) in order to keep finding enough relevant issues to expand the dataset.

In the first round of finding issues with BERT, we selected the top 121 issues with the highest confidence for the property class for labelling. After labelling 121 issues, we decided that the precision of 0.41 for finding property issues was decent, but we preferred it to be higher (see Figure 28).

We also manually labelled the 50 issues that were predicted as property issues (property confidence score > 0.5), but with the lowest confidence scores for the property class. From these issues, we obtained 12 new property issues. In other words, this was a precision of 0.24. This small test confirmed the intuitive idea that the quality of the results indeed decreases as the confidence decreases; this confirmed that we should indeed focus on the issues with the highest confidence.

We then started the second round of finding issues with BERT. Similar to the first round, we used the second version of BERT to classify all issues from the 6 Apache projects (Hadoop, Cassandra, Tajo, HDFS, Mapreduce and Yarn) that did not have a manual label yet. Then we selected the 600 issues with the highest property confidence. Using BERT, we more than doubled the number of property issues in our dataset. With a precision of 0.67 (see Figure 29), we found that BERT was effective in finding property issues.

4.5.1.3.2 Finding Executive Issues

Due to the promising results with the property issues, we wanted to apply a similar approach for finding executive issues, since there were not many executive issues in the initial dataset either. For this first round of collecting executive issues, we used the same trained version of BERT as the one used to find the second round of property issues.

Many executive issues that were found with the Maven dependencies analysis are about version updates of technologies. The following are a few examples:

- CASSANDRA-12032[16]: Update Netty to 4.0.37 (no C* code changes in this ticket)
- CASSANDRA-7128[17]: Upgrade NBHM to use the Boundary maintained version

[16]https://issues.apache.org/jira/browse/CASSANDRA-12032
[17]https://issues.apache.org/jira/browse/CASSANDRA-7128

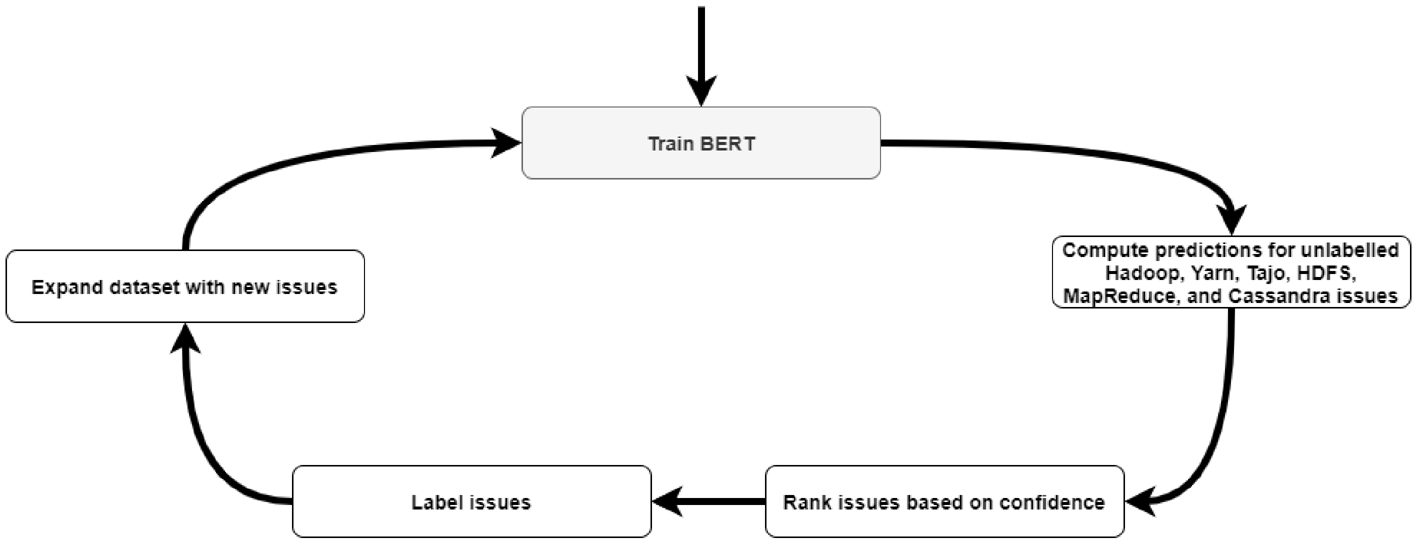

Train BERT
Compute predictions for unlabelled Hadoop, Yarn, Tajo, HDFS, MapReduce, and Cassandra issues
Rank issues based on confidence
Label issues
Expand dataset with new issues

Fig. 13. The iterative approach we used for finding more issues using BERT.

These issues are rather straight-forward to identify by the model and even a keyword search. To find more 'interesting' issues (i.e. issues with more useful architectural knowledge), we came up with the following approach. By selecting issues that the classifier predicts as both executive and existence, we expect to find issues more involved than a simple version bump. This is because these simple version bumps are often solely executive issues, not existence. In total, there were 202 issues with both the existence confidence and executive confidence above 0.5, and we selected all of these to be annotated.

The issues that were found with this approach seemed less trivial and richer in architectural knowledge than the simple version bumps, meaning that our approach had the desired effect. The precision after 202 issues was 0.43 (see Figure 31), which was similar to the Maven dependencies analysis.

Since we could not find any more issues that had an existence confidence and an executive confidence above 0.5, we retrained the BERT model using all the labelled issues we had up till then; i.e. we obtained the third version of the BERT model.

To obtain comparable results to the previous round, we wanted to label around 200 issues for this round as well. However, we had to lower the threshold of the existence confidence in order to find enough issues. With a confidence threshold of 0.5 for executive and 0.34 for existence, we were able to find exactly 200 issues. Although this different threshold did not make it entirely comparable, it did have the desired effect of finding more involved executive issues. In the end, the precision for finding executive issues was 0.53 with this third version of BERT.

Because we could not find much more issues that both had a high existence and executive confidence, we tried another approach. As we wanted to find issues that are not about dependency upgrades etc., we tried a basic keyword filtering to filter out such issues from the results found by the deep learning classifier. For a start, we filtered out issues containing the keywords "upgrade", "update", "bump", "dependency", or "library" in their summary. We also applied the keyword extraction described in [18] in order to find more possible keywords to include in our search. Specifically, we trained a CNN classifier (with the same hyperparameters as in [18]) for classification, and examined high-probability keywords for executive issues. We examined the keywords for issues found using Maven analysis especially closely, because this type of analysis is specifically tailored towards finding issues dealing with dependencies. Based on the results, we found that version number markers were also common keywords. Because of this, we also looked for version numbers of the form `x.y.z` and `x.y` in the issue summary.

To assess the quality of the issues obtained after filtering, we selected a batch of 100 issues with the highest executive confidence. We found out that the list of keywords did not seem to be complete, since we still found a lot of version bumps of technologies in these issues. We therefore decided that, due to the effort required in completing the keyword list, this was not a good approach to find interesting executive issues, and we decided to abandon this approach after these 100 issues were labelled.

4.5.2 Step 2.2: Labelling Issues

4.5.2.1 Basic Annotation Approach

Annotation of new issues was primarily performed by the two authors, assisted by the primary supervisor. The basic methodology we used, was that all issues to be annotated were split equally between the two authors. Issues with any degree of uncertainty were marked for review. This meant that the other author would also annotate these issues. If the two authors did not agree, and could not come to an agreement after discussing the issue, the issue was forwarded to the primary supervisor for annotating. The label provided by the supervisor was then used as the definitive label for the issue.

The issues were labelled based on the architectural design decisions made in their summaries and descriptions. Initially, we did this according to the definitions of existence, executive, and property decisions as defined by Kruchten in [45], as well as by considering labelled issues from the existing dataset as examples. This process was further refined by incrementally creating a coding book of rules to help annotate issues. The issues labelled by the primary supervisor were used to come up with the rules for the coding book. The full coding book can be found in Appendix A. The overall methodology for labelling is also depicted in Figure 14.

In order to improve labelling consistency in the beginning of the data collection process, the first and second author labelled a number of issues together and discussed these before continuing to annotate issues independently. Issues where no agreement could be reached were once again passed on to the primary supervisor. These issues served as a means to come up with the first entries of the coding book. Later, the coding book kept being incrementally expanded.

4.5.2.2 Division of Annotation Work

All work in this section was performed according to Section 4.5.2.1. When we write "the first author annotated these issues",

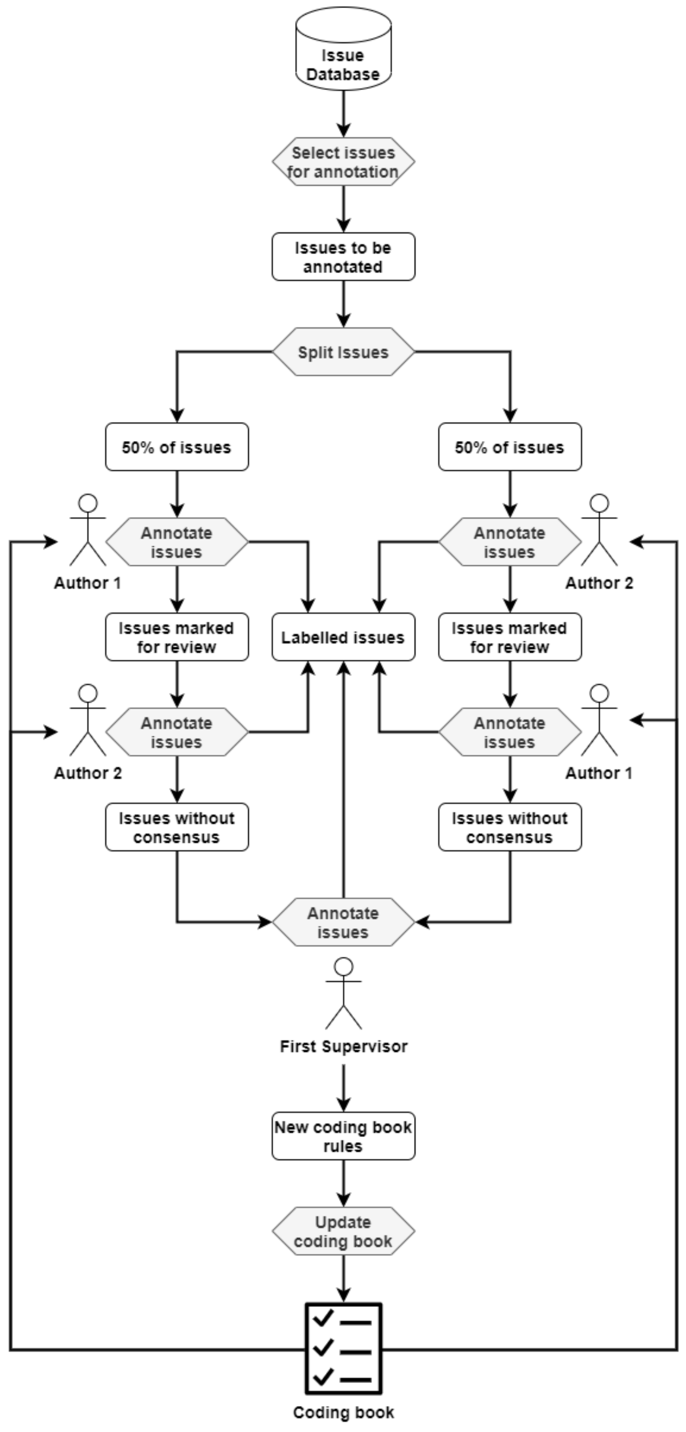


Fig. 14. Graphical depiction of the annotation process we followed.

we mean that the first author annotated all those issues, the second author annotated the issues that were marked for review by the first author, disagreements were discussed and in case of no agreement, the first supervisor would determine the final label. In general, we split all issues equally between the two authors. However, in the remainder of this section, we will discuss some exemptions to this process.

The two sets of 400 random issues (Section 4.5.1.1) were divided as follows. For the first set, the two authors classified the first 100 issues independently and then discussed the labels. After that, the remaining 300 issues were divided evenly between the two authors. For the second set of 400 issues, a similar approach was taken. However, only the first 50 issues were done by both authors in this case.

The issues found by the keyword search engine (Section 4.5.1.2), were primarily labelled by the first author.

For the issues found using the multi-label classifier (Section 4.5.1.3), all 171 issues from the first round of BERT were annotated by the second author.

For the selected property issues found with the second version of BERT, we had to label 600 issues. These issues were split up between the authors as follows. The first 50 issues were labelled by both authors, and the second batch of 50 issues were labelled by the first author. The last 500 issues were split up in batches of 100, where each author labelled 50 issues from each batch.

For all other rounds, we evenly split the issues of each round between the two authors.

4.5.2.3 Labelling Quality Assessment

While labelling issues, we tracked the labels assigned by all individual authors. This allowed us to compare assigned labels and compute the agreement on the labelled issues.

A full description of the issues we collected with this first round of labelling is given in Figure 41 in Appendix B. In this figure, we provide a full breakdown of all issues we found per class, per collection method (and collection iteration).

Additionally, Figure 43 in Appendix B provides a detailed overview of the agreement between annotators for this round of labelling; We computed the agreement between the three annotators (the two authors Jesse and Arjan, and the first supervisor Mohamed). We did this to obtain insights in the quality of the labelling. We considered the agreement between all three possible pairings of annotators, and considered the agreement per class (existence, executive, property, and non-architectural). Hence, for every round of labelling, we computed twelve (confusion matrix, agreement, kappa) triples. Each time, we computed 1) a confusion matrix displaying how the labels between two annotators compare, 2) the agreement, and 3) Cohen's kappa. However, In this section, we will only discuss the average agreement and Kappa score per class, as given in Table 9.

Class	Agreement	Kappa Score
Existence	0.7295	0.3783
Executive	0.8439	0.5553
Property	0.7418	0.4691
Non-Architectural	0.7167	0.2978

Table 9. Average (over all pairings of annotators) agreement and Kappa score per class for the first round of labelling issues.

The agreement scores were computed using the issues which were labelled by multiple people. In particular, this means that the following issues were used to compute the agreement scores presented in this section:

- The first 100 issues from the random sample from the web development projects, which were labelled together by the two authors.

- The first 50 issues from the random sample from the data processing & storage projects, which were labelled together by the two authors.

- All issues which have been marked for review during labelling, and which were thus checked by both authors.

Of all these issues, only the issues without consensus between the two authors were sent to the first supervisor for annotation. On average, this was about 10% of all issues labelled, while in general 30% to 50% of issues were marked for review.

A result of this is that the agreement is calculated using the issues which were the most difficult to annotate. In particular, the issues also labelled by the first supervisor were deemed very difficult to annotate by the two authors. As a result of this, we expect the agreement results in this section to represent the worst case lower bounds for agreement.

In particular, many issues which were sent to the first supervisor were sent because they contained types of issues not present in the coding book. This also means that in many of this difficult cases, a new entry was added to the coding book. This, combined with the fact that we expect the results here to represent lower bounds, should hopefully alleviate some of the quality concerns regarding the dataset.

It is important to emphasise that these difficult issues were examined by at least two annotators. Therefore, even if the agreement might be relatively low, the quality of these labels is not necessarily a cause for concern, as the annotators engaged in discussions regarding these issues; even if the initial labels of the two authors (on which the agreement calculation was based) differs, it is possible the two authors still came to a consensus.

In Table 9, we can see the agreement for the initial expansion of the dataset. First, we note that the raw agreement overall is pretty good, with the lowest agreement being 0.72 for non-architectural issues.

We can see that agreement on executive issues was overall pretty good; the executive class had the highest average agreement (0.84) and kappa score (0.56). For the other classes, there is more disagreement, certainly in terms of kappa score. In particular, according to Figure 43, the two authors assigned the label "existence" too often compared to the first supervisor, and the label "property" not often enough. Based on the poor agreement for the non-architectural class, we also make the observation that it was often difficult to judge whether an issue should be considered architectural.

However, we also note that in general, all Kappa scores are pretty poor. They certainly indicate a better than chance agreement, but for certain classes (especially non-architectural and existence) the kappa score is really poor.

4.5.3 Step 2.3: Correcting Mislabelled Issues in the Existing Dataset

When labelling issues, it is important to reduce the effect of human error. That is why during labelling, all difficult cases were discussed by at least two people. However, it is still possible for systematic errors to occur if both the first and second author had matching opinions differing from the first supervisor. Additional systematic errors may be introduced due to shifts in opinions about how issues should be annotated. There is a large gap in time between the collection of the initial dataset and the work done in this research, meaning that some ideas could have changed. In this section, we will describe how we attempted to mitigate quality concerns about the dataset due to systematic labelling inconsistencies.

4.5.3.1 Systematic Errors in the Original Dataset

4.5.3.1.1 Incorrectly Labelled Technology Version Upgrades

During discussions with the primary supervisor, it became apparent that there had been a shift in the opinion of what a correct classification should be for issues discussing dependency upgrades. In particular, it became apparent that such issues should be considered executive, because they deal with external technologies. In the past, such issues were not always classified as such; they were classified as non-architectural instead. Because of this, we attempted to find issues discussing such changes in the dataset, and relabel them.

We attempted to identify such issues through keyword search. Specifically, we searched for issues which in their summary contained one of the same keywords which we used to filter the result from the BERT search earlier (i.e. we selected all issues with at least one occurrence of "upgrade", "update", "bump", "dependency", "library", or version number of the form `x.y.z` or `x.y` in their summary).

4.5.3.1.2 Incorrectly Labelled Existence Decisions

A large portion of the original dataset originates from the keyword search (top-down) and the source code analysis (bottom-up). As these issues were labelled by a Bachelor student quite a while ago, we were interested in the quality of the labels.

We selected a random sample of 100 issues from the top-down and bottom-up approaches. This sample was labelled by our first supervisor. The full results of the relabelling done by the first supervisor can be found in Figure 15. In particular, we found that issues labelled as solely existence, were in fact often either property or existence and property issues.

This made us decide to relabel all of these issues (i.e. all issues labelled *only* as existence by this student), as to drastically increase the quality of our dataset. Each of the authors and our first supervisor were then assigned 200 issues originating from the top-down and bottom-up approaches that were labelled as solely existence.

4.5.3.2 Systematic Errors in the New Dataset

In order to check the quality of the issues annotated during this research, we selected a batch of 50 issues that were randomly selected from all the issues the authors labelled during their master thesis. These issues were then also labelled by the first supervisor. The results of this analysis can be found in Figure 16.

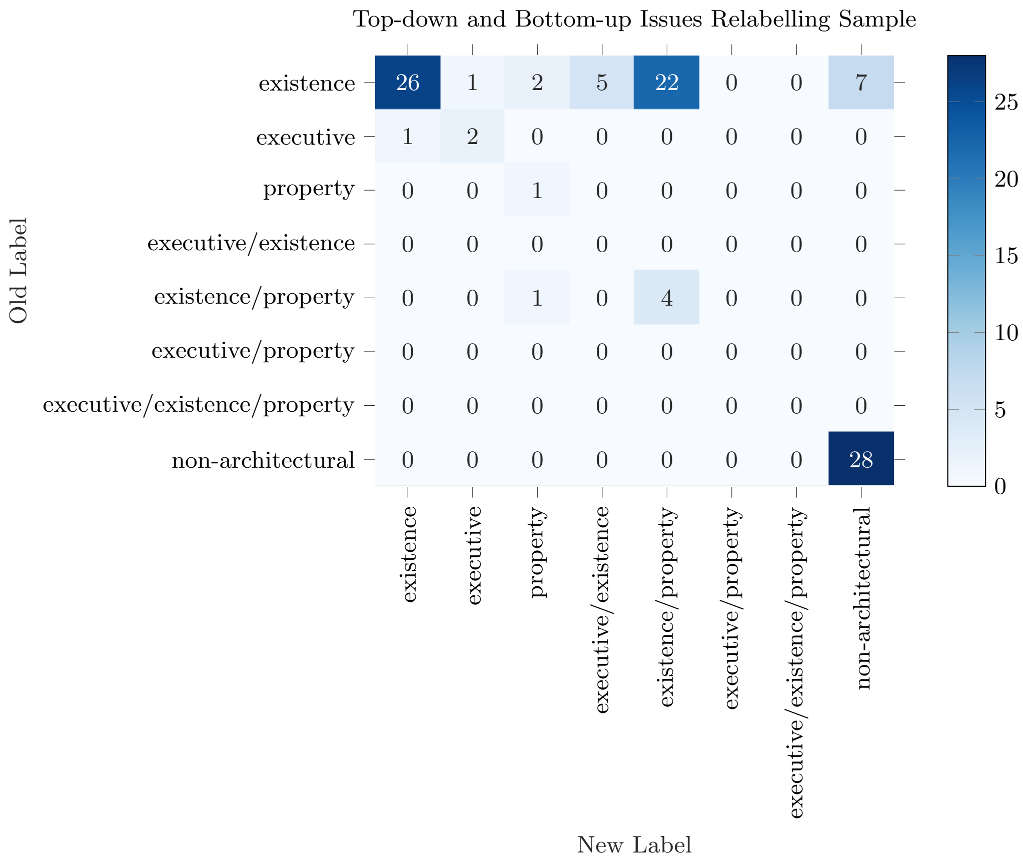
We found that the agreement was better compared to the top-down and bottom-up issues. However, it was clear that we often labelled issues that were solely executive as both executive and existence. As the number of issues that were labelled as both existence and executive during our master thesis was rather small, we decided to relabel these issues as well. Figure 16 also shows that 7 out of the 16 issues that were labelled as existence only were misclassified. Hence, we also relabelled all issues labelled only as existence.

4.5.3.3 Labelling Quality Assessment

Similar to how we did it for the initial set of labels in Section 4.5.2.3, in this section, we will consider the agreement of the annotators on the issues we relabelled.

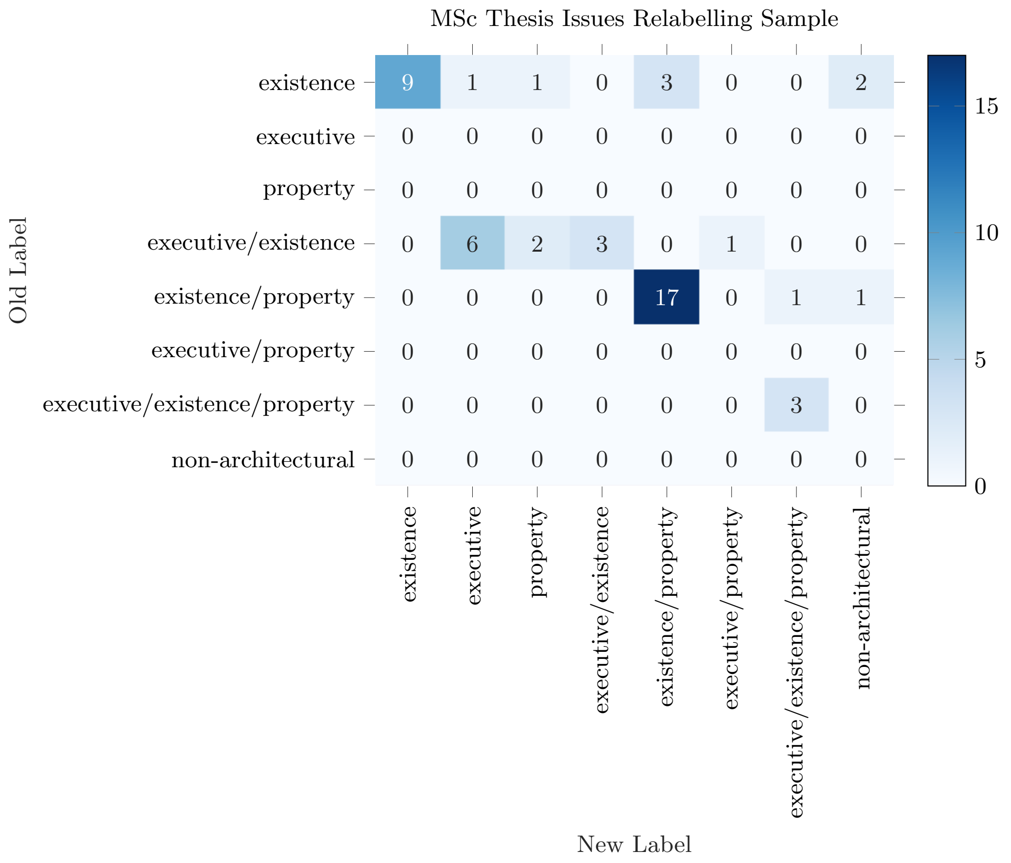
A detailed overview of how relabelling issues changed the amounts of issues per class and method we found, is presented in Figure 42 in Appendix B. Additionally, Figure 44 in Appendix B provides a detailed overview of the agreement for the relabelled issues. The average agreement and Kappa scores are presented in Table 10.

In this case, the issues used to compute the agreement were all issues marked for review; there were no issues that were specifically selected to be labelled jointly by both authors. The agree-



Old Label \ New Label	existence	executive	property	executive/existence	existence/property	executive/property	executive/existence/property	non-architectural
existence	26	1	2	5	22	0	0	7
executive	1	2	0	0	0	0	0	0
property	0	0	1	0	0	0	0	0
executive/existence	0	0	0	0	0	0	0	0
existence/property	0	0	1	0	4	0	0	0
executive/property	0	0	0	0	0	0	0	0
executive/existence/property	0	0	0	0	0	0	0	0
non-architectural	0	0	0	0	0	0	0	28

Fig. 15. Result of the initial pilot relabelling performed by the primary supervisor for the top-down and bottom-up issues.



Old Label \ New Label	existence	executive	property	executive/existence	existence/property	executive/property	executive/existence/property	non-architectural
existence	9	1	1	0	3	0	0	2
executive	0	0	0	0	0	0	0	0
property	0	0	0	0	0	0	0	0
executive/existence	0	6	2	3	0	1	0	0
existence/property	0	0	0	0	17	0	1	1
executive/property	0	0	0	0	0	0	0	0
executive/existence/property	0	0	0	0	0	0	3	0
non-architectural	0	0	0	0	0	0	0	0

Fig. 16. Result of the initial pilot relabelling performed by the primary supervisor for the MSc thesis issues.

Class	Agreement	Kappa Score
Existence	0.6802	0.2771
Executive	0.9116	0.6586
Property	0.7595	0.5239
Non-Architectural	0.8164	0.3279

Table 10. Average (over all pairings of annotators) agreement and Kappa score per class for the second round of labelling issues (i.e. the relabelling of issues from the first round and the initial dataset).

ment with the first supervisor was computed using 1) the issues in the initial set of 50 issues from this research labelled by the first supervisor, and 2) the issues without consensus which were passed along to the first supervisor for classification because of a lack of consensus.

We can immediately observe that the agreement for Executive issues is much better, with an agreement of 0.91 and Kappa score of 0.66. For property decisions, we actually had an overcorrection (see Figure 44); the "property" label is now assigned too often by the two authors. However, in terms of agreement and kappa score, the quality of the labelling has somewhat improved. For non-architectural issues, we can say the same; it was still difficult to correctly judge whether an issue is architectural or not, but in terms of agreement and kappa, quality has slightly gone up.

The most problematic class during the relabelling was the "existence" class. Agreement and Kappa for this class has gone down significantly. We attribute this to the fact that while labelling, the consensus of what makes an issue existence, actually became less clear. We can attribute this to multiple reasons, but the two most important ones were:

1. It became unclear when a change is sufficiently large to affect a component. Based on the description, it can be unclear how big a change is. To make more educated estimates, we started looking at patches and pull requests to estimate the scale of changes. However, this results in multiple problems; 1) this was not done with earlier labelled issues, 2) this contradicts the fact that we should only look at the summary and description, and 3) patch size may not be a good measure of change impact size. For instance, the patch size can be inflated due to many tests being included. Patch size can also make labelling more ambiguous because large patches may be present in issues which do not discuss changes to components, and would thus normally not be labelled as "existence".

2. It became unclear when exactly a discussion is providing sufficient information on changes or additions to warrant assigning the label "existence". For instance, is a component being mentioned enough to label something existence? This had always been a problem, but it became worse due to the fact that estimations of the amount of effort involved in a change became more prevalent during the relabelling process.

4.5.4 Final Dataset

In Table 11, we present the final dataset we obtained after labelling (and relabelling) all the issues. We present a breakdown both by class and by project. Note that there are two new projects among the data storage and processing domain projects (HBase and Submarine): these are the results from issues being moved to different projects. Originally, these were Hadoop and Yarn issues.

Additionally, we also provide Figure 17. This figure represents how many issues were collected during different issue collection rounds, and how these were combined to arrive at the current dataset. Note that the last two phases in the figure (depicted in orange) had not been executed yet at this point; these will be described in Section 4.9.2.

4.5.5 Step 2.4: Measure Precision for Finding ADDs

We want to measure the precision for finding ADDs for different methods, in order to determine which method is the most effective.

4.5.5.1 Step 2.4.1: Measure Precision of Random Sampling

Although random sampling cannot really be considered a search approach, it can give us an estimate of how many ADDs we can expect to find in case we would sample at random. We will simply compute the fraction of decisions of each type, which means we will compute the precision for all the different types.

4.5.5.2 Step 2.4.2: Measure Precision of the Keyword Search

The keyword search approach from [28] has shown promising results in the past. However, almost all issues with the highest search scores have been annotated already. Therefore, we could only select issues with lower scores, as described in Section 4.5.1.2. To determine whether this approach is still efficient for finding ADDs, we calculate the precision@k for finding architectural issues using this approach. Here, precision@k means that we compute the precision of the first k issues for every k, and plot the result as a function of k.

4.5.5.3 Step 2.4.3: Measure Precision of the ML-based Search

Finding ADDs using a deep learning classifier is a new approach that we introduced in this work, meaning that there is no data available regarding the performance of this approach. Due to the nature of deep learning classifiers, where the performance tends to improve upon training with more data, we perform this search approach in multiple iterations, as described in Section 4.5.1.3. As a result, we will be monitoring the precision@k of each iteration to determine whether this approach is efficient for finding architectural issues, as well as to determine whether the precision increases after training with more data.

4.6 Step 3: Optimise and Evaluate Multi-label Deep Learning Models

In this section, we will be describing how we optimised and evaluated our new multi-label classification models, in order to answer **RQ3**. The detailed design for this step is given in Figure 18. The main step is 3.1, in which we design and optimise the exact models we will be using in the remainder of this research. After this follows the evaluation steps 3.2, 3.3, and 3.4.

4.6.1 Step 3.1: Determine Best Multi-label Model Architectures

We experimented with the same features and models as we previously did in [18], except the issue characteristics and ontology bag of words models. This is because these models performed significantly worse than the others. However, we did experiment with BERT. This is because previously, we did not experiment with large language models, while these models have been shown to achieve state-of-the-art performance for various tasks. Hence, in the end we experimented with all the models given in Table 12

In this section, we will explain the full deep learning methodology we used. We will explain the text preprocessing, the feature generation, the hyperparameters we optimised, and the process by which we optimised the hyperparameters.

Domain	**Project**	***Existence***	***Executive***	***Property***	***Existence/Executive***	***Existence/Property***	***Executive/Property***	***Existence/Executive/Property***	***Architectural***	***Non-Architectural***	***Total***
Data Storage & Processing	Cassandra	82	138	115	26	234	44	12	651	850	1501
	Hadoop	58	136	88	36	133	43	19	513	473	986
	HBase	1	0	0	0	0	0	0	1	1	2
	HDFS	74	43	83	15	162	15	11	403	450	853
	MapReduce	18	19	24	1	51	6	2	121	191	312
	Submarine	0	0	0	1	0	0	0	1	0	1
	Tajo	33	40	14	18	41	7	8	161	219	380
	Yarn	127	18	43	10	102	8	7	315	364	679
	Total	393	394	367	107	723	123	59	2166	2548	4714
Web Development	Brooklyn	0	0	0	0	0	0	0	0	10	10
	CloudStack	1	1	0	0	3	0	0	5	120	125
	JSPWiki	0	1	1	0	0	0	0	2	8	10
	Solr	10	8	2	1	6	0	1	28	181	209
	TomEE	0	7	2	0	0	0	0	9	37	46
	Total	11	17	5	1	9	0	1	44	356	400
All	Total	404	411	372	108	732	123	60	2210	2904	5114

Table 11. Dataset composition after collecting more data.

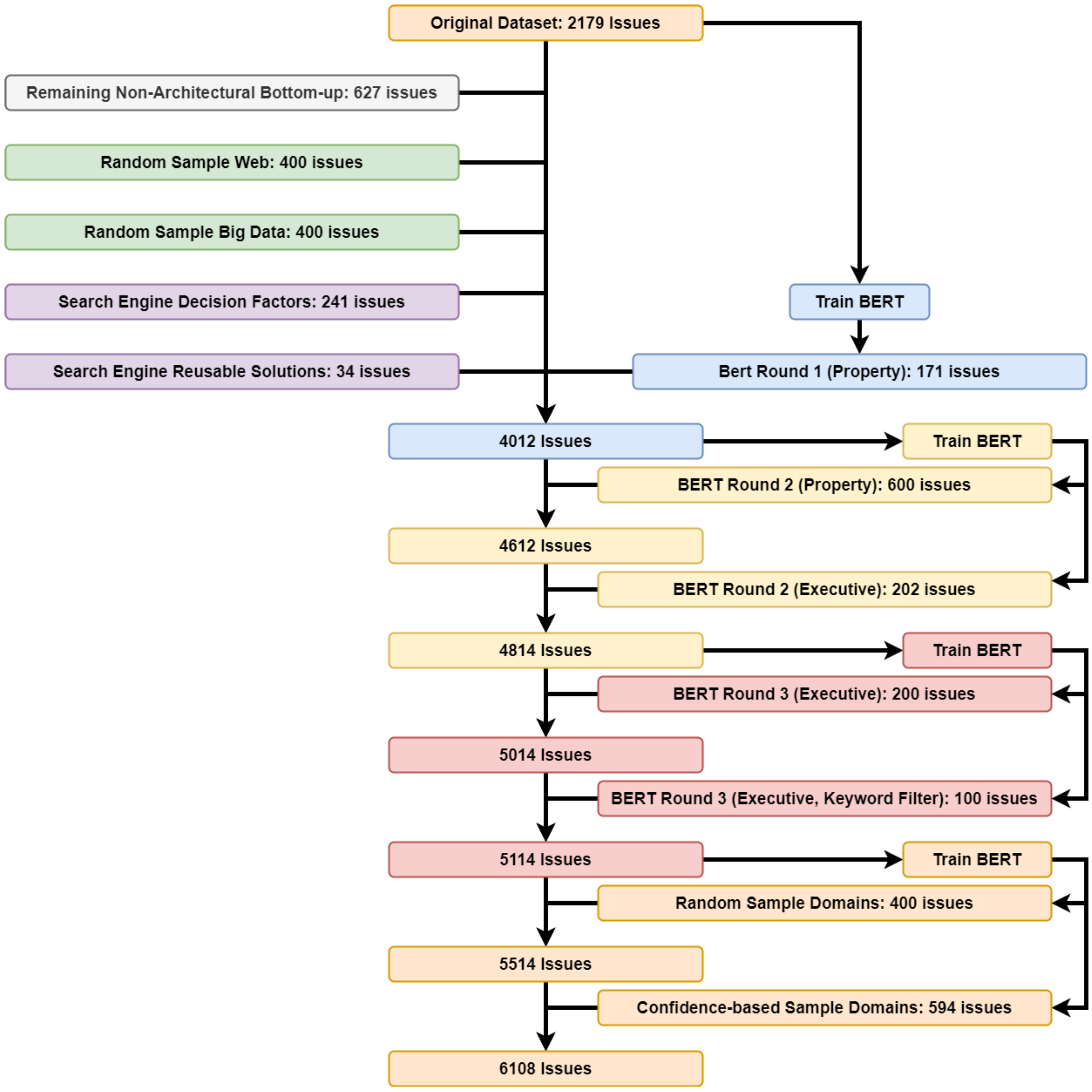

Original Dataset: 2179 Issues
Remaining Non-Architectural Bottom-up: 627 issues
Random Sample Web: 400 issues
Random Sample Big Data: 400 issues
Search Engine Decision Factors: 241 issues
Search Engine Reusable Solutions: 34 issues
Train BERT
Bert Round 1 (Property): 171 issues
4012 Issues
Train BERT
BERT Round 2 (Property): 600 issues
4612 Issues
BERT Round 2 (Executive): 202 issues
4814 Issues
Train BERT
BERT Round 3 (Executive): 200 issues
5014 Issues
BERT Round 3 (Executive, Keyword Filter): 100 issues
5114 Issues
Train BERT
Random Sample Domains: 400 issues
5514 Issues
Confidence-based Sample Domains: 594 issues
6108 Issues

Fig. 17. Overview of how issues were collected in order to arrive at the current dataset.

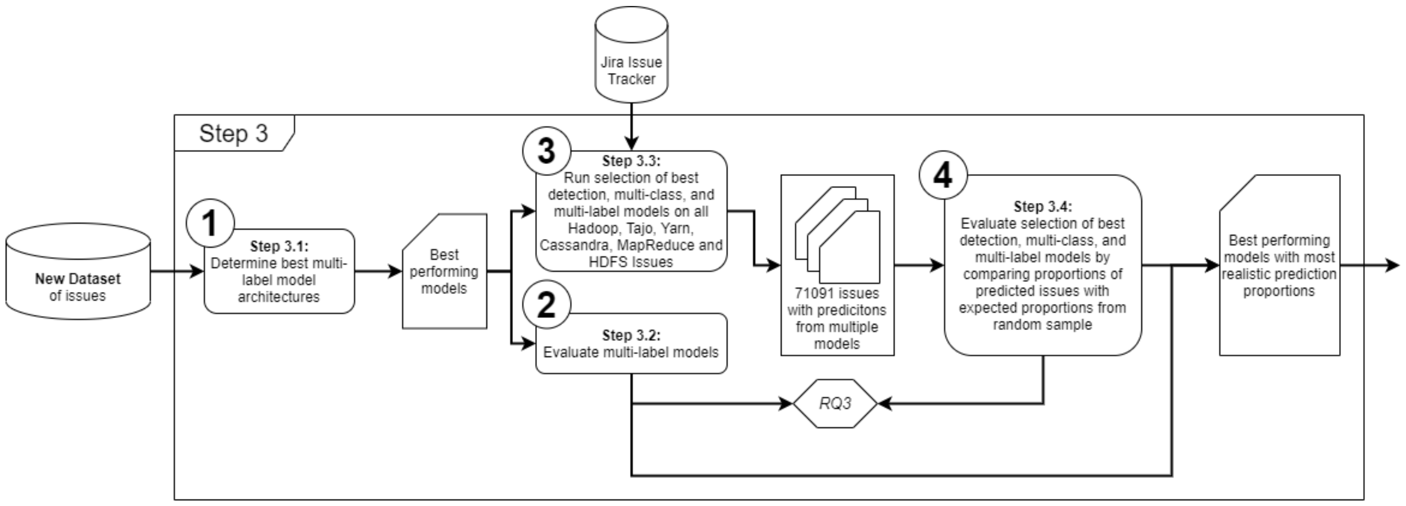

Jira Issue Tracker
Step 3
New Dataset of issues
1
Step 3.1: Determine best multi-label model architectures
Best performing models
3
Step 3.3: Run selection of best detection, multi-class, and multi-label models on all Hadoop, Tajo, Yarn, Cassandra, MapReduce and HDFS Issues
2
Step 3.2: Evaluate multi-label models
71091 issues with predicitons from multiple models
4
Step 3.4: Evaluate selection of best detection, multi-class, and multi-label models by comparing proportions of predicted issues with expected proportions from random sample
Best performing models with most realistic prediction proportions
RQ3

Fig. 18. Detailed design over step 3: Optimise and Evaluate Multi-label Deep Learning Models.

Model Type	Feature Type(s)
FNN	BOW (frequency), BOW (normalised), TF-IDF, Doc2Vec
CNN	Word2Vec
RNN	Word2Vec
BERT	Text

Table 12. List of all the models we experimented with.

4.6.1.1 Preprocessing

We will be developing classifiers which use the summary and description of issues as inputs. The first step when evaluating the deep learning models, is preprocessing the text inputs to remove unnecessary data. For the FNN, CNN, and RNN models we largely followed the same methodology as done in [18]. In particular, we used the same preprocessing. Specifically, we first cleaned the text in the following way:

1. We used heuristically created regular expressions to remove logging output and tracebacks *not* contained in formatting tags from the text. These regular expressions can be found in [18]. The removed portions of text were replaced with special marker words.

2. We removed dates from the text by removing all sequences of three groups of numbers separated by either slashes or dashes (e.g. 1/2/1990, 2005-12-12). We replaced these with the word "date".

3. We removed IP addresses (optionally followed by a port number) from the text. We also allowed addresses where part of the address (i.e. one group) is obscured by `xx`. We replaced these with the word "IP address".

4. We removed links to websites. Links to other issues and source code repositories were replaced with special marker words ("issue link" and "github link"). Other links were replaced with generic link ("web link") markers. We searched for links based on both links with proper Jira formatting[18], as well as by looking for sequences of text (containing no spaces) which contain domain extension names (e.g. `.com` or `.net`)

5. We removed the content of {`code`} and {`noformat`} blocks (and the opening and closing tags themselves). We used the regular expressions already mentioned in point (1) in order to determine whether these blocks contained logging output or tracebacks; in these cases, we replaced the removed text with dedicated marker words. Otherwise, we simply replaced the removed text with generic code block or nonformat block markers.

6. We removed version numbers (e.g. `2.7.0`, `3.6.x`, `v2.1`) from the text and replaced these with the phrase "version-number".

7. We removed file sizes (e.g. 1.2GB) and Amazon instance type names (e.g. `c4.medium`)

8. We removed inline source code (code inside {} brackets). Most of the time, we just used a generic marker for inline code. However, we also used regular expressions to detect potential names of methods, or classes (e.g. `org.apache.hadoop.examples.Grep`, `SnapshotManager#createSnapShot`). lowerCamelCase names were replaced with a marker for method/variable names. UpperCamelCase names were replaced with a marker for class names. Contrary to [18], we did not specifically test for package names and had no special markers for package names. The reason for this is that this would require knowledge of all packages in a project, which will generalise poorly.

9. We searched for lowerCamelCase and UpperCamelCase words not contained in inline code formatting. We also replaced these words with variable/method and class markers, as described above.

10. We tried to search for method and class names not contained in inline code formatting tags by searching for sequences of words separated by dots. This approach is likely to result in some false positives. Hence, we attempted to filter these out by discarding results ending in a file extension and by discarding float literals. The removal of version numbers and Amazon instance names from the text also reduced the number of false positives.

11. We removed formatting tags containing text (e.g. {`panel`}, `*strong*`), but we kept the textual content. We removed all such formatting text present in the Jira formatting help. After this step, no further Jira formatting should be present in the text.

12. We removed filenames not enclosed in tags by searching for sequences of words (possible containing dots, dashes, and

[18]`https://jira.atlassian.com/secure/WikiRendererHelpAction.jspa?section=all`

underscores), concatenated by slashes (/). The removed text was replaced by the phrase "filepath".

13. All remaining numbers and punctuation were removed from the text.

At this point, we had one optional preprocessing step, which we will refer to as "fine-grained technology replacement". With this approach, we replace the names of technologies in issues with special marker words. This approach was based on the observation that the presence of the same technology name may have a different meaning depending on the project. For instance, in Hadoop, an occurrence of "Hadoop" in a title probably has little to nothing to do with a technology decision, while it may be more likely to be technology decision related in projects not directly related to Hadoop.

In order to work around this potential problem, we implemented a fine-grained technology name replacement system. To achieve this, we first came up with a list of technology names. We started with the technology ontology class used in [18]. We further extended this list by adding all project names from our database. We manually inspected the project names before adding them to 1) remove unnecessary suffixed (such as "(retired)"), 2) also include any likely aliases (e.g. "Apache Zookeeper" can also simply be called "Zookeeper", or might be referred to as "ZK").

We then implemented a mechanism which replaces technology names in the following way: If the name of project X (or any alias X' for project X) is encountered in an issue of project X, it is replaced with a marker P. If the name occurs in an issue not belonging to project X, then X is replaced with marker Q. Here, P and Q are configurable markers. For our research, we used $P =$"our development project" and $Q =$"technology".

The summary and description of every issue were cleaned up according to the approach outlines above, with the fine-grained technology replacements as an optional step. Next, the summary and description were concatenated and tokenised. Next, all words were converted to lowercase, lemmatised to remove inflected word forms, and stop words were removed. Next, the preprocessed issues were used to generated features.

As explained previously, due to a bug in our initial code, we have tested the performance of BERT with and without text preprocessing. For this model, we also experimented with the fine-grained technology replacements, but we never applied lowercasing, lemmatisation, or removed stop words, because BERT was not pre-trained with these types of preprocessing.

4.6.1.2 Feature Generation

The types of features we use fall into two categories: bag of words related features (term frequency, normalised term frequency, TF-IDF), and semantic embeddings (Doc2Vec and Word2Vec). We generated features for each in the following ways:

- For BOW (frequency & normalised) and TF-IDF, we collected a corpus of words from all the preprocessed labelled issues. We then generated the feature vectors as described in Section 3.3.2.1.

- For Doc2Vec and Word2Vec, we preprocessed all the ±1.5 million issues from six software domains (to be introduced further in Section 4.9), and trained the embeddings on all these issues. Note that this is different from BOW and TF-IDF; for those, we determined how the features should be generated solely from the labelled issues. For Word2Vec, we used the continuous skip-gram model and trained for five epochs. For Doc2Vec, we used the PV-DBOW model, and also trained for five epochs. For all models, we used vectors of length 300.

4.6.1.3 Model Architectures

In this section, we will describe the model architectures we used, and the corresponding hyperparameter we optimised. We used the same types of model architecture as previously in [18]. These are the model architectures as explained in Section 3.4. However, compared to our previous work, we extended the variety of possible options for various hyperparameters. We also included a number of regularisation options.

In the rest of this thesis we are using abbreviations to refer to certain model architectures:

- BOWF: Fully connected model with bag of words input, with the default frequency encoding.

- BOWN: Fully connected model with bag of words input, with the normalised frequency encoding.

- TF-IDF: Fully connected model with the TF-IDF input encoding.

- DOC2VEC: Fully connected model with a Doc2Vec encoding.

- CNN: Convolutional neural network with a Word2Vec encoding.

- RNN: Recurrent neural network with a Word2Vec encoding.

Table 32 in Appendix C shows a full overview of the hyperparameters that we could optimise for the models. This table also includes default values. Some values are rounded for clarity (since the tuning approach we used often samples values with more decimals than necessary). The exact values of the hyperparameters can be found in the database.

4.6.1.3.1 Shared Hyperparameters

- *Batch Size*: For the internship, we used a batch size that fits all samples in one batch [18]. However, it is generally recommended to use smaller batch sizes [34], such as batch sizes of around 32 to 64 [43]. To be on the safe side, so we do not miss out on performance, we decided to increase this range to be from 8 to 512.

- *Loss Function*: we planned to experiment with three different loss functions: cross-entropy, hinge, and squared hinge. As explained previously, we did not experiment Kullback-Leibler divergence because it is equivalent to cross entropy.

  However, due to a coding error, we accidentally omitted the squared hinge loss from our hyperparameter optimisation. We will get back to the consequences of this in Section 8.

- *Optimizer*: in Section 3.3.4.4, we covered many optimisers available in TensorFlow. We also covered the fact that, given sufficient tuning of the parameters of the optimisers, a more general optimiser (e.g. Adam) will never underperform one of their more specific cases (e.g. SGD). This means that the most promising optimisers to experiment with are AdamW, Nadam, and RMSProp. Finally, we reduced this to only AdamW and Nadam, because it has been shown that these two optimisers often outperform RMSProp [11].

- β_1, β_2, ϵ: these are the hyperparameters of the AdamW and Nadam algorithms as explained in Section 3.3.4.4.

- *Weight Decay (shared by all except BERT)*: The rate by which weights should be decayed while training. We perform a search with a logarithmic sampler. Because such a sampler cannot sample from an interval containing 0, we will always be using weight decay. However, it is possible for the weight decay to be set to a minimal value (10^{-10}).

- *Learning Rate Start, Learning Rate Stop, Learning Rate Step, and Learning Rate Power*: Given an initial learning rate η_i, end learning rate η_e, number of steps s, and a power p, the learning rate η_t used by TensorFlow[19] in epoch t is given by

$$\eta_t = \begin{cases} (\eta_i - \eta_e)\left(1 - \frac{t}{s}\right)^p + \eta_e & t < s \\ \eta_e & t \geq s \end{cases}$$

  The initial learning rate, end learning rate, number of steps, and the power can all be tweaked to determine how the learning rate is decreased in order to improve the training results.

4.6.1.3.2 FNN Model Hyperparameters

All FNN models share the same hyperparameters, and the same search space for hyperparameter optimisation. We have the following hyperparameters for FNN:

- *Number of Dense Layers & Layer Size*: During our internship, we mainly tested rather shallow neural networks [18]. For this work, we wanted to test with deeper networks, up until 5 layers. We optimise the number of hidden layers (0 to 5), and the layer sizes $(2, 4, 8, \ldots, 1024, 2048)$. We adopted the additional requirement that the layer sizes should be descending. As a result, the first hidden layer can be bigger than the input layer, but no subsequent hidden layer can be larger than its preceding layer. This reduces the search space, while still allowing a transformation to a higher dimensional space in the first layer (like support vector machines implicitly do, using the kernel trick [7]).
- *Dense Layer Activation*: We tested a wide variety of activation functions. All hidden layers share the same activation function. We experimented with all activation functions covered in Section 3.3.3.5, except for a few:
  - We excluded the sigmoidal activation function because it is essentially equivalent to the hyperbolic tangent activation function ($\tanh(x) = 2\sigma(x) - 1$).
  - We did not include LeakyReLU in our search, because PReLU is essentially the same, but with the added advantage that it also automatically optimises the value of α.
  - We did not use the softmax function because it is meant for outputting probability distributions as output, not for hidden layers.
- *L1 Regularisation*: for FNNs, we experimented with three different types of L1-regularisation: kernel (weight) regularisation, bias regularisation, and activation regularisation. The first of these penalises large weights. The second penalises large bias (constant) terms. The last one penalises large activations (and, as such, also large weights and large bias terms). Similar to weight decay, we use a logarithmic sampler to sample the coefficients determining the importance of the different penalties, and as such we will always use some degree of L1-regularisation. However, the penalties may become very small.

  We decided to not use L2 regularisation. This is because weight decay has a similar effect as L2 regularisation. In fact, for some optimises (e.g. SGD), the two are fully equivalent; for Adam in particular, weight decay has been shown to lead to better generalisability than L2 regularisation [51]. Additionally, we already have many other measures to prevent overfitting. Not including L2 regularisation reduces the search space of possible models.

[19]`https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/schedules/PolynomialDecay`

- *Dropout*: we experimented with dropout, but only in the hidden layers. Hence, neurons in the input layer could not become disabled. We opted to use dropout and not batch normalisation, because both have a similar effect in FNN, while dropout is computationally significantly more efficient [30].

4.6.1.3.3 CNN Model Hyperparameters

For the CNN model, we had the following hyperparameters:

- *Number of Convolutions, Convolution Size, and Number of Filters*: We experimented with one up to five parallel convolutions, where the size (height) of the filters ranged from 1 to 64. Two convolutions with the same size were not allowed in the search. The number of filters (i.e. the number of convolutions applied in every convolution layer) is equal for all convolution layers, and its set of possible values was given by $1, 2, 4, \ldots, 64$.
- *Convolution Activation Function*: An activation can also be applied to the output of the convolution layer. For this, we experimented with the same activation functions as for regular hidden layers.
- *L1 Regularisation*: We applied L1 regularisation to the weights of the convolution kernel, the bias of the convolutional layers, and the activity of the convolutional layer. This was done in the same way as the L1 regularisation for the fully connected layers. For the same reasons as with FNN, we did not use L2 regularisation.
- *Batch Normalisation*: We experimented with batch normalisation. Specifically, we tested with models which have a batch normalisation layer after every convolutional layer. First, we tested models with and without batch normalisation; either all convolutional layers were followed by a batch normalisation layer, or none. When using batch normalisation, we also optimised the momentum value used to estimate the population mean and variance. We did not experiment with dropout, because batch normalisation has been shown to be more effective for CNNs [30].
- *Fully Connected Layer*: We allowed the concatenation layer in the network to optionally be followed by a single fully connected layer before the final layer. For this layer, we optimised the same hyperparameters as for a FNN network.

4.6.1.3.4 RNN Model Hyperparameters

For the RNN model, we had the following hyperparameters:

- *Number of RNN Layers, RNN Layer Type, and RNN Layer Size*: Compared to the work in [18], we experimented with a slightly more elaborate variety of RNN architectures. Most notably, we experimented with networks with either 1 or 2 bidirectional layers, and we experimented with various different sizes for the layers (16, 32, 64, 128). Research indicates that gated units offer clear performance advantages over simple traditional RNN units [12]. However, there is no clear winner between LSTM and GRU. Therefore, in our experiments, we will evaluate both LSTM units and GRUs, while disregarding the simple RNN units. Note that we allowed networks where the two RNN layers use different types of units. For the recurrent layer activation, we used the hyperbolic tangent as activation function, since this is generally the best choice [63].
- *Fully Connected Layer*: We allowed the concatenation layer in the network to optionally be followed by a single fully connected layer before the final layer. For this layer, we optimised the same hyperparameters as for a FNN network.

4.6.1.3.5 BERT Hyperparameters

For the BERT model, Devlin et al. ([21]) recommended hyperparameter values that should work good for fine-tuning the model. These parameters can be found in Table 33 in Appendix C. In particular, compared to the parameters shared by all other model types, for BERT we often have a reduced search space with some standard values. Specifically:

- The Adam optimiser is used. The parameters for Adam are fixed to $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-7}$.
- We only experimented with constant learning rates. In particular, we only experiment with $2 * 10^{-5}$, $3 * 10^{-5}$, and $5 * 10^{-5}$.
- The loss is fixed to cross entropy.
- The weight decay is fixed to 0.01.
- For the batch size, we only experimented with 16 and 32.
- Since we are performing transfer learning, we do not need to train the model for many epochs. We tried 2, 3, and 4 epochs.
- Contrary to when we used BERT for searching, we did not freeze any layers.

When we used BERT for finding architectural issues, we had limited hardware capabilities. This meant that we could not train BERT using the recommended batch sizes of 16 and 32, but had to use a batch size of 1. However, the performance of BERT with such a small batch size was really poor. To improve the performance, we froze the first 10 layers of the model. However, once we obtained access to a supercomputer, we were able to train and optimise BERT with the recommended hyperparameters [21]. To be clear, we used BERT with 10 frozen layers for finding architectural issues (see Table 8). For all other evaluations, we used an optimised set of the recommend hyperparameters (see Table 33), which did not require freezing any layers.

Finally, for BERT we experimented both with and without text preprocessing. BERT ignores words which are not part of its vocabulary. Hence, we cannot use marker words to replace the formatting, but we have to use proper phrases instead. Because things work differently for BERT, we had to explicitly test whether the preprocessing improved performance or not.

4.6.1.4 Hyperparameter Optimisation

In previous work [18], we spend a lot of manual effort into hyperparameter tuning. To reduce this effort required, we decided to incorporate Keras Tuner[20] into our deep learning tool. Keras Tuner is a library for automatically optimising the TensorFlow[21] models we used. The tuner takes in a range of possible values for the hyperparameters of a model, and explores a portion of the search space of possible combination of hyperparameters in search of the best performing model. The search algorithms were explained in more detail in Section 3.3.7.

Except for BERT, we used the Hyperband algorithm for optimising the hyperparameters. The entire search spaces for the FNN models can be found in Tables 34-37 in Appendix C. The search space for the CNN model can be found in Table 38 in Appendix C. The search space for the RNN can be found in Table 39 in Appendix C. These tables also contain the best hyperparameters found using Hyperband.

For all models, we ran 3 iterations of the algorithm. In each iteration, a little over 250 different random hyperparameter configurations were tested. This number was reduced to a single best set of hyperparameters by means of successive halving. This allowed the algorithm to filter out poor performing sets of hyperparameters early in the training process, to give good performing hyperparameter sets relatively more training time. After every iteration we checked the results, and we did not see much improvement after 3 iterations. Each combination of hyperparameters was tested 3 times, after which the average performance was used to compare the different combinations. We did this to mitigate the randomness of training a deep learning model (e.g. the impact of random initialisation of the weights). Every time we tested a combination of hyperparameters, the model was allowed to train for a maximum of 200 epochs (although Hyperband might train for significantly less). Additionally, we used early stopping; if the performance (in terms of loss) did not improve by more than 0.01 for 5 epochs, the training of a classifier was considered done.

We used 80% of the data for the training set, 10% for the validation set and the last 10% for the test set. The splitting was done in a stratified manner. The training, validation, and test sets were fixed for all models and hyperparameter combinations. This was to make a fair comparison between the different models and different hyperparameter combinations. The data used for the hyperparameter optimisation consisted of all labelled issues from the Apache projects Hadoop, Tajo, Mapreduce, Cassandra, HDFS and Yarn.

For BERT, we performed an exhaustive (i.e. grid) search. This means that all 18 combinations of hyperparameters were tested exhaustively. The search space and optimal hyperparameters are given in Table 33. We used the same training, validation, and test set as for the other models. Additionally, all BERT models were also evaluated three times to mitigate the potential effects of randomness.

For each optimised model, we performed two experiments. One with the preprocessing as described in Section 4.6.1.1 without fine-grained technology replacement, and one with the additional step of fine-grained technology name replacement. Note that the models were first optimised without the fine-grained technology replacement.

4.6.2 Step 3.2: Evaluate Multi-label Models

During our internship, we optimised and evaluated our models using a 10-fold cross validation. For this thesis, we opted to evaluate the models on a holdout test set. This is because, 90% of the dataset is used for optimising the hyperparameters. Evaluating the performance using a 10-fold cross validation on this dataset would mean that we do not actually test the model on data that is entirely foreign to the classifier. In practice, evaluating the performance using a k-fold cross validation also tends to be too optimistic compared to evaluating on a holdout test set [2, 64]. We selected the holdout test set in a stratified manner, as this was found to give a better estimation of the real performance of a classifier [32].

4.6.3 Step 3.3: Compute Predictions on Data Storage & Processing Projects

In order to evaluate whether the classifiers seem to make sensible predictions, we will evaluate the proportions of classes in predictions made by the classifiers. Based on our random sampling, we know the expected proportions for the different types of issues in the six projects the models were trained and evaluated on. In this step, we computed the predictions of the multi-label classifiers on the remaining issues from these six projects. We also included some detection and multi-class classifiers from our earlier work in [18] to see how multi-class classifiers perform compared to these models.

For detection, we selected the CNN and BOWF model. CNN was the best performing model, and we also included BOWF, as this model showed good performance while using completely different input features. For multi-class classification, we used the

[20]`https://keras.io/keras_tuner/`
[21]`https://www.tensorflow.org/`

RNN model and the BOWF model. The RNN model performed the best for the multi-class classification task, and we selected BOWF again for the use of different input features. For each detection and multi-class classification model, we evaluated two variants. The first variant is trained on the original dataset from [18], while the second variant is trained on the new expanded and relabelled dataset.

4.6.4 Step 3.4: Evaluate Models using Proportions of Predictions

In this final step, we evaluated the best performing models based on the performance on the test set, and based on the proportions of the predictions made by the models. The full details of this analysis will be given in Section 6.4. The most promising models in terms of performance were BERT (both with and without preprocessing) and TF-IDF* (TF-IDF with fine-grained technology replacement). However, only BERT also made predictions with proportions close to the expected ones.

4.7 Step 4: Evaluate Need for Additional Data

One of the hypotheses from the previous research [18], was that more data was necessary for improved multi-class classification. We will test this hypothesis by training models varying training dataset sizes, so that we can answer **RQ4**. We experimented only with the best multi-label model: BERT (with preprocessing)

We have a total of 4712 data storage and processing issues that we can use for this experiment. We used 10% of the data for a fixed test set, as to make a fair comparison between the models possible. We used another 10% for a fixed validation set. The rest of the data was used for a growing training set. We increased the size of this training set by 25 issues in each iteration, until we could not increase the training set any further:

- iteration 1: 25 issues
- iteration 2: 50 issues
- ...
- iteration 150: 3750 issues
- iteration 151: 3768 issues

Although the individual results might suffer from the randomness of the deep learning training process, given the small step sizes this should give us the ability to observe a trend line.

4.8 Step 5: Evaluate Generalisability of Multi-label Classifiers to Projects in the Same Domain

For **RQ5**, we want to evaluate the generalisability of the multi-label classifiers. The first way in which we will examine the generalisability of the classifiers, is by examining the generalisability to projects in the same domain, but foreign to the dataset. We will do this using project cross validation, as proposed in [18].

In our cases, this means that we start out with the dataset of data storage & processing issues. These issues are then split into folds according to the projects they belong to. We then perform cross validation. However, we compute the training, validation, and tests sets in a particular way. Suppose that we have projects $p_1, p_2, \ldots, p_k$, and corresponding subsets of issues $D_1, D_2, \ldots, D_k$. Then, on iteration i of the project cross validation, the testing set is D_i and the training and validation set are obtained by splitting

$$\overline{D} = \bigcup_{j \neq i} D_j$$

into a training and validation set. Hence, on every iteration of the validation process, issues from one project are used for the test set, and the issues from the other projects are used for the training and validation sets. This is also depicted in Figure 19. The final performance over all folds is obtained by averaging the performance over every fold. Note that for this analysis, we excluded the issues from HBase and Submarine; these are only three issues in total, and would not constitute a meaningful test set.

The final conclusion from this test was that BERT (with preprocessing) had the best ability to generalise to different project.

4.9 Step 6: Evaluate Generalisability of Multi-class Classifiers to Projects in Different Domains

In this final step, we want to answer **RQ6**. We will mainly focus on evaluating the generalisability to projects of different domains of BERT, the classifier which scored best on the test for generalisability to different projects in the same domain. The detailed design for this step is depicted in Figure 20.

The idea is that we use BERT (with preprocessing) to predict on a large amount of issues from a variety of domains. We then take random samples from these issues, and manually classify these. We then compared these manually annotated labels with the predictions from BERT in order to evaluate the generalisation performance to different domains.

Note that due to a bug in our code, we accidentally used BERT without preprocessing to compute the predictions. We will cover this in more detail in section 8.

4.9.1 Step 6.1: Predict with Multi-label Models on Issues from Different Domains

The first step is to let BERT predict on many issues from various domains. Among all the projects in the database of Montgomery, Lüders, and Maalej, our first supervisor and a Bachelor student (Sarah Druyts) identified six domains:

1. Software development tools
2. DevOps and cloud
3. Data storage and processing
4. Web development
5. Content management
6. SOA and middlewares

Furthermore, from the projects in our database, they identified which projects belonged to which domain. Projects not belonging to any of these domains were not considered for the samples in this step. In the end, we had 1,345,784 issues from these six different domains.

The projects in the training dataset (Hadoop, Tajo, Yarn, MapReduce, HDFS, and Cassandra) all belong to the data storage & processing domain.

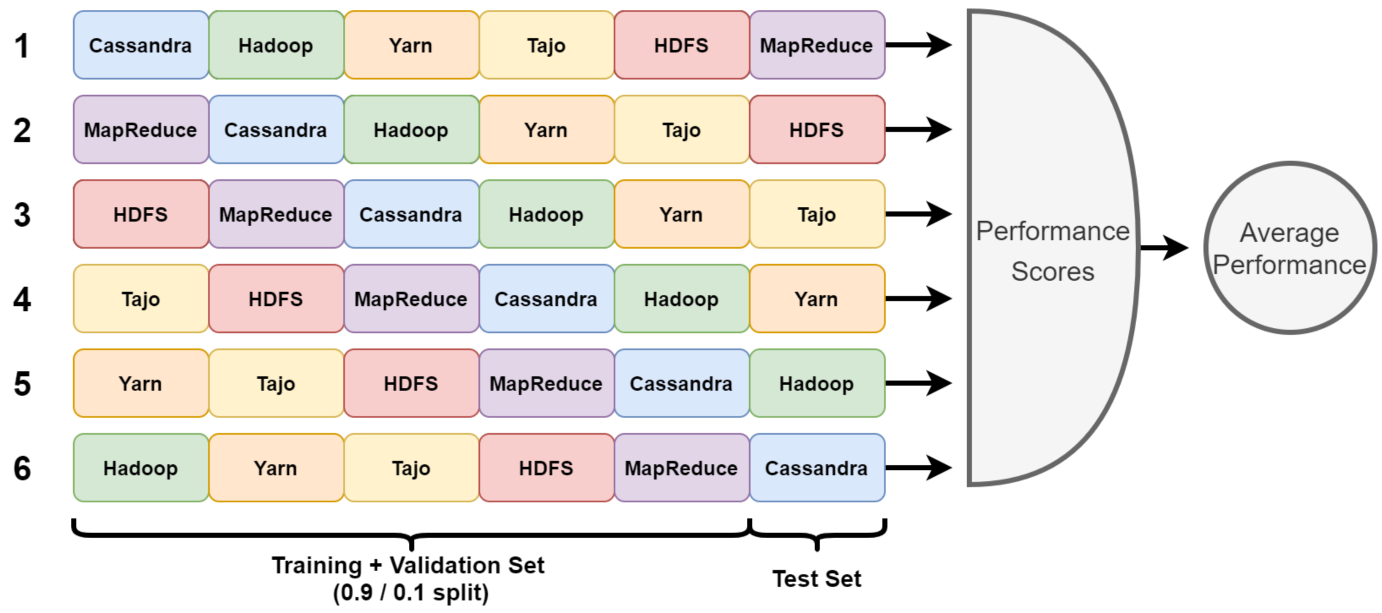


Fig. 19. Graphical depiction of project based cross validation. For each test run, one project is used as the test set, and the remaining projects are used to obtain the training and validation sets.

4.9.2 Step 6.2: Taking Random Samples of Issues from Different Domains

In this section, we describe how we took the random sample from the issues from the different domains. We took two random samples: one fully random samples, and one random sample based on the confidences outputted by the classifier. These two random samples also form the last two steps in arriving at the final dataset of issues presented earlier in Figure 17.

While initially labelling issues from some of these projects, we found that there were a couple of projects that used the Jira issue tracker as a ticketing system for their users, and not for discussions between developers. This meant that, contrary to other projects, these projects 1) are not about programming related problems, and 2) do not contain any architectural knowledge. As such, we decided to exclude these projects, The projects excluded were:

- All projects from the Jira repository (not to be confused with the JiraEcosystem repository)
- The MVNCENTRAL and OSSRH projects from the Sonatype repository

4.9.2.1 Step 6.2.1: Fully Random Sample from the Domains

The first random sample is used for evaluating the generalisability of the model as a plain classifier. The idea is that we take a completely random sample, annotate the sample, and use the performance of the classifier on this random sample as an estimate of the classifier's performance on the entire population of 1.35 million issues we computed the predictions for.

The sample consists of 400 random issues from the different domains. We took this sample in a stratified manner, meaning that we selected more issues for the domains with more issues. To be more specific, we selected the following amounts of issues:

- 50 from software development and tools
- 63 from DevOps and cloud
- 124 from data storage & processing
- 48 from web development
- 74 from content management
- 43 from SOA and middlewares

4.9.2.2 Step 6.2.2: Random Sample from the Domains based on Classifier Confidence

The second sample is used to evaluate the generalisability of the model as a search tool for ADDs. This means that we want to check with what precision the classifier is able to identify existence, executive, and property issues. For each class, i.e. existence, executive and property, we selected the issues that were predicted to be of that class (confidence > 0.5). Then for each class, based on the predicted confidence of the class, we put the issues in 10 bins according to their predicted confidence as follows:

- bin 1: $0.50 < \text{confidence} \leq 0.55$
- bin 2: $0.55 < \text{confidence} \leq 0.60$
- ...
- bin 10: $0.95 < \text{confidence} \leq 1.00$

Then, we selected 33 issues from those bins in a stratified manner. In other words, we selected the amount of issues from every bin proportional to the amount of issues in that bin. Doing this for each class, existence, executive and property, yielded $3 \times 33 = 99$ issues and doing this for all 6 domains yielded $6 \times 99 = 594$ issues. We did not do this for the non-architectural issues, since this sample is used for evaluating the model as a search tool for finding architectural issues.

4.9.3 Step 6.3: Classify Issues

The issues from both samples were split evenly between the two authors, meaning that both authors did 200 issues each from the first sample and 297 issues each from the second sample. These issues were also annotated according to Section 4.5.2.1, meaning that all uncertain cases were double-checked and discussed, and conflicts were resolved by the first supervisor. The final set of issues collected this way is displayed in Table 13.

Sample	Domain	Existence	Executive	Property	Existence/Executive	Existence/Property	Executive/Property	Existence/Executive/Property	Architectural	Non-Architectural	Total
Random	Content Management	1	3	3	0	5	1	0	13	59	72
	Data Storage & Processing	5	6	6	0	3	1	1	22	102	124
	Devops and Cloud	1	3	4	0	0	0	1	9	54	63
	SOA and Middlewares	1	1	2	0	1	0	0	5	38	43
	Software Development Tools	1	2	0	0	3	0	0	6	44	50
	Web Development	1	2	1	0	0	0	0	4	44	48
	Total	10	17	16	0	12	2	2	59	341	400
Confidence	Content Management	11	17	9	0	30	3	3	73	26	99
	Data Storage & Processing	10	17	15	3	21	7	2	75	24	99
	Devops and Cloud	3	14	22	1	9	7	1	57	42	99
	SOA and Middlewares	3	19	11	3	8	8	3	55	44	99
	Software Development Tools	8	16	12	0	13	3	0	52	47	99
	Web Development	7	20	14	1	14	3	2	61	38	99
	Total	42	103	83	8	95	31	11	373	221	594
All	Total	52	120	97	8	107	33	13	432	562	994

Table 13. Overview of issues collected from the random samples from the six domains

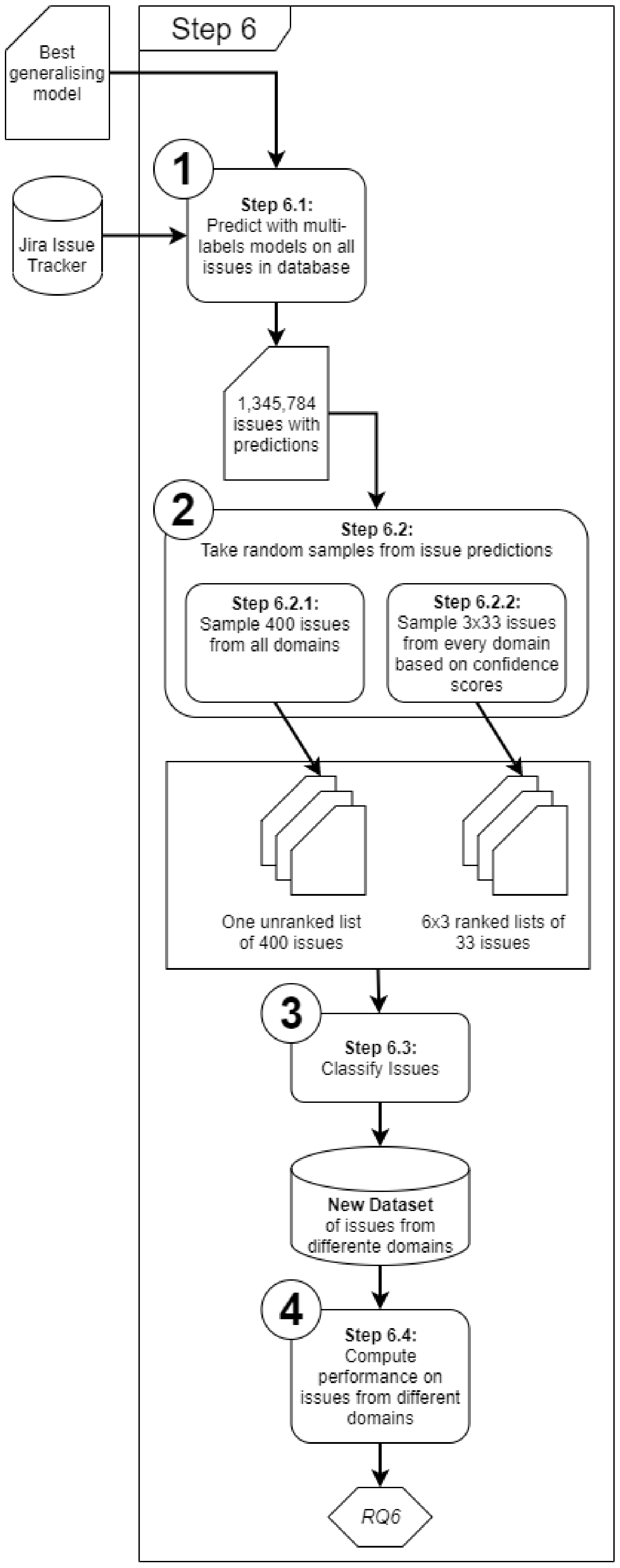

Step 6
Best generalising model
Jira Issue Tracker
1
Step 6.1: Predict with multi-labels models on all issues in database
1,345,784 issues with predictions
2
Step 6.2: Take random samples from issue predictions
Step 6.2.1: Sample 400 issues from all domains
Step 6.2.2: Sample 3x33 issues from every domain based on confidence scores
One unranked list of 400 issues
6x3 ranked lists of 33 issues
3
Step 6.3: Classify Issues
New Dataset of issues from differente domains
4
Step 6.4: Compute performance on issues from different domains
RQ6

Fig. 20. Detail study design for step 6: Evaluate Generalisability of Multi-class Classifiers to Projects in Different Domains.

4.9.3.1 Labelling Quality Assessment

Like we did for the first two labelling rounds, we once again evaluated the agreement for this round of labelling. The detailed agreement statistics can be found in Figure 45 in Appendix B. The average agreement and kappa score per class is given in Table 14.

Class	Agreement	Kappa Score
Existence	0.7616	0.4911
Executive	0.8788	0.6044
Property	0.7576	0.4663
Non-Architectural	0.7108	0.3817

Table 14. Average (over all pairings of annotators) agreement and Kappa score per class for the second round of labelling issues (i.e. the relabelling of issues from the first round).

Like previously, the agreement and Kappa for the "executive" class are still rather good, although slightly worse than before. For existence, the agreement and Kappa score are actually the best they have ever been across all rounds.

For property issues, the agreement has also become slightly better, although the two authors still seem to assign it too often (see Figure 45). We think that this is the case because for some quality attributes (specifically usability, maintainability, and efficiency) it may be difficult to 1) define what constitutes an improvement to these quality attributes, and 2) determine if an issue actually improves these. Since quality attributes may not be mentioned directly, it may be necessary to infer them from the rest of the text in the issue. We found that for the three aforementioned quality attributes, it was particularly easy to make an incorrect judgement. As an example, the two authors often thought a change could potentially improve usability, while the first supervisor did not agree with this.

Finally, the agreement for the non-architectural class is still rather poor. There are still many cases where it is difficult to determine whether the change described in an issue is related to the architecture of the system.

When looking at all labelling quality evaluations we did, we have some general takeaways:

Quality Evaluation Takeaways:

- Agreement is generally quite good.
- Kappa score is generally quite poor.
  - Most types have a kappa score between 0.2 and 0.6
  - We never had kappa scores exceeding the recommend $\kappa = 0.8$
- Executive issues have the best agreement.
- Existence issues are particularly problematic due to an unclear understanding of what makes an issue existence.
- Non-architectural is also a problematic class, which we attribute to the fact that there are many "edge-case" issues.
- The agreement for property issues is not exceptionally bad, but also not exceptionally good.
- The agreement and kappa scores we computed should represent a worst-case scenario.

4.9.4 Step 6.4: Evaluate DL Models

With the labelled issues, we evaluated the performance of BERT on the different domains. For the fully random sample, we computed F_1, precision and recall scores; this means that we evaluate the classifier for use as an actual classifier.

For the random sample based on confidence, we will compute the precision@confidence graph; this means that we will compute the precision as a function of the classifier confidence. This will provide an estimate how useful BERT would be as a search tool in different domains.

4.10 Reproducing and Extending our Results

We want researchers and practitioners to be able to reproduce our results, and provide them the ability to extend our research. We therefore uploaded the code for our entire tool (Maestro) to GitHub[22]. Additionally, we have uploaded an archive containing all data related to our annotated issues and deep learning models[23]. Appendix I provides a brief explanation on how to use Maestro to replicate our results. Additionally, In Appendix J, we provide details on how one can find what issues are (re)labelled during which round, and in Appendix K we provide details on how one can find the trained models and obtained performance metrics we used for presenting our results.

[22]`https://github.com/mining-design-decisions/Maestro`
[23]`https://zenodo.org/record/8225601`

5 Maestro

During our previous work in [18], we developed a command line tool for training and evaluating neural networks. Another student, Sarah Druyts, worked on developing a graphical user interface to improve the usability of the system. However, while working on this research, we decided to further improve the system by adding 1) a centralised database of labels and issues, 2) an API for securely interacting with the database, 3) a client library for interfacing with said database, and 4) a system for collaboratively classifying issues from within the existing graphical user interface. The reasons for these changes were the following:

- *Data Centralisation & Consistency*

  Up until this point, issues were labelled using spreadsheets. Over time, this had led to a multitude of separate spreadsheets containing labels. These spreadsheets were located in different source code repositories. Additionally, more issue labels were located in other repositories. With the addition of a database, we created a single source of truth for all labelled issues.

  Additionally, we were able to convert the dataset to a format where we do no longer use the issue keys to relate labels to issues. We found out that issue keys are not proper unique identifiers in two different ways: 1) the key of an issue may change, and 2) different Jira instances may have projects with the same name. Especially the first point was a problem, because it had already caused some hard-to-detect duplicate entries in the existing dataset.

  The database also serves as a centralised place to store all model configurations, the performance evaluations of models, trained models, the predictions made by trained models, and any other outputs generated by the deep learning system. This means that trained models and prediction results can be now be shared more easily.

- *Usability and Flexibility of the Deep Learning System*

  Previously, the deep learning system was difficult to work with. All input had to be prepared manually using scripts; first issues had to be downloaded, then they had to be preprocessed and put into a specific JSON structure, and only then could they be fed to the deep learning system. This lead to a difficult workflow, and also complicated development of the graphical user interface. By adding the database and moving more preprocessing code into the deep learning system itself, we were able to make it easier to feed new data into the deep learning system for training or prediction.

- *Data Availability*

  Up until this point, we were dependent on the Jira API to download issue data, such as the summaries and descriptions of issues. With the database, we were able to collect data from multiple Jira instances in a single place. We are also no longer limited by rate limits imposed on Jira APIs. Hence, the addition of the database increased the availability of the issue data.

- *Improved Labelling Workflow*

  The addition of the database enabled the addition of a centralised labelling workflow without the use of external tools and spreadsheets. It became possible to classify issues in the graphical user interface itself, and store the discussions between annotators centrally. Additionally, the dataset of labels could now be updated in real time while labelling; previously, spreadsheets had to be merged into the dataset manually.

The database for the system was based on a database of issues created by Montgomery, Lüders, and Maalej in [58], which was brought to our attention by our first supervisor. Montgomery, Lüders, and Maalej created a MongoDB archive containing the data from the issues from multiple Jira instances. In total, data was collected from 1822 different projects, resulting in a database containing around 2.7 million issues. The database contains all public issue data available from the Jira API. This includes not only issue summary and description, but also all comments and other issue characteristics. Henceforth, we will refer to this initial database as the JiraRepos database.

First, we enhanced the JiraRepos database with a script for updating the contents of the database. While the JiraRepos dataset was relatively up-to-date (January 2022), our dataset of manual labelled issues contained issues that were newer. Montgomery, Lüders, and Maalej did not provide a script to update the JiraRepos dataset without downloading all data again. As such, we created a script that prevents us from having to download all issues again, by only downloading issues that were created or updated after January 2022. Another problem we had is the fact that issues from some projects, such as Apache CloudStack [24], are not downloadable without authentication with the API. The original dataset by Montgomery, Lüders, and Maalej was obtained without authentication. Hence, we also updated the script so that it can use authentication. We identified the Apache projects (since we only had credentials for projects from the Apache ecosystem) that had this problem and downloaded all issues from these projects as well. In order to make this script easy to use for our tools, we incorporated the script into an endpoint of our database API.

In order to adapt the database to our use cases, we also stored other types of data. First, we store the manually assigned labels of issues. We also store comments from annotators in order to preserve their reasoning and discussions. Second, we store information that is relevant for our machine learning models. This information includes the configuration of the models, (pre-trained) word embeddings, saved models, performance metrics of the models and the predictions of the models. Finally, we have a tagging system; tags are additional metadata which can be attached to issues. For instance, issues can have tags to denote how they were found, or by whom they were labelled.

Since we want to prevent unknown people modifying our database, we created a database API that requires authentication for every endpoint that modifies the database. This means that everyone can retrieve the data from the database, but only trusted people can modify the data. This database API also allowed us to create endpoints that make interaction with the database easier for the deep learning tool and the UI, as it introduces a layer of abstraction on top of the database. Furthermore, we use HTTPS for secure communication between the client and the server.

We create a daily backup, in case something goes wrong. This might be due to user error, malicious parties or because we use a server without replication, due to cost savings. This automatic backup is stored on Google Drive.

All aforementioned changes have resulted in our tool, which we call Maestro. Maestro is a tool for finding and exploring architectural design decisions. The tool allows users to search for issues using keywords, design and evaluate classifiers, use classifiers to find issues, view statistics about issues, and manually label more issues in order to expand the existing dataset while supporting discussions between annotators. In this chapter, we briefly summarised the work we did on the backend of the tool. For more details on Maestro as a tool, including its envisioned use cases and workflow, we refer to its accompanying paper [53] (since the work has been accepted for publication, but has not been published yet, the paper can be found in Appendix D). Information about the architecture, and how to use and run Maestro, can

[24]`https://cloudstack.apache.org/`

be found in our repository[25]. Additionally, the archives containing the data for the JiraRepos database, and the data regarding our deep learning models and annotated issues can be accessed here[26].

[25]https://github.com/mining-design-decisions/Maestro
[26]https://zenodo.org/record/8225601

6 Results

In this section, we will present and discuss the results we obtained. We start off with discussing the results of labelling issues. We will cover the results research questions by research question.

6.1 RQ1: Measure Precision for Finding ADDs Using Random Sampling

We classified two batches of 400 randomly samples issues (see Section 4.5.1.1). The first batch of issues was taken from web development projects (Apache Solr, CloudStack, TomEE, JSPWiki and Brooklyn), and the second batch was taken from data storage & processing related projects (Apache Hadoop, Cassandra, Yarn, Mapreduce, HDFS and Tajo). The results for the web projects can be found in Figure 21. The results for the data storage & processing projects can be found in Figure 22. Additionally, we took a random sample from six different software domains. Although the purpose of this random sample was to answer **RQ6**, it is also relevant for this research question. The results of this random sample are shown in Figure 23.

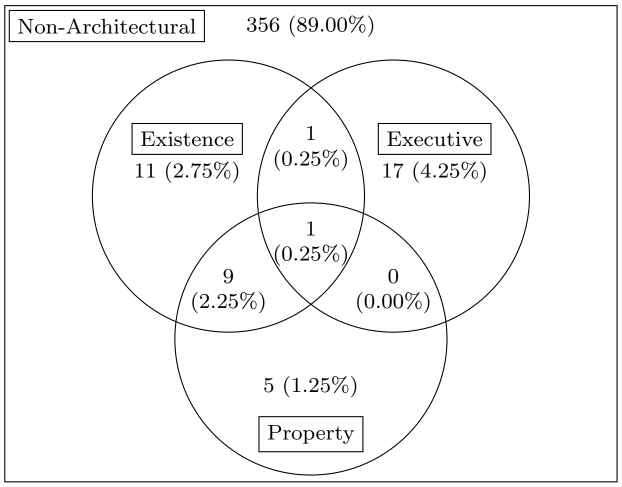


Fig. 21. Distribution of issues found by taking a random sample from the Apache projects Solr, JSPWiki, CloudStack, Brooklyn, and TomEE (total amount of issues: 400) belonging to the web development domain.

For the web development projects, we found that 11% of the issues were architectural. For the data storage and processing issues, we found this to be 12%. Furthermore, the random sample we took from all six identified domains, consisted of 14.5% architectural issues. This confirms the suspicion that in general, the number of architectural issues is low compared to the number of non-architectural issues.

This shows that finding architectural issues a hard task; randomly selecting issues is inefficient for finding large amounts of architectural knowledge. Hence, tools such as a keyword search engine, Maven POM file analyser, static source code analysis, and machine learning classifiers, are essential for efficiently finding architectural issues. Although the fraction of issues containing ADDs is small, given that the six domains contain around 1.5 million issues in total, we suspect that one could find around 150,000-225,000 architectural issues in these six domains. In total, the database contains around 3 million issues, possibly containing 300,000-450,000 architectural issues. However, we did not take a sample from the other 1.5 million issues, so we cannot be sure about these numbers. These numbers show that issue tracking systems are rich in architectural knowledge, making them a

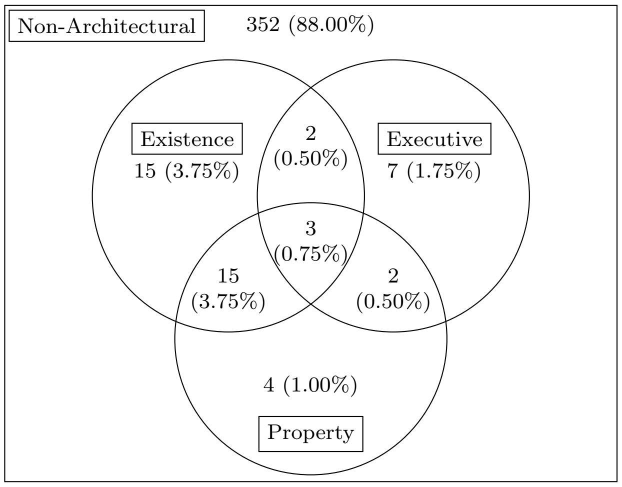


Fig. 22. Distribution of issues found by taking a random sample from the Apache projects Hadoop, Tajo, Yarn, HDFS, MapReduce, and Cassandra (total amount of issues: 400) belonging to the data storage & processing domain.

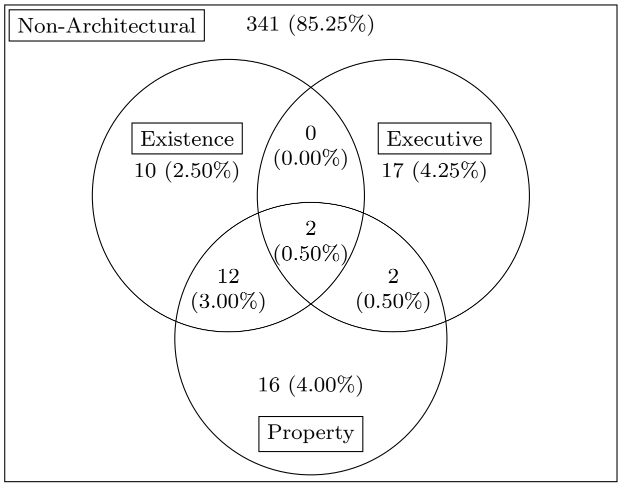


Fig. 23. Distribution of issues found by taking a random sample from six software domains (total amount of issues: 400).

good source for research on software architecture.

Due to the small amounts of data we have for the architectural design decision subtypes (existence, executive, and property), we cannot make strong claims regarding the proportions of these subtypes in random samples. This is mainly because the uncertainty induced by our 95% confidence interval is still relatively large, with how small these percentages are. For example's sake, consider the property class for the web development projects (Figure 21). There are 15 property issues. This means that the total proportion of property issues would be estimated as $\hat{p} = 15/400 = 0.0375$. However, for the standard error, we have

$$\mathrm{SE}(\hat{p}) = \sqrt{\frac{\hat{p}(1-\hat{p})}{400}} = \sqrt{\frac{5775}{400^3}}$$

This means that $1.96 * \mathrm{SE}(\hat{p}) \approx 0.019$. Hence, the 95% confidence interval for the true population mean is $[0.0185, 0.0565]$. Hence, we know (with 95% certainty) that the true proportion of property issues must be between 1.85% and 5.65%. Throughout the remainder of this discussion, we will use the maximum likelihood estimates (e.g. 3.75% for property issues in the web development domain). However, we should keep in mind that there is still room for quite some uncertainty because we are dealing with small percentages.

Comparing Figures 22 and 21, it seems that between domains, the proportions of the subtypes can differ a lot. For example, for the six projects in the data storage & processing domain (Figure 22), developers have to make more component (existence) decisions and decisions are often focused on the quality of the system (many issues contain at least a property decision). Projects like Hadoop and Cassandra are pretty stand-alone projects, making the number of executive issues small. Furthermore, these projects require efficient data processing and storage, showing the need for many property and existence decisions. In contrast, for the web development domain (Figure 21), it seems that developers have to deal more often with external factors (executive). When looking at the random sample of issues from all six domains in Figure 23, it seems that, in general, developers most often have to make property decisions, while existence and executive decisions are made almost equally often. Also, we see that existence decisions are often made to improve a certain quality attribute of the system (property), because many architectural issues contain both an existence and property decision.

RQ1 Takeaways:

- The fraction of architectural issues differs based on projects and domains, but is in the range of 10%-15%:
  - For projects in the data storage & processing domain (specifically, Hadoop, HDFS, Yarn, Tajo, MapReduce, and Cassandra), around 12% of issues is architectural.
  - For projects in the web development domain (specifically, Solr, JSPWiki, CloudStack, Brooklyn, and TomEE), around 11% of issues are architectural.
  - Across all six identified domains, around 14.5% of issues are architectural.
- The fractions of existence, executive, and property decisions are small and seem to differ more substantially per domain.
- The small fraction of architectural issues show that search tools are needed for finding architectural issues efficiently.

6.2 RQ2: Evaluate deep learning as a search tool for architectural issues

In this section, we will be discussing the results we obtained from acquiring issues from keywords searches and deep learning classifiers, and then labelling those issues.

6.2.1 Re-evaluating Existing Approaches

One of the main results in this section is an evaluation of deep learning as a search approach. For this evaluation, we will be comparing with other known search approaches (source code analysis, Maven POM file analysis, and keyword search). However, to make this comparison fair, we also re-evaluated the precision@k for those approaches using the relabelled dataset. The results of doing this can be found in Figure 24. Additionally, we also evaluated the precision@k for every type of keyword search separately. These results can be found in Figure 25. Compared to the original results for source code analysis ([28]), keyword search ([28]), and Maven POM file analysis ([25]), we make the following observations:

- The precision@k has become worse for existence issues for all approaches. We attribute this to the fact that we relabelled many existence issues, and some were classified to totally different types (e.g. just property, or non-architectural).
- The precision@k for finding property issues for static source code analysis has slightly improved. We attribute this to the fact that many issues were relabelled from "existence" to "existence/property".
- The precision@k for finding property issues with the search engine has significantly improved. We also attribute this to the fact that many "existence" issues were relabelled to either "property" or "existence/property". We can also see this for all four individual searches.
- For keyword searches and static source code analysis, the precision@k for finding executive issues has slightly improved. This is because a number of dependency upgrade issues were relabelled from "non-architectural" to "executive".
- When considering the average precision@k for the approaches to identify architectural issues, little has changed. The only change is that the precision@k seems to actually decrease faster than before for the keyword search approach.

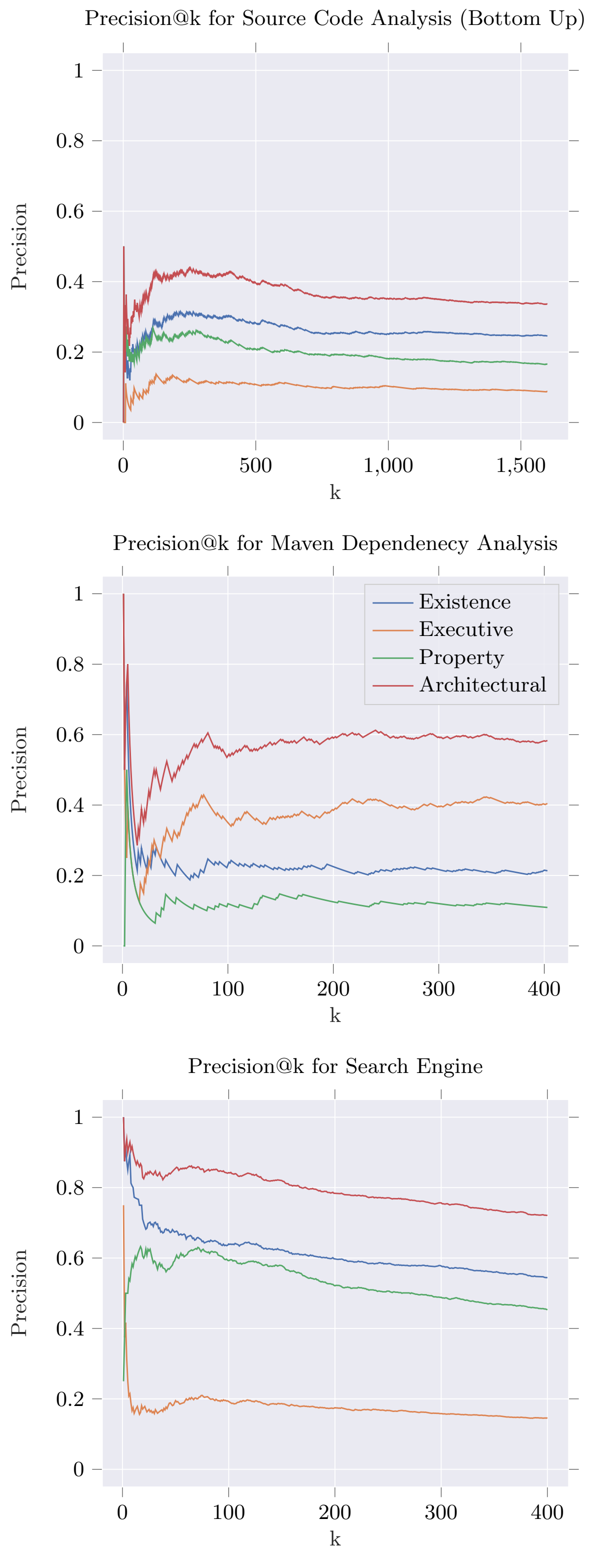


Fig. 24. Precision@k for other approaches for finding architectural design decisions, where precision@k was computed based on the relabelling done in this research. Top: source code analysis. Middle: Maven POM file analysis. Bottom: Keyword search. Note that for the keyword search, the precision@k is computed as the average precision@k for the four separately performed keyword searches. The results for each separate search can be found in Figure 25.

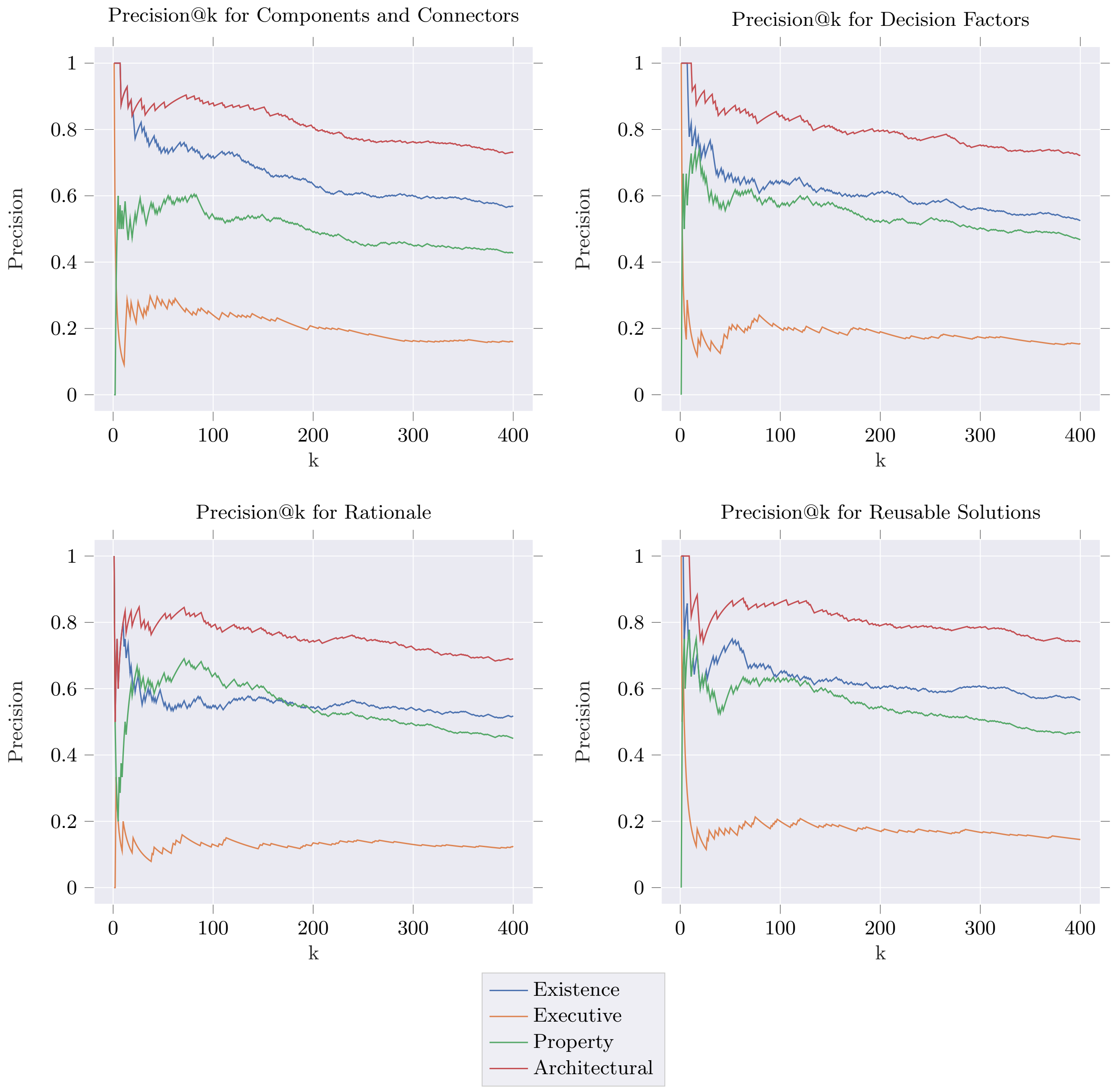


Fig. 25. Precision@k for the different keyword searches performed in [28], where the precision@k was computed based on the relabelling done in this research.

6.2.2 New Keyword Search

First, we tried to expand our dataset using keyword searches. We computed the precision@k for both lists of issues we labelled. The precision@k for the reusable solutions can be found in Figure 26. The precision@k for the decision factors can be found in Figure 27.

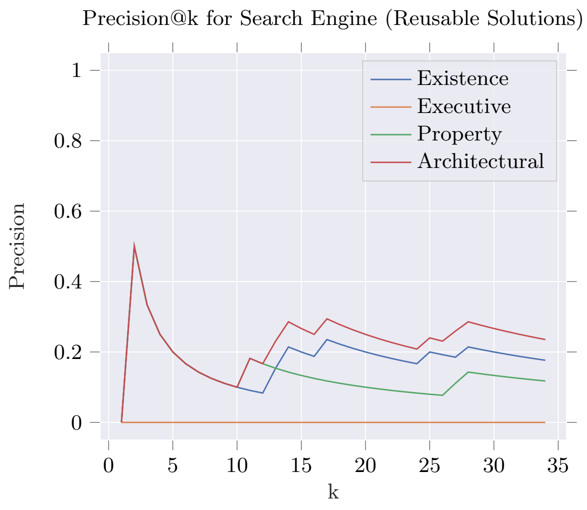


Fig. 26. Precision of the k issues with the highest search scores using the reusable solutions keywords.

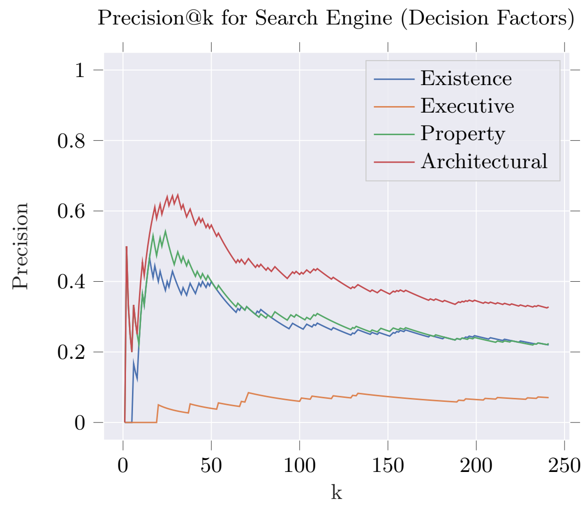


Fig. 27. Precision of the k issues with the highest search scores using the decision factors keywords.

We started with keyword search because we were aware that the Maven dependencies analysis method was exhausted, i.e. it could not find more issues. However, this also seems to be the case for the keyword searches to some extent. Previously, keyword searches achieved a precision@400 of 45.3% for finding property issues. This precision was obtained using the issues with the highest search scores. Now that the issues with a high search score were already labelled, we had to label issues with lower search scores. This probably resulted in the low precision for keyword searches in this research. With the reusable solutions keywords, we obtained a precision@34 of 11.8% for finding property issues and with the decision factors keywords, the precision@241 was 22.4%. Although this is more than twice as efficient as analysing issues from random samples, it is still infeasible for finding a lot of property issues. Given that our goal was to obtain around 500 additional property issues, we would have to label around 2500 issues with that precision.

6.2.3 Deep Learning for Finding Property Issues

This low precision was the direct motivation to start experimenting with deep learning, and in particular BERT, for finding architectural issues, leading to **RQ2**. The first round of BERT issues was done in two batches. First, the top-121 issues were classified. Next, the bottom 50 issues were classified. The resulting precision@k for the top 121 and bottom 50 issues are given in Figure 28.

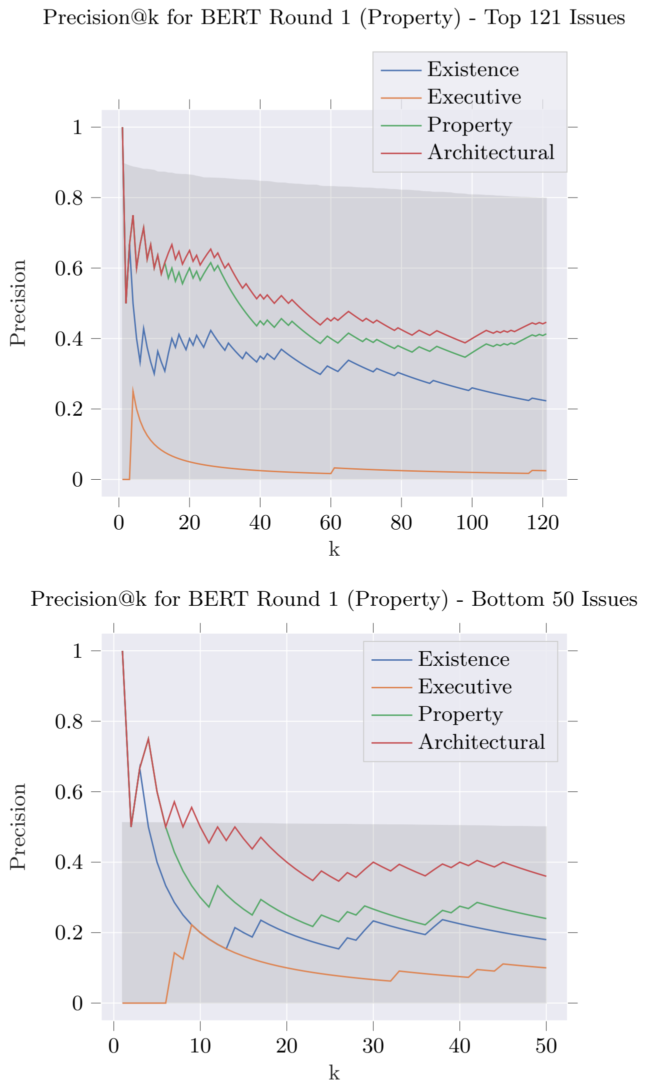


Fig. 28. (Top) Precision of the k issues with the highest property confidence. (Bottom) Precision of the k issues with the lowest property confidence above 0.5. The legend is valid for both plots. The confidences are from the first version of the BERT model, which was used for finding property issues. This BERT model did not use any text preprocessing.

Looking at Figure 28 (top), we see that the precision@121 of 41.3% on the top 121 issues is already almost two times higher than what we achieved with the new keyword searches. In other words, with this version of BERT, we already reduced the required effort for finding property issues in half. The precision@50 for the 50 issues with the lowest confidence scores was only 24%; this is worse than the top 121 issues, after labelling even fewer issues. However, this is still higher than the new keyword searches. Most importantly, it shows that the confidence of the classifier is a good indication for the likelihood of an issue to contain a property decision. This test therefore confirms the intuitive idea that, to achieve the highest precision in finding architectural issues, one has to select the issues with the highest confidence scores.

Despite the much higher precision of 41.3% compared to keyword searches, we decided to retrain BERT and test whether this showed any improvement in terms of precision. In this second round of finding property issues using BERT, we classified the top 600 issues with the highest confidence for the property class. The precision@k for the list of labelled issues is given in Figure 29.

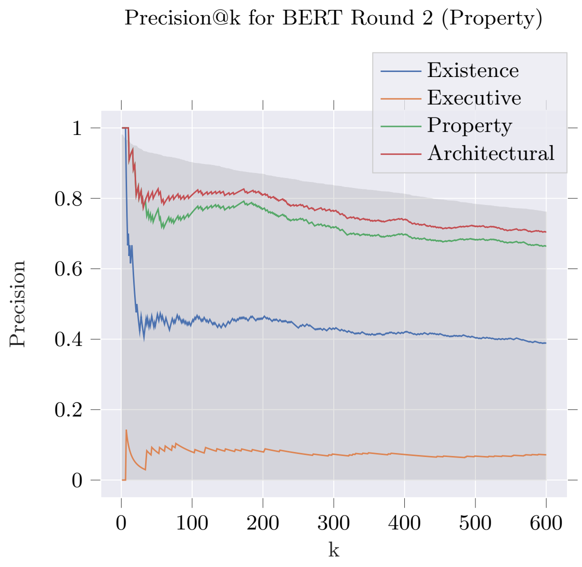


Fig. 29. Precision of the k issues with the highest property confidence. The confidences are from the second version of the BERT model, which was used for finding property issues. This BERT model did not use any text preprocessing.

From the 600 issues we labelled, 399 of them were in fact property issues. This leads to a precision@600 of 66.5% for this second version of BERT. This makes deep learning classifiers almost three times as effective as keyword searches in terms of precision, and more than 11 times as effective as using random sampling. Using the first version of BERT, we extended the dataset with 62 newly found property issues, and with the second version of BERT we found 399 new property issues. Initially, the dataset contained only 265 property issues, so BERT allowed us to more than double this amount.

Figure 30 shows the precision for finding property issues using different methods. We included the results of the second version of BERT, because this was the most recent version of BERT used for finding property issues. Clearly, BERT outperforms any other method on this task. Keyword searches showed decent performance as well, but static SC analysis and Maven dependencies analysis show poor performance. Therefore, for efficiently finding property issues, it seems that BERT is the best options.

6.2.4 Deep Learning for Finding Executive Issues

Given the high precision for finding property issues, we used this approach for finding executive issues as well. To filter out issues about simple dependency upgrades, we decided to label all 202 issues with a confidence > 0.5 for both the executive and existence classes. We hypothesised that issues solely about upgrading dependencies are assigned a low confidence for existence, meaning we can filter those out by setting a relatively high threshold for the existence confidence. The precision@k for this list of labelled issues is given in Figure 31.

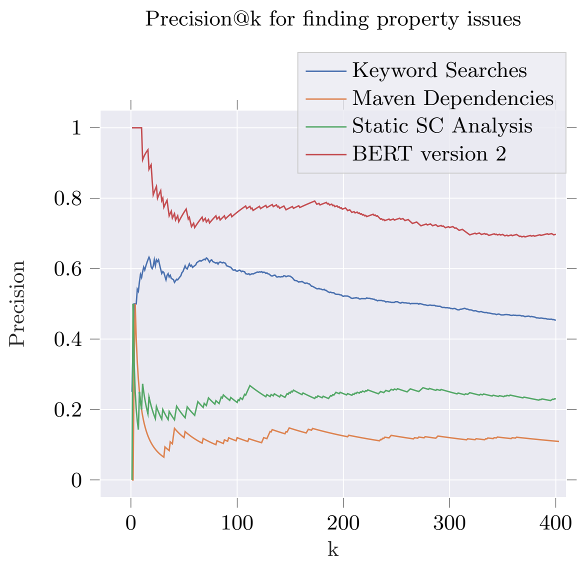


Fig. 30. Precision for finding *property* issues for different search methods.

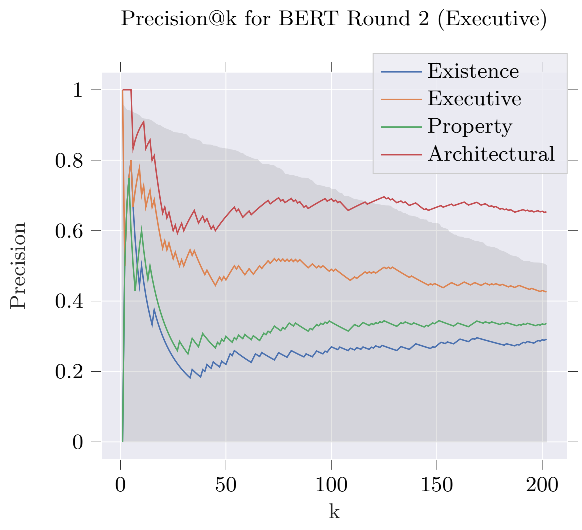


Fig. 31. Precision of the k issues with the highest executive confidence and for which the confidence for both executive and existence were above 0.5. The confidences are from the second version of the BERT model, which was used for finding executive issues. This BERT model did not use any text preprocessing.

Previously, the Maven dependencies analysis was the only method that obtained a decent precision for finding executive issues. The Maven dependencies analysis had a 0.4 precision@400, whereas keyword searches and source code analysis were below 0.2 precision for all k. However, the Maven dependencies analysis was exhausted, meaning that all Maven dependency changes were already analysed previously. At the moment, a deep learning classifier is therefore the only possibly effective method for finding executive issues. Using the second version of BERT, we were able to find executive issues with a precision@202 of 42.6%. This shows that the deep learning classifier is as efficient as the Maven dependencies analysis, while also being able to find more issues than that method. Furthermore, the filter we invented was found to be effective for excluding straight forward executive issues about technology version bumps. This filter works by setting a threshold for the existence class, confirming our theory that these straight forward executive issues are given a low existence confidence score by the classifier.

Because deep learning also seemed to be effective for finding executive issues, we decided to retrain the BERT model using the newly acquired data. Using this third version of BERT, we classified all 200 issues with a confidence > 0.5 for the executive class and a confidence > 0.34 for the existence class. As described in our study design, we used a threshold of 0.34 in order to find 200 issues. The precision@k for this labelled list of issues is given in Figure 32.

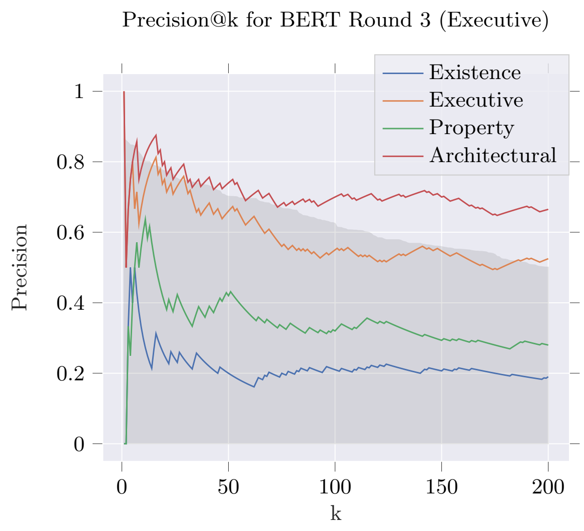


Fig. 32. Precision of the k issues with the highest executive confidence and for which the existence confidence was above 0.34 and the executive confidence was above 0.5. The confidences are from the third version of the BERT model, which was used for finding executive decisions. This BERT model did not use any text preprocessing.

Despite the lowered confidence threshold for the existence class, this search had the desired effect of filtering out simple version bumps. Additionally, the precision@200 increased by 10% to 52.5% for finding executive issues. This iterative approach, of labelling issues and retraining the model, seems to be beneficial for finding additional executive issues. For both the property and executive issues, we also saw that this iterative process increased the precision for finding the respective issues. This is a benefit compared to the other methods, because the other methods can easily get exhausted without having a solution to this problem. For example, the keyword searches are limited to a fixed set of keywords. First, composing this set of keywords involves domain knowledge and effort. Second, when searching with this set of keywords gets exhausted, one would have to spend additional effort in updating these sets of keywords to make keyword searches effective again, although success is not guaranteed. Deep learning classifiers on the other hand can simply be retrained, after which they should have better search performance.

To further expand the dataset with executive issues, we classified 100 issues in order of highest confidence for the executive class, but excluding the issues containing keywords such as "upgrade" in the summary. The corresponding precision@k is given in Figure 33.

We found that the keyword filter we used was incomplete and thus ineffective, leading to many issues about simple versions bumps in this sample. Nonetheless, the precision@100 was 48.5%, which was still efficient. With these three rounds of labelling for executive issues, we found 86 executive in the first round, 105 in the second round, and 49 issues in the last round. This nearly doubled the amount of executive issues in our dataset from 295 to 535 executive issues. Given that the Maven dependencies analysis method was exhausted and the precision of the other existing methods was below 0.2, deep learning was more than two times as effective as existing search methods and more than 11 times as effective as finding executive issues using random sampling.

Figure 34 shows the precision of different methods for finding executive issues. We included the most recent version of BERT we used for finding executive issues, which was version 3. Again, BERT clearly outperforms any other method. Maven dependencies analysis is also a good option for this task, but recall that all issues found by this method were already annotated, meaning this method is exhausted. Static SC analysis and keyword searches show poor performance for this task. Similar as for property issues, BERT seems to be the best choice for finding executive issues efficiently.

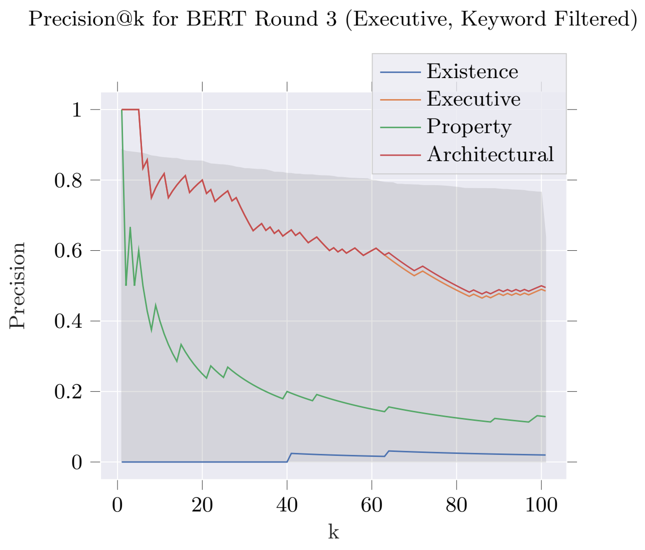


Fig. 33. Precision of the k issues with the highest executive confidence. These issues were filtered based on a set of keywords in order to filter out issues about simple version bumps of technologies. The confidences are from the third version of the BERT model, which was used for finding executive decisions. This BERT model did not use any text preprocessing.

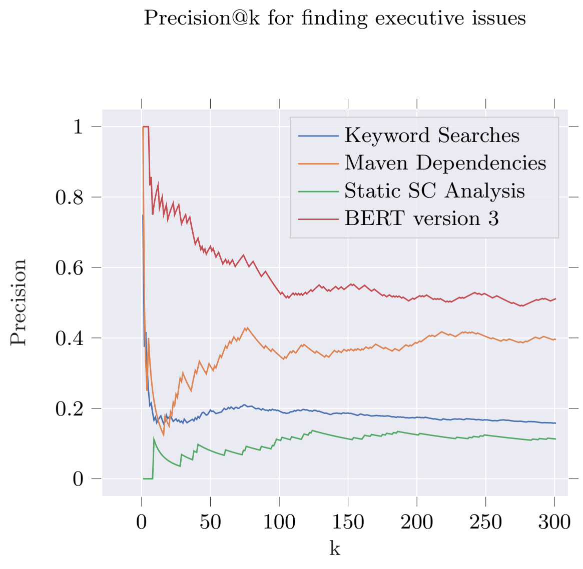


Fig. 34. Precision for finding *executive* issues for different search methods.

6.2.5 Summary of the Search Rounds

In total, BERT helped us collect 809 architectural issues, from which 368 were existence, 291 executive, and 510 property. Because the classifier is not entirely accurate, we also collected 464 non-architectural issues.

In general, it seems that deep learning classifiers are able to find architectural issues with a higher precision than existing methods, such as keyword search, Maven analysis, and source code analysis. Besides, the Maven analysis was completely exhausted, and the keyword search also seems to be exhausted to a large extent when looking at the decreased precision. On the other hand, deep learning classifiers were able to still find many architectural issues with a high precision.

RQ2 Takeaways:

- Deep learning achieves better precision@k for finding property and executive issues than static source code analysis, Maven dependency analysis, and keyword searches.
- Deep learning has better precision@k for finding architectural issues in general than Maven dependency analysis and static source code analysis. Keyword searches seem to be as effective as deep learning, until the point where it is almost exhausted. Deep learning also suffers from exhaustion, but at a slower rate than keyword searches.
- Deep learning is able to find architectural issues successfully when keyword searches and Maven dependency analysis have already been exhausted.

6.3 Evaluating Classifier Performance

In this section, we introduce some concepts which are necessary to fully understand how we evaluated our classifiers. Specifically, we introduce a few variations on the regular macro scores, we introduce a concept called ‘classification as detection’, and we compare the performances of the classifiers with a *best-guessing* classifier.

6.3.1 Performance Metrics

In Section 3.3.5.2, we outlined several metrics for the evaluation of machine learning classifiers. In particular, for multi-class problems, we outlined macro metrics: these metrics are computed as the average of some metric over all classes.

For the regular macro precision, recall, and F_1-score scores, we decided to include the performance of the non-architectural class. This is because in practice this is an important class to get right, also for classification and not only for detection. In fact, often it might be even more important to get this class right compared to the others; depending on the use cases, it may be more important for the classifier to distinguish architectural from non-architectural, than it is for the classifier to correctly classify the exact types of architectural design decisions in architectural issues.

However, for completeness, we decided to also include a macro score for the three architectural classes only, which we call *positive macro*.

Furthermore, we included *weighted macro scores.* Instead of taking the average of the four class performances, we use a weighted average. These weights are obtained from the random samples and are used to create a performance score that should better represent the performance of the classifier in a practical setting, because the scores are weighted according to how often a certain type of issue occurs in practice in an issue tracking system. The exact weights we used depends on the dataset we use for evaluation.

For **RQ3**, we evaluate the classifier on our expanded dataset containing issues from the data storage & processing domain, and we evaluate the classifiers on the random sample of that domain. Therefore, for both evaluations, we will be using the weights from the random sample of the data storage & processing domain. For **RQ6**, we evaluated the classifier on two random samples. For both random samples, we used the weights obtained from the fully random sample from the six software domains. Table 15 contains the weights for each of the samples. Note that the table is to give an overview of the weights, for our calculations we used more precise weights.

Domain	Class	Count	Weight
Data storage & processing	Existence	35	0.082
	Executive	14	0.033
	Property	24	0.056
	Non-Architectural	352	0.828
All six domains	Existence	24	0.057
	Executive	21	0.050
	Property	32	0.077
	Non-Architectural	341	0.816

Table 15. Weights that are used to calculate the weighted macro scores. The weights we use depend on the domain(s) to which the test set belongs, on which we evaluated a classifier.

Additionally, we introduced *classification as detection.* This metric interprets the output of a multi-label classifier as if it was a detection model. This means that if at least one of the confidences of the existence, property or executive class is greater than 0.5, we interpret the output as *architectural.* If all confidences are below 0.5, we interpret the output as *non-architectural.* This makes it possible to make a comparison with the detection models from our internship [18]. Unless we specifically mention that the results are from the internship models, we use *detection* to refer to *classification as detection.*

6.3.2 Best-guessing Classifier Performance

In the coming sections, we will be evaluating classifiers on different datasets. Sometimes the performance of a classifier may seem better than it is. For example, 66% of the issues were architectural in our initial dataset [18]. It is possible for a classifier to obtain a precision of 0.66 and a recall of 1.0 on that dataset, by predicting all issues as architectural. While the performance of the classifier on paper may seem to be decent, it is practically useless.

To counteract this problem, we will be comparing the performances of our classifiers with a so-called *best-guessing* classifier. The performance of this classifier is the maximal performance achievable, without looking at the issues themselves. Therefore, for each class, this best-guessing classifier will either always output false or it will always output true.

If the classifier never predicts class X, then the precision for class X will be 0. On the contrary, if the classifier predicts all issues to be of class X, then the precision will be the fraction of issues in the dataset that belongs to class X[27]. The recall for class X is either 0 if the classifier never outputs that class, or 1 if the classifier always outputs that class. Hence, in general, it is beneficial for the best guessing classifier to output true for as many classes as possible in order to maximise (macro) F_1 score.

As an example, we will demonstrate how we calculated the performance of the best-guessing classifier for the existence class in our expanded dataset. From the 4713 labelled issues, we have 1282 existence issues. The fraction of existence issues is therefore 0.272. This means that if one would select a random issue from this dataset, there is a 0.272 chance that this issue will be existence. A best-guessing classifier therefore has a precision of 0.272, because there is a 0.272 chance of correctly guessing that an issue is existence. Recall measures how many of the existence issues are missed by the classifier. To maximise this for

[27]Technically, the fraction of issues in the set belonging to class X. However, we can assume equality due to the use of a stratified training/validation/test split.

the best-guessing classifier, it is obviously desired to never miss any existence issues. Hence, when the best-guessing classifier predicts all issues to be existence, it never 'misses' an existence issue (i.e. the recall is 1.000) and 27.2% of the issues will be predicted correctly as existence (i.e. the precision is 0.272), yielding an F_1 score of 0.428. Note that this best-guessing classifier is predicting all issues as existence, regardless of the content of the issue. This classifier is therefore entirely biased towards the existence class.

For the best-guessing performance for the macro score, we should decide whether to predict all issues as non-architectural or as existence and executive and property. In the case of the weighted macro scores, it is beneficial to predict all issues as non-architectural, due to the large weight for the non-architectural class. For the positive macro score, it is better to predict all issues as existence, executive and property. For the regular macro score, it depends on the dataset. If a large portion of the dataset is non-architectural, for example in the random samples, it is beneficial to predict all issues as non-architectural. Otherwise, it is better to predict all issues as existence, executive and property.

The best-guessing detection performance is similar to the class scores, but now we consider only the architectural and non-architectural class. In particular, since this is a binary classification problem, the only class of interest is the architectural class. Therefore, a best-guessing classifier should predict all issues as architectural for this specific task. The precision for detection will be the fraction of issues that is architectural, and the recall will be 1.

For each metric, we have provided the best-guessing scores in Table 16 for our expanded dataset. Similarly, for the random samples from the data storage & processing domain and all six domains can be found in Tables 17 and 18, respectively. Note that in the sections where we compare the scores of the classifiers with best-guessing, we are comparing the F_1 scores.

Metric	Counts	F_1	P	R
Existence	1282	0.428	0.272	1.000
Executive	683	0.253	0.145	1.000
Property	1272	0.425	0.425	1.000
Non-Architectural	2547	0.702	0.540	1.000
Macro	–	0.277	0.172	0.750
Positive macro	–	0.369	0.229	1.000
Weighted macro	–	0.581	0.447	0.828
Detection	2166	0.630	0.460	1.000

Table 16. Best-guessing performances for the expanded dataset, excluding the issues outside the data storage & processing domain. F_1 is the F_1 score, P is the precision, and R is the recall.

Metric	Counts	F_1	P	R
Existence	35	0.161	0.087	1.000
Executive	14	0.068	0.035	1.000
Property	24	0.113	0.060	1.000
Non-Architectural	352	0.936	0.880	1.000
Macro	–	0.234	0.220	0.750
Positive macro	–	0.114	0.061	1.000
Weighted macro	–	0.775	0.729	0.828
Detection	48	0.214	0.120	1.000

Table 17. Best-guessing performances for the random sample from the data storage & processing domain. F_1 is the F_1 score, P is the precision, and R is the recall.

Metric	Counts	F_1	P	R
Existence	24	0.109	0.058	1.000
Executive	21	0.095	0.050	1.000
Property	32	0.169	0.092	1.000
Non-Architectural	341	0.916	0.845	1.000
Macro	–	0.229	0.211	0.750
Positive macro	–	0.124	0.067	1.000
Weighted macro	–	0.747	0.689	0.828
Detection	59	0.268	0.155	1.000

Table 18. Best-guessing performances for the random sample from all six domains. F_1 is the F_1 score, P is the precision, and R is the recall.

6.4 RQ3: Evaluating Multi-Label Classifiers

6.4.1 Comparison of Classifiers

In this section, we will be discussing the performance of the multi-label classifiers we experimented with. We created overviews of the performances in Tables 19, 20. These two tables give the performance of the classifier on a test set obtained through a stratified split (since we always used the same test set, we will refer to this as "the fixed test set"). The first table gives the performance for multi-label classification, the second one for detection.

Additionally, we calculated the detection performance of our internship models [18] on the new dataset in Table 21.

We also included tables showing more detailed performance metrics. The detailed metrics for the best model (BERT) can be found in Table 24. The detailed metrics for the other models can be found in Appendix E in Tables 41-53.

Model	F_1	P	R
BERT	0.676	0.726	0.644
BERT†	0.671	0.673	0.674
BERT*	0.628	0.689	0.597
TF-IDF*	0.625	0.639	0.620
BOWN*	0.611	0.625	0.601
TF-IDF	0.609	0.603	0.620
BOWN	0.607	0.599	0.617
BOWF	0.600	0.622	0.585
RNN	0.590	0.610	0.584
DOC2VEC	0.585	0.649	0.550
BOWF*	0.581	0.631	0.548
RNN*	0.577	0.614	0.551
DOC2VEC*	0.568	0.632	0.537
CNN	0.562	0.555	0.569
CNN*	0.440	0.446	0.587

Table 19. Macro scores on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. Results with a * are obtained from models trained using fine-grained technology replacement. Results with a † are obtained without text preprocessing. F_1 is the macro F_1 score, P is the macro precision, and R is the macro recall.

Looking at Tables 19 and 20, we see that BERT is the best performing model. The performance is the best, both with and without text preprocessing. The best macro F_1 score of 0.676 is achieved with text preprocessing, and the best detection F_1 score of 0.753 is achieved without text preprocessing. The TF-IDF model with the fine-grained technology replacements (TF-IDF* from now) is the next best model, with a macro F_1 score of 0.625 and a detection F_1 score of 0.722. Although this TF-IDF* model is a rather simple and computationally cheap model, it still is able to achieve good performance. However, it is almost 0.05 behind BERT in terms of macro F_1 score and 0.03 behind BERT in terms of detection F_1 score. Therefore, unless BERT

Model	F_1	P	R
BERT†	0.753	0.713	0.799
BERT	0.751	0.756	0.745
TF-IDF*	0.722	0.694	0.752
BOWN*	0.718	0.699	0.738
TF-IDF	0.716	0.676	0.762
BOWF	0.712	0.733	0.692
BERT*	0.706	0.723	0.690
RNN	0.698	0.694	0.701
BOWF*	0.687	0.747	0.636
RNN*	0.680	0.713	0.650
BOWN	0.677	0.661	0.694
DOC2VEC*	0.667	0.768	0.589
DOC2VEC	0.652	0.788	0.556
CNN*	0.638	0.497	0.888
CNN	0.635	0.672	0.603

Table 20. Detection scores on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. Results with a * are obtained from models trained using fine-grained technology replacement. Results with a † are obtained without text preprocessing. F_1 is the detection F_1 score, P is the detection precision, and R is the detection recall.

Model	F_1	P	R
Voting Detection	0.736	0.716	0.758
Stacking Detection	0.728	0.688	0.773
RNN Detection	0.726	0.719	0.733
CNN Detection	0.695	0.686	0.705
BOWF Detection	0.679	0.633	0.733
DOC2VEC Detection	0.676	0.711	0.645
Combination Detection	0.673	0.682	0.664
TF-IDF Detection	0.667	0.632	0.705
BOWN Detection	0.662	0.624	0.705

Table 21. Detection scores of the models from the internship [18] on the fixed test set from the new dataset, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the detection F_1 score, P is the detection precision, and R is the detection recall.

is infeasible due to hardware constraints, it seems that BERT should be preferred over TF-IDF*.

Term frequency models show good performance all over the board. TF-IDF(*), BOWF and BOWN(*) models achieve macro F_1 scores between 0.600 and 0.625, making them all better than CNN, RNN and DOC2VEC models. RNN is only slightly behind BOWN with a macro F_1 score of 0.590, and DOC2VEC is slightly behind that with a macro F_1 score of 0.585. CNN was not performing well during this test, with a macro F_1 score of 0.562. Similarly, for detection we also see good performance of the TF-IDF(*), BOWN* and BOWF models, with F_1 scores ranging between 0.722 and 0.712. This is again closely followed by RNN, with a detection F_1 score of 0.698. The gap to DOC2VEC* is larger for detection, because that model only achieves a detection F_1 score of 0.667. CNN* showed poor performance with only a 0.638 detection F_1 score.

We think BERT is performing this good simply because it is a very powerful state-of-the-art model, with lots of parameters, and which has been trained on a lot of data. Especially the latter factor explains why it is more capable to deal with complicated texts than other models.

We suspect that bag of words related models (BOWF(*), BOWN(*), TF-IDF(*)) are performing good because they are closely related to keyword searches; we think that the models may be too attuned to keywords used with the search engine, or other common keywords in general. For instance, "performance" and "security" are common indicators for property issues, and "update" and "upgrade" are common indicators for executive issues.

We attempted to verify this hypothesis. We did this by evaluating the models again, but with different training and test sets. Specifically, we used the issues we randomly sampled from Hadoop, Yarn, Tajo, HDFS, MapReduce, and Cassandra as the test set; the remaining issues from those six projects were used to obtain the training and validation sets (with a stratified 90%/10% split). In this analysis, we saw that the performance of bag of words related models dropped more substantially than those for BERT and RNN; these results can be found in Tables 22 and 23 (with more elaborate results per model in Appendix G). However, these results should be taken with a bit of scepticism. We also performed the experiments for **RQ4** (i.e. testing with different amount of training data) using this set (i.e. the random sample from the six data storage & processing projects) as test data, and the results of those test runs (Figures 46 and 47 in Appendix F) were so inconsistent that we believe we should not draw any strong conclusions from Tables 22 and 23. We believe these scores are inconsistent because there are very few architectural issues in a random sample. We discuss this, alongside the drop in performance for all models, in more detail in Section 6.7. In particular, we did not use these results to see what models performed the best for multi-label classification.

Now we move our attention back to Table 19. For RNN, we expect that this model may not be performing particularly bad, but the other models particularly good. Nevertheless, the performance is rather low compared to our earlier work in [18]. We do not have a single clear idea why this could be the case, but we think it could be related to more unclear issues that have made it harder for the classifiers to achieve good performance on the new dataset.

For Doc2Vec, we expect that a similar thing as for RNN is going on. We expect the same for CNN. However, for the CNN model, we have the additional suspicion that the particular CNN architecture we used may be less suited for multi-label classification. The CNN used assigns a score to n-grams (in particular, 49-grams and 8-grams in our case), and for each n, only 8 scores (number of filters) are passed through from the max-pooling layer. This means that only 16 outputs are fed into the output layer. In particular, for multi-class classification, this may simply be too little information. Intuitively, for tasks with only a single output, such as detection and multi-class classification, every kernel in the convolutional layers can be special-tailored towards a particular class; this is not the case for multi-label classification. Hence, intuitively, it would not be surprising if this particular architecture would perform more poorly for multi-label classification

In Table 21, we can see the performance of the detection models we experimented with previously in [18], but trained and evaluated on our new dataset. We tested with a selection of the best models, including a voting ensemble and stacking ensemble model, both based on the BOWF, CNN, and RNN models from [18]. One thing that stands out is that BERT achieves higher performance than the internship models that were specifically trained for the detection task. For most multi-label models, it is the case that they obtain similar performance to the internship models. Although we have not trained BERT specifically for the detection task, it seems that there is no loss in detection performance when training multi-label models compared to detection models. This makes multi-label models particularly attractive, due to the added benefit of being able to predict the architectural subtypes, while at the same time achieving good detection performance.

The old models show lower performance compared to the work we did in [18]. We suspect this can mainly be attributed to the fact that our new dataset is more balanced; it is no longer possible for classifiers to obtain good performance by assigning (nearly) all samples to a single class.

Model	F_1	P	R
RNN	0.613	0.574	0.695
BERT	0.613	0.623	0.607
BERT†	0.549	0.575	0.572
RNN*	0.547	0.511	0.620
BERT*	0.546	0.554	0.549
BOWF*	0.534	0.544	0.541
BOWF	0.532	0.589	0.497
TF-IDF	0.522	0.489	0.589
BOWN	0.492	0.438	0.685
BOWN*	0.490	0.487	0.498
DOC2VEC	0.484	0.539	0.448
TF-IDF*	0.482	0.447	0.554
DOC2VEC*	0.479	0.544	0.439
CNN	0.085	0.046	0.750
CNN*	0.085	0.046	0.750

Table 22. Macro scores on the random sample of the data storage & processing domain. Results with a * are obtained from models trained using fine-grained technology replacement. Results with a † are obtained without text preprocessing. F_1 is the macro F_1 score, P is the macro precision, and R is the macro recall.

Model	F_1	P	R
RNN	0.633	0.528	0.792
BERT	0.598	0.592	0.607
BERT*	0.566	0.517	0.625
BOWF*	0.547	0.500	0.604
BOWF	0.538	0.556	0.521
DOC2VEC	0.535	0.605	0.479
BERT†	0.534	0.456	0.646
RNN*	0.533	0.414	0.750
DOC2VEC*	0.465	0.526	0.417
TF-IDF	0.463	0.360	0.646
BOWN*	0.436	0.387	0.500
BOWN	0.430	0.309	0.708
TF-IDF*	0.423	0.326	0.604
CNN	0.214	0.120	1.000
CNN*	0.214	0.120	1.000

Table 23. Detection scores on the random sample of the data storage & processing domain. Results with a * are obtained from models trained using fine-grained technology replacement. Results with a † are obtained without text preprocessing. F_1 is the detection F_1 score, P is the detection precision, and R is the detection recall.

6.4.2 Detailed Performance for BERT

Because BERT was the best model in most of our tests, we will specifically focus on this model for the more detailed performance metrics (see Table 24).

First, we consider the performance of BERT if we were to use the classification model as a detection model. We see that BERT has a F_1 score of 0.753 for detecting architectural issues. More specifically, BERT is able to catch 80.4% of the architectural issues and 70.8% of the predicted architectural issues were in fact architectural. These results seem to be worse than the 0.823 F_1 score from [18]. However, as explained, it seems that the task has also become harder. The dataset from [18] consisted of 66% architectural issues. This makes the 0.823 F_1 score only a 4% improvement over a best-guessing classifier that predicts all issues as architectural (this 'best-guessing' classifier has a 0.66 precision with 1.00 recall, and a F_1 score of 0.795). For our new dataset that we used in this test, only 46% of the issues were architectural. This makes it so that the improvement over a best-guessing classifier predicting all issues as architectural is 19% for our new classifier.

The class specific scores for BERT in Table 24 show even larger improvements over a best-guessing classifier. In terms of raw scores, existence and executive seems to be rather balanced,

Set	Metric	F_1	P	R	Impr. over best-guessing
Train	Detection	0.982	0.975	0.989	+56%
	Macro	0.960	0.951	0.970	+247%
	Positive macro	0.952	0.939	0.966	+158%
	Weighted macro	0.979	0.980	0.979	+69%
	Existence	0.946	0.921	0.972	+121%
	Executive	0.950	0.924	0.978	+276%
	Property	0.959	0.971	0.948	+126%
	Non-Architectural	0.985	0.988	0.982	+40%
Val	Detection	0.804	0.763	0.850	+28%
	Macro	0.709	0.713	0.709	+156%
	Positive macro	0.674	0.670	0.681	+83%
	Weighted macro	0.790	0.809	0.774	+36%
	Existence	0.657	0.614	0.706	+53%
	Executive	0.700	0.690	0.710	+177%
	Property	0.664	0.707	0.626	+56%
	Non-Architectural	0.816	0.840	0.793	+16%
Test	Detection	0.753	0.708	0.804	+19%
	Macro	0.671	0.673	0.674	+142%
	Positive macro	0.638	0.625	0.655	+73%
	Weighted macro	0.748	0.782	0.719	+29%
	Existence	0.651	0.614	0.691	+52%
	Executive	0.653	0.610	0.701	+158%
	Property	0.609	0.651	0.573	+43%
	Non-Architectural	0.771	0.815	0.733	+10%

Table 24. BERT† performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. These results were obtained without text preprocessing. F_1 is the F_1 score, P is the precision, and R is the recall.

with a recall around 0.69-0.70 and a precision of around 0.61. The performance improvement over the best-guessing classifier is 52% for existence, whereas it is 158% for executive. This is because we have 1282 existence issues in the dataset, compared to only 683 executive issues. This makes the precision for a best-guessing classifier much lower for the executive class. Comparing the precisions and recalls with the property class scores, we see that the precision is higher for property (0.651), but the recall is lower (0.573). We believe the precision could be explained by the fact that property decisions can be obvious due to keywords such as 'performance improvement' or 'security improvement'. However, a lack of such keywords might make it too hard for the classifier to detect the property decision.

The performance improvement of 10% over the best-guessing classifier for the non-architectural class is much lower compared to the other classes. This is mainly due to the fact that a best-guessing classifier obtains a high precision of 0.54, because 54% of the dataset is non-architectural. Nonetheless, we see good performance by the classifier, with the classifier catching 73.3% of the non-architectural issues with a precision of 0.815.

We also included a confusion matrix on the test set in Figure 35. In this matrix, we see that most mistakes are detection mistakes, meaning that architectural issues are predicted as non-architectural and vice versa. Looking at the other mistakes, we see that many existence and/or property issues are predicted incorrectly. It might be that existence and property decisions are the hardest to predict correctly. Another reason for this might be a problem with the labelling of the issues. Although we have tried to mitigate this problem by relabelling issues, it might be the case that there are numerous issues with a wrong label.

6.4.3 Evaluating Proportions of Predictions

After completing the research in [18], we wanted to predict issues with the trained classifiers. However, we found that some classi-

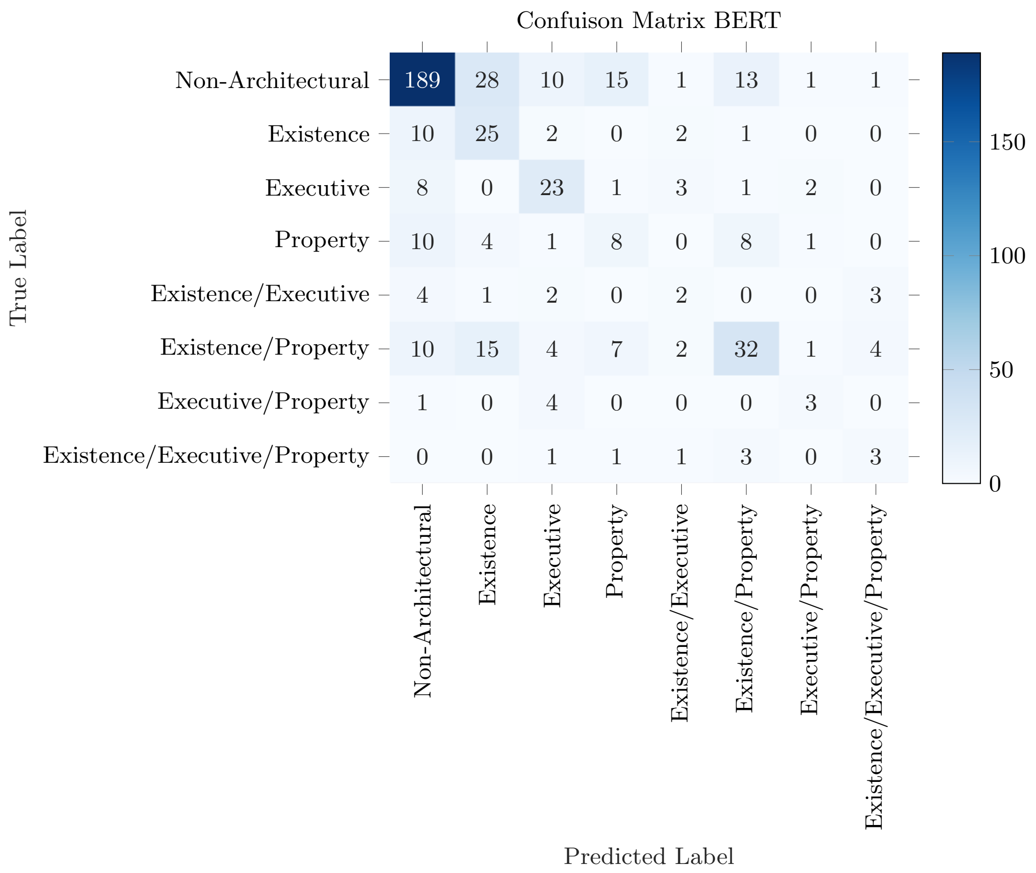


Fig. 35. Confusion matrix for BERT† on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. These results were obtained without text preprocessing. On the y-axis we see the true labels and on the x-axis we see the predicted labels.

fiers predicted 60-70% of the issues as architectural. This made us decide to perform a practical sanity check to verify whether classifiers predict 'realistic' amounts of architectural issues. For brevity, we included only three multi-label models: BERT, RNN, and TF-IDF*. This is because these were the best performing multi-label models in their respective categories (BERT, semantic embedding based, bag of words based). Although we decided to not evaluate the performance of multi-class classifiers, we did decide to include the performance of such classifiers here, in order to demonstrate that this is a necessary sanity check, and to show how other types of models compare to them.

With the random sample from the Apache projects Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn, we found that around 12% of the issues are architectural. When we compute the predictions of a classifier on all issues from these projects, we therefore also expect around 12% of issues to be predicted as architectural.

Table 25 shows all the predicted and expected proportions. Because we excluded the issues that were in the dataset (because the classifiers would trivially have good performance on training data), we have to compensate for that. Our dataset contains 2166 architectural issues from the above-mentioned Apache projects, while the dataset only consists of 4713 labelled issues. Assuming that around 12% of the 71093 issues from these projects are architectural, we excluded a fourth of the total number of architectural issues by excluding the labelled issues. If we then compensate for that, $\frac{8531-2166}{71093-4713} = 9.59\%$, we find that 9.59% of the remaining issues should be predicted as architectural. For the architectural subclasses, we performed similar calculations.

The results of this sanity check show that this check is indeed necessary. Some of the models from [18] predict almost half of the issues as architectural. Although they are not the 60-70% we found previously, they still are unrealistically high numbers. Furthermore, a classifier predicting half of the issues as architectural is not useful in practice, because the maximum possible precision of such a classifier would be around 20%.

From this test, it seems that RNN and BERT are the best models in this regard. They are off by less than 2% for architectural issues, which is a little on the high side. CNN is also close to the expected amount, but CNN is only a detection model. In general, it seems that detection models perform better in this test than the multi-class classification models. Multi-label classification seems to be better than multi-class as well.

When looking at the class-specific scores of BERT and RNN, it seems that both models are not too far off; they are all within 2.5% of the expected proportions. BERT is especially close for the existence class. BERT is also somewhat close for the executive class, although too many issues are predicted as executive. This might be because the classifier is too focused on technology names and version numbers, while issues containing them might not always be executive. Many property issues are missed by BERT (without text preprocessing), while RNN and BERT (with text preprocessing) are marking too many issues as property. This is also in line with earlier observations that BERT (with text preprocessing) and RNN have high recall for property, while BERT (without text preprocessing) has low recall for property. We are not sure what the reason for this behaviour can be, so conclusive answers would require additional research.

Another part of the solution seems to be our new expanded dataset. Models trained on the new dataset consistently outperform the same models trained on the dataset from [18] (in terms of predicted proportions). This is most likely a result of the more balanced dataset: 46% of the issues are architectural in the new dataset, whereas this was 66% in the old dataset. In the old dataset, it was more rewarding to predict many issues as architectural compared to the new dataset. Additionally, deep

	Architectural	Existence	Executive	Property
Expected proportions data storage & processing	9.59%	7.44%	2.72%	4.51%
RNN Multi-Label Classification	10.6%	5.0%	1.8%	7.7%
BERT† Multi-Label Classification	11.3%	7.1%	3.4%	2.9%
BERT Multi-Label Classification	12.0%	7.0%	3.0%	7.8%
CNN Detection new dataset	12.2%	–	–	–
BOWF Detection new dataset	19.3%	–	–	–
TF-IDF* Multi-Label Classification	21.5%	6.4%	12.1%	7.1%
CNN Detection	25.1%	–	–	–
RNN Multi-Class Classification new dataset	25.5%	9.4%	7.7%	8.5%
BOWF Detection	33.5%	–	–	–
RNN Multi-Class Classification	38.6%	11.8%	6.3%	20.5%
BOWF Multi-Class Classification new dataset	48.2%	17.0%	21.0%	10.1%
BOWF Multi-Class Classification	49.3%	17.1%	22.0%	10.2%

Table 25. Predicted and expected proportions of architectural issues and their subclasses. Note that we included multi-class classifiers to demonstrate the usefulness of this sanity check. Results with a † are obtained without text preprocessing.

learning models tend to perform better when trained with more data, which might be part of the solution as well.

RQ3 Takeaways:

- BERT (with preprocessing) is the top-performing model.
    - BERT (all variants) have better performance than our other semantic embedding or bag of words related models.
    - Both BERT and RNN make predictions with realistic proportions.
    - BERT, as a multi-label classifier, is better in detecting architectural issues than classifiers that are specifically trained for detecting architectural issues.
- Expanding the dataset and using multi-label classification models results in models making predictions closer to expected real world proportions.

6.5 RQ4: Evaluating Need for Additional Data

In [18], we found that we could improve the performance of the classifiers by training them with more data. Now that we have expanded the dataset, we evaluated whether this is still the case.

Multi-label classification with the BERT† model seems to be rather saturated (see Figure 36). The trend line becomes quite flat by training the classifier with more data. This seems to indicate that the performance cannot be improved much by labelling more issues.

We also created another graph for BERT†, which interprets the classification output as if it were a detection task (see Figure 37. Looking at the trend line in that graph, we do see a somewhat steeper line in the end than we saw for multi-label classification. This means that, if we would increase the dataset size, we could gain some performance benefit. However, due to the small slope of the trend line, we expect this would take a lot of effort.

It seems that we can gain small improvements in terms of the detection performance of BERT†, by training the classifier with more issues. However, this would require substantial labelling effort, which given the performance gains, might not be worth it. Also, when testing the classifier on a random sample, there seems to at most a little improvement possible.

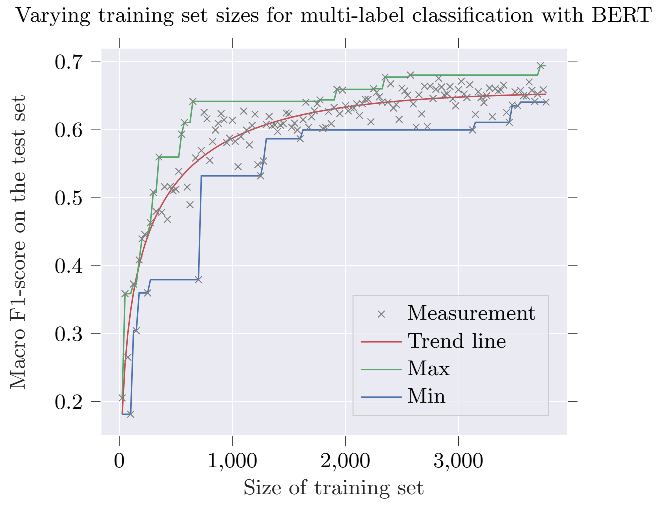


Fig. 36. Varying training set sizes for multi-label classification with BERT†, with a fixed test set. These results were obtained without text preprocessing.

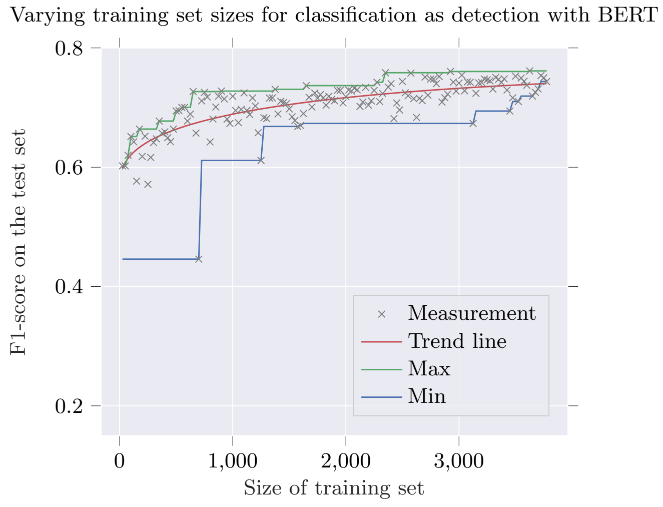


Fig. 37. Varying training set sizes for classification as detection with BERT†, with a fixed test set. These results were obtained without text preprocessing.

RQ4 Takeaways:

- With substantial labelling effort, it might be possible to slightly increase the detection performance for BERT.
- Generally, the multi-label classifiers have reached their maximum potential in terms of performance.
  - Simply increasing the size of the dataset does not seem to be effective for improving the multi-label classifier's performance.
  - Other strategies are needed to improve the performance.

Model	F_1	P	R
BERT	0.747 ± 0.036	0.723 ± 0.047	0.781 ± 0.090
RNN	0.721 ± 0.035	0.713 ± 0.053	0.737 ± 0.070
BERT†	0.715 ± 0.053	0.729 ± 0.068	0.727 ± 0.146
RNN*	0.713 ± 0.038	0.716 ± 0.065	0.719 ± 0.086
TF-IDF	0.696 ± 0.025	0.633 ± 0.044	0.781 ± 0.068
TF-IDF*	0.685 ± 0.028	0.649 ± 0.034	0.730 ± 0.060
BOWN	0.680 ± 0.032	0.646 ± 0.034	0.724 ± 0.083
BERT*	0.675 ± 0.061	0.710 ± 0.058	0.659 ± 0.127
BOWN*	0.666 ± 0.038	0.660 ± 0.027	0.675 ± 0.071
BOWF*	0.631 ± 0.052	0.742 ± 0.036	0.555 ± 0.082
CNN*	0.621 ± 0.041	0.478 ± 0.085	0.930 ± 0.119
BOWF	0.618 ± 0.085	0.773 ± 0.045	0.523 ± 0.110
DOC2VEC	0.617 ± 0.060	0.725 ± 0.067	0.550 ± 0.106
DOC2VEC*	0.616 ± 0.055	0.740 ± 0.065	0.536 ± 0.088
CNN	0.614 ± 0.049	0.602 ± 0.134	0.721 ± 0.248

Table 27. Detection scores for the cross-project validation. Results with a * are obtained from models trained using fine-grained technology replacement. Results with a † are obtained without text preprocessing. F_1 is the detection F_1 score, P is the detection precision, and R is the detection recall.

6.6 RQ5: Cross-Project Generalisability

In practice, it would be useful to be able to use an already trained classifier on any other software project, without the need to train the classifier specifically on that project. We therefore evaluated the capacity of the models to generalise to projects foreign to the training and validation set, but within the same domain. We tested this using cross-project validation, for which the results are shown in Tables 26 and 27.

Model	F_1	P	R
BERT	0.653 ± 0.031	0.666 ± 0.034	0.658 ± 0.031
BERT†	0.637 ± 0.036	0.657 ± 0.040	0.643 ± 0.061
RNN	0.604 ± 0.017	0.627 ± 0.036	0.610 ± 0.016
RNN$*$	0.603 ± 0.027	0.618 ± 0.053	0.611 ± 0.044
TF-IDF	0.577 ± 0.014	0.566 ± 0.024	0.609 ± 0.032
TF-IDF$*$	0.569 ± 0.036	0.568 ± 0.042	0.584 ± 0.038
BERT*	0.563 ± 0.033	0.619 ± 0.054	0.547 ± 0.043
BOWN	0.560 ± 0.015	0.568 ± 0.019	0.568 ± 0.028
BOWN$*$	0.551 ± 0.031	0.562 ± 0.034	0.553 ± 0.045
BOWF	0.548 ± 0.077	0.629 ± 0.057	0.520 ± 0.069
BOWF$*$	0.543 ± 0.036	0.597 ± 0.034	0.518 ± 0.033
DOC2VEC$*$	0.536 ± 0.056	0.607 ± 0.057	0.515 ± 0.056
DOC2VEC	0.527 ± 0.046	0.587 ± 0.050	0.514 ± 0.050
CNN	0.408 ± 0.121	0.423 ± 0.206	0.556 ± 0.128
CNN$*$	0.269 ± 0.082	0.237 ± 0.122	0.571 ± 0.162

Table 26. Macro scores for the cross-project validation. Results with a * are obtained from models trained using fine-grained technology replacement. Results with a † are obtained without text preprocessing. F_1 is the macro F_1 score, P is the macro precision, and R is the macro recall.

For this test, we see again major performance benefits by using BERT compared to the other models. In particular, BERT achieves an average F_1 score of 0.653, whereas the next best model, RNN, only achieves an average F_1 score of 0.604. The BOW models show performances between 0.577 and 0.548, followed by DOC2VEC with 0.536, and CNN performed very poor with an average F_1 score of 0.408.

For the detection task, we also see that RNN is being outperformed by BERT, with detection F_1 scores of 0.721 and 0.747 respectively. The closest term frequency model is TF-IDF with a F_1 score of 0.696.

For BERT, it seems that text preprocessing allows for a noticeable gain in performance. It might be the case that certain text formatting, or a lack thereof, give an indication that an issue is more likely to be architectural in some projects, which might not be the case in other projects. However, we have not verified any of this, so this could potentially be future work.

Fine-grained technology replacement seems to have a small negative impact on most models. For BERT, this is a big negative impact. In terms of generalisability across projects, it therefore seems to be beneficial to not replace things like technology names. We are not sure of the cause of this behaviour, but it seems that replacing the technology names leads to a loss of useful information for the classifiers.

Because BERT was the best performing model for both multi-label classification and for the detection task, we will focus our attention on this model. Comparing the cross-project performance with the regular performance from **RQ3**, we only see a performance drop of 0.02 in terms of macro F_1 score, and a drop of only 0.005 for the detection F_1 score. These are performance drops of approximately only 3.5% and 0.5%, respectively. This means that on average, around 96.5% of the multi-label performance is in fact transferable to projects not used for training. For detection this is even higher, where on average, around 99.5% of the performance is transferable.

We do have to make a note that some projects, i.e. HDFS, Hadoop, Yarn and Mapreduce, belong to the Hadoop ecosystem, meaning that they share the same code base, and potentially share the same developers as well. Additionally, Apache Tajo makes use of Hadoop, and is therefore also potentially related to the Hadoop ecosystem, but to a much lesser extent. This might make it easier for the classifier to achieve good performance on such projects, because similar keywords could be used in such an ecosystem. For that reason, we also included a graph showing the performance on each of the projects in Figure 38. This figure shows that the performances for Apache Cassandra and Tajo are in fact about average compared to the other projects, showing that the performance is also transferable to non-related or less-related projects. However, there is quite a gap between the precision and recall performance for Cassandra and Tajo. This might be the effect of the fact that a large part of the training set belongs to the Hadoop ecosystem. Interestingly, the precision for Tajo is relatively high compared to its recall, while the precision for Cassandra is relatively low compared to its recall. Unfortunately, we currently do not have a grounded explanation for these observations.

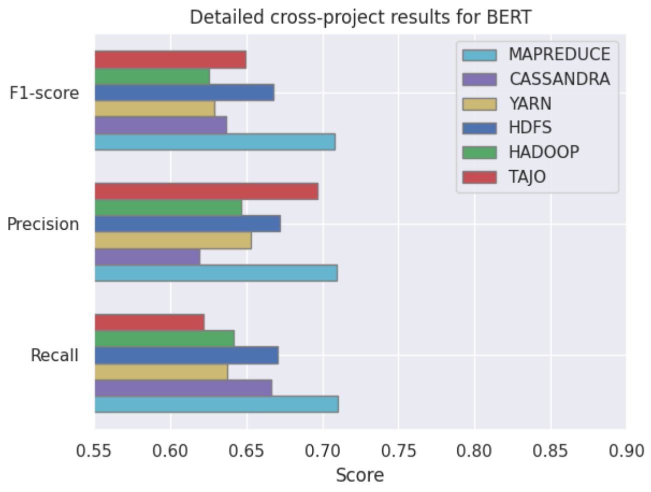


Fig. 38. Detailed cross-project performance of BERT.

With the results of this test, we have shown that the performance is transferable to projects not used for training the classifier. In fact, we only saw a performance drop of about 3.5% for multi-label performance and around a 0.5% drop for detection. This shows that an already trained classifier can in fact be used on projects foreign to the training data, but in the same domain and ecosystem.

RQ5 Takeaways:

- BERT has the best generalisability to different projects of all classifiers we tested.
  - 96.5% of the multi-label classification performance was transferable to other projects.
  - 99.5% of the detection performance was transferable to other projects.
- BERT can be an effective classifier for projects that are not used for training the classifier, given that the project is in the same domain.

6.7 RQ6: Cross-Domain Generalisability

For **RQ6**, we evaluated the classifier on two random samples. For the first random sample, we focused on the performance of the classifier as a plain classifier on different software domains. For the second random sample, we focused on evaluating the classifiers as a search tool for finding architectural issues.

6.7.1 Fully Random Sample

In Table 29, we can see the performance of BERT (without text preprocessing) on the random sample of 400 issues from the six domains. In Table 31 we see the performance of BERT (with text preprocessing) on the same dataset. For completeness, Appendix G contains the performances of the other models we tested on the random sample from six software domains. We decided to look specifically at BERT with text preprocessing, because in most of the tests we performed, this was the best performing model. BERT without text preprocessing was used to obtain the second random sample from this research question (i.e. the random sample based on the classifier's confidences), due to a bug which was found too late by the authors. In hindsight, it would probably have been slightly better to use BERT with text preprocessing. However, because we used BERT without text preprocessing for the second random sample, we decided to also include its performance for the 'fully' random sample as well.

Additionally, we have Tables 28 and 30 which depict the performance of the BERT without text preprocessing and the BERT with text preprocessing models, respectively, on the sample of 400 issues from the data storage & processing domain. These tables serve to put the results from Tables 29 and 31 into context.

When we compare the results in Tables 28, 29, 30 and 31 with the performance scores outlined for the same BERT models in Sections 6.4 and 6.6, we can immediately see that the performance scores are considerably lower. At a first glance, one could think this is because the models generalise poorly to different domains. However, Tables 28 and 30 contain the performances of the BERT models specifically on a test set consisting of issues from the projects on which the model was also trained. The performance scores in this table are comparable to those in Tables 29 and 31.

We have two possible reasons which could explain the poor performance of the BERT models, which we see in Tables 28 and 30. The first reason is that there might be some systematic bias in our dataset. All the issues in the dataset, even the non-architectural ones, were found using techniques specifically designed to find architectural issues. Hence, the classifier might be biased towards such issues. If we then test the classifier on issues from a random sample (i.e. a test set without this bias), the classifier's performance could be poorer. Another explanation is that there are very few architectural issues in these random samples. Precision and recall are computed using the true positive count, false positive count, and false negative count. With a random sample and classifiers with close-to-realistic prediction proportions, all these numbers will be fairly small (due to the large numbers of non-architectural issues, the true negative count is by far the highest). As a result of this, the scores can easily be affected by particularly "poor" or "lucky" samples (see Appendix H for the exact counts and full details). However, given the fact that on both random samples we have somewhat similar performance scores, we think the first reason is more likely.

In an attempt to try to reduce the effect of the second reason, we also computed the 95% confidence intervals for the detection task and the class-specific scores for the classification task, and displayed these in Tables 28, 29, 30 and 31.

We start with discussing the detection performance of BERT without text preprocessing on the random sample from the six domains (Table 29). Of course, the F_1 score of 0.508 is considerably worse than the one of 0.753 we had in Section 6.4 on the fixed test set. However, when we compare with the performance of BERT on the random sample from the data storage & processing domain (Table 28), the performance is actually rather comparable. The F_1 score on the six domains is 0.026 lower; the precision seems somewhat better, and the recall seems slightly worse. The performance is still also twice as good as the best-guessing classifier. For the other BERT model with text preprocessing, we see again see a large performance gap compared to what we achieved in Section 6.4. However, for this model, there is also a large performance gap between the random samples. The F_1 score on the random sample from the six domains (Table 31) is 0.132 lower than on the data storage & processing domain (Table 30). Most of the loss seems to come from a large loss in terms of recall. The BERT model without text preprocessing suffered from the same problem, although the loss for that model was slightly smaller. We are unsure what causes this loss of recall.

We now move our attention to the metrics for the multi-class classification for BERT without text preprocessing. All macro

Metric	F_1	Precision (95% confidence interval)	Recall (95% confidence interval)	Improvement over best-guessing
Detection	0.534	0.456 ([0.338, 0.574])	0.646 ([0.511, 0.781])	+150%
Macro	0.549	0.575	0.572	+135%
Positive macro	0.425	0.451	0.464	+273%
Weighted macro	0.839	0.868	0.821	+8%
Existence	0.533	0.500 ([0.345, 0.655])	0.571 ([0.408, 0.735])	+231%
Executive	0.400	0.308 ([0.130, 0.485])	0.571 ([0.312, 0.831])	+488%
Property	0.343	0.545 ([0.251, 0.840])	0.250 ([0.077, 0.423])	+203%
Non-Architectural	0.921	0.949 ([0.925, 0.973])	0.895 ([0.863, 0.927])	-2%

Table 28. BERT (without preprocessing) performance on the random sample from the data storage & processing domain. F_1 is the F_1 score. For the class specific precision and recall, and for the detection precision and recall, we also provide the 95% confidence intervals for those values. For more details, see Table 71 in Appendix H.

Metric	F_1	Precision (95% confidence interval)	Recall (95% confidence interval)	Improvement over best-guessing
Detection	0.508	0.508 ([0.381, 0.636])	0.508 ([0.381, 0.636])	+90%
Macro	0.469	0.583	0.469	+105%
Positive macro	0.321	0.473	0.320	+159%
Weighted macro	0.805	0.840	0.801	+8%
Existence	0.308	0.400 ([0.152, 0.648])	0.250 ([0.077, 0.423])	+182%
Executive	0.355	0.268 ([0.133, 0.404])	0.524 ([0.310, 0.737])	+274%
Property	0.300	0.750 ([0.450, 1.000])	0.188 ([0.052, 0.323])	+78%
Non-Architectural	0.915	0.915 ([0.885, 0.945])	0.915 ([0.885, 0.945])	+0%

Table 29. BERT (without preprocessing) performance on the random sample from the six domains. F_1 is the F_1 score. For the class specific precision and recall, and for the detection precision and recall, we also provide the 95% confidence intervals for those values. For more details, see Table 72 in Appendix H.

Metric	F_1	Precision (95% confidence interval)	Recall (95% confidence interval)	Improvement over best-guessing
Detection	0.598	0.592 ([0.454, 0.730])	0.604 ([0.466, 0.743])	+179%
Macro	0.613	0.623	0.607	+162%
Positive macro	0.502	0.516	0.495	+341%
Weighted macro	0.871	0.876	0.868	+12%
Existence	0.562	0.621 ([0.444, 0.797])	0.514 ([0.349, 0.680])	+249%
Executive	0.444	0.462 ([0.191, 0.733])	0.429 ([0.169, 0.688])	+554%
Property	0.500	0.464 ([0.280, 0.649])	0.542 ([0.342, 0.741])	+342%
Non-Architectural	0.945	0.946 ([0.922, 0.970])	0.943 ([0.919, 0.967])	+1%

Table 30. BERT (with preprocessing) performance on the random sample from the data storage & processing domain. F_1 is the F_1 score. For the class specific precision and recall, and for the detection precision and recall, we also provide the 95% confidence intervals for those values. For more details, see Table 74 in Appendix H.

Metric	F_1	Precision (95% confidence interval)	Recall (95% confidence interval)	Improvement over best-guessing
Detection	0.466	0.545 ([0.398, 0.693])	0.407 ([0.281, 0.532])	+74%
Macro	0.489	0.557	0.465	+114%
Positive macro	0.345	0.442	0.306	+178%
Weighted macro	0.815	0.820	0.823	+9%
Existence	0.278	0.417 ([0.138, 0.696])	0.208 ([0.046, 0.371])	+155%
Executive	0.383	0.346 ([0.163, 0.529])	0.429 ([0.217, 0.640])	+303%
Property	0.375	0.562 ([0.319, 0.806])	0.281 ([0.126, 0.437])	+122%
Non-Architectural	0.921	0.902 ([0.871, 0.933])	0.941 ([0.916, 0.966])	+1%

Table 31. BERT (with preprocessing) performance on the random sample from the six domains. F_1 is the F_1 score. For the class specific precision and recall, and for the detection precision and recall, we also provide the 95% confidence intervals for those values. For more details, see Table 75 in Appendix H.

F_1 scores in Table 28 have become worse than those in 29; in all cases, both precision and recall have decreased. The relative performance gain compared to a best-guessing classifier has also become worse, although the BERT performance on the six domains is still considerably better than that of the best-guessing classifier. We can make similar observations for the BERT model with text preprocessing in Tables 30 and 31.

Moving our attention to the class-specific scores (Table 29) of BERT without text preprocessing, it seems that the performance for existence decisions is not easily transferable to different domains. While the classifier obtains a 0.533 F_1 score for the random sample from the data storage & processing issues, the performance is only 0.308 across the six domains. Especially the recall has become worse. In fact, this is one of the rare cases where there is actually a statistically significant difference between the two scores, based on the confidence intervals. This shows that component decisions are probably very specific to a domain or even a project, making it hard to detect existence decisions in other domains or projects. This drop in performance is even larger for the BERT model with text preprocessing. While it achieved an F_1 score of 0.562 on the data storage & processing issues, it only achieved an F_1 score of 0.278 on the random sample from the six domains.

For both BERT models, the performance for executive decisions between the data storage & processing domain test and the test on the six domains is somewhat comparable, although performance has gone down on the six domains. We note that both BERT models are scoring achieving the highest performance on the executive class in terms of recall compared to the existence and property classes; this is likely the result of the fact that many executive issues about dependency upgrades are easy to catch by the classifier.

For the property issues in the random sample from the six domains, we see a much improved precision and a much worse recall for both BERT models compared to the data storage & processing domain. We think the high precision can be attributed to the fact that many property decisions are done in terms of quality attributes, which are easy to catch by classifiers. The lower recall shows the classifiers have trouble identifying all property issues across the six domains though. The classifier might be a bit too focused on how developers from the data storage & processing domain make property decisions. This might potentially be an interesting topic for future research; finding out whether there is a difference in how developers express design decisions across different domains.

The main contributor to the macro scores is the non-architectural class. Because many issues are non-architectural, it is easy to get a high score for this class, as is shown by the small performance differences (1%) with the best-guessing classifier. Nonetheless, the classifiers show major performance increases over the best-guessing classifier. This is because each type of design decision is made only in a small fraction of the issues.

6.7.2 Random Sample Based on Confidence

Our second random sample is more focused towards evaluating the classifier as a search tool for architectural issues. To show this, we created a graph showing the average precision (from the highest confidence to the lowest) versus the confidence of the classifier for the given class (Figure 39). An interesting finding is that a confidence value of only slightly above 0, already leads to a relatively high precision for finding architectural issues. This might be because of the sigmoidal output we used for the classifier. Issues for which the classifier is really certain have confidences very close to 0 or 1. Only issues for which the classifier is not certain have confidences more in the middle; this is a result of the steep curve present in sigmoidal activation functions.

For large confidence values, a small decrease in confidence leads to a large decrease in precision. What this means in practice is that you have to use the issues with very high confidence values, in order to find architectural issues with a very high precision. However, also for lower confidence values, we see good precision scores. In fact, for the default confidence threshold of 0.5, we see that 63% of the predicted architectural issues, were in fact architectural issues. For property this was slightly higher with 65%, executive was somewhere in between with 56%, and the precision for existence was the worst with 43%.

Going back to the second random sample, we have created Figure 40. This figure shows the precisions for each of the classes and for each domain. What we see in this graph is that the precision for the data storage & processing domain is generally one of the highest for each of the classes. This is expected, because the classifier is trained specifically on issues from this domain. The classifier performs really well on the content management domain. We suspect that this domain might be rather similar compared to the data storage & processing domain in terms of design decisions. Another hypothesis is that design decisions in this domain might be expressed more explicitly, making it easier for the classifier to catch the design decisions. However, this could be a topic for future work.

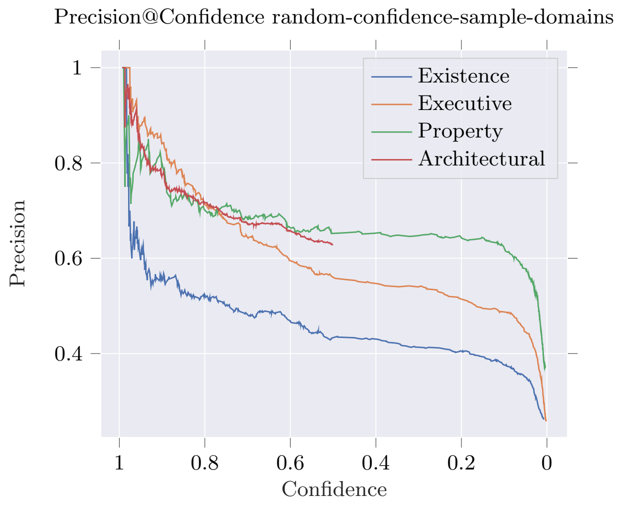


Fig. 39. Precision@confidence graph for the random sample of 594 issues from the six domains, based on the confidence scores of BERT. Note that the confidence for "Archiecetural" was computed as the maximum of all thee confidences outputted by the classifier.

What is interesting about the existence decisions, is the fact that the classifier performs really poor on most domains. For example, the precision is only 0.2 for the devops and cloud domain. We suspect that existence decisions can differ a lot between the domains, and potentially also between projects as well. It would be interesting to make a qualitative comparison between the design decisions made in different domains, in order to see whether there is any truth in this hypothesis.

Property and executive decisions do not show much difference in terms of precision between the domains. We also suspect that these two types of decisions are in fact more similar between the domains compared to existence decisions. For example, many executive decisions involve selecting or dealing with external technologies. This is unlikely to be very different between domains. Moreover, property decisions are about improving overarching traits of the system. Keywords involved in these decisions, such as *performance*, *security*, and *improvement*, are rather obvious for the classifier and should not differ much between domains as well.

The graphs have shown that the classifier is rather efficient for finding executive and property issues in basically any domain we tested. The precision for executive is higher than 44% for any domain, for property and architectural issues the precision is higher than 53% for any domain. Only existence issues show really poor generalisability across the domains. However, even for the worst performing domain, devops and cloud, the precision is almost four times better than finding existence issues using random sampling.

RQ6 Takeaways:

- The performance of BERT drops significantly when evaluated on a random sample
  - One possible cause is the small amount of architectural issues in such a sample, combined with randomness.
  - Another possible cause is that most issues in the dataset have some sort of bias due to being selected with specific search methods.
- When comparing the performance of BERT on a random sample from the data storage & processing domains with a random sample from the six domains
  - BERT performs noticeable worse on the six domains.
  - BERT still performs reasonably well compared to best-guessing classifiers.
- BERT is an effective search tool in other domains as well.
  - A precision of 63% for finding architectural issues.
  - A precision of 43% for finding existence issues.
  - A precision of 56% for finding executive issues.
  - A precision of 65% for finding property issues.
- Existence decisions seem to be the most different between domains.

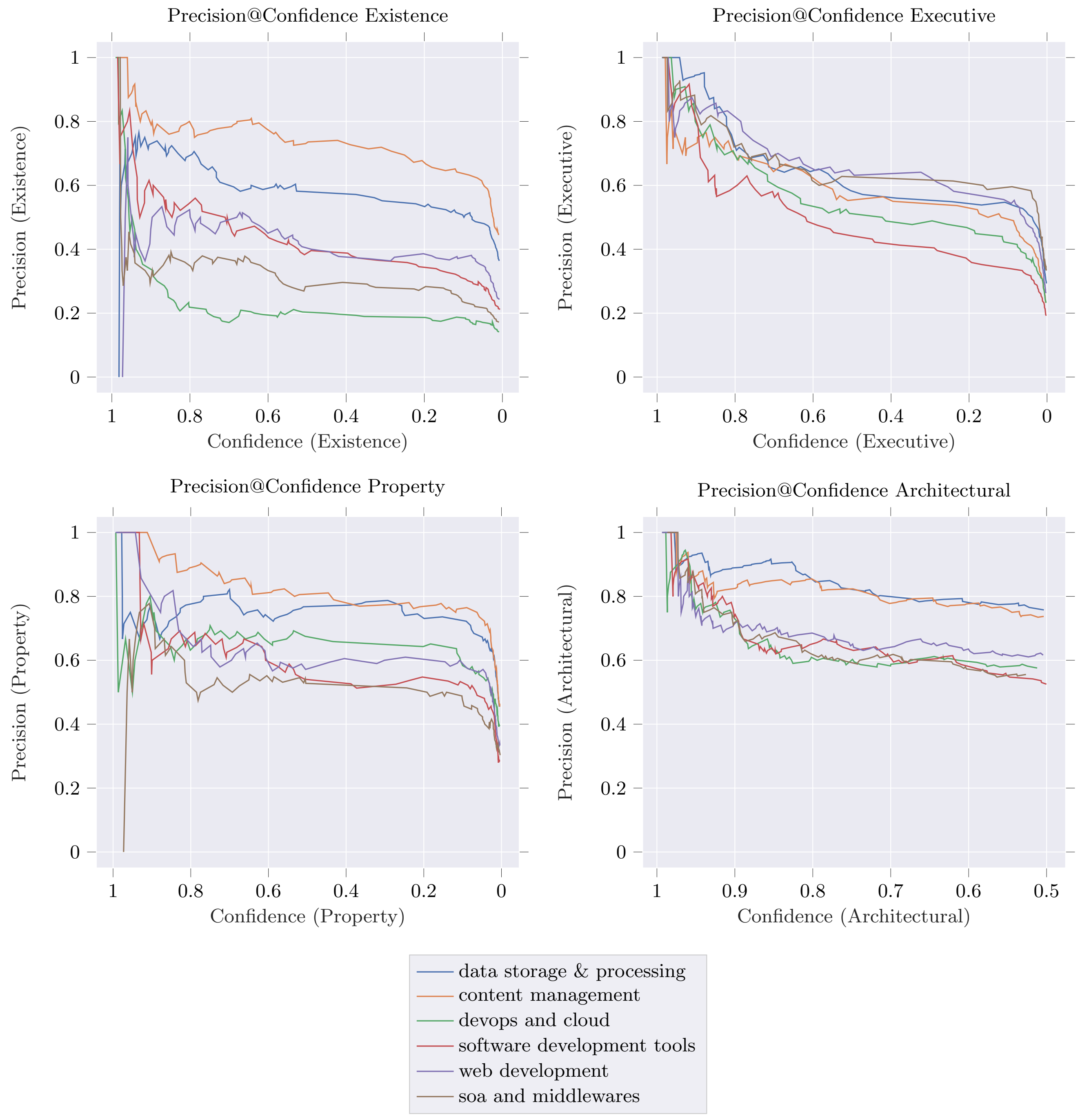


Fig. 40. Precision@confidence graph for the random sample of 594 issues from the six domains, based on the confidence scores of BERT. This graph contains the precision of each of the classes for each domain separately. Note that the confidence for "Architecetural" was computed as the maximum of all thee confidences outputted by the classifier.

7 Discussion

In this section, we will discuss what the findings of our research mean for researchers and practitioners, and what its broader implications could be.

7.1 Implications for Researchers

In answer to **RQ1**, we found that architectural issues in issue tracking systems are rare; around 10%-15%. This means that random search approaches are an infeasible approach for researchers to find large quantities of architectural knowledge in issue tracking systems. Hence, for studies requiring a lot of architectural issues, dedicated search tools are necessary.

The results we found for **RQ2** can in turn help researchers to make informed decisions about what search tools to use. In this work, we compared our proposed deep learning based search with keyword search, Maven analysis, and static source code analysis. Deep learning and keyword analysis are both types of top-down search: they search for architectural knowledge in issue descriptions themselves. Static source code analysis and Maven analysis are bottom-up search methods: they analyse source code changes, and link these changes to issues in order to identify architectural issues. We found that the top top-down methods achieve better precision@k than the two bottom-up methods. The main exception is that Maven analysis is also able to find executive issues with high precision. However, many of these issues fall into the category of dependency version bumps. We believe the true strength of deep learning is its ability to search for issues with specific labels. For instance, we were able to find more "interesting" executive issues by selecting executive issues with high existence scores.

Overall, we recommend that researchers use top-down approaches to find architectural issues. These approaches are able to find more architectural issues than bottom-up approaches. The main exception to this rule would be if researchers were looking specifically for some type of architectural issue which is not easily identified using top-down approaches – although this would first require researchers to actually pinpoint the existence of such issues. Researchers could also consider combining multiple techniques to see how well they perform for specific purposes. For instance, we expect there might be benefit to combining deep learning with keyword searches to find more specific architectural information.

Finally, we make the informal recommendation that researchers wishing to analyse architectural knowledge in open source systems focus their attention on the Apache and Spring ecosystems. While annotating, we found that these projects seem to have the most clear discussions on architecture.

The results of **RQ3** show that multi-label classifiers outperform binary classifiers that are specifically trained for detecting architectural issues. We therefore recommend that researchers use multi-label models instead, because besides detecting architectural issues, they also classify architectural issues into their subtypes (i.e. existence, executive, and property).

Researchers can also use **RQ3** to decide on the best model for classifying issues. In case researchers want to apply a classifier on issues from projects used to train the classifier, our recommendation will be to use BERT with text preprocessing. This was overall the best performing model. Additionally, BERT predicts realistic amounts of architectural issues. While this is not a guarantee that BERT performs well in practice, it at least suggests that it could be useful for researchers wanting to apply this classifier in practice.

Researchers could look at **RQ5** if they want to apply a classifier on projects that are not necessarily used for training the classifier. Again, we recommend BERT with preprocessing for this application. The results showed that, compared to **RQ3**, on average nearly all performance was transferable to projects not used for training. We therefore suspect that these classifiers can useful for researchers on basically any project in the domain the classifier was trained on.

RQ6 shows that the performance on random samples is much worse compared to **RQ3**. Because these samples are small, we are not sure whether this performance is poor due to an 'unlucky' random sample, or whether the classifiers might be less performant in practice than we expected. Additionally, there was a noticeable performance drop when applying the classifier on six domains, compared to the performance on the data storage & processing domain. Nonetheless, the classifiers show major improvements over a best-guessing classifier, suggesting they might still be useful for researchers. Due to the suspected randomness in the results, we cannot make a clear recommendation. However, given the results from **RQ3** and **RQ5**, we suspect that BERT with text preprocessing is the safest option to get good performance when applying the classifier on different domains.

RQ6 also showed that DL classifiers have a high precision for finding architectural issues in six domains. DL classifiers can therefore be helpful for researchers wanting to find architectural issues in such domains. Although we evaluated BERT without text preprocessing, given that most evaluations suggest that BERT with text preprocessing is slightly better, we recommend researchers to use BERT with text preprocessing for finding architectural issues in different domains.

While we recommend to use BERT as the classifier for basically any task, we do have to note that it is a computationally heavy model. Therefore, if computing power is a limiting factor, RNN is also a good alternative, because it showed good generalisability to different projects in the same domain. If RNN is not feasible, or if the search is performed on projects also contained in the training set, TF-IDF* (TF-IDF with fine-grained technology replacement) can also serve as a computationally cheap alternative.

RQ4 guides researchers with data collection. The results we obtained here show that expanding the dataset with issues from the six projects already contained in the dataset is unlikely to result in any real performance gains. Hence, this is not a worthwhile line of research to pursue. A more feasible approach is likely to be working on diversifying the dataset by adding issues from different projects, domains, and ecosystems.

7.2 Implications for Practitioners

For practitioners, **RQ1** has similar implications as for researchers. Since architectural knowledge in issue trackers is sparse, it can be difficult to locate architectural knowledge. This shows that, unless practitioners know specifically what they have to search for, they also have to resort to dedicated search tools.

RQ2 helps guide practitioners in the search tools to use. In general, we advise the use of deep learning for finding existence and executive issues, and keyword searches or deep learning for property issues. Practitioners may also benefit for specific information on technology choices and upgrades. In such cases, Maven analysis could also be used.

Especially for practitioners, we would advise the use of hybrid techniques. We observed that deep learning searches are able to capture a wider variety of issues (i.e. they are exhausted less quickly than keyword search and Maven analysis). On the other hand, keyword searches could potentially be used to search for more targeted information. Because of this, there might be benefit to using a combination of keyword search and deep learning. The results from **RQ3**, **RQ5**, and **RQ6** can support in making a choice regarding what classifier to use, with the same considerations as we outlined for researchers.

One of the largest hurdles for practitioners is finding actually relevant architectural knowledge. While labelling issues, we encountered a significant amount of issues that would be considered architectural according to our coding book, but which do not necessarily contain useful information. Additionally, for many projects, the information can be very domain specific. This means that the information found in issues may only be useful for practitioners with significant experience regarding certain techniques (for example, compaction and gossip in Cassandra would require prior knowledge on those topics to fully understand and use the knowledge in many issues). Because of this, directly searching for issues containing architectural knowledge is not a suitable method for architectural knowledge re-use for architects who are unfamiliar with the techniques being used. The search tools are best used to fine-tune the design of similar systems, or for other purposes where the practitioner is already familiar with the technologies used in the system being searched for architectural knowledge.

We believe these search tools might be most useful for knowledge re-discovery, and not necessarily knowledge re-use. We imagine that there are scenarios where the lack of rationale for certain design choices might make it difficult to continue development of the system. We believe that search tools for architectural knowledge in their current form would be most useful in cases where details about specific design choices in the same system are desired.

8 Threats to Validity

In this section, we discuss factors that may affect the validity of this research.

8.1 External Validity

8.1.1 Dataset Construction Bias of the Original Dataset

The initial dataset was collected using Maven POM file analysis ([17]), keyword searches ([28]), and static source code analysis ([70]). It might be possible that these methods only capture certain kinds of non-architectural and architectural issues, but totally miss others. This would mean that certain types of issues were not represented in the original dataset.

In this research, we mainly used deep learning classifiers to find additional architectural issues. However, the results found by these classifiers naturally find issues similar to those in the training dataset. This means that any biases resulting from the original three methods will most likely still be present in the dataset created in this research.

We also found some issues using random sampling. These issues do not have this selection bias, and thus may contain types of architectural issues otherwise not present in the dataset. However, the amount of architectural issues in these samples was very small; this means that these samples likely did little to nothing to mitigate these biases for architectural issues in particular. However, because the number of non-architectural issues in these samples was large, we may expect that the dataset now also contains a significant sample of non-architectural issues found using this non-biased selection method.

We evaluated all our deep learning classifiers on two different test sets: on a test set obtained through a stratified random split of the full dataset, and a test set obtained by taking the 400 issues from the random sample from the data storage & processing domain. In Section 6.4, we saw that the performance is worse both in terms of precision and recall when using the latter as a test set. This may be an indicator that models trained on issues found using Maven analysis, keyword searches, source code analysis, and deep learning indeed have some sort of bias. Though, we should be careful with this conclusion, because the amount of architectural issues in the random sample is also small.

8.1.2 Data Leakage

Data leakage is the phenomenon that information about the test set is contained in the training set. If this occurs, the performance on the test set might be too optimistic. Data leakage can be a problem of the nature of the data, or of the way the experiments were conducted. In this research, we identified two potential sources of data leakage which can be attributed to the nature of our dataset:

8.1.2.1 Relationships between Issues

An anonymous reviewer for a paper on finding ADDs, co-authored by the two authors of this work, pointed out that temporal dependencies between issues may constitute a form of data leakage (i.e. information about the test set is contained in the training set). After deliberating, we identified that may be related to an even bigger problem, which both influences bias of the dataset, and affects data leakage.

Issues in issue trackers often do not come alone. Some issues duplicate others, some issues may introduce bugs, some issues contain follow-up work, some issues are prerequisites for others, and some issues have child/parent relationships.

The anonymous reviewer was specifically alluding to follow-up work and prerequisite work; in these cases, issues in the training dataset may mention issues in the test dataset, or vice versa. Since the key of an issue is not an input for the classifiers, this does not necessarily constitute data leakage.

However, follow-up work, prerequisite work, and especially child/parent issues, do tend to form clusters of very closely related issues. Consider for instance HDFS-1052[28]. This particular issue has 91 child issues. All these 92 issues somehow deal with the same topic: architectural changes to improve scalability of HDFS. Because this is such a large issue, it is also related to 7 other issues. If these issues are all similar enough, or contain enough similar concepts, a classifier might be able to get these issues right "too easily" if they are in the test set (provided enough of the issues are contained in the dataset).

This example illustrates how related issues may constitute data leakage, which may affect the validity of the performance scores we obtained.

8.1.2.2 Duplicate Issues

Some issues are contained multiple times in the dataset, because identical issues were created. Anecdotally, we found one issue which was duplicated around 15 times (!). Additionally, sometimes a user creates multiple issues which are almost identical, but not quite. A deep learning classifier is likely to give identical or similar predictions for such issues. If an issue is contained in the training dataset, and a duplicate is contained in the set dataset, this constitutes a form of data leakage.

8.2 Construction Validity

8.2.1 Size of the Random Sample from the Six Domains

In order to test the generalisability of multi-label classifiers to different domains for **RQ6**, we took two random samples: one fully random sample of 400 issues, and one random sample based on confidence of $6 \times 3 \times 33 = 594$ issues. In retrospect, we determined that the sample of 400 issues was too small for a good performance evaluation. The main problem is that in such a random sample, very few issues are architectural. This means that there is a large degree of uncertainty in the computed precision and recall scores.

The specific problem is that precision and recall are computed using the true positive count, false positive count, and false negative count (see Appendix H). With the small amount of architectural issues we have, these numbers are all low (the majority of the issues are non-architectural, which contribute to the true negative count). As a result, the precision and recall scores have large standard errors, leading to a lot of uncertainty in the results.

In order to avoid this, either a larger sample size should be used, or a different evaluation method should be used.

8.2.2 Evaluation of Deep Learning Models

For evaluating our DL models, we used a fixed holdout test set. However, it seems that neither a holdout test set nor a k-fold cross validation give good estimations for the practical performance of the models [32]. Considering that our dataset is rather small, analysing the results on a holdout test set as well as the results of a k-fold cross validation, might have given us better insights into the true performance of the DL models.

However, given that neither of the two methods give good estimations of the true performance of a classifier [32], and the fact that our dataset is potentially biased (Section 8.1.1), a larger random sample as the test set might have been a better option. This would have required a lot of manual labelling effort, but it would at least have given a better insight into the practical applicability of the classifiers.

[28]`https://issues.apache.org/jira/browse/HDFS-1052`

8.2.3 Experimental Mistakes

We identified a number of mistakes in how we conducted our experiments. When we discovered these, we no longer had time to correct them. Hence, we present them here.

8.2.3.1 Suboptimal Settings for BERT during Dataset Expansion

While using BERT to collect more architectural issues, we used different hyperparameters than for the final version of BERT we evaluated later. This is because we were still running BERT locally, which constrained us to smaller batch sizes. As a result of this, the conclusions we reached with regard to deep learning for finding architectural issues (i.e. **RQ2** could be slightly different from for the BERT model we experimented with for **RQ3** and onward.

8.2.3.2 TF-IDF Trained Sub-optimally

Generating TF-IDF features is a two-step process: first, the term weights must be computed by computing the IDF for each word. Next, term frequencies are computed and multiplied by the IDF. The key here is that IDF computation can be done separately from the first step. The IDF weights can either be computed on the training dataset itself, or on a larger corpus of issues. We initially intended to compute the IDF weights on all issues from Hadoop, Yarn, Tajo, HDFS, MapReduce, and Cassandra. However, we inadvertently only computed them based on the issues from those project, which were contained in our actual dataset. This mistake could have implicated for the generalisability of the TF-IDF model. Due to the way we determined the IDF, the IDF weights could be too specific, leading to poorer generalisability.

8.2.3.3 No Experiments with Squared Hinge Loss

During our hyperparameter optimisation, we were planning to experiment with cross entropy loss, hinge loss, and squared hinge loss. However, we accidentally excluded the squared hinge loss. Since cross entropy was essentially always the best performing loss function, we do not believe this has any effects on our results.

8.2.3.4 Incorrect BERT Model for **RQ6**

In order to answer **RQ6**, we planned on using the BERT model with preprocessing. However, due to a bug in our code, we accidentally used the BERT model without preprocessing.

The results for **RQ5** showed that BERT with preprocessing generalises better to different projects in the same domain without preprocessing. Based on this, we hope that the same holds across domains. This would mean the results we got for **RQ6** are pessimistic lower bounds for the results we should expect for BERT with preprocessing.

8.3 Reliability

8.3.1 Quality of Issue Labelling

We used qualitative techniques while labelling more issues. There is inherent risk of human error with such an approach. In this section, we discuss some of the threats introduced by our issue labelling approach.

8.3.1.1 Low Kappa Score

In Sections 4.5.2.3, 4.5.3 and 4.9.3.1, we elaborated on the agreement of our issue labelling. We observed that the Kappa score was generally quite low. Additionally, the confusion matrices also showed that there was generally a not insignificant amount of disagreement between reviewers.

If different annotators label issues differently, this could lead to worse classifier performance; The classifier may “get confused” by contradictory labelling, or if its predictions had been correct if the issue were labelled by a different annotator. Additionally, inconsistent labelling also leads to reasonable doubts regarding the accuracy of the performance evaluation of BERT across different domains.

We attempted to mitigate this threat by 1) double-checking with other annotators in case of uncertainty, and 2) checking for systematic disagreements and attempting to correct these. However, the agreement scores were still relatively poor, leading to doubts about the quality of the labelling. One redeeming factor here is that the agreement was computed on the most difficult in the dataset, hopefully leading to a pessimistic lower bound.

8.3.1.2 Ambiguous Definition of Existence Decisions

As explained in Section 4.5.3, while relabelling systematic labelling errors, we found that the definition of existence issues had become inconsistent. Additionally, for determining whether an issue was existence, we sometimes looked at other information than only the summary and the description of an issue. The classifier has no access to such information, and would thus not be able to use it. Hence, an inconsistent (or at least ill-defined) definition of existence, alongside a reliance on additional information, may have led to the inclusion of issues labelled as existence which cannot properly be identified by classifiers as such.

9 Conclusion

In this thesis, we have expanded an existing dataset of labelled issues from [18]. With random sampling, we found that only 10-15% of the issues in an issue tracking system contain architectural design decisions. To efficiently find architectural issues, researchers and practitioners need search tools. We have experimented with using a deep learning classifier (BERT) as a search tool, which outperformed existing tools, such as keyword searches, Maven dependencies analysis, and source code analysis. One of the main reason for this is that existing tools showed signs of exhaustion, meaning that they could not find architectural issues efficiently any more. Furthermore, we demonstrated that deep learning is also an effective search tool in different software domains, without having to train them on specific domains.

Moreover, we have improved the performance of the classifiers compared to previous work in [18]. Additionally, we have performed different evaluation techniques to evaluate the performance of the classifiers in a more practical setting. We have found that the new multi-label classifiers should perform better in practice than existing classifiers from [18]. Also, we have found that almost all the classifier's performance is transferable to projects in the same domain, that have not been used for training the classifier. When applied on projects in different domains, around 80% of the performance is transferable. This shows that researchers and practitioners could use the trained classifier on a wide variety of projects, not necessarily belonging to the same software domain.

9.1 Future Work

In this section, we will describe a number of possibilities for future research. We categorised these possibilities into improvements for this research, alternative deep learning applications for ADDs, analysis of ADDs, and improvements to Maestro.

9.1.1 Deep Learning Improvements

In this section, we will propose various methods to improve the current deep learning classifiers. Specifically, we will propose various methods to improve classifier performance.

9.1.1.1 Determine Cause of Incorrect Classifications

We saw that the performance of the classifiers on random samples and on issues from different domains is significantly worse than the performance on a simple test set taken from our existing dataset. This implies that there might still be room to improve the performance of the classifiers.

We think that, before performing experiments in the hope of improving classifier performance, it would be a good idea to investigate likely causes of poor performance. Future work could therefore focus on performing qualitative analysis on issues which are incorrectly classified by classifiers. This way, it would be possible to determine whether improvement should be done through collecting more issues, relabelling the existing dataset, or something entirely different.

9.1.1.2 Improve Labelling of the Dataset

As discussed previously, the agreement we got while labelling issues was not great. This might lead to concerns regarding the quality of the dataset. Incorrect or inconsistent labelling might also have a negative effect on classifier performance.

As such, future work could focus on isolating common labelling mistakes in the current dataset, and re-labelling issues affected by those mistakes. The fact that we tagged issues with both their acquisition method(s) and label authors could help in such an effort.

9.1.1.3 Extend Dataset of issues

The results of **RQ4** showed that collecting more issues will not necessarily improve the performance of the classifier. However, this analysis did not take into account what types of issues would be added to the dataset. Since we still had relatively poor performance on the random samples and on issues from different domains, collecting more data may still result in performance improvements. We suggest two ways in which additional data could be collected:

1. Collect data from a wider variety of projects, domains, and ecosystems. The more varied the training data, the better we can expect generalisability to be.

2. Device new ways to find architectural issues. As explained in Section 8.1.1, our current dataset might be biased because of the methods used to find architectural issues. If this turns out to be true (e.g. confirmed through qualitative analysis), then finding new ways to discover architectural issues, which are not discover-able using the other methods, could lead to better generalisability performance of the classifiers.

9.1.2 Deep Learning Refinements

In this section, we will discuss a number of different ways in which deep learning can be leveraged for identifying architectural knowledge in issue tracking systems.

9.1.2.1 Perform Sentence- or Paragraph-level Classification of ADDs

In the current dataset, issues are labelled on the issue-level. However, some parts of an issue might contain architectural knowledge, while others may not. Hence, it could be useful to classify issues with a more fine-grained classification of issue descriptions. For instance, issues could be annotated on the paragraph or sentence level. Other researchers have done similar things before. For instance, Viviani et al. classified pull requests on the paragraph level; entire comments in pull requests were found to be too course-grained, while sentences were found to be too fine-grained [81]. An additional benefit of developing classifiers, which can more narrowly pinpoint architectural knowledge, is that they might also be more useful in the automatic extraction of architectural knowledge in future research.

9.1.2.2 Perform Regression on "Degree of ADD-ness"

While labelling issues, we observed that some issues are more obviously architectural than others. For instance, in some issues, it is very clear that they contain an existence decisions; in other issues, it might be very subtle or very minor. Our first supervisor proposed the idea of regression on the degree of "ADD-ness". For instance, an issue could be annotated as being 0.2 existence, 0.6 property, and 0.0 executive. Implementing such a regression task could be a next step in separating less useful from useful issues. Here, usefulness would then be determined by how the coding book used for that annotating procedure defines usefulness; it could for instance be the amount and clarity of the architectural knowledge.

9.1.3 Investigating ADDs

In this research, we created two sizeable datasets of architectural issues: one dataset of manually labelled issues, and one dataset of issues found using deep learning classifiers. Both these datasets can be used for further analysis to improve our understanding of architectural knowledge in issue tracking systems.

9.1.3.1 Investigate How ADDs Evolve Over Time

One thing which can be investigated in future research, is how architectural issues evolve over time as a system evolves. Some projects use issue tracking systems for the entirety of their lifetime, and different types and amounts of ADDs may be discussed in different lifetime phases of the system. Such knowledge could possibly be used to further refine search methods; perhaps specific types of ADDs are more common in specific lifetime phases. Additionally, this might gain insight into common pitfalls or oversights during a certain part of the design phase, leading to architectural problems later in the system's lifetime. Such insight might help practitioners prevent such mistakes.

9.1.3.2 Investigate Concepts Discussed in ADDs

Our first supervisor pitched the idea to use LDA (latent Dirichlet allocation) to determine what kind of topics are discussed in architectural issues. Future research focusing on this could use the classifier generated dataset of issues to investigate on a massive scale, across different domains and ecosystems, what kind of architectural topics software engineers discuss in issue trackers.

9.1.3.3 Examine how ADDs Differ in Different Projects, Domains, and Ecosystems

Future research could analyse whether there is a difference in how architecture is discussed in different projects, domains, and ecosystems. Such research can provide insights in the (possibly different) ways software engineers discuss software architecture.

Such research could also provide insights in ways to improve the generalisability of classifiers. If we have better knowledge of how ADDs differ between projects, domains, and ecosystems, we might be able to come up with techniques to deal with those differences. If that is not feasible, it could still provide insights what kind of issues are currently "missing" from the training dataset.

9.1.4 Further Improvements to Maestro

Finally, we propose various ways to improve the Maestro tool. This work is less research focus, but more programming focused.

9.1.4.1 Evaluate Usefulness for Practitioners

Currently, Maestro has only been used in research to expand our dataset of architectural issues, and for a statistical analysis of all architectural issues identified using BERT. The latter work was done by fellow student Sarah Druyts for her bachelor's thesis. This means that Maestro has only been used by researchers. However, in [53], we mention how Maestro is meant for both researchers and practitioners. Hence, future work could investigate the usefulness of Maestro for practitioners. Specifically, future work could investigate whether Maestro is useful for rediscovering and re-using architectural knowledge when solving specific problems.

9.1.4.2 Refactor the Tool to Improve Re-usability, Extendability, and Scalability

Throughout its development, Maestro has undergone various refactoring rounds. However, there are still various problems which need to be addressed. Here, we present a list of desired changes. However, to fully understand these, we recommend the reader first familiarise themselves with the architecture of Maestro[29].

- **Refactor how testing features and features for prediction are generated**:

  When generating features for the test set or predictions, we need to generate the features with the same settings as the training set. The feature generation code was designed so that by default, it uses the configuration options which would be set when training a model. When generating features for testing or prediction, we have to pass in a different configuration state, and everywhere in the code, we have to check which state should be used. This is brittle, difficult to maintain, and should be refactored.

- **Make the database more scalable**:

  Currently, some operations in the database are particularly slow. Especially 'join' operations are slow in MongoDB. In order to speed these up, we use indexes. However, the current database schema requires many indices, causing the database to hit its index limit. Either the database schema should be refactored to use fewer indices (and ideally improve performance), or the API should be ported to use a different database (e.g. an SQL database).

- **Refactor model saving**:

  The current implementation used to store trained feature generators and models is difficult to follow and maintain. This should be refactored to improve maintainability and make it easier to store very complex models (e.g. auto-encoders).

- **Refactor deep learning input encoding handling**:

  Currently, the deep learning code knows about four hard-coded output encodings. Here, an output encoding is what we have been referring to as a "deep learning task". For instance, we have output encoding for detection, multi-class, and multi-label. The current implementation has all these hard-coded, including the class corresponding to each output. We would like to change this to make it dynamically configurable. The work required for this change would be relatively little, but would require significant regression testing. Making this change would be the first major step towards using the deep learning manager as a general deep learning tool, instead of a tool specifically for training deep learning models for finding ADDs.

- **Implement user-defined labels**:

  The only labels which can be assigned to an issue are "non-archictural", "existence", "executive", and "property". We imagine that there are also use cases where users would want to assign other labels; either they could want more find-grained classification, or they could want to design classifiers for different issues (e.g. identify issues specifically mentioning tactics). This would require support in the user interface, deep learning manager, database, database API, and database API client. Doing this would also be the second (and last) change needed to convert the deep learning manager into a general deep learning tool.

- **Per-annotator labels**:

  Currently, an issue can have only one label. If researchers wish to compute agreement scores, they either have to store the labels per annotators elsewhere, or leave automatically identifiable comments containing the labels for each annotator.

  Currently, Maestro provides a utility script for computing inter-annotator agreement[30]. However, this script required annotators to leave a label in the format `label: X/Y/Z`, where X, Y, and Z are labels. This is not very flexible and user-friendly. Per-annotator labels would be a more user-friendly approach.

[29] `https://github.com/mining-design-decisions/Maestro/blob/main/docs/architecture/index.md`

[30] `https://github.com/mining-design-decisions/Maestro/tree/main/agreement`

- **Built-in agreement calculations**:

  Related to the per-annotator labels, is the improvement that agreement calculations could be performed from within the UI, instead of relying on an external script.

- **Automatic form generation and hyperparameter optimisation in the UI**:

  Currently, the user interface of Maestro has hard-coded pages for creating model configurations. However, the deep learning manager has dynamic endpoints for retrieving all possible configuration options available for models. These endpoints could be used to dynamically generate the forms for model creation in the user interface. This would improve the maintainability of the system because changes to the model configurations would not require changes in the user interface.

  Related to this is that the user interface has no option to perform hyperparameter optimisation. The hyperparameter optimisation is analogous to normal model creation, except for one detail: instead of one value, a search space of values is provided for every configuration option. This would also be trivial to implement with dynamic form generation.

- **Improve hybrid search approaches**:

  In its current state, Maestro is mostly focused on separate keyword search and separate deep learning search. It also has very primitive support for a hybrid search: the results of the keyword search can be filtered based on a hard-coded deep learning classifier. Future work could focus on making this classifier configurable, and potentially investigating the usefulness of such configurability. Additionally, future work could focus on implementing filtering based on multiple classifiers.

- **Quality of life changes to the user interface**:

  Some elements in the user interface (e.g. the tags associated with an issue) take up a lot of space. This should be changed to reduce visual noise. Additionally, inputting long queries is cumbersome with the current interface. A general set of quality of life improvements to the user interface could greatly improve the user experience.

- **Trained model deprecation policy**:

  Trained models are currently stored in the database. Certain changes in the deep learning manager, such as the addition of a new parameter to a feature generator, may cause older trained models to become unusable with the new version of Maestro. Currently, the system does not automatically handle this. This should be changed; there should be a system to inspect and invalidate outdated trained models. This is currently a non-trivial change, because schema-wise, the predictions made by these outdated models would also be deleted if these models were deleted from the database.

10 Acknowledgements

- The authors would like to thank the first supervisor, Mohamed Soliman, for the assistance with issue labelling, for checking various drafts of the thesis, and overall for the guidance throughout the entire project.

- The authors would like to thank Sarah Druyts for developing the user interface for Maestro, and for performing work on the classification of projects in the database into domains.

- The authors would like to thank the second supervisor, Paris Avgeriou, for the discussion during the master's symposium, which has lead to various additions to the discussion in this work. Additionally, Paris Avgeriou funded our cloud VM we used for hosting our online database, for which we are very grateful.

- The authors would like to thank the Center for Information Technology of the University of Groningen for providing access to the Hábrók high performance computing cluster.

References

[1] Manjunatha Badiger and Jose Alex Mathew. “Retrospective Review of Activation Functions in Artificial Neural Networks”. en. In: *Proceedings of Third International Conference on Communication, Computing and Electronics Systems*. Ed. by V. Bindhu, João Manuel R. S. Tavares, and Ke-Lin Du. Lecture Notes in Electrical Engineering. Singapore: Springer, 2022, pp. 905–919. ISBN: 9789811688621. DOI: 10.1007/978-981-16-8862-1_59.

[2] Grzegorz Baron. “Comparison of cross-validation and test sets approaches to evaluation of classifiers in authorship attribution domain”. In: *Computer and Information Sciences: 31st International Symposium, ISCIS 2016, Kraków, Poland, October 27–28, 2016, Proceedings 31*. Springer, 2016, pp. 81–89.

[3] Manoj Bhat et al. “ADeX: A Tool for Automatic Curation of Design Decision Knowledge for Architectural Decision Recommendations”. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. Mar. 2019, pp. 158–161. DOI: 10.1109/ICSA-C.2019.00035.

[4] Manoj Bhat et al. “Automatic extraction of design decisions from issue management systems: A machine learning based approach”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10475 LNCS (2017). Publisher: Springer Verlag ISBN: 9783319658308, pp. 138–154. ISSN: 16113349. DOI: 10.1007/978-3-319-65831-5_10/COVER. URL: https://link-springer-com.proxy-ub.rug.nl/chapter/10.1007/978-3-319-65831-5_10 (visited on 02/15/2023).

[5] Tingting Bi et al. “Architecture information communication in two OSS projects: The why, who, when, and what”. en. In: *Journal of Systems and Software* 181 (Nov. 2021), p. 111035. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111035. URL: https://www.sciencedirect.com/science/article/pii/S0164121221001321 (visited on 07/20/2023).

[6] Tingting Bi et al. “Mining Architecture Tactics and Quality Attributes knowledge in Stack Overflow”. en. In: *Journal of Systems and Software* 180 (Oct. 2021), p. 111005. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111005. URL: https://www.sciencedirect.com/science/article/pii/S0164121221001023 (visited on 07/20/2023).

[7] Michael Biehl. *The Shallow and the Deep: A biased introduction to neural networks and old school machine learning*. University of Groningen, 2022.

[8] Nils Bjorck et al. “Understanding Batch Normalization”. In: *Advances in Neural Information Processing Systems*. Vol. 31. Curran Associates, Inc., 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/36072923bfc3cf47745d704feb489480-Abstract.html (visited on 06/15/2023).

[9] Rafael Capilla et al. “10 years of software architecture knowledge management: Practice and future”. en. In: *Journal of Systems and Software* 116 (Sept. 2015), pp. 191–205. ISSN: 01641212. DOI: 10.1016/j.jss.2015.08.054. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121215002034 (visited on 06/09/2023).

[10] Kyunghyun Cho et al. *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*. arXiv:1409.1259 [cs, stat]. Oct. 2014. DOI: 10.48550/arXiv.1409.1259. URL: http://arxiv.org/abs/1409.1259 (visited on 06/08/2023).

[11] Dami Choi et al. *On Empirical Comparisons of Optimizers for Deep Learning*. en. Oct. 2019. URL: https://arxiv.org/abs/1910.05446v3 (visited on 06/13/2023).

[12] Junyoung Chung et al. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. arXiv:1412.3555 [cs]. Dec. 2014. DOI: 10.48550/arXiv.1412.3555. URL: http://arxiv.org/abs/1412.3555 (visited on 06/08/2023).

[13] Lorenzo Ciampiconi et al. *A survey and taxonomy of loss functions in machine learning*. arXiv:2301.05579 [cs]. Jan. 2023. DOI: 10.48550/arXiv.2301.05579. URL: http://arxiv.org/abs/2301.05579 (visited on 05/29/2023).

[14] Jacob Cohen. “A Coefficient of Agreement for Nominal Scales”. en. In: *Educational and Psychological Measurement* 20.1 (Apr. 1960). Publisher: SAGE Publications Inc, pp. 37–46. ISSN: 0013-1644. DOI: 10.1177/001316446002000104. URL: https://doi.org/10.1177/001316446002000104 (visited on 06/16/2023).

[15] Tim Cooijmans et al. *Recurrent Batch Normalization*. arXiv:1603.09025 [cs]. Feb. 2017. DOI: 10.48550/arXiv.1603.09025. URL: http://arxiv.org/abs/1603.09025 (visited on 06/14/2023).

[16] Koby Crammer and Yoram Singer. “On the algorithmic implementation of multiclass kernel-based vector machines”. In: *The Journal of Machine Learning Research* 2 (Mar. 2002), pp. 265–292. ISSN: 1532-4435.

[17] Arjan Dekker. “Exploring Technology Design Decisions in Issue Tracking Systems”. en. BSc Thesis. University of Groningen, 2021. URL: https://fse.studenttheses.ub.rug.nl/25456/ (visited on 07/28/2023).

[18] Arjan Dekker and Jesse Maarleveld. “Mining for Architectural Design Decisions in Issue Tracking Systems using Deep Learning Approaches”. MSc Internship Report. Groningen: University of Groningen, 2022. URL: https://fse.studenttheses.ub.rug.nl/28689/ (visited on 03/28/2023).

[19] Arjan Dekker and Jesse Maarleveld. *Mining for Architectural Design Decisions in Issue Tracking Systems using Deep Learning Approaches - Errata*. URL: https://github.com/mining-design-decisions/mining-design-decisions.

[20] Arjan Dekker et al. *Detecting & Classifying Architectural Design Decisions in Issues from Issue Trackers using Issue Properties*. Semester Report. University of Groningen, Feb. 2023. URL: https://github.com/mining-design-decisions/exploring-issue-properties.

[21] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv:1810.04805 [cs]. May 2019. DOI: 10.48550/arXiv.1810.04805. URL: http://arxiv.org/abs/1810.04805 (visited on 05/16/2023).

[22] Musengamana Jean de Dieu et al. *Mining Architectural Information: A Systematic Mapping Study*. arXiv:2212.13179 [cs]. June 2023. DOI: 10.48550/arXiv.2212.13179. URL: http://arxiv.org/abs/2212.13179 (visited on 07/20/2023).

[23] Wei Ding et al. “Understanding the Causes of Architecture Changes Using OSS Mailing Lists”. en. In: *International Journal of Software Engineering and Knowledge Engineering* 25.09n10 (Nov. 2015), pp. 1633–1651. ISSN: 0218-1940, 1793-6403. DOI: 10.1142/S0218194015400367. URL: https://www.worldscientific.com/doi/abs/10.1142/S0218194015400367 (visited on 07/20/2023).

[24] Timothy Dozat. “Incorporating Nesterov Momentum into Adam”. en. In: (Feb. 2016). URL: https://openreview.net/forum?id=OM0jvwB8jIp57ZJjtNEZ (visited on 06/13/2023).

[25] Sarah Druyts. *Exploring Architectural Design Decisions in Issue Tracking Systems.* Honours Programme Report. University of Groningen, 2022. URL: https://github.com/Shadania/Jira_Arch/blob/main/Jira_Arch_Report.pdf.

[26] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. "Activation functions in deep learning: A comprehensive survey and benchmark". en. In: *Neurocomputing* 503 (Sept. 2022), pp. 92–108. ISSN: 0925-2312. DOI: 10.1016/j.neucom.2022.06.111. URL: https://www.sciencedirect.com/science/article/pii/S0925231222008426 (visited on 06/01/2023).

[27] Vasiliki Efstathiou, Christos Chatzilenas, and Diomidis Spinellis. "Word embeddings for the software engineering domain". In: *Proceedings of the 15th International Conference on Mining Software Repositories.* MSR '18. New York, NY, USA: Association for Computing Machinery, May 2018, pp. 38–41. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196448. URL: https://dl.acm.org/doi/10.1145/3196398.3196448 (visited on 07/03/2023).

[28] Said Faroghi. "Mining architectural knowledge in issue tracking systems". en. BSc Thesis. University of Groningen, 2022. URL: https://fse.studenttheses.ub.rug.nl/26603/ (visited on 07/28/2023).

[29] Peter I. Frazier. *A Tutorial on Bayesian Optimization.* arXiv:1807.02811 [cs, math, stat]. July 2018. DOI: 10.48550/arXiv.1807.02811. URL: http://arxiv.org/abs/1807.02811 (visited on 07/04/2023).

[30] Christian Garbin, Xingquan Zhu, and Oge Marques. "Dropout vs. batch normalization: an empirical study of their impact to deep learning". en. In: *Multimedia Tools and Applications* 79.19 (May 2020), pp. 12777–12815. ISSN: 1573-7721. DOI: 10.1007/s11042-019-08453-9. URL: https://doi.org/10.1007/s11042-019-08453-9 (visited on 06/14/2023).

[31] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* 1st ed. MIT Press, 2016. ISBN: 978-0-262-03561-3. URL: http://www.deeplearningbook.org.

[32] Martin Gütlein et al. "A large-scale empirical evaluation of cross-validation and external test set validation in (Q) SAR". In: *Molecular Informatics* 32.5-6 (2013). Publisher: Wiley Online Library, pp. 516–528.

[33] Moritz Hardt, Benjamin Recht, and Yoram Singer. *Train faster, generalize better: Stability of stochastic gradient descent.* arXiv:1509.01240 [cs, math, stat]. Feb. 2016. DOI: 10.48550/arXiv.1509.01240. URL: http://arxiv.org/abs/1509.01240 (visited on 05/30/2023).

[34] Fengxiang He, Tongliang Liu, and Dacheng Tao. "Control batch size and learning rate to generalize well: Theoretical and empirical evidence". In: *Advances in neural information processing systems* 32 (2019).

[35] Uwe van Heesch and Paris Avgeriou. "Mature architecting-a survey about the reasoning process of professional architects". In: *2011 Ninth Working IEEE/IFIP Conference on Software Architecture.* IEEE, 2011, pp. 260–269.

[36] Uwe van Heesch and Paris Avgeriou. "Mature Architecting - A Survey about the Reasoning Process of Professional Architects". In: *2011 Ninth Working IEEE/IFIP Conference on Software Architecture.* June 2011, pp. 260–269. DOI: 10.1109/WICSA.2011.42.

[37] Dan Hendrycks and Kevin Gimpel. *Gaussian Error Linear Units (GELUs).* arXiv:1606.08415 [cs]. July 2020. DOI: 10.48550/arXiv.1606.08415. URL: http://arxiv.org/abs/1606.08415 (visited on 06/02/2023).

[38] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen". In: *Diploma, Technische Universität München* 91.1 (1991).

[39] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[40] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". en. In: *Proceedings of the 32nd International Conference on Machine Learning.* ISSN: 1938-7228. PMLR, June 2015, pp. 448–456. URL: https://proceedings.mlr.press/v37/ioffe15.html (visited on 06/14/2023).

[41] Li-Ping Jing, Hou-Kuan Huang, and Hong-Bo Shi. "Improved feature selection approach TFIDF in text mining". In: *Proceedings. International Conference on Machine Learning and Cybernetics.* Vol. 2. Nov. 2002, 944–946 vol.2. DOI: 10.1109/ICMLC.2002.1174522.

[42] Justin M. Johnson and Taghi M. Khoshgoftaar. "Survey on deep learning with class imbalance". In: *Journal of Big Data* 6.1 (Dec. 2019). Publisher: SpringerOpen, pp. 1–54. ISSN: 21961115. DOI: 10.1186/S40537-019-0192-5/TABLES/18. URL: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-019-0192-5 (visited on 02/15/2023).

[43] Ibrahem Kandel and Mauro Castelli. "The effect of batch size on the generalizability of the convolutional neural networks on a histopathology dataset". In: *ICT express* 6.4 (2020). Publisher: Elsevier, pp. 312–315.

[44] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* arXiv:1412.6980 [cs]. Jan. 2017. DOI: 10.48550/arXiv.1412.6980. URL: http://arxiv.org/abs/1412.6980 (visited on 06/13/2023).

[45] Philippe Kruchten. "An ontology of architectural design decisions in software intensive systems". In: *2nd Groningen workshop on software variability* (2004). (Visited on 02/15/2023).

[46] Philippe Kruchten, Patricia Lago, and Hans Van Vliet. "Building up and reasoning about architectural knowledge". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4214 LNCS (2006). Publisher: Springer, Berlin, Heidelberg ISBN: 3540488197, pp. 43–58. ISSN: 03029743. DOI: 10.1007/11921998_8/COVER. URL: https://link-springer-com.proxy-ub.rug.nl/chapter/10.1007/11921998_8 (visited on 02/15/2023).

[47] César Laurent et al. *Batch Normalized Recurrent Neural Networks.* arXiv:1510.01378 [cs, stat]. Oct. 2015. DOI: 10.48550/arXiv.1510.01378. URL: http://arxiv.org/abs/1510.01378 (visited on 06/14/2023).

[48] Quoc V. Le and Tomas Mikolov. *Distributed Representations of Sentences and Documents.* arXiv:1405.4053 [cs]. May 2014. DOI: 10.48550/arXiv.1405.4053. URL: http://arxiv.org/abs/1405.4053 (visited on 06/13/2023).

[49] Lisha Li et al. "Hyperband: A novel bandit-based approach to hyperparameter optimization". In: *The journal of machine learning research* 18.1 (2017). Publisher: JMLR. org, pp. 6765–6816.

[50] Xueying Li, Peng Liang, and Tianqing Liu. "Decisions and Their Making in OSS Development: An Exploratory Study Using the Hibernate Developer Mailing List". In: *2019 26th Asia-Pacific Software Engineering Conference (APSEC).* ISSN: 2640-0715. Dec. 2019, pp. 323–330. DOI: 10.1109/APSEC48747.2019.00051.

[51] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization.* arXiv:1711.05101 [cs, math]. Jan. 2019. DOI: 10.48550/arXiv.1711.05101. URL: http://arxiv.org/abs/1711.05101 (visited on 06/13/2023).

[52] Ping Luo et al. *Towards Understanding Regularization in Batch Normalization.* arXiv:1809.00846 [cs, stat]. Apr. 2019. DOI: 10.48550/arXiv.1809.00846. URL: http://arxiv.org/abs/1809.00846 (visited on 06/15/2023).

[53] Jesse Maarleveld et al. "Maestro: A Tool to Find and Explore Architectural Design Decisions in Issue Tracking Systems". In: *Software Architecture. ECSA 2023 Tracks and Workshops.* Lecture Notes in Computer Science. To be published. 2023.

[54] Christian Manteuffel, Paris Avgeriou, and Roelof Hamberg. "An exploratory case study on reusing architecture decisions in software-intensive system projects". en. In: *Journal of Systems and Software* 144 (Oct. 2018), pp. 60–83. ISSN: 0164-1212. DOI: 10.1016/j.jss.2018.05.064. URL: https://www.sciencedirect.com/science/article/pii/S0164121218301110 (visited on 05/16/2023).

[55] Mary L. McHugh. "Interrater reliability: the kappa statistic". eng. In: *Biochemia Medica* 22.3 (2012), pp. 276–282. ISSN: 1330-0962.

[56] Cornelia Miesbauer and Rainer Weinreich. "Classification of Design Decisions – An Expert Survey in Practice". en. In: *Software Architecture.* Ed. by Khalil Drira. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 130–145. ISBN: 978-3-642-39031-9. DOI: 10.1007/978-3-642-39031-9_12.

[57] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space.* arXiv:1301.3781 [cs]. Sept. 2013. DOI: 10.48550/arXiv.1301.3781. URL: http://arxiv.org/abs/1301.3781 (visited on 06/14/2023).

[58] Lloyd Montgomery, Clara Lüders, and Walid Maalej. "An Alternative Issue Tracking Dataset of Public Jira Repositories". In: *Proceedings of the 19th International Conference on Mining Software Repositories.* arXiv:2201.08368 [cs]. May 2022, pp. 73–77. DOI: 10.1145/3524842.3528486. URL: http://arxiv.org/abs/2201.08368 (visited on 03/30/2023).

[59] Anh Nguyen et al. "An Analysis of State-of-the-art Activation Functions For Supervised Deep Neural Network". In: *2021 International Conference on System Science and Engineering (ICSSE).* ISSN: 2325-0925. Aug. 2021, pp. 215–220. DOI: 10.1109/ICSSE52999.2021.9538437.

[60] Chigozie Nwankpa et al. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning.* arXiv:1811.03378 [cs]. Nov. 2018. DOI: 10.48550/arXiv.1811.03378. URL: http://arxiv.org/abs/1811.03378 (visited on 06/01/2023).

[61] Prajit Ramachandran, Barret Zoph, and Quoc Le. "Swish: a Self-Gated Activation Function". In: (Oct. 2017).

[62] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. *Searching for Activation Functions.* arXiv:1710.05941 [cs] version: 1. Oct. 2017. DOI: 10.48550/arXiv.1710.05941. URL: http://arxiv.org/abs/1710.05941 (visited on 06/02/2023).

[63] Masud Rana, Md. Mohsin Uddin, and Md. Mohaimnul Hoque. "Effects of Activation Functions and Optimizers on Stock Price Prediction using LSTM Recurrent Networks". In: *Proceedings of the 2019 3rd International Conference on Computer Science and Artificial Intelligence.* CSAI '19. New York, NY, USA: Association for Computing Machinery, Mar. 2020, pp. 354–358. ISBN: 978-1-4503-7627-3. DOI: 10.1145/3374587.3374622. URL: https://dl.acm.org/doi/10.1145/3374587.3374622 (visited on 06/08/2023).

[64] R. Bharat Rao, Glenn Fung, and Romer Rosales. "On the dangers of cross-validation. An experimental evaluation". In: *Proceedings of the 2008 SIAM international conference on data mining.* SIAM, 2008, pp. 588–596.

[65] Xiaoxue Ren et al. "Neural Network-based Detection of Self-Admitted Technical Debt: From Performance to Explainability". In: *ACM Transactions on Software Engineering and Methodology* 28.3 (July 2019), 15:1–15:45. ISSN: 1049-331X. DOI: 10.1145/3324916. URL: https://dl.acm.org/doi/10.1145/3324916 (visited on 06/23/2023).

[66] M. Schuster and K.K. Paliwal. "Bidirectional recurrent neural networks". In: *IEEE Transactions on Signal Processing* 45.11 (Nov. 1997). Conference Name: IEEE Transactions on Signal Processing, pp. 2673–2681. ISSN: 1941-0476. DOI: 10.1109/78.650093.

[67] Arman Shahbazian, Daye Nam, and Nenad Medvidovic. "Toward predicting architectural significance of implementation issues". In: *Proceedings - International Conference on Software Engineering* (May 2018). Publisher: IEEE Computer Society ISBN: 9781450357166, pp. 215–219. ISSN: 02705257. DOI: 10.1145/3196398.3196440. (Visited on 02/15/2023).

[68] Arman Shahbazian et al. "Recovering Architectural Design Decisions". In: *2018 IEEE International Conference on Software Architecture (ICSA).* Apr. 2018, pp. 95–9509. DOI: 10.1109/ICSA.2018.00019.

[69] Noam Shazeer and Mitchell Stern. *Adafactor: Adaptive Learning Rates with Sublinear Memory Cost.* arXiv:1804.04235 [cs, stat]. Apr. 2018. DOI: 10.48550/arXiv.1804.04235. URL: http://arxiv.org/abs/1804.04235 (visited on 06/14/2023).

[70] Mohamed Soliman, Matthias Galster, and Paris Avgeriou. "An Exploratory Study on Architectural Knowledge in Issue Tracking Systems". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 12857 LNCS (2021). arXiv: 2106.11140 Publisher: Springer Science and Business Media Deutschland GmbH ISBN: 9783030860431, pp. 117–133. ISSN: 16113349. DOI: 10.1007/978-3-030-86044-8_8/FIGURES/3. URL: https://link-springer-com.proxy-ub.rug.nl/chapter/10.1007/978-3-030-86044-8_8 (visited on 02/15/2023).

[71] Mohamed Soliman, Kirsten Gericke, and Paris Avgeriou. "Where and What do Software Architects blog? : An Exploratory Study on Architectural Knowledge in Blogs, and their Relevance to Design Steps". In: *2023 IEEE 20th International Conference on Software Architecture (ICSA).* Mar. 2023, pp. 129–140. DOI: 10.1109/ICSA56044.2023.00020.

[72] Mohamed Soliman et al. "Architectural knowledge for technology decisions in developer communities: An exploratory study with StackOverflow". In: *Proceedings - 2016 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016* (July 2016). Publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781509021314, pp. 128–133. DOI: 10.1109/WICSA.2016.13. (Visited on 02/15/2023).

[73] Mohamed Soliman et al. *Exploring Web Search Engines to Find Architectural Knowledge.* arXiv:2103.11705 [cs]. Mar. 2021. DOI: 10.48550/arXiv.2103.11705. URL: http://arxiv.org/abs/2103.11705 (visited on 07/21/2023).

[74] Mohamed Soliman et al. "Improving the Search for Architecture Knowledge in Online Developer Communities". In: *2018 IEEE International Conference on Software Architecture (ICSA).* Apr. 2018, pp. 186–18609. DOI: 10.1109/ICSA.2018.00028.

[75] Nitish Srivastava et al. “Dropout: a simple way to prevent neural networks from overfitting”. In: *The Journal of Machine Learning Research* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435.

[76] F. Tian et al. “Automatic identification of architecture smell discussions from stack overflow”. en. In: *https://ksiresearch.org/seke/sekeproc.html*. Accepted: 2021-09-27T07:34:16Z ISSN: 2325-9000. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2020. ISBN: 978-1-891706-50-9. DOI: `10.18293/SEKE2020-084`. URL: `https://digital.library.adelaide.edu.au/dspace/handle/2440/132351` (visited on 07/20/2023).

[77] Fangchao Tian, Peng Liang, and Muhammad Ali Babar. “How Developers Discuss Architecture Smells? An Exploratory Study on Stack Overflow”. In: *2019 IEEE International Conference on Software Architecture (ICSA)*. Mar. 2019, pp. 91–100. DOI: `10.1109/ICSA.2019.00018`.

[78] Hans Van Vliet. *Software Engineering: Principles and Practice*. 3rd ed. Jon Wiley & Sons, Ltd, 2008. ISBN: 978-0-470-03146-9.

[79] S. Vani and T. V. Madhusudhana Rao. “An Experimental Approach towards the Performance Assessment of Various Optimizers on Convolutional Neural Network”. In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. Apr. 2019, pp. 331–336. DOI: `10.1109/ICOEI.2019.8862686`.

[80] John Verzani. *Using R for Introductory Statistics*. 2nd ed. CRC Press, 2014. ISBN: 978-1-4665-9073-1.

[81] Giovanni Viviani et al. “Locating Latent Design Information in Developer Discussions: A Study on Pull Requests”. In: *IEEE Transactions on Software Engineering* 47.7 (July 2021). Conference Name: IEEE Transactions on Software Engineering, pp. 1402–1413. ISSN: 1939-3520. DOI: `10.1109/TSE.2019.2924006`.

[82] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. en. Dec. 2012. URL: `https://arxiv.org/abs/1212.5701v1` (visited on 06/13/2023).

[83] Lei Zhao, Musa Mammadov, and John Yearwood. “From Convex to Nonconvex: A Loss Function Analysis for Binary Classification”. In: *2010 IEEE International Conference on Data Mining Workshops*. ISSN: 2375-9259. Dec. 2010, pp. 1281–1288. DOI: `10.1109/ICDMW.2010.57`.

A Coding Book

Coding Book

Arjan Dekker & Jesse Maarleveld

March 2023

1 Preface: Definitions & Terminology

1.1 Basic Definitions

The basic definitions used for Existence, Executive, and Property issues are based on the following paper by Kruchten:

Kruchten, Philippe. 'An Ontology of Architectural Design Decisions in Software Intensive Systems'. 2nd Groningen Workshop on Software Variability, 2004.

For all issue labelling, the basic definitions outlined in that paper should be kept in mind. The rules in this coding book mostly define and substantiate how we interpreted these definitions and how issues should be assigned to them.

1.2 Significant Enough Changes/Significant Effort etc

Often, if an issue is architectural or not depends on the effort involved. What we mean by this, is that something is hard to change. For instance, the addition of a new component is architectural if making large changes later is difficult. Hence, the key question we ask if necessary is "Would it be difficult to change the implementation?".

The following issues were considered existence:

- An addition of a new partition strategy to Cassandra: CASSANDRA-8866
- Blacklisting of ill-performing nodes in Hadoop: HDFS-289
- Encryption of SSTables in Cassandra: CASSANDRA-9633
- Refactoring to expose an interface: HADOOP-15038
- CASSANDRA-5283
- YARN-4619
- CASSANDRA-4011
- CASSANDRA-14213

The following issues were considered architectural, but not existence:

- Changes to memory handling in HDFS:
- Moving some functionality to a separate thread: HDFS-16016
- Throttling in HDFS: HDFS-9723
- Optimisation of small repair streams (seems major, but not enough conclusive evidence to warrant existence): CASSANDRA-13290
- Better load distribution among threats: CASSANDRA-4292
- Umbrella for OS-level optimizations: HADOOP-7714
- CASSANDRA-12104
- /CASSANDRA-13291

2 Guidelines for Certain Issue Types

- User requests should be evaluated based on the types of decisions discussed in the request.

3 Architectural

ARCH-1 Use-cases/requirements are architectural (HADOOP-9659)

ARCH-2 Umbrella tasks that contain many links to subtasks are architectural. Depending on the focus of the task, it can be existence, property and/or executive (HADOOP-15977 , HADOOP-1771777)

4 Existence

EXI-1 Large behavioural changes to commands/operations are existence (HADOOP-15845 , YARN-613)

EXI-2 The introduction of tactics is existence if the implementation requires the addition/modification/removal of components or behaviour between components (HADOOP-18458)

EXI-3 The addition of major new functionality is existence, even if the implementation might be relatively straightforward (CASSANDRA-17059)

EXI-4 Refactoring may be existence if done on a scale where many places/components are affected (CASSANDRA-8609 , HDFS-6315)

EXI-5 Property issues which motivate the addition/removal/modification of existing components or interactions are often also existence, given that the changes are sufficiently large (CASSANDRA-8609 , YARN-8673)

EXI-6 New features with significant implementation challenges/non-trivial implementation/large implementation effort are often existence (CASSANDRA-1339)

EXI-7 Major restructuring/sub-project creation is existence (CASSANDRA-1228)

EXI-8 The implementation of large utility tools is existence (HDFS-8968)

EXI-9 Parallelism is often existence, unless the implementation is trivial (CASSANDRA-2901)

EXI-10 Changes to the interaction between components is existence (HDFS-7607 , CASSANDRA-8345 , CASSANDRA-6752 , CASSANDRA-4761)

EXI-11 Changes spanning multiple components are existence (CASSANDRA-9633)

EXI-12 Vague or small change descriptions are still existence, if components and/or protocols are mentioned (MAPREDUCE-5189 , HDFS-2181 , YARN-3409)

EXI-13 Large non-trivial code changes (in patch) are existence (YARN-6620 , YARN-3998)

EXI-14 Issues without patch if changes to components and/or protocols are described (HDFS-6658)

NON-EXI-1 Trivial code movement is not architectural (YARN-2107), unless a proper architectural reason is specified (HADOOP-9649), or the code being moved is large enough (HDFS-12259).

NON-EXI-2 Small changes to internal interfaces are not existence (CASSANDRA-6248)

NON-EXI-3 Very small interface additions (e.g. "hookds") are not existence (CASSANDRA-5545)

NON-EXI-4 Very vague change descriptions not mentioning components and/or protocls are not existence (HADOOP-1986)

5 Executive

NB: Protocols, tools, hardware and interfaces can be considered a technology

EXEC-1 Issue deals with external technologies (e.g. support for technology Y, CASSANDRA-1193) are executive, unless the changes are minor (CASSANDRA-11519). However, in case much effort is needed to analyse the effects of such changes is large, the issue can also be considered executive (CASSANDRA-3031)

EXEC-2 Dependency additions/upgrades/removals are executive (HADOOP-17947 , CASSANDRA-13291)

EXEC-3 Creation of utility tools for development (e.g. benchmarking) is executive (HDFS-8968 , HADOOP-12725)

EXEC-4 External code contributions (e.g. merging existing projects) is executive (HADOOP-2878 , YARN-2670)

EXEC-5 Requests or proposals from companies or other users to support technologies is executive (CASSANDRA-11703 , HADOOP-9484)

NON-EXEC-1 Code to work around some flaw in a dependency, external tool etc. is not executive (HADOOP-17597)

NON-EXEC-2 User requests are not executive (HADOOP-17268)

6 Property

PROP-1 Quality attribute enhancements are often property, if applied on a high-enough level (e.g. parallelism, HTTPS connection re-use), and provided that the improvement is one of the main goals of the issue (not mentioned off-handedly) (TAJO-1970). Note that improving quality attributes is sometimes implicitly described by words such as refactoring.

PROP-2 Discussions (as in, discussion without implementation) on quality attributes (e.g. security) are property if the change being is major enough (e.g. implementation would be considered existence), even if in the end no implementation is done or proposed. Investigations also fall under this category. (CASSANDRA-7129 , CASSANDRA-7045)

PROP-3 Refactoring might be property if done on a system-level (i.e. implication in many places), and/or expressed in terms of quality attributes (CASSANDRA-8609 , HDFS-2353)

PROP-4 The implementation of tools/utilities meant to measure (and later improve) quality attributes is property (HDFS-8968)

PROP-5 Parallelism is in general property if done for performance, even if the implementation is simple (CASSANDRA-2901)

PROP-6 Umbrella issues focusing on the improvement of quality attributes are property, even if no clear decisions are made in the issue itself. (MAPREDUCE-563 , YARN-2745)

PROP-7 Mitigation of security vulnerabilities is property (CASSANDRA-15121 , TAJO-1214), as well as changing security settings (HADOOP-16779)

PROP-8 Reporting a problem w.r.t. a quality attribute is property (HADOOP-55)

PROP-9 Usability issues are generally property (HDFS-2849)

NON-PROP-1 A single interface or API being X (where X is a quality attribute) is not immediately property (CASSANDRA-754)

NON-PROP-2 Backwards-compatibility related issues are generally not property (CASSANDRA-10990)

NON-PROP-3 Quality attribute improvement or discussion is not property if the change is trivial (e.g. configuration change) (CASSANDRA-14678)

NON-PROP-4 Indirect improvements ("this change would enable the development of faster algorithms") are not property (TAJO-196)

7 Non-Architectural

NON-ARCH-1 Not any of the above

NON-ARCH-2 Small changes are non-architectural

NON-ARCH-3 Configuration changes are not architectural, even if this may lead to the ability to use more/different technologies (HADOOP-14417)

NON-ARCH-4 The addition of small utility tools which do not provide or require novel functionality, is not architectural (SOLR-11179)

NON-ARCH-5 License/legal use clarification of source code is not architectural (CLOUDSTACK-161)

NON-ARCH-6 Small changes in/additions of error handling are not architectural, even though it may lead to new behaviour in case the error is encountered (SOLR-3505 , CASSANDRA-18042)

NON-ARCH-7 Small utility additions/new features need not be architectural, even though they introduce new parts of a public interface (SOLR-10485 , SOLR-11338)

NON-ARCH-8 Tests/QA are non architectural CLOUDSTACK-1000)

NON-ARCH-9 Formalities/release guidelines are not architectural (JSPWIKI-559)

NON-ARCH-10 (Detailed) implementation issues without design discussion are not architectural (HDFS-5616)

NON-ARCH-11 Coding standard issues (e.g. "not compliant with SQL syntax") are not architectural (TAJO-1970)

NON-ARCH-12 Minor behavioural clarifications are not architectural (CASSANDRA-9131)

NON-ARCH-13 Umbrella issues without their own decisions or implementation details are non-architectural, even if the large effort being coordinated requires substantial architectural changes.

NON-ARCH-14 Small code refactorings are non-architectural.

NON-ARCH-15 Small bugs/bug reports are non-architectural, given that the solution is a relatively straightforward code change.

NON-ARCH-16 "TODO Lists" are non-architectural (HADOOP-5064)

B Detailed Result of Issue Labelling

In this appendix, we have various figures describing the result of labelling issues in more detail. Figures 41 and 42 display the amounts of issues we found per label in the initial expansion of our dataset. The first figure gives the amounts before relabelling; the second figure the amount after relabelling. The second figures thus shows how the relabelling has changed the results of our data collections efforts.

Next, Figures 43, 44, and 45 provide detailed statistics regarding the agreement while labelling. Figure 43 gives the agreement on the initial set of issues we collected to expand the dataset, before performing relabelling. Figure 44 presents the agreement on the issues we relabelled during the relabelling process. Finally, Figure 45 presents the agreement on the random samples from the six domains.

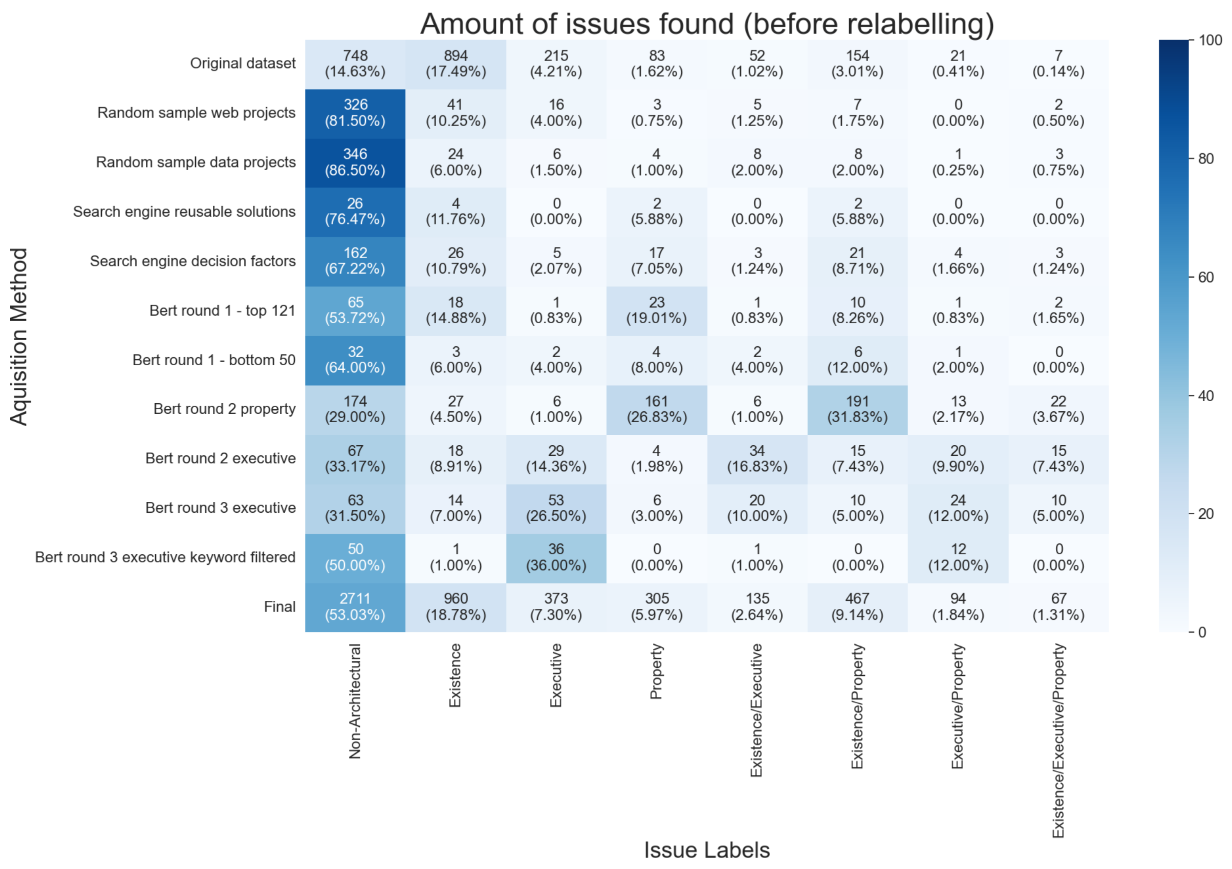
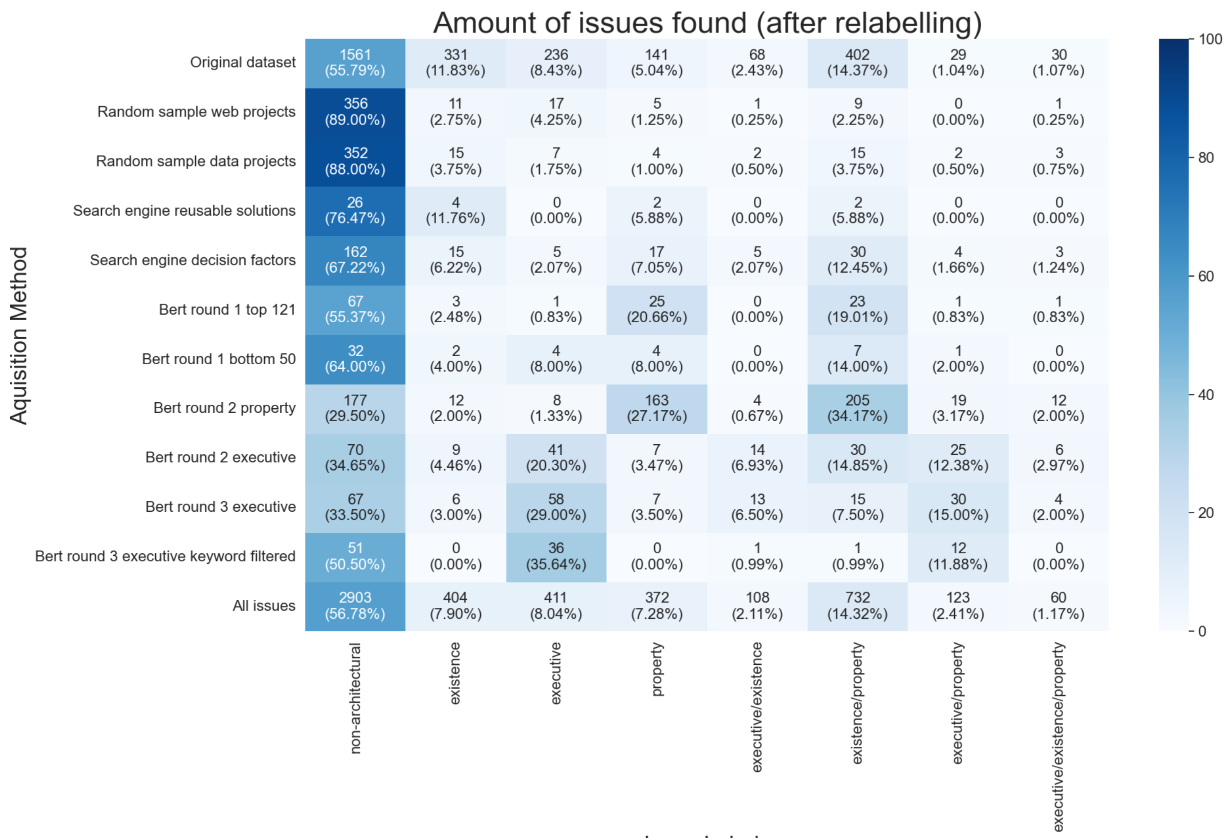


Fig. 41. Issues found per labelling round before relabelling.



	non-architectural	existence	executive	property	executive/existence	existence/property	executive/property	executive/existence/property
Original dataset	1561 (55.79%)	331 (11.83%)	236 (8.43%)	141 (5.04%)	68 (2.43%)	402 (14.37%)	29 (1.04%)	30 (1.07%)
Random sample web projects	356 (89.00%)	11 (2.75%)	17 (4.25%)	5 (1.25%)	1 (0.25%)	9 (2.25%)	0 (0.00%)	1 (0.25%)
Random sample data projects	352 (88.00%)	15 (3.75%)	7 (1.75%)	4 (1.00%)	2 (0.50%)	15 (3.75%)	2 (0.50%)	3 (0.75%)
Search engine reusable solutions	26 (76.47%)	4 (11.76%)	0 (0.00%)	2 (5.88%)	0 (0.00%)	2 (5.88%)	0 (0.00%)	0 (0.00%)
Search engine decision factors	162 (67.22%)	15 (6.22%)	5 (2.07%)	17 (7.05%)	5 (2.07%)	30 (12.45%)	4 (1.66%)	3 (1.24%)
Bert round 1 top 121	67 (55.37%)	3 (2.48%)	1 (0.83%)	25 (20.66%)	0 (0.00%)	23 (19.01%)	1 (0.83%)	1 (0.83%)
Bert round 1 bottom 50	32 (64.00%)	2 (4.00%)	4 (8.00%)	4 (8.00%)	0 (0.00%)	7 (14.00%)	1 (2.00%)	0 (0.00%)
Bert round 2 property	177 (29.50%)	12 (2.00%)	8 (1.33%)	163 (27.17%)	4 (0.67%)	205 (34.17%)	19 (3.17%)	12 (2.00%)
Bert round 2 executive	70 (34.65%)	9 (4.46%)	41 (20.30%)	7 (3.47%)	14 (6.93%)	30 (14.85%)	25 (12.38%)	6 (2.97%)
Bert round 3 executive	67 (33.50%)	6 (3.00%)	58 (29.00%)	7 (3.50%)	13 (6.50%)	15 (7.50%)	30 (15.00%)	4 (2.00%)
Bert round 3 executive keyword filtered	51 (50.50%)	0 (0.00%)	36 (35.64%)	0 (0.00%)	1 (0.99%)	1 (0.99%)	12 (11.88%)	0 (0.00%)
All issues	2903 (56.78%)	404 (7.90%)	411 (8.04%)	372 (7.28%)	108 (2.11%)	732 (14.32%)	123 (2.41%)	60 (1.17%)

Fig. 42. Issues found per labelling round, after relabelling. This is thus a version of Figure 41 adjusted for the relabelling we performed. This figure gives an overview of how relabelling changed the amounts of issues found per class.

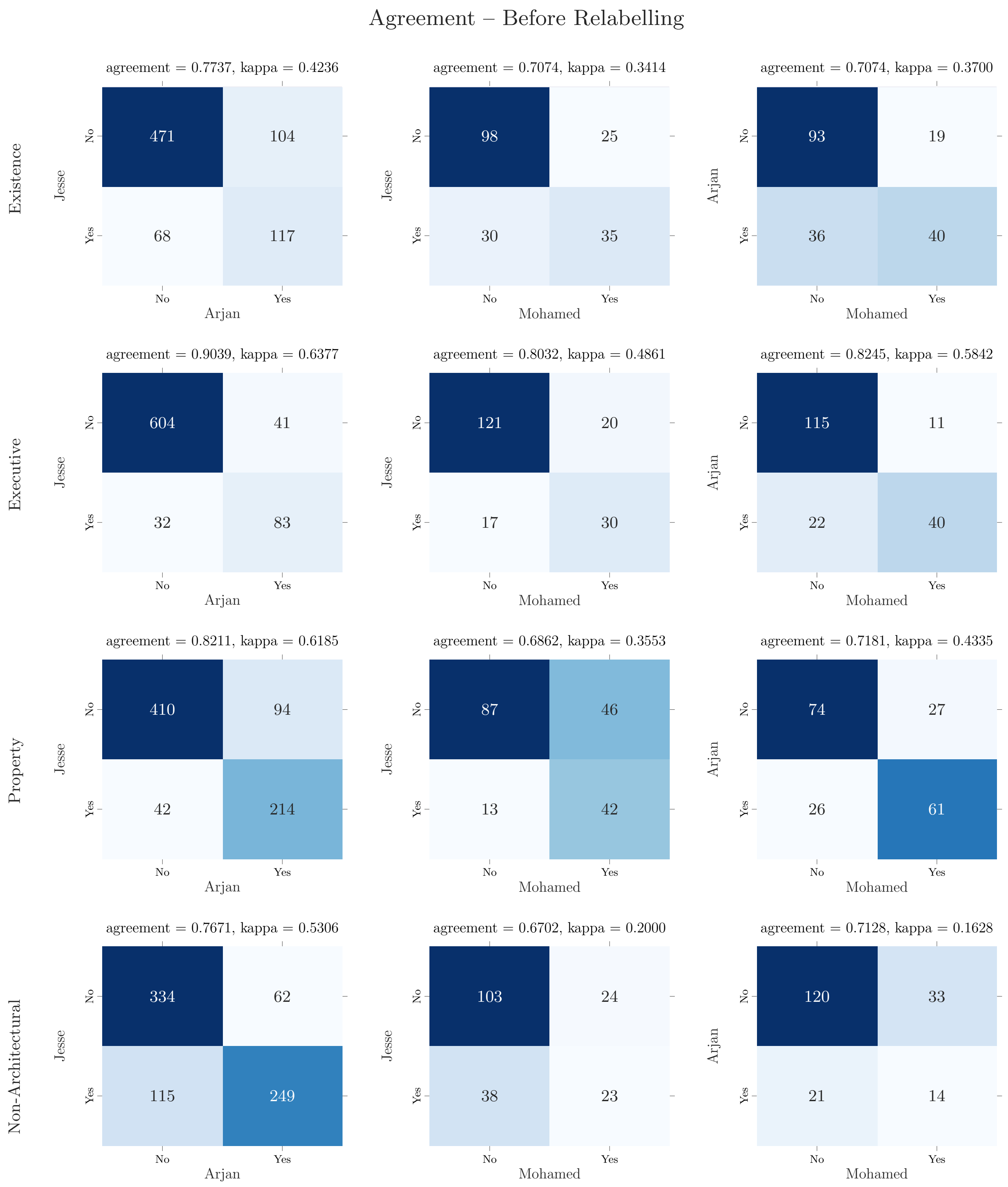
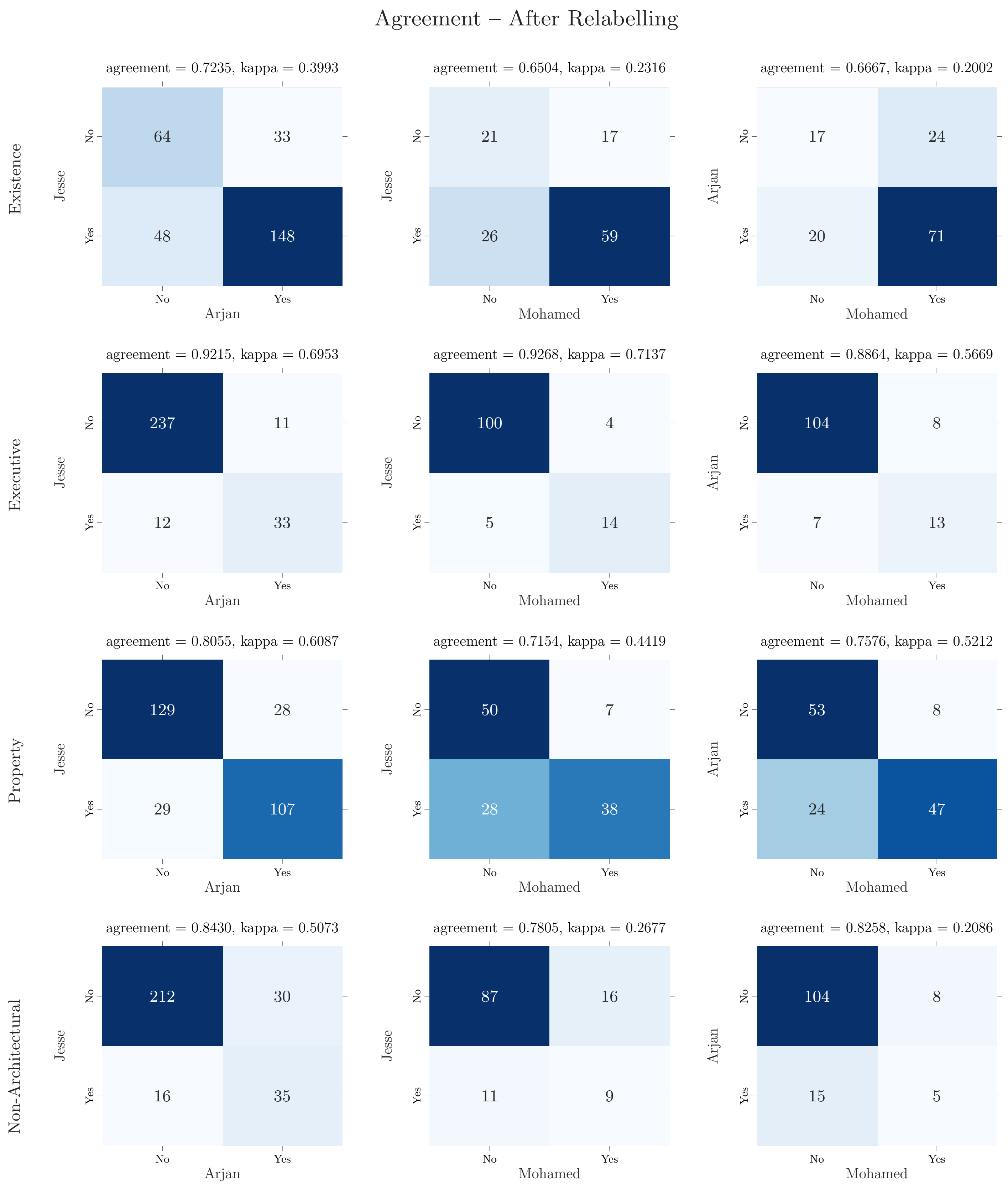


Fig. 43. Agreement after issue collection (before relabelling)



Category	Pair (rows / columns)	No/No	No/Yes	Yes/No	Yes/Yes
Existence	Jesse / Arjan	64	33	48	148
Existence	Jesse / Mohamed	21	17	26	59
Existence	Arjan / Mohamed	17	24	20	71
Executive	Jesse / Arjan	237	11	12	33
Executive	Jesse / Mohamed	100	4	5	14
Executive	Arjan / Mohamed	104	8	7	13
Property	Jesse / Arjan	129	28	29	107
Property	Jesse / Mohamed	50	7	28	38
Property	Arjan / Mohamed	53	8	24	47
Non-Architectural	Jesse / Arjan	212	30	16	35
Non-Architectural	Jesse / Mohamed	87	16	11	9
Non-Architectural	Arjan / Mohamed	104	8	15	5

Fig. 44. Agreement on the relabelled issues.

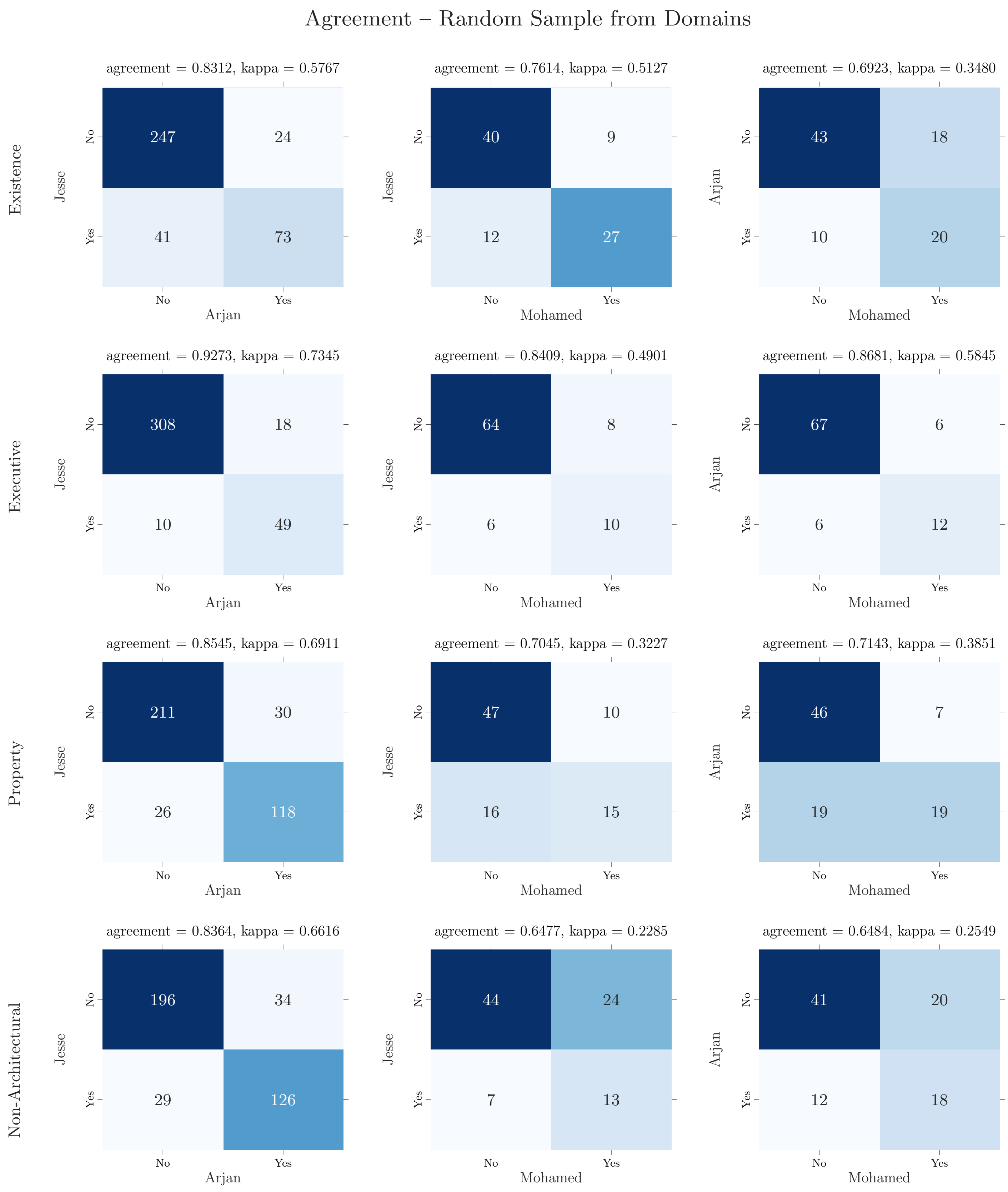


Fig. 45. Agreement on the issues sampled from the six domains.

C Hyperparameter Tables

This section start with an overview of the hyperparameters that we could optimise. This table is followed by the tables for the hyperparameters we optimised for each individual model, including the best hyperparameter values we found for each model during the optimisation process.

Hyperparameter(s)	Model Type(s)	Default value	Description
batch size	ALL	32	How many input samples are used for each training step.
number of dense layers	CNN, RNN, FNN	–	The number of dense layers in the model.
dense layer activation	CNN, RNN, FNN	linear	The activation function to use for all dense layers in the model. Note that the activation function is the same for each dense layer.
dense layer activation alpha	CNN, RNN, FNN	0	Alpha value for the ELU and LeakyReLU activations.
kernel l1	CNN, RNN, FNN	0	L1 regularisation for the kernel weights.
bias l1	CNN, RNN, FNN	0	L1 regularisation for the bias weights.
activity l1	CNN, RNN, FNN	0	L1 regularisation for the activity.
dense layer i size	CNN, RNN, FNN	–	Number of units in the i-th dense layer.
dense layer i dropout	CNN, RNN, FNN	0	Dropout for the i-th dense layer.
number of convolutions	CNN	–	Number of convolution kernels.
convolution i size	CNN	–	Size of the i-th convolution kernel.
filters	CNN	–	Number of filters for each convolution.
convolution activation	CNN	–	Activation of the kernel layers.
convolution activation alpha	CNN	–	Alpha value for the ELU and LeakyReLU activations.
convolution kernel l1	CNN	0	L1 regularisation for the kernel weights.
convolution bias l1	CNN	0	L1 regularisation for the bias weights.
convolution activity l1	CNN	0	L1 regularisation for the activity.
convolution batch normalisation	CNN	false	Batch normalisation for the convolution layers.
convolution i batch normalisation momentum	CNN	0.99	Momentum for the batch normalisation.
number of rnn layers	RNN	–	Number of RNN layers
rnn layer i type	RNN	–	The type of layer (i.e. LSTM or GRU) for the i-th rnn layer. Note that this can be different for each RNN layer.
rnn layer i size	RNN	–	Size of the i-th RNN layer. Note that this can be different for each RNN layer.
optimiser	ALL	adam	Optimisation algorithm used for training.
weight decay	CNN, RNN, FNN	0.0	A parameter of the optimiser that determines by how much large weights in the network are being penalised.
beta-1	ALL	0.9	Initial decay rate for the first moment of the gradient.
beta-2	ALL	0.999	Initial decay rate for the second moment of the gradient.
epsilon	ALL	1e-7	Epsilon value for the adam/nadam/adamw optimisers.
learning rate start	ALL	5e-3	Determines by how much the weights can change during a training step at the start of training.
learning rate stop	ALL	5e-4	Determines by how much the weights can change during a training step after a specified number of training steps have been completed (see next hyperparameter).
learning rate steps	ALL	470	How many training steps it takes to go from *learning rate start* to *learning rate stop*. A higher value means that more steps are required for the learning rate to decay.
learning rate power	ALL	1	Determines the shape of the learning rate decay. A power of 2 means a polynomial decay with power 2. The result is that initially the learning rate decreases rapidly, and after that it keeps decreasing more slowly.
epochs	ALL	–	Number of epochs to train the model.
early stopping min delta	ALL	–	Minimum performance improvement required to not trigger early stopping.
early stopping patience	ALL	–	Number of epochs without improvement to stop training.
loss	ALL	crossentropy	Loss function that is being minimized during training.
max tokens	BERT	512	The amount of tokens each input sample has. This only works for BERT.
padding	BERT	true	Issues that are shorter than 512 tokens are padded using special tokens. This only works for BERT.
truncation	BERT	true	Issues that cannot be encoded into 512 tokens are being truncated. This is only works for BERT.
number of frozen layers	BERT	0	How many of the BERT encoder layers are being frozen.

Table 32. This table contains an overview of the hyperparameters that we could optimise. Besides the default value, it also contains a short description for each hyperparameter.

Hyperparameter	Search space	Best Value
batch size	16, 32	16
number of frozen layers	–	0
optimiser	–	adam
beta 1	–	0.9
beta 2	–	0.999
epsilon	–	1e-7
weight decay	–	0.01
learning rate start	2e-5, 3e-5, 5e-5	3e-5
learning rate stop	–	equal to learning rate start
learning rate steps	–	value does not matter
learning rate power	–	0
epochs	2, 3, 4	4
loss	–	crossentropy
max tokens	–	512
padding	–	true
truncation	–	true

Table 33. Hyperparameters that we tuned for BERT, including the best values we found after tuning.

Hyperparameter	Search space	Best Value
batch size	8, 16, 32, . . . , 512	256
number of dense layers	1 - 5	2
dense layer activation	linear, relu, elu, tanh, softsign, prelu, selu, gelu, swish, softplus	swish
dense layer activation alpha	0.01 - 1.0*	–
kernel l1	1e-10 - 0.1*	3.5203e-6
bias l1	1e-10 - 0.1*	8.0251e-7
activation l1	1e-10 - 0.1*	5.6641e-4
dense layer 1 size	2, 4, 8, ..., 2048	8
dense layer 1 dropout	0.0, 0.05, ..., 0.5	0.05
dense layer 2 size	2, 4, 8, ..., 2048	1024
dense layer 2 dropout	0.0, 0.05, ..., 0.5	0
optimiser	adamw, nadam	nadam
weight decay	1e-10 - 0.5*	1.0112e-5
beta-1	1e-10 - 0.999*	6.2501e-10
beta-2	1e-10 - 0.999*	2.0632e-4
epsilon	1e-10 - 0.01*	1.5045e-9
learning rate start	0.01 - 1.0*	2.3883e-2
learning rate stop	1e-5 - 0.1*	3.1211e-2
learning rate steps	1 - 10000*	33
learning rate power	1 - 2	1.8592
epochs	–	200
early stopping min delta	–	0.01
early stopping patience	–	5
loss	crossentropy, hinge	crossentropy

Table 34. Hyperparameters that we tuned for BOWF, including the best values we found after tuning. Search spaces with a * are sampled using a logarithmic function.

Hyperparameter	Search space	Best Value
batch size	8, 16, 32, . . . , 512	16
number of dense layers	1 - 5	1
dense layer activation	linear, relu, elu, tanh, softsign, prelu, selu, gelu, swish, softplus	relu
dense layer activation alpha	0.01 - 1.0*	–
kernel l1	1e-10 - 0.1*	3.6096e-8
bias l1	1e-10 - 0.1*	4.1990e-9
activation l1	1e-10 - 0.1*	8.9417e-5
dense layer 1 size	2, 4, 8, ..., 2048	512
dense layer 1 dropout	0.0, 0.05, ..., 0.5	0.05
optimiser	adamw, nadam	nadam
weight decay	1e-10 - 0.5*	9.3321e-5
beta-1	1e-10 - 0.999*	9.0221e-2
beta-2	1e-10 - 0.999*	7.0132e-9
epsilon	1e-10 - 0.01*	1.3311e-4
learning rate start	0.01 - 1.0*	4.4118e-2
learning rate stop	1e-5 - 0.1*	2.2659e-3
learning rate steps	1 - 10000*	631
learning rate power	1 - 2	1.7064
epochs	–	200
early stopping min delta	–	0.01
early stopping patience	–	5
loss	crossentropy, hinge	crossentropy

Table 35. Hyperparameters that we tuned for BOWN, including the best values we found after tuning. Search spaces with a * are sampled using a logarithmic function.

Hyperparameter	Search space	Best Value
batch size	8, 16, 32, . . . , 512	16
number of dense layers	1 - 5	1
dense layer activation	linear, relu, elu, tanh, softsign, prelu, selu, gelu, swish, softplus	relu
dense layer activation alpha	0.01 - 1.0*	–
kernel l1	1e-10 - 0.1*	3.6096e-8
bias l1	1e-10 - 0.1*	4.1990e-9
activation l1	1e-10 - 0.1*	8.9417e-5
dense layer 1 size	2, 4, 8, ..., 2048	512
dense layer 1 dropout	0.0, 0.05, ..., 0.5	0.05
optimiser	adamw, nadam	nadam
weight decay	1e-10 - 0.5*	9.3321e-5
beta-1	1e-10 - 0.999*	9.0221e-2
beta-2	1e-10 - 0.999*	7.0132e-9
epsilon	1e-10 - 0.01*	1.3311e-4
learning rate start	0.01 - 1.0*	4.4118e-2
learning rate stop	1e-5 - 0.1*	2.2659e-3
learning rate steps	1 - 10000*	631
learning rate power	1 - 2	1.7064
epochs	–	200
early stopping min delta	–	0.01
early stopping patience	–	5
loss	crossentropy, hinge	crossentropy

Table 36. Hyperparameters that we tuned for TF-IDF, including the best values we found after tuning. Search spaces with a * are sampled using a logarithmic function.

Hyperparameter	Search space	Best Value
batch size	8, 16, 32, . . . , 512	32
number of dense layers	1 - 5	4
dense layer activation	linear, relu, elu, tanh, softsign, prelu, selu, gelu, swish, softplus	tanh
dense layer activation alpha	0.01 - 1.0*	–
kernel l1	1e-10 - 0.1*	3.3919e-9
bias l1	1e-10 - 0.1*	1.3285e-7
activation l1	1e-10 - 0.1*	2.5887e-10
dense layer 1 size	2, 4, 8, ..., 2048	512
dense layer 1 dropout	0.0, 0.05, ..., 0.5	0.25
dense layer 2 size	2, 4, 8, ..., 2048	2048
dense layer 2 dropout	0.0, 0.05, ..., 0.5	0.05
dense layer 3 size	2, 4, 8, ..., 2048	32
dense layer 3 dropout	0.0, 0.05, ..., 0.5	0.4
dense layer 4 size	2, 4, 8, ..., 2048	2048
dense layer 4 dropout	0.0, 0.05, ..., 0.5	0
optimiser	adamw, nadam	nadam
weight decay	1e-10 - 0.5*	4.9950e-5
beta-1	1e-10 - 0.999*	9.6102e-2
beta-2	1e-10 - 0.999*	7.3496e-9
epsilon	1e-10 - 0.01*	6.4399e-4
learning rate start	0.01 - 1.0*	1.0251e-2
learning rate stop	1e-5 - 0.1*	5.0890e-4
learning rate steps	1 - 10000*	5
learning rate power	1 - 2	1.3785
epochs	–	200
early stopping min delta	–	0.01
early stopping patience	–	5
loss	crossentropy, hinge	crossentropy

Table 37. Hyperparameters that we tuned for DOC2VEC, including the best values we found after tuning. Search spaces with a * are sampled using a logarithmic function.

Hyperparameter	Search space	Best Value
batch size	32, 64, 128, . . . , 512	256
number of dense layers	0, 1	1
dense layer activation	linear, relu, elu, tanh, softsign, prelu, selu, gelu, swish, softplus	selu
dense layer activation alpha	0.01 - 1.0*	–
dense layer i size	2, 4, 8, ..., 2048	64
number of convolutions	1 - 5	2
convolution 1 size	1 - 64	49
convolution 2 size	1 - 64	8
filters	1, 2, 4, ..., 64	8
convolution activation	linear, relu, elu, tanh, softsign, prelu, selu, gelu, swish, softplus	swish
convolution activation alpha	0.01 - 1.0*	–
convolution kernel l1	1e-10 - 0.1*	1.6615e-8
convolution bias l1	1e-10 - 0.1*	1.8959e-8
convolution activity l1	1e-10 - 0.1*	6.1055e-8
convolution batch normalisation	false, true	false
convolution 1 batch normalisation momentum	0.01 - 0.99*	–
convolution 2 batch normalisation momentum	0.01 - 0.99*	–
optimiser	adamw, nadam	nadam
weight decay	1e-10 - 0.5*	3.6116e-4
beta-1	1e-10 - 0.999*	8.4755e-4
beta-2	1e-10 - 0.999*	1.2244e-6
epsilon	1e-10 - 0.01*	6.7890e-10
learning rate start	0.01 - 1.0*	1.6069e-1
learning rate stop	1e-5 - 0.1*	2.8635e-3
learning rate steps	1 - 10000*	36
learning rate power	1 - 2	1
epochs	–	200
early stopping min delta	–	0.01
early stopping patience	–	5
loss	crossentropy, hinge	crossentropy

Table 38. Hyperparameters that we tuned for CNN, including the best values we found after tuning. Search spaces with a * are sampled using a logarithmic function.

Hyperparameter	Search space	Best Value
batch size	8, 16, 32, . . . , 512	32
number of dense layers	0 - 1	0
dense layer activation	linear, relu, elu, tanh, softsign, prelu, selu, gelu, swish, softplus	–
dense layer activation alpha	0.01 - 1.0*	–
kernel l1	1e-10 - 0.1*	–
bias l1	1e-10 - 0.1*	–
activation l1	1e-10 - 0.1*	–
dense layer 1 size	2, 4, 8, ..., 2048	–
dense layer 1 dropout	0.0, 0.05, ..., 0.5	–
number of rnn layers	1 - 2	2
rnn layer 1 type	GRU, LSTM	LSTM
rnn layer 1 size	16, 32, 64, 128	128
rnn layer 2 type	GRU, LSTM	GRU
rnn layer 2 size	16, 32, 64, 128	16
optimiser	adamw, nadam	adamw
weight decay	1e-10 - 0.5*	1.0996e-8
beta-1	1e-10 - 0.999*	2.9464e-4
beta-2	1e-10 - 0.999*	4.4085e-3
epsilon	1e-10 - 0.01*	8.0932e-3
learning rate start	0.01 - 1.0*	1.5716e-2
learning rate stop	1e-5 - 0.1*	3.5921e-3
learning rate steps	1 - 10000*	4
learning rate power	1 - 2	1.7859
epochs	–	200
early stopping min delta	–	0.01
early stopping patience	–	5
loss	crossentropy, hinge	crossentropy

Table 39. Hyperparameters that we tuned for RNN, including the best values we found after tuning. Search spaces with a * are sampled using a logarithmic function.

D Maestro Tool Paper

Maestro: A Tool to Find and Explore Architectural Design Decisions in Issue Tracking Systems

Jesse Maarleveld, Arjan Dekker, Sarah Druyts, and Mohamed Soliman

University of Groningen (RUG), Groningen, The Netherlands
{j.maarleveld,a.j.dekker.5,s.druyts}@student.rug.nl, m.a.m.soliman@rug.nl

Abstract. Software engineers commonly re-use architectural design decisions (ADDs) from their previous experience. However, in practice, software engineers still depend on adhoc mechanisms to re-use ADDs. Recent studies show that software engineers discuss ADDs in issue tracking system, which could be useful for software engineers to make new ADDs. Nevertheless, it is rather challenging to find ADDs among the big amount of issues in issue trackers. Therefore, we introduce Maestro, an open source tool for finding, annotating, and exploring ADDs in issue tracking systems. The tool allows researchers and practitioners to find and analyze issues containing ADDs in issue trackers. Maestro provides annotation mechanisms, deep learning components, keywords-based search engine and a user-interface that can be easily used by researchers and practitioners to find and analyze ADDs in issue trackers.

Keywords: Architectural design decisions · issue tracking system.

1 Introduction

Software engineers tend to reuse the knowledge from previously made Architectural Design Decisions (ADDs) [14], such as ADDs on components design (e.g. through patterns ([5])), technology ADDs [20], and ADDs on tactics to address quality requirements (e.g. authentication mechanisms as security tactics) [2]. For instance, software engineers can learn from the drawbacks (e.g. performance issues) of solutions decided in previous ADDs. The re-use of knowledge from previous ADDs could help software engineers to effectively design new systems and mitigate risks.

While re-using ADDs could be useful in practice, empirical studies show that software engineers do not commonly document ADDs [14]. For instance, researchers proposed a wide variety of tools to manage and document ADDs [6, 23, 24]. However, software engineers still tend to maintain their knowledge on ADDs in their head (i.e. tacit) without explicit documentation [6]. On the other hand, software engineers communicate and discuss ADDs *informally* to resolve issues (e.g. new features[1] or improvements[2]) in issue tracking systems

[1] https://issues.apache.org/jira/browse/HADOOP-13944
[2] https://issues.apache.org/jira/browse/CASSANDRA-12245

(e.g. Jira) [19, 3]. We call issues containing such discussions *architectural issues*. The discussions on ADDs in architectural issues contain useful knowledge, which software engineers could potentially re-use to make new ADDs.

While architectural issues could potentially be useful for software engineers, they are not tagged by software engineers [19], which make them hard to find and explore in between the vast majority of issues on programming and bugs. Therefore, researchers utilised different approaches (e.g. machine learning [3], source code analysis [19], and qualitative analysis [19]) to find and explore architectural issues, each with different pros and cons. However, the diversity of the different approaches require researchers and practitioners to execute each approach separately, and possibly manually combine their results to effectively find and explore architectural issues. To execute each approach separately is a complex, error prone and time-consuming process, which require expertise in different fields like machine learning and qualitative analysis.

In this paper, we propose Maestro: An open source tool[3] to find and explore ADDs in issue tracking systems. Maestro combines four different approaches to find and explore ADDs in a single process: keyword-based searches, deep learning, qualitative analysis, and statistical analysis. In addition, Maestro allows importing results from other approaches such as source code analysis. In Maestro, we distinguish between different types of ADDs according to Kruchten et al. [13]: existence (component related), executive (process and technology related), and property (quality related). Maestro is designed to be extensible and easy to use for both researchers and practitioners. For instance, software engineering researchers can train and run deep learning models without expertise on programming deep learning models. Maestro can be deployed remotely or locally, which provides flexibility for researchers and practitioners to run the tool.

The rest of the paper is organised as follows: In Section 2, we discuss the use cases of Maestro. In Section 3, we discuss the architecture of Maestro. We explain our experiences and evaluation of Maestro in Section 4, and compare it with related work in Section 5. Finally, we conclude the paper in Section 6.

2 Use Cases

Maestro serves both researchers and practitioners to find and explore ADDs in issue tracking systems. Researchers can use Maestro for empirical analysis; practitioners can use Maestro to re-discover and re-use architecetural knowledge. Fig 1 shows an overview of the use cases supported by Maestro and their relationships. We explain each use-case below:

UC1 **Select candidate issues for qualitative analysis**: Researchers can select certain issues to be manually analysed (in UC2). The nomination of the selected issues can come from different sources: 1) predictions made by deep learning classifiers (in UC4). 2) issues resulting from keywords-searching (in

[3] Available from: `https://github.com/mining-design-decisions/Maestro`

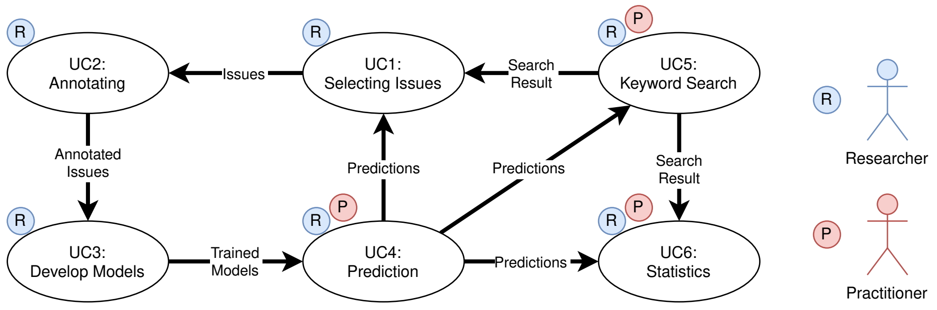


Fig. 1. Use cases supported by Maestro, annotated with relevant actors per use case. Arrows show how results from one use case (or activity) are used by other use cases.

UC5). 3) issues identified from other tools (e.g. source code analysis [19, 18]), and 4) issues selected randomly similar to Bhat et al. [3].

UC2 Annotate issues with types of ADDs: Researchers can analyse selected issues (from UC1) using qualitative methods (e.g. grounded theory [21]), and annotate them based on the types of ADDs within issues. Using the tool, multiple remotely located researchers can discuss types of ADDs using an online conversation associated with each issue. The UI provides the researchers with the summaries and descriptions of issues, the assigned types of ADDs, and a discussion thread per issue. The conversations between researchers can be used incrementally to create a coding book for annotating architectural issues. Furthermore, the tool supports researchers to calculate agreement measures such as Kappa [9] to ensure high quality of the qualitative analysis. The annotated issues can be directly used to develop new deep learning models (in UC3).

UC3 Develop deep learning models to identify types of ADDs in issues: First, researchers can design classifiers by choosing from different types of feature generation (e.g. Word embedding [15] and Word Frequency), deep learning architectures (e.g. RNN [12, 8, 7], CNN [17], and BERT [11]), which can be automatically tuned using the flexible user interface of the tool. Second, researchers can train designed classifiers using the annotated issues (from UC2), and compute their accuracy (e.g. in terms of F_1 score) to automatically identify types of ADDs in issues.

UC4 Predict types of ADDs in issues: Both practitioners or researchers can use the trained classifiers (from UC3) to predict types of ADDs in new, previously un-annotated, issues. Specifically, practitioners can find past ADDs in issues of existing projects, understand their rationale, and re-use their knowledge to make new ADDs. Researchers could further analyse these issues using qualitative analysis (in UC2) or statistical analysis (in UC6).

UC5 Search for ADDs using keywords: Both practitioners or researchers can search for architectural issues using classical keywords-based search (i.e. information retrieval). Moreover, the tool facilitates filtering search results based on the predictions of classifiers (from UC4). In this way, practitioners could effectively find issues that discuss certain types of ADDs. At the same time, researchers can focus their qualitative and statistical analysis (in UC2 and UC6) on issues that discuss certain types of ADDs.

UC6 Perform statistical analysis on ADDs: Researchers and practitioners could perform statistical analysis on architectural issues. For example, practitioners could determine the duration of issues that involve certain types of ADDs. This can help practitioners to estimate the duration of future ADDs based on their type. As another example, researchers might be interested to determine the amount of knowledge on certain types of ADDs in the descriptions and comments of architectural issues.

3 Architecture of Maestro

Maestro consists of four layers, each contains multiple components. The logical architecture is depicted in Figure 2, and the physical architecture in Figure 3. We explain below each layer in more details:

- The **Persistence Layer** contains four different databases: 1) a database that contains data on issues (e.g. summary and description), which we based on the dataset from Montgomery et al. [16]. 2) a database that contains data related to the manual annotation of issues (e.g. manual labels and discussions between researchers), and all deep learning related data (e.g. trained models, their configurations, performance scores), 3) a database that contains cached statistics data, and 4) a database for usernames and passwords.
- The **Data Access Layer** provides secure access to the databases using authentication tactics. Furthermore, it contains components that can update the issues database with new issues from issue trackers (current only Jira is supported) to support the extensibility of the system. We re-used the component created by Montgomery et al. [16], and enhanced it to be extensible.
- The **Processing Layer** contains two major components: 1) The *Keywords Search Engine* provides a centralised API for performing keyword searches (UC5) using Apache Lucene, which allows re-use of pre-computed indices. 2) The *Deep Learning Manager* acts as the backend for all deep learning related functionality outlined in UC2 and UC3. The deep learning was designed to be extensible. In Fig 2, every pipeline makes use of one or more entities. New entities, such as new feature generators or neural networks, can be easily added by adding new entity classes which are instantiated through factories.
- The **User Interface** provides an interface for the user to fulfil all use-cases in Section 2. For instance, to achieve UC3, the UI presents different options for each deep learning model and provides a user-friendly interface to provide parameter values. Through the UI, researchers could initiate the training of machine learning models, and view accuracy scores in a concise overview. Moreover, researchers could manually view and classify issues (UC2). Further details on the UI can be viewed in our video[4]. The UI is designed according to

[4] `https://www.youtube.com/watch?v=sztY5it5Lb4`

the Model-View-Controller (MVC) pattern, and depends on the processing layer and the data access layer (see Figure 2).

Components can be deployed locally or remotely (Fig 3), allowing data centralisation and offloading of computationally intensive tasks to other devices.

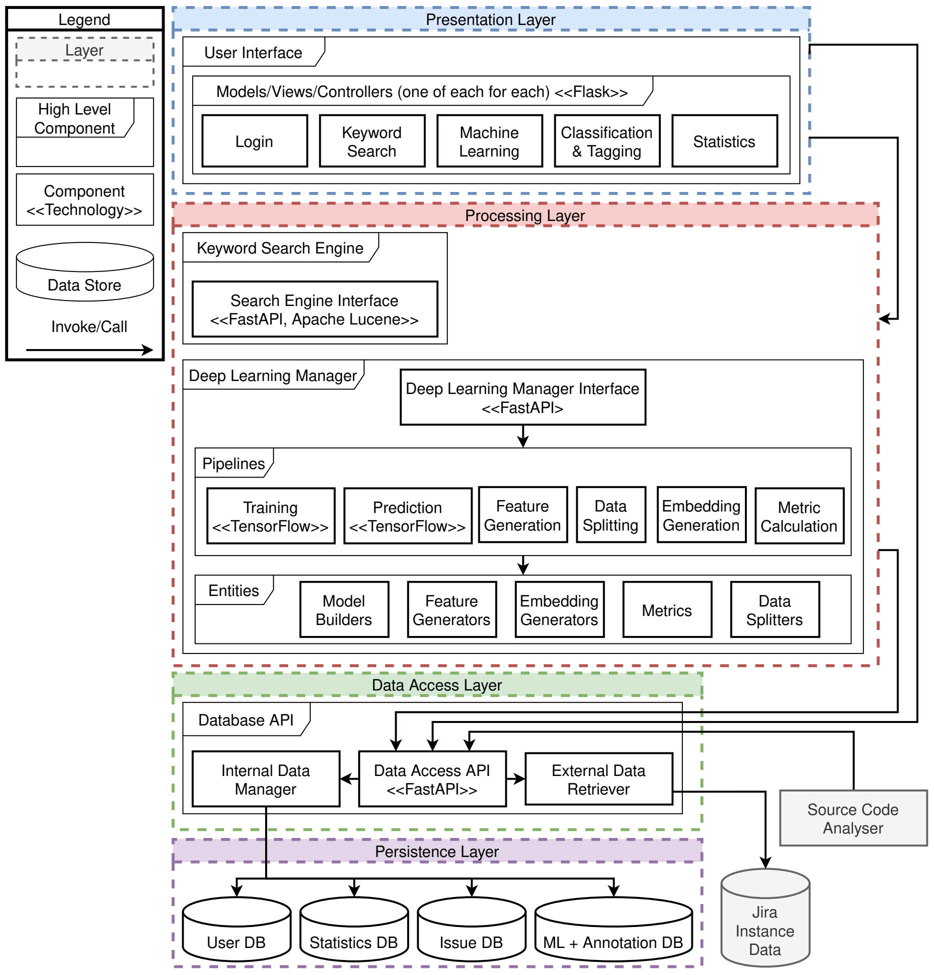


Fig. 2. The logical architecture of Maestro. The "high level components" are larger components with smaller sub-components.

4 Research Process to Develop Maestro

Maestro is a result of a research project spanning more than 2 years of efforts [10] that aims to explore ADDs in issue tracking systems. The four authors of

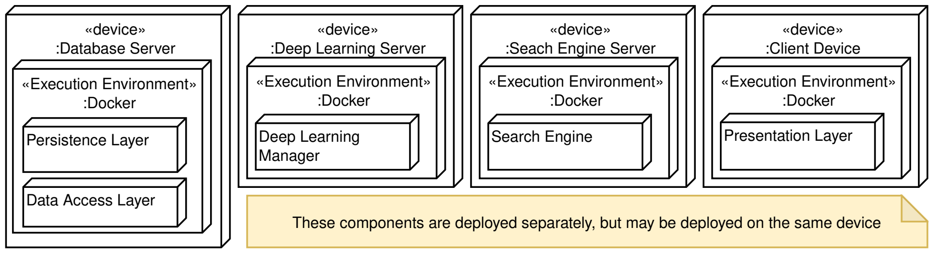


Fig. 3. The physical architecture of Maestro.

this paper, as well as two other independent researchers, participated in this project. Our research follows an action research method [1], where researchers investigated the problem of finding and exploring architectural issues in issue trackers, and simultaneously developed approaches to find and analyse architectural issues. In detail, we followed four phases, each consists of an action and an evaluation steps. We explain below each phase and step, and associate them to the use-cases (UC) in Section 2. We explain how these phases lead to the development of Maestro, and illustrate how it can be used in research.

– **Phase 1 - Random sampling to find architectural issues**:
  Action: We selected a random sample of 400 issues from six different open-source projects, and analysed them using qualitative analysis [21].
  Evaluation: The percentage of architectural issues range between 10-15% of the random sample, which shows that random sampling is not an effective approach to find ADDs in issue trackers.
– **Phase 2 - Keywords-search and source code analysis**:
  Action: Because random sampling was ineffective to find architectural issues, we experimented with two further approaches: searching using keywords from literature (**UC5**), and source code analysis [19]. Using both approaches, we selected 2179 candidate issues (**UC1**) from six open source projects from the Big Data domain (e.g. Apache Hadoop) to be manually analysed using qualitative analysis. For each issue, we downloaded its title and description in an excel sheet, and annotated the types of ADDs in their descriptions according to Kruchten et al. [13]: Existence, property and executive. Disagreements between researchers were discussed in separate meetings.
  Evaluation: Keywords searching and source code analysis were effective to find existence ADDs (precision $> 50\%$), but suffered from low precision to find property and executive ADDs. Moreover, during the qualitative analysis, we realised that it is challenging to annotate large number of issues using Excel sheets, because some issues are long and contain formatting symbols, which cannot be correctly visualised. It was also challenging to track our discussions on issues during our meetings. These discussions were important to write and improve our coding book to annotate ADDs in issue trackers.
– **Phase 3 - Machine learning to find architectural issues**:
  Action: Because keywords-search and source code analysis were not effec-

tive to find property and executive architectural issues, we trained different deep learning models to automatically classify architectural issues (**UC3**). We then used the model with the best accuracy (i.e. "BERT" model) to predict the types of issues (**UC4**), which have not been previously manually analysed. Accordingly, We sorted the issues identified from "BERT" model depending on the confidences obtained from the model to analyse manually (**UC1**). We developed the user interface of the tool to display and sort list of issues based on the confidences generated by deep learning models. Furthermore, we developed a dedicated user interface to annotate and tag issues based on the types of ADDs in their description (**UC2**).
Evaluation: The tool showed significant usefulness to annotate issues, because researchers (allocated remotely) could directly view, discuss and classify issues in one process. According to our experience, using the tool was better than relying on excel sheets, especially in visualising long and complex issues. Moreover, the tool allows to discuss issues, and instantly add issues to the training set without any need to run other scripts or upload data, which prevent faults such as forgetting to include issues or inserting duplicate issues (i.e. the tool provides a consistent overview of all labelled issues for all users). Additionally, during annotations, the tool allows adding tags to issues, which helped us to mark issues that require a second opinion on their classification, and enabled us to track information about who annotated which issues, and how these issues were found (e.g. using keywords searching – **UC5**). This tagging functionality helped us to more easily identify groups of potentially miss-annotated issues. Furthermore, the UI brings notable enhancements to train deep learning models. Previously, we had to manually create configurations for each model, which was error-prone and tedious. However, the UI now clearly presents all available options for each model to facilitate creation, training and evaluation. Using this new functionality of the tool, we performed UC2-UC4 in 3 iterations to expand our dataset to reach 2210 architectural issues and 2903 non-architectural issues.

– **Phase 4 - Find architectural issues from different domains**:
Action: In the previous phases, we explored ADDs in six open-source projects from the Big Data domain. In this phase, we explore ADDs in projects from different domains other than Big Data. Thus, we re-used a recent dataset from Montgomery et al. [16], which contains more than 2.7 million Jira issues from 1352 projects that belong to six different domains including Big Data, Cloud Computing, SOA, and DevOps. We trained and executed the best performing model (i.e. "BERT") to identify architectural issues and predict the types of ADDs in all issues in the dataset (**UC4**). We also developed a statistical analysis functionality in the tool (**UC6**) to visualise the types of ADDs in the different domains, as well as the characteristics of architectural issues such as time to resolve and the amount of discussion in comments.
Evaluation: Using the tool, we identified 250,708 architectural issues from the six domains. Moreover, we determined the most common types of ADDs per domain, and compared characteristics of architectural issues per domain. For example, issues that discuss property ADDs were most involved and took

longer time to resolve. The statistical functionality in the tool (**UC6**) shows its usefulness to explore ADDs in a massive number of architectural issues.

5 Related Work

Several traditional architectural knowledge management tools have been previously proposed [22]. These tools store and document ADDs in repositories and templates, which need to be manually populated. On the other hand, our proposed tool Maestro focus on ADDs discussed in issue tracking systems.

The closest tool to Maestro is ADeX [4], which can classify architectural issues using machine learning. Moreover, ADeX can recommend developers for making certain ADDs based on personal expertise. While both tools ADeX and Maestro aim to find and explore ADDs in issue trackers, our proposed tool Maestro is different than ADeX in the following points:

– Maestro allows researchers to apply qualitative analysis (in UC2), and add manually classified issues to the training dataset. Moreover, Maestro supports keywords-based searches (in UC5), which allows researchers to easily expand their dataset of architectural issues through a snowballing process. This process is not supported by ADeX.
– Maestro provides a user-friendly UI to train and evaluate *new deep learning models* (in UC3 and UC4), which can help researchers to evolve models for classifying architectural issues. This flexibility is not provided by ADeX, which provides pre-trained machine learning models for classification. The accuracy of the pre-trained model is fixed based on Bhat et al. [3].
– Maestro has been evaluated on a large dataset of issues with 2.7 million issues from different domains, which show its scalability and usefulness to run on projects from different domains. In contrast, ADeX has been applied on two open source projects.
– Maestro is open source[5] and is designed to be extended by other researchers or practitioners. In contrast, the source code of ADeX is not referenced by the authors of ADeX.

6 Conclusion

We developed Maestro, an open source tool for finding, and exploring architectural issues that discuss design decisions. Our experience with Maestro showed its usefulness to find and annotate 5113 issues, and develop deep learning models that automatically classified 250,708 architectural issues. Contrary to existing tools, Maestro supports researchers to find and annotate architectural issues through keywords searching, deep learning models and snowballing. Our future work focuses on evaluating Maestro with practitioners to evaluate its usefulness to re-use ADDs from issue trackers. Furthermore, we aim to use Maestro to further expand our dataset with new issues from different projects, and different issue trackers. This can improve the accuracy and generalizability of Maestro.

[5] `https://github.com/mining-design-decisions/Maestro`

References

1. Baskerville, R.L., Wood-Harper, A.T.: A Critical Perspective on Action Research as a Method for Information Systems Research. In: Willcocks, L.P., Sauer, C., Lacity, M.C. (eds.) Enacting Research Methods in Information Systems: Volume 2, pp. 169–190. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-29269-4_7, https://doi.org/10.1007/978-3-319-29269-4_7
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Professional (2003), google-Books-ID: mdiIu8Kk1WMC
3. Bhat, M., Shumaiev, K., Biesdorf, A., Hohenstein, U., Matthes, F.: Automatic extraction of design decisions from issue management systems: A machine learning based approach. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **10475 LNCS**, 138–154 (2017). https://doi.org/10.1007/978-3-319-65831-5_10, publisher: Springer Verlag ISBN: 9783319658308
4. Bhat, M., Tinnes, C., Shumaiev, K., Biesdorf, A., Hohenstein, U., Matthes, F.: ADeX: A Tool for Automatic Curation of Design Decision Knowledge for Architectural Decision Recommendations. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C). pp. 158–161 (Mar 2019). https://doi.org/10.1109/ICSA-C.2019.00035
5. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley, Chichester, UK (1996)
6. Capilla, R., Jansen, A., Tang, A., Avgeriou, P., Babar, M.A.: 10 years of software architecture knowledge management: Practice and future. Journal of Systems and Software **116**, 191–205 (Sep 2015). https://doi.org/10.1016/j.jss.2015.08.054
7. Cho, K., van Merrienboer, B., Bahdanau, D., Bengio, Y.: On the Properties of Neural Machine Translation: Encoder-Decoder Approaches (Oct 2014). https://doi.org/10.48550/arXiv.1409.1259, arXiv:1409.1259 [cs, stat]
8. Chung, J., Gulcehre, C., Cho, K., Bengio, Y.: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling (Dec 2014). https://doi.org/10.48550/arXiv.1412.3555, arXiv:1412.3555 [cs]
9. Cohen, J.: A Coefficient of Agreement for Nominal Scales. Educational and Psychological Measurement **20**(1), 37–46 (Apr 1960). https://doi.org/10.1177/001316446002000104, publisher: SAGE Publications Inc
10. Dekker, A., Maarleveld, J.: Mining for Architectural Design Decisions in Issue Tracking Systems using Deep Learning Approaches. MSc Internship Report, University of Groningen, Groningen (2022), https://fse.studenttheses.ub.rug.nl/28689/
11. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (May 2019). https://doi.org/10.48550/arXiv.1810.04805, http://arxiv.org/abs/1810.04805, arXiv:1810.04805 [cs]
12. Hochreiter, S., Schmidhuber, J.: Long Short-term Memory. Neural computation **9**, 1735–80 (Dec 1997). https://doi.org/10.1162/neco.1997.9.8.1735
13. Kruchten, P.: An ontology of architectural design decisions in software intensive systems. 2nd Groningen workshop on software variability (2004)
14. Manteuffel, C., Avgeriou, P., Hamberg, R.: An exploratory case study on reusing architecture decisions in software-intensive system projects. Journal of Systems and Software **144**, 60–83 (Oct 2018). https://doi.org/10.1016/j.jss.2018.05.064

15. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient Estimation of Word Representations in Vector Space (Sep 2013). https://doi.org/10.48550/arXiv.1301.3781, http://arxiv.org/abs/1301.3781, arXiv:1301.3781 [cs]
16. Montgomery, L., Lüders, C., Maalej, W.: An Alternative Issue Tracking Dataset of Public Jira Repositories. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 73–77 (May 2022). https://doi.org/10.1145/3524842.3528486, arXiv:2201.08368 [cs]
17. Ren, X., Xing, Z., Xia, X., Lo, D., Wang, X., Grundy, J.: Neural Network-based Detection of Self-Admitted Technical Debt: From Performance to Explainability. ACM Transactions on Software Engineering and Methodology **28**(3), 15:1–15:45 (Jul 2019). https://doi.org/10.1145/3324916
18. Shahbazian, A., Kyu Lee, Y., Le, D., Brun, Y., Medvidovic, N.: Recovering Architectural Design Decisions. Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018 pp. 95–104 (Jul 2018). https://doi.org/10.1109/ICSA.2018.00019, publisher: Institute of Electrical and Electronics Engineers Inc. ISBN: 9781538663981
19. Soliman, M., Galster, M., Avgeriou, P.: An Exploratory Study on Architectural Knowledge in Issue Tracking Systems. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **12857 LNCS**, 117–133 (2021). https://doi.org/10.1007/978-3-030-86044-8_8/FIGURES/3, arXiv: 2106.11140 Publisher: Springer Science and Business Media Deutschland GmbH ISBN: 9783030860431
20. Soliman, M., Riebisch, M., Zdun, U.: Enriching Architecture Knowledge with Technology Design Decisions. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture. pp. 135–144 (May 2015). https://doi.org/10.1109/WICSA.2015.14
21. Stol, K.J., Ralph, P., Fitzgerald, B.: Grounded theory in software engineering research: a critical review and guidelines. In: Proceedings of the 38th International Conference on Software Engineering. pp. 120–131. ICSE '16, Association for Computing Machinery, New York, NY, USA (May 2016). https://doi.org/10.1145/2884781.2884833
22. Tang, A., Avgeriou, P., Jansen, A., Capilla, R., Ali Babar, M.: A comparative study of architecture knowledge management tools. Journal of Systems and Software **83**(3), 352–370 (Mar 2010). https://doi.org/10.1016/j.jss.2009.08.032
23. Weinreich, R., Groher, I.: Software architecture knowledge management approaches and their support for knowledge management activities: A systematic literature review. Information and Software Technology **80**, 265–286 (Dec 2016). https://doi.org/10.1016/j.infsof.2016.09.007
24. Weinreich, R., Groher, I., Miesbauer, C.: An expert survey on kinds, influence factors and documentation of design decisions in practice. Future Generation Computer Systems **47**, 145–160 (Jun 2015). https://doi.org/10.1016/j.future.2014.12.002

E Extra Tables for RQ3

This section contains the extra tables for **RQ3**. Specifically, it contains the detailed performance metrics on the fixed test set for each of the models we have tested.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.966	0.980	0.952	+53%
	Macro	0.949	0.961	0.939	+243%
	Positive macro	0.941	0.962	0.923	+155%
	Weighted macro	0.966	0.958	0.975	+66%
	Existence	0.927	0.969	0.889	+117%
	Executive	0.947	0.982	0.914	+274%
	Property	0.949	0.933	0.966	+123%
	Non-Architectural	0.972	0.958	0.986	+38%
Val	Detection	0.746	0.784	0.712	+18%
	Macro	0.640	0.703	0.603	+131%
	Positive macro	0.585	0.682	0.523	+59%
	Weighted macro	0.764	0.752	0.785	+31%
	Existence	0.502	0.658	0.406	+17%
	Executive	0.569	0.702	0.478	+125%
	Property	0.685	0.685	0.685	+61%
	Non-Architectural	0.803	0.768	0.841	+14%
Test	Detection	0.751	0.756	0.745	+19%
	Macro	0.676	0.726	0.644	+144%
	Positive macro	0.636	0.709	0.587	+72%
	Weighted macro	0.766	0.765	0.772	+32%
	Existence	0.581	0.694	0.500	+36%
	Executive	0.667	0.796	0.574	+164%
	Property	0.662	0.637	0.688	+56%
	Non-Architectural	0.795	0.779	0.812	+13%

Table 40. BERT performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.935	0.956	0.915	+48%
	Macro	0.896	0.907	0.889	+223%
	Positive macro	0.881	0.905	0.861	+139%
	Weighted macro	0.930	0.908	0.954	+60%
	Existence	0.866	0.844	0.890	+102%
	Executive	0.904	0.973	0.844	+257%
	Property	0.873	0.899	0.849	+105%
	Non-Architectural	0.941	0.913	0.972	+34%
Val	Detection	0.729	0.761	0.699	+16%
	Macro	0.611	0.656	0.584	+120%
	Positive macro	0.554	0.630	0.500	+50%
	Weighted macro	0.745	0.717	0.781	+28%
	Existence	0.608	0.623	0.594	+42%
	Executive	0.464	0.605	0.377	+84%
	Property	0.590	0.663	0.531	+39%
	Non-Architectural	0.781	0.734	0.833	+11%
Test	Detection	0.706	0.723	0.690	+12%
	Macro	0.628	0.689	0.597	+127%
	Positive macro	0.581	0.680	0.517	+57%
	Weighted macro	0.742	0.708	0.786	+28%
	Existence	0.638	0.678	0.603	+49%
	Executive	0.514	0.730	0.397	+103%
	Property	0.590	0.633	0.552	+39%
	Non-Architectural	0.771	0.716	0.836	+10%

Table 41. BERT with fine-grained technology replacement performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.904	0.879	0.931	+44%
	Macro	0.864	0.864	0.865	+212%
	Positive macro	0.848	0.840	0.857	+130%
	Weighted macro	0.902	0.920	0.885	+55%
	Existence	0.830	0.821	0.839	+94%
	Executive	0.862	0.880	0.845	+241%
	Property	0.851	0.819	0.886	+100%
	Non-Architectural	0.914	0.938	0.892	+30%
Val	Detection	0.726	0.696	0.759	+15%
	Macro	0.646	0.633	0.663	+133%
	Positive macro	0.616	0.587	0.648	+67%
	Weighted macro	0.717	0.738	0.698	+23%
	Existence	0.595	0.559	0.635	+39%
	Executive	0.634	0.616	0.652	+151%
	Property	0.619	0.585	0.656	+46%
	Non-Architectural	0.739	0.771	0.709	+5%
Test	Detection	0.716	0.676	0.762	+14%
	Macro	0.609	0.603	0.620	+120%
	Positive macro	0.567	0.545	0.594	+54%
	Weighted macro	0.706	0.738	0.680	+22%
	Existence	0.534	0.525	0.544	+25%
	Executive	0.556	0.561	0.552	+120%
	Property	0.609	0.548	0.685	+43%
	Non-Architectural	0.736	0.779	0.698	+5%

Table 42. TF-IDF performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.909	0.899	0.919	+44%
	Macro	0.867	0.876	0.858	+213%
	Positive macro	0.849	0.858	0.841	+130%
	Weighted macro	0.908	0.916	0.899	+56%
	Existence	0.826	0.836	0.817	+93%
	Executive	0.861	0.892	0.832	+240%
	Property	0.859	0.846	0.873	+102%
	Non-Architectural	0.921	0.929	0.912	+31%
Val	Detection	0.688	0.685	0.691	+9%
	Macro	0.602	0.603	0.604	+117%
	Positive macro	0.562	0.561	0.564	+52%
	Weighted macro	0.695	0.696	0.694	+20%
	Existence	0.534	0.515	0.556	+25%
	Executive	0.595	0.629	0.565	+135%
	Property	0.556	0.540	0.573	+31%
	Non-Architectural	0.724	0.727	0.721	+3%
Test	Detection	0.722	0.694	0.752	+15%
	Macro	0.625	0.639	0.620	+126%
	Positive macro	0.583	0.592	0.585	+58%
	Weighted macro	0.720	0.742	0.701	+24%
	Existence	0.541	0.524	0.559	+26%
	Executive	0.627	0.725	0.552	+148%
	Property	0.580	0.526	0.645	+36%
	Non-Architectural	0.751	0.779	0.725	+7%

Table 43. TF-IDF with fine-grained technology replacement performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.994	0.994	0.993	+58%
	Macro	0.988	0.986	0.989	+257%
	Positive macro	0.985	0.984	0.987	+167%
	Weighted macro	0.993	0.992	0.994	+71%
	Existence	0.985	0.981	0.989	+130%
	Executive	0.984	0.984	0.984	+289%
	Property	0.987	0.986	0.988	+132%
	Non-Architectural	0.995	0.994	0.995	+42%
Val	Detection	0.689	0.685	0.694	+9%
	Macro	0.592	0.583	0.605	+114%
	Positive macro	0.548	0.534	0.565	+48%
	Weighted macro	0.695	0.698	0.693	+20%
	Existence	0.534	0.540	0.528	+25%
	Executive	0.573	0.518	0.642	+127%
	Property	0.535	0.544	0.527	+26%
	Non-Architectural	0.727	0.731	0.722	+3%
Test	Detection	0.677	0.661	0.694	+7%
	Macro	0.607	0.599	0.617	+119%
	Positive macro	0.571	0.555	0.589	+55%
	Weighted macro	0.690	0.701	0.680	+19%
	Existence	0.568	0.556	0.581	+33%
	Executive	0.569	0.547	0.594	+125%
	Property	0.576	0.561	0.592	+36%
	Non-Architectural	0.715	0.731	0.699	+2%

Table 44. BOWN performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.833	0.826	0.841	+32%
	Macro	0.773	0.786	0.762	+179%
	Positive macro	0.746	0.760	0.733	+102%
	Weighted macro	0.837	0.843	0.831	+44%
	Existence	0.743	0.725	0.762	+74%
	Executive	0.739	0.784	0.698	+192%
	Property	0.755	0.770	0.739	+78%
	Non-Architectural	0.856	0.862	0.849	+22%
Val	Detection	0.704	0.700	0.709	+12%
	Macro	0.613	0.616	0.613	+121%
	Positive macro	0.571	0.575	0.572	+55%
	Weighted macro	0.708	0.710	0.706	+22%
	Existence	0.553	0.510	0.603	+29%
	Executive	0.606	0.635	0.580	+140%
	Property	0.556	0.579	0.534	+31%
	Non-Architectural	0.737	0.742	0.733	+5%
Test	Detection	0.718	0.699	0.738	+14%
	Macro	0.611	0.625	0.601	+121%
	Positive macro	0.563	0.576	0.556	+53%
	Weighted macro	0.721	0.736	0.707	+24%
	Existence	0.564	0.549	0.581	+32%
	Executive	0.579	0.648	0.522	+129%
	Property	0.547	0.530	0.565	+29%
	Non-Architectural	0.754	0.772	0.736	+7%

Table 45. BOWN with fine-grained technology replacement performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.877	0.922	0.836	+39%
	Macro	0.853	0.886	0.827	+208%
	Positive macro	0.836	0.891	0.789	+127%
	Weighted macro	0.891	0.874	0.912	+53%
	Existence	0.794	0.882	0.722	+86%
	Executive	0.842	0.888	0.801	+233%
	Property	0.872	0.901	0.845	+105%
	Non-Architectural	0.904	0.871	0.940	+29%
Val	Detection	0.667	0.730	0.614	+6%
	Macro	0.603	0.632	0.584	+118%
	Positive macro	0.555	0.608	0.512	+50%
	Weighted macro	0.713	0.685	0.748	+23%
	Existence	0.514	0.594	0.452	+20%
	Executive	0.606	0.635	0.580	+140%
	Property	0.545	0.595	0.504	+28%
	Non-Architectural	0.749	0.703	0.801	+7%
Test	Detection	0.712	0.733	0.692	+13%
	Macro	0.600	0.622	0.585	+117%
	Positive macro	0.543	0.577	0.516	+47%
	Weighted macro	0.732	0.726	0.740	+26%
	Existence	0.496	0.578	0.434	+16%
	Executive	0.559	0.551	0.567	+121%
	Property	0.574	0.602	0.548	+35%
	Non-Architectural	0.773	0.756	0.791	+10%

Table 46. BOWF performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.838	0.911	0.776	+33%
	Macro	0.802	0.863	0.760	+189%
	Positive macro	0.775	0.874	0.701	+110%
	Weighted macro	0.863	0.836	0.898	+48%
	Existence	0.749	0.810	0.697	+75%
	Executive	0.725	0.904	0.605	+187%
	Property	0.852	0.909	0.801	+100%
	Non-Architectural	0.880	0.831	0.936	+25%
Val	Detection	0.670	0.766	0.595	+6%
	Macro	0.583	0.635	0.552	+111%
	Positive macro	0.522	0.612	0.456	+42%
	Weighted macro	0.723	0.687	0.774	+25%
	Existence	0.516	0.600	0.452	+21%
	Executive	0.538	0.640	0.464	+113%
	Property	0.513	0.596	0.450	+21%
	Non-Architectural	0.766	0.703	0.841	+9%
Test	Detection	0.687	0.747	0.636	+9%
	Macro	0.581	0.631	0.548	+110%
	Positive macro	0.517	0.598	0.457	+40%
	Weighted macro	0.730	0.707	0.760	+26%
	Existence	0.520	0.582	0.471	+22%
	Executive	0.518	0.644	0.433	+105%
	Property	0.513	0.569	0.468	+21%
	Non-Architectural	0.774	0.731	0.822	+10%

Table 47. BOWF with fine-grained technology replacement performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.750	0.746	0.753	+19%
	Macro	0.668	0.680	0.671	+141%
	Positive macro	0.629	0.644	0.634	+71%
	Weighted macro	0.759	0.762	0.759	+31%
	Existence	0.624	0.633	0.615	+46%
	Executive	0.587	0.701	0.505	+132%
	Property	0.678	0.598	0.782	+59%
	Non-Architectural	0.785	0.789	0.782	+12%
Val	Detection	0.738	0.754	0.723	+17%
	Macro	0.653	0.672	0.644	+136%
	Positive macro	0.611	0.641	0.595	+66%
	Weighted macro	0.749	0.742	0.758	+29%
	Existence	0.587	0.633	0.548	+37%
	Executive	0.639	0.736	0.565	+153%
	Property	0.607	0.553	0.672	+43%
	Non-Architectural	0.779	0.765	0.793	+11%
Test	Detection	0.698	0.694	0.701	+11%
	Macro	0.590	0.610	0.584	+113%
	Positive macro	0.537	0.564	0.531	+46%
	Weighted macro	0.714	0.719	0.712	+23%
	Existence	0.589	0.623	0.559	+38%
	Executive	0.464	0.578	0.388	+84%
	Property	0.557	0.491	0.645	+31%
	Non-Architectural	0.747	0.750	0.744	+6%

Table 48. RNN performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.731	0.749	0.714	+16%
	Macro	0.657	0.682	0.638	+137%
	Positive macro	0.615	0.653	0.586	+67%
	Weighted macro	0.753	0.747	0.761	+30%
	Existence	0.595	0.662	0.540	+39%
	Executive	0.582	0.648	0.528	+130%
	Property	0.669	0.650	0.688	+57%
	Non-Architectural	0.781	0.766	0.797	+11%
Val	Detection	0.686	0.732	0.645	+9%
	Macro	0.632	0.667	0.606	+128%
	Positive macro	0.591	0.649	0.544	+60%
	Weighted macro	0.725	0.705	0.749	+25%
	Existence	0.576	0.641	0.524	+35%
	Executive	0.618	0.704	0.551	+144%
	Property	0.579	0.603	0.557	+36%
	Non-Architectural	0.754	0.718	0.793	+7%
Test	Detection	0.680	0.713	0.650	+8%
	Macro	0.577	0.614	0.551	+108%
	Positive macro	0.518	0.575	0.473	+40%
	Weighted macro	0.715	0.704	0.730	+23%
	Existence	0.527	0.612	0.463	+23%
	Executive	0.487	0.558	0.433	+93%
	Property	0.539	0.556	0.524	+27%
	Non-Architectural	0.755	0.729	0.783	+8%

Table 49. RNN with fine-grained technology replacement performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.693	0.832	0.594	+10%
	Macro	0.642	0.728	0.598	+132%
	Positive macro	0.589	0.730	0.498	+60%
	Weighted macro	0.766	0.722	0.833	+32%
	Existence	0.604	0.704	0.528	+41%
	Executive	0.536	0.768	0.411	+112%
	Property	0.627	0.720	0.556	+48%
	Non-Architectural	0.801	0.722	0.898	+14%
Val	Detection	0.641	0.797	0.536	+2%
	Macro	0.567	0.657	0.526	+105%
	Positive macro	0.499	0.648	0.408	+35%
	Weighted macro	0.724	0.676	0.801	+25%
	Existence	0.533	0.648	0.452	+24%
	Executive	0.523	0.737	0.406	+107%
	Property	0.442	0.558	0.366	+4%
	Non-Architectural	0.770	0.684	0.880	+10%
Test	Detection	0.652	0.788	0.556	+4%
	Macro	0.585	0.649	0.550	+111%
	Positive macro	0.519	0.631	0.442	+41%
	Weighted macro	0.737	0.693	0.802	+27%
	Existence	0.544	0.674	0.456	+27%
	Executive	0.500	0.622	0.418	+98%
	Property	0.514	0.596	0.452	+21%
	Non-Architectural	0.781	0.704	0.876	+11%

Table 50. DOC2VEC performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.686	0.809	0.595	+9%
	Macro	0.646	0.727	0.603	+133%
	Positive macro	0.598	0.730	0.510	+62%
	Weighted macro	0.759	0.721	0.818	+31%
	Existence	0.585	0.748	0.480	+37%
	Executive	0.561	0.732	0.455	+122%
	Property	0.647	0.708	0.595	+52%
	Non-Architectural	0.792	0.719	0.881	+13%
Val	Detection	0.637	0.756	0.550	+1%
	Macro	0.574	0.647	0.536	+107%
	Positive macro	0.513	0.635	0.433	+39%
	Weighted macro	0.712	0.671	0.773	+23%
	Existence	0.495	0.619	0.413	+16%
	Executive	0.541	0.714	0.435	+114%
	Property	0.504	0.573	0.450	+19%
	Non-Architectural	0.754	0.682	0.845	+7%
Test	Detection	0.667	0.768	0.589	+6%
	Macro	0.568	0.632	0.537	+105%
	Positive macro	0.498	0.605	0.431	+35%
	Weighted macro	0.731	0.697	0.782	+26%
	Existence	0.513	0.644	0.426	+20%
	Executive	0.438	0.605	0.343	+73%
	Property	0.544	0.565	0.524	+28%
	Non-Architectural	0.777	0.714	0.853	+11%

Table 51. DOC2VEC with fine-grained technology replacement performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.714	0.739	0.692	+13%
	Macro	0.669	0.662	0.676	+141%
	Positive macro	0.635	0.632	0.638	+72%
	Weighted macro	0.747	0.730	0.765	+29%
	Existence	0.615	0.608	0.622	+44%
	Executive	0.657	0.662	0.653	+160%
	Property	0.632	0.625	0.639	+49%
	Non-Architectural	0.771	0.751	0.792	+10%
Val	Detection	0.641	0.682	0.605	+2%
	Macro	0.563	0.564	0.565	+103%
	Positive macro	0.512	0.524	0.502	+39%
	Weighted macro	0.683	0.658	0.711	+18%
	Existence	0.535	0.531	0.540	+25%
	Executive	0.493	0.493	0.493	+95%
	Property	0.508	0.549	0.473	+20%
	Non-Architectural	0.717	0.685	0.753	+2%
Test	Detection	0.635	0.672	0.603	+1%
	Macro	0.562	0.555	0.569	+103%
	Positive macro	0.507	0.508	0.507	+37%
	Weighted macro	0.689	0.666	0.714	+19%
	Existence	0.543	0.549	0.537	+27%
	Executive	0.475	0.458	0.493	+88%
	Property	0.504	0.517	0.492	+19%
	Non-Architectural	0.725	0.696	0.756	+3%

Table 52. CNN performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

Set	Metric	F1	P	R	Impr. over best guessing
Train	Detection	0.670	0.521	0.938	+6%
	Macro	0.502	0.509	0.692	+81%
	Positive macro	0.535	0.401	0.834	+45%
	Weighted macro	0.429	0.763	0.361	-26%
	Existence	0.558	0.445	0.749	+30%
	Executive	0.452	0.304	0.879	+79%
	Property	0.596	0.452	0.872	+40%
	Non-Architectural	0.404	0.834	0.266	-42%
Val	Detection	0.667	0.512	0.955	+6%
	Macro	0.445	0.479	0.634	+61%
	Positive macro	0.485	0.361	0.778	+31%
	Weighted macro	0.358	0.758	0.299	-38%
	Existence	0.532	0.421	0.722	+24%
	Executive	0.389	0.253	0.841	+54%
	Property	0.533	0.407	0.771	+25%
	Non-Architectural	0.327	0.836	0.203	-53%
Test	Detection	0.638	0.497	0.888	+1%
	Macro	0.440	0.446	0.587	+59%
	Positive macro	0.460	0.351	0.697	+25%
	Weighted macro	0.397	0.672	0.330	-32%
	Existence	0.515	0.427	0.647	+20%
	Executive	0.364	0.246	0.701	+44%
	Property	0.501	0.379	0.742	+18%
	Non-Architectural	0.379	0.733	0.256	-46%

Table 53. CNN with fine-grained technology replacement performance on the fixed test set, containing issues from the projects Apache Hadoop, Cassandra, Tajo, Mapreduce, HDFS and Yarn. F_1 is the F_1 score, P is the precision, and R is the recall.

F Extra Plots for RQ4

This section contains the plots for BERT trained on varying dataset sizes, with the random sample from the data storage & processing domain as the test set. These plots are an addition of the results for **RQ4**.

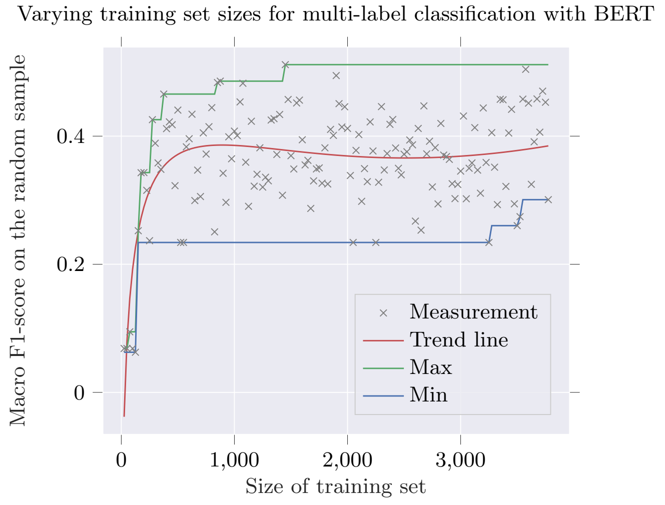


Fig. 46. Varying training set sizes for multi-label classification with BERT†, with the random sample of the data storage & processing domain as the test set. These results were obtained without text preprocessing.

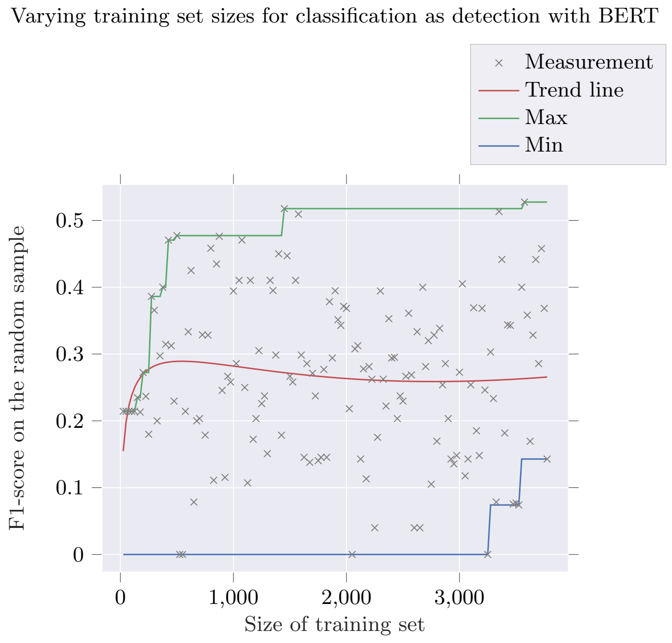


Fig. 47. Varying training set sizes for Detection with BERT†, with the random sample of the data storage & processing domain as the test set. These results were obtained without text preprocessing.

G Extra Tables for RQ6

This section contains the additional plots for **RQ6**. The first part consists of performance overviews of the models on the random sample from the six domains for both the multi-label classification performance and detection performance. The second part consists of detailed performance metrics on all three random samples (data storage & processing domain, web development domain, and the six domains) for each model we tested.

Model	F_1	P	R
BERT	0.489	0.557	0.465
BERT†	0.469	0.583	0.469
RNN	0.462	0.475	0.452
DOC2VEC*	0.442	0.490	0.428
BOWN	0.435	0.396	0.537
TF-IDF	0.427	0.395	0.531
BERT*	0.417	0.505	0.406
TF-IDF*	0.414	0.395	0.465
RNN*	0.404	0.492	0.403
BOWN*	0.403	0.418	0.407
BOWF*	0.386	0.416	0.373
BOWF	0.314	0.373	0.300
DOC2VEC	0.307	0.354	0.296
CNN	0.090	0.048	0.750
CNN*	0.090	0.048	0.750

Table 54. Macro scores on the random sample of all domains. Results with a * are obtained from models trained using fine-grained technology replacement. Results with a † are obtained without text preprocessing. F_1 is the macro F_1 score, P is the macro precision, and R is the macro recall.

Model	F_1	P	R
BERT†	0.508	0.508	0.508
BERT	0.466	0.545	0.407
RNN	0.455	0.490	0.424
BOWN	0.410	0.324	0.559
RNN*	0.407	0.407	0.407
DOC2VEC*	0.400	0.411	0.390
TF-IDF	0.393	0.298	0.576
TF-IDF*	0.376	0.311	0.475
BERT*	0.348	0.357	0.339
BOWN*	0.342	0.345	0.339
BOWF*	0.314	0.372	0.271
BOWF	0.220	0.391	0.153
DOC2VEC	0.206	0.263	0.169
CNN	0.257	0.147	1.000
CNN*	0.257	0.147	1.000

Table 55. Detection scores on the random sample of all domains. Results with a * are obtained from models trained using fine-grained technology replacement. Results with a † are obtained without text preprocessing. F_1 is the detection F_1 score, P is the detection precision, and R is the detection recall.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.534	0.456	0.646	+150%
	Macro	0.549	0.575	0.572	+135%
	Positive macro	0.425	0.451	0.464	+273%
	Weighted macro	0.839	0.868	0.821	+8%
	Existence	0.533	0.500	0.571	+231%
	Executive	0.400	0.308	0.571	+488%
	Property	0.343	0.545	0.250	+203%
	Non-Architectural	0.921	0.949	0.895	-2%
Web development	Detection	0.478	0.391	0.614	+141%
	Macro	0.531	0.681	0.589	+126%
	Positive macro	0.403	0.592	0.492	+353%
	Weighted macro	0.846	0.895	0.833	+4%
	Existence	0.423	0.367	0.500	+307%
	Executive	0.552	0.410	0.842	+506%
	Property	0.235	1.000	0.133	+227%
	Non-Architectural	0.914	0.949	0.882	-3%
All six domains	Detection	0.508	0.508	0.508	+90%
	Macro	0.469	0.583	0.469	+105%
	Positive macro	0.321	0.473	0.320	+159%
	Weighted macro	0.805	0.840	0.801	+8%
	Existence	0.308	0.400	0.250	+182%
	Executive	0.355	0.268	0.524	+274%
	Property	0.300	0.750	0.188	+78%
	Non-Architectural	0.915	0.915	0.915	+0%

Table 56. BERT† performance on the random samples. These results were obtained without text preprocessing.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.598	0.592	0.604	+179%
	Macro	0.613	0.623	0.607	+162%
	Positive macro	0.502	0.516	0.495	+341%
	Weighted macro	0.871	0.876	0.868	+12%
	Existence	0.562	0.621	0.514	+249%
	Executive	0.444	0.462	0.429	+554%
	Property	0.500	0.464	0.542	+342%
	Non-Architectural	0.945	0.946	0.943	+1%
Web development	Detection	0.469	0.514	0.432	+137%
	Macro	0.546	0.592	0.515	+132%
	Positive macro	0.414	0.479	0.370	+365%
	Weighted macro	0.869	0.870	0.871	+7%
	Existence	0.368	0.438	0.318	+254%
	Executive	0.541	0.556	0.526	+494%
	Property	0.333	0.444	0.267	+363%
	Non-Architectural	0.940	0.931	0.949	+0%
All six domains	Detection	0.466	0.545	0.407	+74%
	Macro	0.489	0.557	0.465	+114%
	Positive macro	0.345	0.442	0.306	+178%
	Weighted macro	0.815	0.820	0.823	+9%
	Existence	0.278	0.417	0.208	+155%
	Executive	0.383	0.346	0.429	+303%
	Property	0.375	0.562	0.281	+122%
	Non-Architectural	0.921	0.902	0.941	+1%

Table 57. BERT performance on the random samples.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.463	0.360	0.646	+116%
	Macro	0.522	0.489	0.589	+123%
	Positive macro	0.399	0.336	0.504	+250%
	Weighted macro	0.811	0.846	0.786	+5%
	Existence	0.427	0.400	0.457	+165%
	Executive	0.286	0.214	0.429	+320%
	Property	0.484	0.395	0.625	+328%
	Non-Architectural	0.892	0.946	0.844	-5%
Web development	Detection	0.387	0.274	0.659	+95%
	Macro	0.424	0.384	0.573	+80%
	Positive macro	0.279	0.196	0.502	+213%
	Weighted macro	0.780	0.847	0.745	-4%
	Existence	0.277	0.209	0.409	+166%
	Executive	0.348	0.240	0.632	+282%
	Property	0.212	0.137	0.467	+195%
	Non-Architectural	0.858	0.949	0.784	-9%
All six domains	Detection	0.393	0.298	0.576	+47%
	Macro	0.427	0.395	0.531	+87%
	Positive macro	0.292	0.223	0.453	+136%
	Weighted macro	0.734	0.787	0.706	-2%
	Existence	0.246	0.195	0.333	+126%
	Executive	0.306	0.203	0.619	+222%
	Property	0.325	0.271	0.406	+92%
	Non-Architectural	0.833	0.913	0.765	-9%

Table 58. TF-IDF performance on the random samples.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.423	0.326	0.604	+98%
	Macro	0.482	0.447	0.554	+106%
	Positive macro	0.349	0.283	0.462	+206%
	Weighted macro	0.792	0.829	0.767	+2%
	Existence	0.386	0.333	0.457	+139%
	Executive	0.279	0.207	0.429	+310%
	Property	0.381	0.308	0.500	+237%
	Non-Architectural	0.881	0.939	0.830	-6%
Web development	Detection	0.375	0.270	0.614	+89%
	Macro	0.430	0.390	0.554	+83%
	Positive macro	0.286	0.206	0.473	+222%
	Weighted macro	0.785	0.844	0.752	-4%
	Existence	0.267	0.189	0.455	+156%
	Executive	0.407	0.300	0.632	+347%
	Property	0.185	0.128	0.333	+157%
	Non-Architectural	0.863	0.943	0.795	-8%
All six domains	Detection	0.376	0.311	0.475	+40%
	Macro	0.414	0.395	0.465	+81%
	Positive macro	0.266	0.226	0.347	+115%
	Weighted macro	0.747	0.776	0.727	-0%
	Existence	0.194	0.158	0.250	+78%
	Executive	0.375	0.279	0.571	+295%
	Property	0.230	0.241	0.219	+36%
	Non-Architectural	0.857	0.900	0.818	-6%

Table 59. TF-IDF* performance on the random samples.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.430	0.309	0.708	+101%
	Macro	0.492	0.438	0.685	+110%
	Positive macro	0.369	0.267	0.652	+224%
	Weighted macro	0.781	0.839	0.760	+1%
	Existence	0.489	0.390	0.657	+204%
	Executive	0.286	0.179	0.714	+320%
	Property	0.333	0.233	0.583	+195%
	Non-Architectural	0.860	0.952	0.784	-8%
Web development	Detection	0.346	0.241	0.614	+75%
	Macro	0.407	0.377	0.564	+73%
	Positive macro	0.263	0.189	0.498	+195%
	Weighted macro	0.764	0.840	0.726	-6%
	Existence	0.327	0.273	0.409	+215%
	Executive	0.299	0.191	0.684	+228%
	Property	0.162	0.102	0.400	+125%
	Non-Architectural	0.842	0.941	0.761	-11%
All six domains	Detection	0.410	0.324	0.559	+53%
	Macro	0.435	0.396	0.537	+90%
	Positive macro	0.296	0.223	0.450	+138%
	Weighted macro	0.750	0.787	0.734	+0%
	Existence	0.286	0.231	0.375	+162%
	Executive	0.253	0.172	0.476	+166%
	Property	0.348	0.267	0.500	+106%
	Non-Architectural	0.851	0.913	0.798	-7%

Table 60. BOWN performance on the random samples.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.436	0.387	0.500	+104%
	Macro	0.490	0.487	0.498	+109%
	Positive macro	0.350	0.339	0.367	+207%
	Weighted macro	0.820	0.835	0.807	+6%
	Existence	0.455	0.484	0.429	+182%
	Executive	0.188	0.167	0.214	+176%
	Property	0.407	0.367	0.458	+261%
	Non-Architectural	0.910	0.929	0.892	-3%
Web development	Detection	0.403	0.312	0.568	+104%
	Macro	0.435	0.400	0.523	+85%
	Positive macro	0.283	0.220	0.415	+218%
	Weighted macro	0.808	0.843	0.786	-1%
	Existence	0.208	0.192	0.227	+100%
	Executive	0.433	0.317	0.684	+376%
	Property	0.208	0.152	0.333	+189%
	Non-Architectural	0.891	0.941	0.846	-5%
All six domains	Detection	0.342	0.345	0.339	+28%
	Macro	0.403	0.418	0.407	+76%
	Positive macro	0.242	0.262	0.247	+95%
	Weighted macro	0.768	0.772	0.768	+3%
	Existence	0.150	0.188	0.125	+38%
	Executive	0.340	0.281	0.429	+257%
	Property	0.235	0.316	0.188	+39%
	Non-Architectural	0.887	0.886	0.889	-3%

Table 61. BOWN* performance on the random samples.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.538	0.556	0.521	+151%
	Macro	0.532	0.589	0.497	+127%
	Positive macro	0.396	0.473	0.348	+247%
	Weighted macro	0.850	0.863	0.842	+10%
	Existence	0.421	0.545	0.343	+162%
	Executive	0.267	0.250	0.286	+292%
	Property	0.500	0.625	0.417	+342%
	Non-Architectural	0.939	0.935	0.943	+0%
Web development	Detection	0.324	0.458	0.250	+63%
	Macro	0.467	0.602	0.419	+99%
	Positive macro	0.311	0.499	0.237	+249%
	Weighted macro	0.852	0.858	0.864	+5%
	Existence	0.286	0.667	0.182	+175%
	Executive	0.312	0.385	0.263	+243%
	Property	0.333	0.444	0.267	+363%
	Non-Architectural	0.937	0.912	0.963	-1%
All six domains	Detection	0.220	0.391	0.153	-18%
	Macro	0.314	0.373	0.300	+37%
	Positive macro	0.115	0.209	0.080	-8%
	Weighted macro	0.764	0.747	0.797	+2%
	Existence	0.121	0.222	0.083	+11%
	Executive	0.125	0.182	0.095	+32%
	Property	0.098	0.222	0.062	-42%
	Non-Architectural	0.911	0.867	0.959	-1%

Table 62. BOWF performance on the random samples.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.547	0.500	0.604	+156%
	Macro	0.534	0.544	0.541	+128%
	Positive macro	0.402	0.411	0.415	+252%
	Weighted macro	0.841	0.858	0.828	+9%
	Existence	0.426	0.500	0.371	+165%
	Executive	0.378	0.304	0.500	+456%
	Property	0.400	0.429	0.375	+254%
	Non-Architectural	0.931	0.944	0.918	-1%
Web development	Detection	0.330	0.319	0.341	+67%
	Macro	0.422	0.414	0.432	+80%
	Positive macro	0.258	0.246	0.272	+190%
	Weighted macro	0.824	0.826	0.823	+1%
	Existence	0.182	0.182	0.182	+75%
	Executive	0.350	0.333	0.368	+285%
	Property	0.242	0.222	0.267	+237%
	Non-Architectural	0.914	0.918	0.910	-3%
All six domains	Detection	0.314	0.372	0.271	+17%
	Macro	0.386	0.416	0.373	+69%
	Positive macro	0.215	0.262	0.190	+73%
	Weighted macro	0.773	0.766	0.785	+3%
	Existence	0.227	0.250	0.208	+109%
	Executive	0.244	0.250	0.238	+157%
	Property	0.174	0.286	0.125	+3%
	Non-Architectural	0.900	0.880	0.921	-2%

Table 63. BOWF* performance on the random samples.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.633	0.528	0.792	+196%
	Macro	0.613	0.574	0.695	+162%
	Positive macro	0.506	0.442	0.625	+344%
	Weighted macro	0.867	0.885	0.859	+12%
	Existence	0.571	0.571	0.571	+255%
	Executive	0.364	0.316	0.429	+435%
	Property	0.583	0.438	0.875	+416%
	Non-Architectural	0.935	0.970	0.903	-0%
Web development	Detection	0.396	0.339	0.477	+100%
	Macro	0.513	0.487	0.566	+118%
	Positive macro	0.382	0.339	0.460	+329%
	Weighted macro	0.837	0.853	0.828	+3%
	Existence	0.364	0.303	0.455	+250%
	Executive	0.526	0.526	0.526	+478%
	Property	0.255	0.188	0.400	+255%
	Non-Architectural	0.908	0.932	0.885	-4%
All six domains	Detection	0.455	0.490	0.424	+70%
	Macro	0.462	0.475	0.452	+102%
	Positive macro	0.312	0.333	0.295	+151%
	Weighted macro	0.803	0.799	0.809	+8%
	Existence	0.333	0.333	0.333	+206%
	Executive	0.263	0.294	0.238	+177%
	Property	0.339	0.370	0.312	+101%
	Non-Architectural	0.913	0.903	0.924	-0%

Table 64. RNN performance on the random samples.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.533	0.414	0.750	+149%
	Macro	0.547	0.511	0.620	+134%
	Positive macro	0.428	0.360	0.542	+276%
	Weighted macro	0.829	0.861	0.812	+7%
	Existence	0.481	0.362	0.714	+199%
	Executive	0.258	0.235	0.286	+280%
	Property	0.545	0.484	0.625	+383%
	Non-Architectural	0.905	0.962	0.855	-3%
Web development	Detection	0.389	0.319	0.500	+97%
	Macro	0.471	0.459	0.510	+100%
	Positive macro	0.328	0.300	0.391	+268%
	Weighted macro	0.823	0.848	0.806	+1%
	Existence	0.310	0.224	0.500	+198%
	Executive	0.486	0.500	0.474	+435%
	Property	0.188	0.176	0.200	+160%
	Non-Architectural	0.900	0.934	0.868	-5%
All six domains	Detection	0.407	0.407	0.407	+52%
	Macro	0.404	0.492	0.403	+77%
	Positive macro	0.240	0.357	0.239	+94%
	Weighted macro	0.777	0.804	0.775	+4%
	Existence	0.299	0.233	0.417	+174%
	Executive	0.171	0.214	0.143	+80%
	Property	0.250	0.625	0.156	+48%
	Non-Architectural	0.897	0.897	0.897	-2%

Table 65. RNN* performance on the random samples.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.535	0.605	0.479	+150%
	Macro	0.484	0.539	0.448	+107%
	Positive macro	0.331	0.409	0.278	+190%
	Weighted macro	0.846	0.850	0.847	+9%
	Existence	0.467	0.560	0.400	+190%
	Executive	0.167	0.200	0.143	+145%
	Property	0.359	0.467	0.292	+218%
	Non-Architectural	0.944	0.931	0.957	+1%
Web development	Detection	0.238	0.250	0.227	+20%
	Macro	0.327	0.330	0.325	+39%
	Positive macro	0.133	0.138	0.128	+49%
	Weighted macro	0.807	0.803	0.811	-1%
	Existence	0.227	0.227	0.227	+119%
	Executive	0.171	0.188	0.158	+88%
	Property	0.000	0.000	0.000	-100%
	Non-Architectural	0.911	0.906	0.916	-3%
All six domains	Detection	0.206	0.263	0.169	-23%
	Macro	0.307	0.354	0.296	+34%
	Positive macro	0.113	0.183	0.089	-9%
	Weighted macro	0.748	0.743	0.766	+0%
	Existence	0.143	0.167	0.125	+31%
	Executive	0.049	0.050	0.048	-49%
	Property	0.146	0.333	0.094	-13%
	Non-Architectural	0.890	0.865	0.918	-3%

Table 66. DOC2VEC performance on the random samples.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.465	0.526	0.417	+117%
	Macro	0.479	0.544	0.439	+105%
	Positive macro	0.327	0.418	0.269	+187%
	Weighted macro	0.835	0.841	0.835	+8%
	Existence	0.414	0.522	0.343	+157%
	Executive	0.261	0.333	0.214	+284%
	Property	0.308	0.400	0.250	+172%
	Non-Architectural	0.936	0.923	0.949	-0%
Web development	Detection	0.226	0.194	0.273	+14%
	Macro	0.325	0.316	0.341	+38%
	Positive macro	0.139	0.119	0.168	+56%
	Weighted macro	0.782	0.799	0.767	-4%
	Existence	0.189	0.161	0.227	+81%
	Executive	0.163	0.133	0.211	+79%
	Property	0.065	0.062	0.067	-10%
	Non-Architectural	0.882	0.905	0.860	-6%
All six domains	Detection	0.400	0.411	0.390	+49%
	Macro	0.442	0.490	0.428	+93%
	Positive macro	0.290	0.355	0.270	+134%
	Weighted macro	0.789	0.802	0.787	+6%
	Existence	0.408	0.400	0.417	+274%
	Executive	0.113	0.094	0.143	+19%
	Property	0.348	0.571	0.250	+106%
	Non-Architectural	0.899	0.895	0.903	-2%

Table 67. DOC2VEC* performance on the random samples.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.214	0.120	1.000	+0%
	Macro	0.085	0.046	0.750	-63%
	Positive macro	0.114	0.061	1.000	-0%
	Weighted macro	0.022	0.012	0.172	-97%
	Existence	0.161	0.087	1.000	-0%
	Executive	0.068	0.035	1.000	-1%
	Property	0.113	0.060	1.000	+0%
	Non-Architectural	0.000	0.000	0.000	-100%
Web developmen	Detection	0.198	0.110	1.000	+0%
	Macro	0.067	0.035	0.750	-72%
	Positive macro	0.089	0.047	1.000	+0%
	Weighted macro	0.012	0.006	0.136	-98%
	Existence	0.104	0.055	1.000	+0%
	Executive	0.091	0.048	1.000	-0%
	Property	0.072	0.037	1.000	+0%
	Non-Architectural	0.000	0.000	0.000	-100%
All six domains	Detection	0.257	0.147	1.000	-4%
	Macro	0.090	0.048	0.750	-61%
	Positive macro	0.120	0.064	1.000	-3%
	Weighted macro	0.023	0.012	0.184	-97%
	Existence	0.113	0.060	1.000	+4%
	Executive	0.100	0.052	1.000	+5%
	Property	0.148	0.080	1.000	-12%
	Non-Architectural	0.000	0.000	0.000	-100%

Table 68. CNN performance on the random samples.

Domain	Metric	F1	P	R	Impr. over best guessing
Data storage & processing	Detection	0.214	0.120	1.000	+0%
	Macro	0.085	0.046	0.750	-63%
	Positive macro	0.114	0.061	1.000	-0%
	Weighted macro	0.022	0.012	0.172	-97%
	Existence	0.161	0.087	1.000	-0%
	Executive	0.068	0.035	1.000	-1%
	Property	0.113	0.060	1.000	+0%
	Non-Architectural	0.000	0.000	0.000	-100%
Web development	Detection	0.198	0.110	1.000	+0%
	Macro	0.067	0.035	0.750	-72%
	Positive macro	0.089	0.047	1.000	+0%
	Weighted macro	0.012	0.006	0.136	-98%
	Existence	0.104	0.055	1.000	+0%
	Executive	0.091	0.048	1.000	-0%
	Property	0.072	0.037	1.000	+0%
	Non-Architectural	0.000	0.000	0.000	-100%
All six domains	Detection	0.257	0.147	1.000	-4%
	Macro	0.090	0.048	0.750	-61%
	Positive macro	0.120	0.064	1.000	-3%
	Weighted macro	0.023	0.012	0.184	-97%
	Existence	0.113	0.060	1.000	+4%
	Executive	0.100	0.052	1.000	+5%
	Property	0.148	0.080	1.000	-12%
	Non-Architectural	0.000	0.000	0.000	-100%

Table 69. CNN* performance on the random samples.

H Confidence Intervals for Random Samples

In this appendix, we present the performance of the BERT model (both with and without preprocessing) on the three random samples of issues we collected and labelled during this work. Table 70 contains the results for the sample from the web development domain for the BERT model without preprocessing. Table 71 contains the results from the sample from the data storage & processing domain for the BERT model without preprocessing. Finally, Table 72 contains the result from the random sample taken from all six domains for the BERT model without preprocessing. Tables 73, 74, 75 contain the results for the BERT model with preprocessing, in the same order.

In each table, we present the precision and recall, alongside their corresponding 95% confidence intervals for every class. For each class, we give the metric (precision or recall), the numbers of issues involved in computing the metric, the standard error, and the resulting confidence interval. Recall that

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \qquad \text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Hence, the amount of issues involved in computing the precision is the true positive count plus the false positive count. For recall, this is the true positive count plus the false negative count.

Next, given a fraction $\hat{p}$ and amount of issues n, the standard error was computed as

$$\text{SE}(\hat{p}) = \sqrt{\frac{\hat{p}(1-\hat{p})}{n}}$$

Finally, the 95% confidence interval was computed as

$$I = [\max\{0, \hat{p} - 1.96 * \text{SE}(\hat{p})\}, \min\{1, \hat{p} + 1.96 * \text{SE}(\hat{p})\}]$$

Note that for every table, the "Architectural" class gives the precision and recall (and related statistics) for the detection task; i.e. for the performance when the multi-class model's output is interpreted as a binary output for the detection of architectural issues.

		Existence	Executive	Property	Non-Architectural	Architectural
Precision	Precision	0.367	0.410	1.000	0.949	0.391
	TP + FP	30	39	2	331	69
	Standard Error	0.088	0.079	0	0.012	0.059
	Confidence Interval	[0.1942, 0.5391]	[0.2559, 0.5646]	[1.0000, 1.0000]	[0.9249, 0.9724]	[0.2761, 0.5065]
Recall	Recall	0.500	0.842	0.133	0.882	0.614
	TP + FN	22	19	15	356	44
	Standard Error	0.107	0.084	0.088	0.017	0.073
	Confidence Interval	[0.2911, 0.7089]	[0.6781, 1.0000]	[0.0000, 0.3054]	[0.8485, 0.9155]	[0.4698, 0.7575]

Table 70. Confidence Intervals of the precision and recall per class for the performance of BERT (without preprocessing) on the random sample of issues from the web development domain.

		Existence	Executive	Property	Non-Architectural	Architectural
Precision	Precision	0.500	0.308	0.545	0.949	0.456
	TP + FP	40	26	11	332	68
	Standard Error	0.079	0.090	0.150	0.012	0.060
	Confidence Interval	[0.3450, 0.6550]	[0.1303, 0.4851]	[0.2512, 0.8397]	[0.9251, 0.9725]	[0.3375, 0.5743]
Recall	Recall	0.571	0.571	0.250	0.895	0.646
	TP + FN	35	14	24	352	48
	Standard Error	0.084	0.132	0.088	0.016	0.069
	Confidence Interval	[0.4075, 0.7354]	[0.3122, 0.8307]	[0.0768, 0.4232]	[0.8628, 0.9269]	[0.5105, 0.7811]

Table 71. Confidence Intervals of the precision and recall per class for the performance of BERT (without preprocessing) on the random sample of issues from the data storage & processing domain.

		Existence	Executive	Property	Non-Architectural	Architectural
Precision	Precision	0.400	0.268	0.750	0.915	0.508
	TP + FN	15	41	8	341	59
	Standard Error	0.127	0.069	0.153	0.015	0.065
	Confidence Interval	[0.1521, 0.6479]	[0.1327, 0.4039]	[0.4499, 1.0000]	[0.8853, 0.9446]	[0.3809, 0.6360]
Recall	Recall	0.250	0.524	0.188	0.915	0.508
	TP + FN	24	21	32	341	59
	Standard Error	0.088	0.109	0.069	0.015	0.065
	Confidence Interval	[0.0768, 0.4232]	[0.3102, 0.7374]	[0.0523, 0.3227]	[0.8853, 0.9446]	[0.3809, 0.6360]

Table 72. Confidence Intervals of the precision and recall per class for the performance of BERT (without preprocessing) on the random sample of issues taken from all six domains.

		Existence	Executive	Property	Non-Architectural	Architectural
Precision	Precision	0.438	0.556	0.444	0.931	0.513
	TP + FN	16	18	9	363	37
	Standard Error	0.124	0.117	0.166	0.013	0.082
	Confidence Interval	[0.1944, 0.6806]	[0.3260, 0.7851]	[0.1198, 0.7691]	[0.9051, 0.9572]	[0.3525, 0.6746]
Recall	Recall	0.318	0.526	0.267	0.949	0.432
	TP + FN	22	19	15	356	44
	Standard Error	0.099	0.115	0.114	0.012	0.075
	Confidence Interval	[0.1235, 0.5128]	[0.3018, 0.7508]	[0.0429, 0.4905]	[0.9267, 0.9722]	[0.2855, 0.5782]

Table 73. Confidence Intervals of the precision and recall per class for the performance of BERT (with preprocessing) on the random sample of issues taken from the web development domain.

		Existence	Executive	Property	Non-Architectural	Architectural
Precision	Precision	0.621	0.462	0.464	0.946	0.592
	TP + FN	29	13	28	351	49
	Standard Error	0.090	0.138	0.094	0.012	0.070
	Confidence Interval	[0.4441, 0.7973]	[0.1905, 0.7325]	[0.2796, 0.6490]	[0.9222, 0.9695]	[0.4542, 0.7295]
Recall	Recall	0.514	0.429	0.542	0.943	0.604
	TP + FN	35	14	24	352	48
	Standard Error	0.085	0.132	0.102	0.012	0.071
	Confidence Interval	[0.3487, 0.6799]	[0.1693, 0.6878]	[0.3423, 0.7410]	[0.9190, 0.9674]	[0.4658, 0.7425]

Table 74. Confidence Intervals of the precision and recall per class for the performance of BERT (with preprocessing) on the random sample of issues taken from the data storage & processing domain.

		Existence	Executive	Property	Non-Architectural	Architectural
Precision	Precision	0.417	0.346	0.562	0.902	0.545
	TP + FN	12	26	16	356	44
	Standard Error	0.142	0.093	0.124	0.016	0.075
	Confidence Interval	[0.1377, 0.6956]	[0.1633, 0.5290]	[0.3194, 0.8056]	[0.8708, 0.9326]	[0.3983, 0.6926]
Recall	Recall	0.208	0.429	0.281	0.941	0.407
	TP + FN	24	21	32	341	59
	Standard Error	0.083	0.108	0.080	0.013	0.064
	Confidence Interval	[0.0459, 0.3708]	[0.2169, 0.6402]	[0.1255, 0.4370]	[0.9164, 0.9663]	[0.2814, 0.5321]

Table 75. Confidence Intervals of the precision and recall per class for the performance of BERT (with preprocessing) on the random sample of issues taken from all six domains.

I Replication

In this section, we provide a brief guide on how to reproduce our results using Maestro.

First, Maestro must be installed. Installation guides can be found in the main Maestro GitHub repository[31]. Additionally, detailed installation instructions for the deep learning component are also available[32]. These instructions also explain how to deploy the deep learning manager on high performance computing clusters.

I.1 Viewing Issues

In order to get a list of the issues we collected, Go to the the "Classify Issues" tab and press "Create Query". Next, un-check the "Simplified query?" box. Next, enter a query in the box. Queries are written using Mongo query syntax. The following is a good starting query:

```
{
  "$and": [
    {"tags": {"$eq": "has-label"}},
    {
      "$or": [
        ...
      ]
    }
  ]
}
```

Here, the contents of the inner "$or" should be a list of tags (in the `{"tags": {"$eq": <tag>}}` format). A list of available tags per round of issue collecting we performed can be found in Appendix J.

When creating a query for one of the "bert-round-X" tags, it is also possible to select the corresponding BERT model in the "Select ML Models" section. There are three available models: "bert-round-1", "bert-round-2", and "bert-round-3". We advise only using a single tag (e.g. only "bert-round-2-property") when selecting one of these models. Selecting a model will make Maestro display the classifier predictions alongside the issues.

When not using an ML model, press the "Remove this ML Model Option" button.

Finally, give the query a name and save it. Pressing the name of the query in the "Classify Issues" tab will now bring up the list of selected issues.

I.2 Reviewing Model Performance Metrics & Checking Model Settings

In order to double check model performance metrics or check model settings, go the "ML models" tab, and then the "ML Models" sub-tab. This will bring up a list of available model configurations stored in the database. A list of all configurations created during our research is given in Appendix K. When clicking on the name of the model, Maestro will bring up a screen containing two types of data: the performance of the model the last time it was trained, and the settings (hyperparameters) with which the model was trained. Although Appendix K does provide the name of all the models, we advise double checking the model ID in the address bar.

Note: everything from this point onward will require you to be logged in to the UI. Instructions for creating an account can be found in the user documentation[33].

The "(Re)train Model" button can be used to train the model again, in order to validate that its performance is reproducible.

A new model can be created by going to the "ML Models" sub-tab again, and pressing the "Create New Model" button. The hyperparameters for the tabs "Classifier" and "Training" can be found in Appendix C. For the hyperparameters in the "Pre-Processing" tab, most settings can be kept as defaults. The only difference is that the "Input Mode" must be set to determine the feature generation, and a corresponding Word Embedding must be selected (the name "word embedding" here is mostly a historical artefact. For instance, for bag of words models, the word embedding is simply a dictionary of known words). The identifiers of the embeddings we used can be found in the configurations of the different models (in the Maestro configuration view).

In case one wants to re-generate the word embeddings too, this can be done by going to the "ML Models" tab and then the "Word Embeddings" sub-tab. The hyperparameters for the different embeddings are given in Table 76.

[31]`https://github.com/mining-design-decisions/Maestro`
[32]`https://github.com/mining-design-decisions/Maestro/blob/main/docs/usage/dl_manager/index.md`
[33]`https://github.com/mining-design-decisions/Maestro/blob/main/docs/usage/issues_db_api/README.md#users`

Embedding	Used by	Hyperparameters
Word2Vec	CNN, RNN	Trained on all issues from the six domains. Use stemming: No. Use lemmatization: Yes. Use part-of-speech (pos) tagging: No. Formatting handling: Markers. Use ontologies: No. Vector length: 300. Min count: 5. Algorithm: skip-gram.
Doc2Vec	Doc2Vec	Trained on all issues from the six domains. Use stemming: No. Use lemmatization: Yes. Use part-of-speech (pos) tagging: No. Formatting handling: Markers. Use ontologies: No. Vector length: 300. Min count: 5. Algorithm: skip-gram.
IDFGenerator	TF-IDF	Trained on all labelled issues from the data storage & processing domain. Use stemming: No. Use lemmatization: Yes. Use part-of-speech (pos) tagging: No. Formatting handling: Markers. Use ontologies: No. Min doc count: 0
DictionaryGenerator	BOW (Frequency & Normalised)	Trained on all labelled issues from the data storage & processing domain. Use stemming: No. Use lemmatization: Yes. Use part-of-speech (pos) tagging: No. Formatting handling: Markers. Use ontologies: No. Min doc count: 0

Table 76. Hyperparameters for the different embeddings we used.

For training on all issues from the six domains, the following query can be used:

```
{
  "$or": [
    {"tags": {"$eq": "project-merged_domain=software development tools"}},
    {"tags": {"$eq": "project-merged_domain=devops and cloud"}},
    {"tags": {"$eq": "project-merged_domain=data storage & processing"}},
    {"tags": {"$eq": "project-merged_domain=web development"}},
    {"tags": {"$eq": "project-merged_domain=content management"}},
    {"tags": {"$eq": "project-merged_domain=soa and middlewares"}}
  ]
}
```

For training on all labelled issues from the data processing & storage domain, the following query can be used:

```
{
  "$and": [
    {"tags": {"$eq": "has-label"}},
    {"tags": {"$eq": "project-merged_domain=data storage & processing"}}
  ]
}
```

Enabling fine-grained technology replacement is currently not directly supported by the Maestro UI, and requires manipulation of the *saved* embedding and model configurations through the database API. The Python script below can be used to enable fine-grained technology replacements. Note that the script requires the database client library[34] to be installed.

```python
import issue_db_api

database_url = 'https://url.to.database'
database_username = 'your-username'
database_password = 'your-password'

target = 'embedding'    # 'embedding' or 'model'
target_id = 'some-id'   # Obtained from the UI

repo = issue_db_api.IssueRepository(database_url,
                                    credentials=(database_username, database_password))

if target == 'embedding':
    obj = repo.get_embedding_by_id(target_id)
else:
    assert target == 'model'
    obj = repo.get_model_by_id(target_id)

new_config = obj.config | {
    'replace-this-technology-mapping': '6491b09727a779d274330603',
    'this-technology-replacement': 'development project',
    'replace-other-technologies-list': '6491b0be27a779d274330605',
    'other-technology-replacement': 'technology'
}

obj.config = new_config     # Uploads new config to the database
```

[34]https://github.com/mining-design-decisions/maestro-issue-db-api-client

J Database Tags

In Table 77 we provide the tags we used in the database to specify what issues were labelled during which round of labelling.

Tag	Description
maven	Issues found using Maven POM dependencies analysis.
top-down	Issues found using keyword searches.
bottom-up	Issues found using static source code analysis.
bottom-up-discovered	Issues found using static source code analysis, but these were discovered after our internship [18].
random-sample-data-projects	Issues found with a random sample of 400 issues from the Apache projects Hadoop, Cassandra, Tajo, HDFS, MapReduce, and Yarn.
random-sample-web-projects	Issues found with a random sample of 400 issues from the Apache projects Solr, CloudStack, JSPWiki, Brooklyn, and TomEE.
search-engine-reusable-solutions-2023-02-22	Issues that were found with keyword searches on 2023-02-22, using the reusable solutions keywords.
search-engine-decision-factors-2023-02-22	Issues that were found with keyword searches on 2023-02-22, using the decision factors keywords.
bert-round-1	Issues that were found with BERT version 1, for the first round of finding property issues with BERT. The 121 labelled issues with the highest confidence have the refined tag *bert-round-1-top-121.* The 50 labelled issues with the lowest confidence are tagged with the refined tag *bert-round-1-bottom-50.*
bert-round-2-property	Issues that were found with BERT version 2, for the second round of finding property issues with BERT.
bert-round-2-executive	Issues that were found with BERT version 2, for the first round of finding executive issues with BERT.
bert-round-3-executive	Issues that were found with BERT version 3, for the second round of finding executive issues with BERT.
bert-round-3-executive-keyword-filtered	Issues that were found with BERT version 3, for the third round of finding executive issues with BERT.
random-sample-domains	Issues found using a random sample of 400 issues from six software domains.
random-confidence-sample-domains	Issues found using a random sample, based on the confidence of a deep learning classifier. These issues are also selected from six software domains.
relabeling-non-arch-version-bumps	Issues we relabelled, because we changed our opinion on how we should label issues about technology version bumps.
relabeling-internship	Issues we labelled during our internship [18], but we decided to relabel certain problematic issues.
relabeling-top-down-bottom-up-*	Issues found with the keyword searches and static source code analysis, which were found to be of low quality and hence relabeled. Each of the labellers labelled a subset of these issues. These subsets can be found by replacing * with the name of the labeller (jesse, arjan, or mohamed).
relabeling-master-thesis	Issues labelled during this thesis, which were found to be of low quality and are therefore relabelled. Each of the labellers labelled a subset of these issues. These subsets can be found by replacing * with the name of the labeller (jesse, arjan, or mohamed).
relabelling-msc-thesis-existence	Issues labelled as solely existence during this thesis, which were relabelled due to having low quality labels.

Table 77. This table contains the tags we used in the database to specify what issues were (re)labelled during which round of labelling.

K Trained Models

This section contains the IDs of the models, versions, and performances we used. These IDs can be used to find the corresponding models, versions, and performances in our database. Table 78 contains the multi-label models we trained on the issues from the data storage & processing domain. Tables 79 and 80 contain the detection and multi-class models from our internship [18] that are trained on that same dataset. We also trained multi-label models on that dataset, but excluding the issues from the random sample, because we used the random sample as the test set for that evaluation. These models are shown in Table 81. Table 82 contains the models we evaluated using varying training set sizes. Finally, Table 83 contains the models we used for cross-project validation.

Model	Model ID	Version ID	Model Name
BERT without text preprocessing	648ee4526b3fde4b1b33e099	648f1f6f6b3fde4b1b3429cf	BERT best settings seed=4
BERT with text preprocessing	64c8abf36f7bb6a08a6e4041	64c8ca5e6f7bb6a08a6e4c7b	BERT best settings (with formatting removal)
BOWF	648f21ba6b3fde4b1b342fe9	648f21f86b3fde4b1b342fea	BOWF best params
BOWN	648f2a9c6b3fde4b1b343003	64c40b8276ddee87785b4afd	BOWN best params
TFIDF	648f430a6b3fde4b1b343075	648f43456b3fde4b1b343076	TFIDF best params
DOC2VEC	648f75de6b3fde4b1b3430e9	648f77276b3fde4b1b3430ea	DOC2VEC best params
CNN	648f79876b3fde4b1b34313e	64901b5a6b3fde4b1b343140	CNN best params
RNN	648f7ecf6b3fde4b1b34313f	64908d6027a779d2743303e2	RNN best params
BERT* without text preprocessing	649215cb27a779d27433882a	6492366827a779d274338850	BERT best settings seed=4 with replacement
BERT* with text preprocessing	64c8abfa6f7bb6a08a6e4042	64c8cc056f7bb6a08a6e5295	BERT best settings with replacement
BOWF*	6491b91227a779d27433060a	6491beb627a779d274335287	BOWF best params with replacement
BOWN*	6492020027a779d274336cc2	649203e027a779d274336d25	BOWN best params with replacement
TFIDF*	6491b64f27a779d274330608	6491bcb027a779d2743338b3	TFIDF best params with replacement
DOC2VEC*	64921e0927a779d274338830	6492a11f27a779d27433a9de	DOC2VEC best params with replacement
CNN*	64921f2827a779d274338834	6492a45b27a779d27433c473	CNN best params with replacement
RNN*	64921f9127a779d274338837	6492aaf527a779d27433c7a4	RNN best params with replacement

Table 78. This table contains the models we trained on the issues from the data storage & processing domain. Models with a * have fine-grained technology replacement.

Model	Model ID	Version ID	Model Name
BOWF Detection	64728704d364b0625efa6fb8	64c3c0fc76ddee87785b451b	Internship BOWFrequency Detection - New Dataset
BOWN Detection	64728704d364b0625efa6fb9	64c3c31776ddee87785b452a	Internship BOWNormalized Detection - New Dataset
TFIDF Detection	64728705d364b0625efa6fbb	64c3c3c476ddee87785b4548	Internship TFIDF Detection - New Dataset
DOC2VEC Detection	64728705d364b0625efa6fbd	64c3c41876ddee87785b4573	Internship Doc2Vec Detection - New Dataset
CNN Detection	64728706d364b0625efa6fc6	64c3ca7276ddee87785b48cf	Internship CNN Detection - New Dataset
RNN Detection	64728706d364b0625efa6fc9	64c3d44876ddee87785b4a18	Internship RNN Detection - New Dataset
Combination ensemble (BOWFreq/CNN/RNN)	64728706d364b0625efa6fca	64c3e25576ddee87785b4a45	Internship Detection Ensemble (BOWFreq/CNN/RNN) - Combination - New Dataset
Voting ensemble (BOWFreq/CNN/RNN)	64728706d364b0625efa6fcb	64c3e3f076ddee87785b4a76	Internship Detection Ensemble (BOWFreq/CNN/RNN) - Voting - New Dataset
Stacking ensemble (BOWFreq/CNN/RNN)	64728706d364b0625efa6fcc	647338d6d364b0625efa7574	Internship Detection Ensemble (BOWFreq/CNN/RNN) - Stacking - New Dataset

Table 79. This table contains the detection models from our internship [18] that we trained on the issues from the data storage & processing domain.

Model	Model ID	Version ID	Model Name
BOWF	64728705d364b0625efa6fbe	64c3c67976ddee87785b45db	Internship BOWFrequency Classification - New Dataset
BOWN	64728705d364b0625efa6fbf	64c3c7cc76ddee87785b4654	Internship BOWNormalized Classification - New Dataset
TFIDF	64728705d364b0625efa6fc2	64c3c9b876ddee87785b47f9	Internship TFIDF Classification - New Dataset
DOC2VEC	64728705d364b0625efa6fc1	64c3c94976ddee87785b4746	Internship Doc2Vec Classification - New Dataset
CNN	64728705d364b0625efa6fc4	64c3ca4776ddee87785b4878	Internship CNN Classification - New Dataset
RNN	64728706d364b0625efa6fc8	64c3d11476ddee87785b4903	Internship RNN Classification - New Dataset

Table 80. This table contains the multi-class models from our internship [18] that we trained on the issues from the data storage & processing domain.

Model	Model ID	Version ID	Model Name
BERT without text preprocessing	64bbf5bb5d41aef11824f350	64c227365d41aef1182505cf	BERT best settings seed=4 (without random sample)
BERT with text preprocessing	64c8c70f6f7bb6a08a6e4c79	64c8e6536f7bb6a08a6e5f17	BERT best settings (without random sample)
BOWF	64c50fcd76ddee87785b6a2a	64c50ffe76ddee87785b6a2b	BOWF best params (without random sample)
BOWN	64c50d9f76ddee87785b4f60	64c50f4f76ddee87785b6937	BOWN best params (without random sample)
TFIDF	64c50ca976ddee87785b4eec	64c50d3876ddee87785b4eed	TFIDF best params (without random sample)
DOC2VEC	64c512d476ddee87785b6c6e	64c5165076ddee87785b6ec3	DOC2VEC best params (without random sample)
CNN	64c516b476ddee87785b8866	64c5187276ddee87785ba1e4	CNN best params (without random sample)
RNN	64c5106f76ddee87785b6a4f	64c512a976ddee87785b6a51	RNN best params (without random sample)
BERT* without text preprocessing	649215cb27a779d27433882a	6492366827a779d274338850	BERT best settings seed=4 with replacement
BERT* with text preprocessing	64c8c7436f7bb6a08a6e4c7a	64c8e8076f7bb6a08a6e5f1d	BERT best settings with replacement (without random sample)
BOWF*	64c5100a76ddee87785b6a3b	64c5102c76ddee87785b6a3c	BOWF best params with replacement (without random sample)
BOWN*	64c50f0976ddee87785b4f61	64c50fb476ddee87785b69a0	BOWN best params with replacement (without random sample)
TFIDF*	64c50d6276ddee87785b4f5f	64c50f1a76ddee87785b4f62	TFIDF best params with replacement (without random sample)
DOC2VEC*	64c514f376ddee87785b6ec2	64c5180f76ddee87785b8867	DOC2VEC best params with replacement (without random sample)
CNN*	64c5185a76ddee87785ba1e3	64c518ac76ddee87785ba3ea	CNN best params with replacement (without random sample)
RNN*	64c5109a76ddee87785b6a50	64c5143476ddee87785b6c6f	RNN best params with replacement (without random sample)

Table 81. This table contains the models we trained on the issues from the data storage & processing domain, but excluding the issues from the random sample. Models with a * have fine-grained technology replacement.

Model	Model ID	Performance ID	Model Name
BERT without text preprocessing	6496cca0286e80e8e244b4ec	649796316794484c00b4cd65	BERT best settings (varying dataset size)
TFIDF	6496c93c286e80e8e244b4eb	6496d3ae286e80e8e244b4ed	TFIDF best params (varying dataset size)
RNN	64c6522e6f7bb6a08a6e4040	64cc20676f7bb6a08a6eedaa	RNN best settings (varying dataset size)
Internship BOWF Detection	64970037286e80e8e244cdd6	649726a6286e80e8e244dad4	Internship BOWF Detection (varying dataset size)
Internship CNN Detection	6496fef0286e80e8e244cdd5	64973235286e80e8e244e052	Internship CNN Detection (varying dataset size)
Internship BOWF Classification	6496ed4e286e80e8e244b990	64970c60286e80e8e244cdd7	Internship BOWF Classification3Simplified (varying dataset size)
Internship RNN SO Classification	6496f72e286e80e8e244b996	649ac79c277b34078bc74c4d	Internship RNN SO Classification3Simplified (varying dataset size)
BERT without text preprocessing (random sample as the test set)	64c243f85d41aef118250be8	64c4e99a76ddee87785b4b70	BERT best settings (varying dataset size, random sample as test set)

Table 82. This table contains the models we evaluated using different training dataset sizes. It also includes a model that we evaluated using the random sample as the test set.

Model	Model ID	Performance ID	Model Name
BERT without text preprocessing	6492161f27a779d27433882c	649241f527a779d274338fe8	BERT best settings seed=4 (cross-project)
BERT with text preprocessing	64c8c63d6f7bb6a08a6e4c77	64c8d7fa6f7bb6a08a6e58af	BERT best settings (cross-project)
BOWF	6492026727a779d274336cc3	6492030527a779d274336cc5	BOWF best params (cross-project)
BOWN	6492038027a779d274336d23	649204ac27a779d2743386f9	BOWN best params (cross-project)
TFIDF	6491c38d27a779d274336bee	6491cb3227a779d274336c57	TFIDF best params (cross-project)
DOC2VEC	64921d6827a779d27433882f	64929f9627a779d27433a975	DOC2VEC best params (cross-project)
CNN	64921edf27a779d274338832	6492a3cd27a779d27433c3c1	CNN best params (cross-project)
RNN	64921f5227a779d274338835	6492a78927a779d27433c6c2	RNN best params (cross-project)
BERT* without text preprocessing	649215ee27a779d27433882b	64924d6827a779d27433a94b	BERT best settings seed=4 with replacement (cross-project)
BERT* with text preprocessing	64c8c6866f7bb6a08a6e4c78	64c8e3d06f7bb6a08a6e58da	BERT best settings with replacement (cross-project)
BOWF*	649202b927a779d274336cc4	649205cd27a779d27433875d	BOWF best params with replacement (cross-project)
BOWN*	649203b327a779d274336d24	649208da27a779d2743387c4	BOWN best params with replacement (cross-project)
TFIDF*	6491c45727a779d274336bef	6491c69127a779d274336bf0	TFIDF best params with replacement (cross-project)
DOC2VEC*	64921e8327a779d274338831	6492a36327a779d27433c361	DOC2VEC best params with replacement (cross-project)
CNN*	64921f0b27a779d274338833	6492a42827a779d27433c425	CNN best params with replacement (cross-project)
RNN*	64921f7a27a779d274338836	6492aa6827a779d27433c738	RNN best params with replacement (cross-project)

Table 83. This table contains the models we used for the cross-project validation. Models with a * have fine-grained technology replacement.