Mining for Architectural Design Decisions in Issue Tracking Systems using Deep Learning Approaches

Arjan Dekker (s3726169) & Jesse Maarleveld (s3816567)

Supervisors: Mohamed Soliman & Paris Avgeriou

July 2022

1 Introduction

Issue tracking systems such as JIRA allow developers to discuss ideas about software projects. Some of these ideas may contain information relevant for the architecture of the project. Specifically, some issues may contain information about design decisions. Design decisions are an important form or architectural knowledge, because it is important to not only understand the architecture of a software project, but also *why* the architecture is as it is [5]. However, such architectural design decisions (ADDs) are often not explicitly documented [2]. As such, it can be difficult to nearly impossible to retroactively obtain these ADDs.

Issue trackers, such as JIRA, allow developers to discuss problems in or enhancements for software in so-called issues. These issues may contain architectural knowledge in the form of ADDs [1]. The major challenge is identifying and classifying these so-called architectural issues – issues containing one or multiple ADDs.

Previous researchers have used traditional machine learning (ML) approaches to identify and further classify architectural issues, with promising results [1]. In this report, we will investigate the effectiveness of applying modern deep learning (DL) approaches for the same purpose.

Powerful and generalizable automated methods for extracting ADDs from issue tracking systems could be beneficial. They can save considerable manual effort because identifying architectural knowledge can be a time consuming process. This information can be used to re-discover information about an architecture. Future researchers can also use deep learning tools to mine architectural issues on a larger scale. The information from them can be analyzed for future research, possibly leading to the development of architectural knowledge repositories – thus leading to a wider spread ability to share architectural knowledge amongst developers than currently available [2].

In section 2, we will cover the background of ADDs and issues in issue tracking systems in more detail. We will then introduce the research questions in section 3. Next, in sections 4 and 5, we will introduce the datasets we will be working with in more detail. In sections 6, 7, 8, and 9 we describe the main parts of our approach. Sections 10 and 11 explain further optimizations to our approach. We will then answer the research questions in sections 12, 13, 14, and 15.

2 Background

2.1 Architectural Design Decisions

Architectural Design Decisions are decisions that contain some form of information relevant to the architecture of the system. Kruchten [4] identified three categories of ADDs: Executive Decisions, Existence Decisions, and Property Decisions. We will briefly explain each type below.

2.1.1 Executive Decisions

Executive decisions are decisions driven by the (business) environment of the system. Such decisions may effect the development process or the organization developing the software itself. Often, such decisions also deal with a choice of technology [4]. An example of an executive issue is given in figure 1.

2.1.2 Existence Decisions

Existence decisions are ADDs which state that some functionality or component will show up in the system. When this is a statement about components, the decision is called structural. If the statement is about the interaction between components, the decision is called behavioral. A special case is a negative expression of an existence decision – a decision stating that some component or functionality will not show up in the system. Such decisions are called ban decisions [4]. An example of an existence decision is given in figure 2.

2.1.3 Property Decisions

Property decisions are ADDs which state enduring, overarching traits or qualities of the system. An example of this is a decision to not use any software using the GNU GPLv3 license as part of a project, because this license prohibits closed source distribution. Property decisions are often not

Figure 1: Example of an executive issue. The issue proposes to replace one technology with another (parts in red). Justification is given (in blue). This is hence an ADD which contains AK about a choice of technology.

Figure 2: Example of an existence issue. The issues states a problem (blue), discusses some aspects of a solution (green), and explains how implementing that solution leads to the addition of new components and new functionality to the system (red).

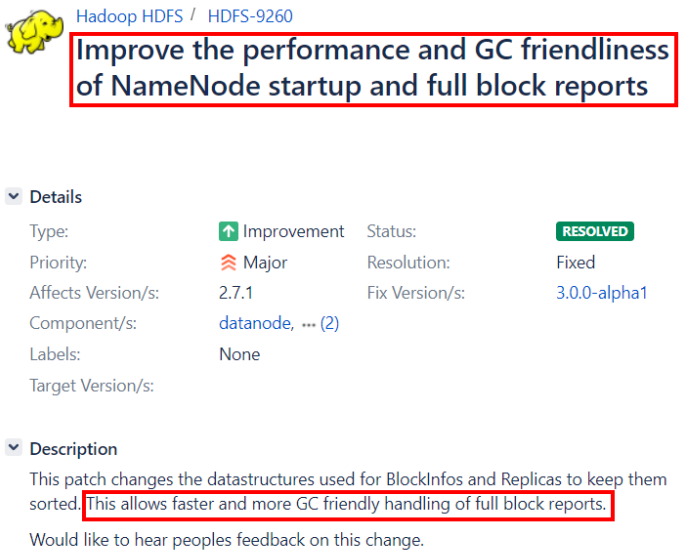

Figure 3: Example of a property issue. This issue proposes increasing the performance of certain processes within the system. Implicitly, this decision states that performance is important, making this a property decision.

written down and may be difficult to recognize as such. Often, they deal with one or more quality attribute [4]. An example of a property issue is given in figure 3.

2.2 Issue Properties

Issues in issue tracking systems contain various attributes which can be used for classification. In figures 1, 2, and 3 we already saw that issues have a title and description. These are the sources of information which allow us humans to classify issues are architectural. However, issues also have other properties which can be used for classification. Figure 4 contains examples of all the possible attributes we tested with. First of all, there is the type of the issue (blue). A common example of this is the type "bug". Next, there is the priority, indicating how urgent an issue is (magenta). The component field (green) indicates which components of the software are affected by the issue. Labels (black) can also be attached to an issue. These labels contain information not expressed in the other attributes. Labels can express intent (e.g. "supportability"), but also that an issue is appropriate to ease someone into contribution to open source projects (e.g. "beginner"). We also have the status field (light blue). This field indicates the current status of the issue – meaning whether is is active, or has been closed. The resolution field (pink) is closely related to the status, because it contains the final resolution. Possible examples include "won't fix" or "fixed" for bugs. There is also the votes field (brown), which indicates the amount of votes on an issue. Votes can be used by community members than an issue is somehow important. There is also the watches field (grey), which indicates the amount of people following ("watching") the

issue because they find it important.

Issues can also have attachments (yellow). For this work, we are only interested in the number of attachments. Similarly, issues may have links to other issues (orange). For instance, an overarching coordinating issue may have links to other smaller issues providing part of the implementation in some larger effort. We are once again only interested in the number of links to other issues. Closely related to these links, is the parent status. The issue in figure 4, has a parent issue (indicated by the red field). This means that it is part of some overarching larger effort. For the purpose of this research, we used the fact whether an issue has a parent as a feature.

3 Research Questions

In this report, we will attempt to answer the following research questions:

- ***(RQ1)*** *How accurate are deep learning approaches to identify and classify architectural issues?*

  Bhat et al. [1] used traditional machine learning approaches to identify and classify architectural issues. However, modern machine learning approaches like deep learning and word embedding opened the way to improve the accuracy of textual classification. Therefore, we ask this RQ to evaluate the accuracy of modern machine learning approaches to identify and classify architectural issues.

- ***(RQ2)*** *How would the training data-set of architectural issues impact the generalizability of deep learning approaches to identify and classify architectural issues?*

  The issues in the dataset we are using were obtained using bottom-up and top-down search approaches, and are thus not necessarily representative of issues in issue trackers in general [3]. Contrarily, Bhat et al [1] use a random sampling of issues from projects which they classify later. We are interested in answering how well the methods generalize to one dataset when training on the other (both ways). A well-generalizable approach is desirable because it allows us to extract other issues from issue trackers which are not necessarily similar to issues in the training dataset.

- ***(RQ3)*** *How generalizable are deep learning approaches to identify and classify architectural issues from different projects?*

  We are potentially interested in using deep learning to obtain issues from new projects not contained in the dataset. An important requirement to do this, is that the deep learning methods generalize well to projects not in the dataset. Hence, we ask this research question to evaluate the generalizability of deep learning to projects foreign to the training set.

- ***(RQ4)*** *What are the keywords used by deep learning approaches to identify and classify architectural issues?*

  We are interested in knowing how the deep learning models classify architectural issues. Most importantly, we are interested in knowing what words in the description and title of issues are important in determining the final classification. This interest is two-fold. We want to know whether the deep learning models use words humans would expect or use when classifying issues. On the other hand, it could be that the deep learning methods find new insightful keywords which human classifiers would not easily think of, which could lead to new insights. Therefore, we ask this research question to evaluate what keywords are important for deep learning algorithms when identifying and classifying issues.

4 Dataset Description

The dataset we used was based on [3]. In this work, a dataset containing both architectural issues and non-architectural issues was collected using top-down and bottom-up approaches [3]. The issues were obtained from multiple projects: Cassandra, Tajo, Hadoop Common, Hadoop HDFS, Hadoop Map Reduce, and Hadoop Yarn. Originally, his dataset contained a total number of 1846 issues, where 931 issues were non-architectural, and 915 were architectural. The architectural issues were further subdivided, into 587 existence issues, 199 property issues, and 129 executive issues.

Note that in the dataset, issues could have multiple labels (e.g. an issue could be both executive and existence). However, in order to simplify the classification task for the classifier, we only took the most important label. Here, executive has the highest priority, then property, and finally existence. This is because executive decisions tend to drive property issues, which in turn drive existence issues.

Part of our work involved checking the classification of the non-architectural issues. The primary supervisor found that in general, architectural issues were classified reasonably well (around 5% disagreement in a random sampling), while the non-architectural issues were classified somewhat worse (up to 30% disagreement in a random sampling). Based on this observation, we re-classified the non-architectural issues which did not have the label "bug". This is because issues with that labels were generally correctly classified. This means that a total of 712 issues had to be re-classified.

For the classification, the two authors first classified the same 50 issues independently and discussed disagreements. The primary supervisor then checked the re-classification of this 50 issues. The goal of this step was to establish a mutually understood baseline for the classification. Next, each author classified 331 of the remaining issues. All cases where one author was unsure were also checked by the other

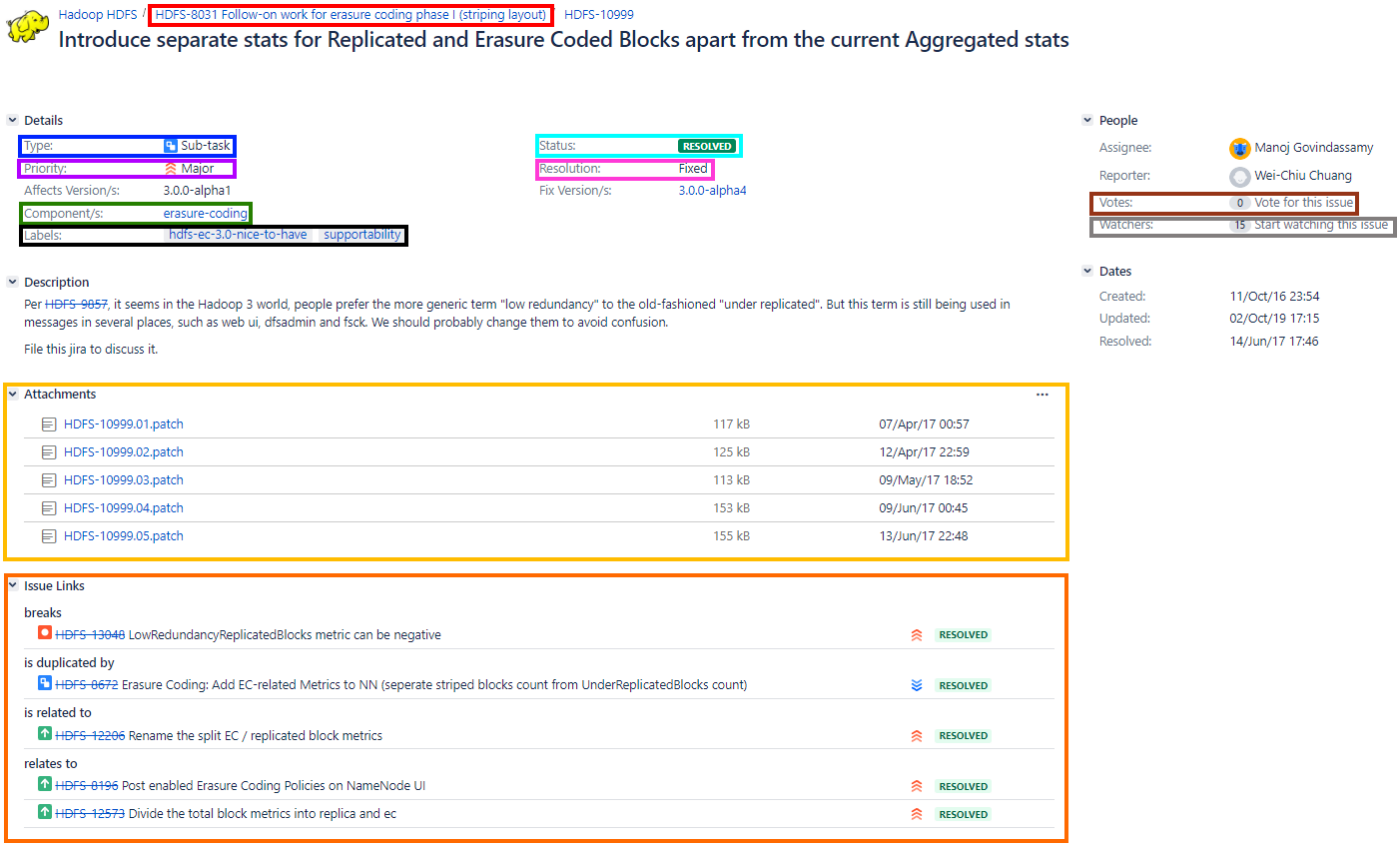

Figure 4: Examples of the possible properties an issue can have.

author. In case of disagreement, a final classification was determined by the primary supervisor.

In the end, 324 of the 712 issues were identified to be architectural. 272 were found to be existence issues, 26 were executive, and 26 were property issues.

Finally, another addition of 403 issues was made by another student project. These issues were obtained by analyzing Maven files. A total of 155 non-architectural issues, 69 existence issues, 21 property issues, and 158 executive issues were added. Additional, 9 of the existing issues were re-classified.

The final dataset we obtained this way is described in table 1.

5 Analyzing the Bhat Dataset

Part of the work we did, was to perform a qualitative analysis of the dataset collected by Bhat et al. This analysis had multiple goals. One of these goals was assessing the viability of using the dataset of Bhat et al. for evaluating the performance of our own deep learning methods. We performed the analysis by following the following steps:

1. *Locate and download Bhat´s dataset*: Bhat et al. do not provide their issues data-set in [1]. Thus, we contacted the primary author of the paper (i.e. Manoj Bhat) to ask for the data-set. Manoj Bhat kindly provided us with the issues dataset[1].

2. *Evaluate classifications on architectural issues*: We evaluated our agreement with the classification done by Bhat et al. We took a stratified sample of 300 issues from Bhat´s dataset. The sample ensures a representative distribution for all types of issues as classified by Bhat et al.: non-architectural issues, structural, behavioral and ban decisions. We independently re-classified the sample of issues using the definitions of Kruchten et al. [4] to determine the agreement on issues´ classifications. The two authors of this report and the supervisor worked together on this step. The authors classified 175 issues, with an overlap of 50 issues between them. Both authors discussed any disagreements in the overlapping issues. Moreover, issues which were difficult to classify were discussed between the two authors and the supervisor. In cases where no consensus could be reached by the two authors, the the supervisor provided the decisive classification.

   While Bhat et al. classified issues based solely on the existence types of design decisions, we found some issues in Bhat´s dataset that contain Executive and Property design decisions. These issues were re-classified based on their types of design decisions and according to definitions of Kruchten et al. [4]. As a result of this qualitative analysis, we determined the agreement on the different types of issues. Table 2 shows the agreement between our classification and Bhat´s classification. We can observe that the highest agreement is on splitting issues between architectural and non-architectural issues. However, we have lower agreement on the types of architectural issues. For instance, we did not agree with Bhat´s classification on issues that discuss Ban design decisions. There were also some issues for which there was too little information in the issue to come to a classifiaction.

The main conclusions from this analysis are that the dataset used by Bhat et al. can be used to test the detection of architectural issues. However, because of the large disagreement, the dataset of Bhat et al. will not be used in evaluating the performance of deep learning methods on the classification task. Table 3 gives a simple breakdown of the issues in this dataset of Bhat et al.

6 Prepare baseline

To evaluate our proposed machine learning approaches, we compare our approaches with the approach from Bhat et al.. We have chosen the approach from Bhat et al., because they classify issues among types of design decisions. However, Bhat et al. use a different dataset of issues (see Table section 5), and classify issues among the three types of existence design decisions: structural, behavioral and ban. It is also important to note that Bhat et al. have a two step approach for classifying issues. First, they use a classifier to determine whether an issue is a design decision. This process is called detection. Next, they use a second classifier to determine the type of the design decision. They refer to this as classification.

To accurately compare the approach from Bhat et al. with our proposed approaches, we replicated the approach from Bhat et al.. In this way, we could apply Bhat's traditional machine learning approaches on our dataset, and apply our proposed deep learning approaches on Bhat's dataset. We followed these steps to replicate the work from Bhat et al.:

1. *Acquire Bhat´s machine learning approach*: Bhat et al. [1] do not provide the implementation of their machine learning approach. Thus, we contacted the primary author of the paper (i.e. Manoj Bhat), who kindly provided us with a helpful source code implementation of their machine learning approach[2]. Since the repository has been updated after the work of Bhat et al. was published, we looked at older versions of the repository to understand Bhat´s approach. Specifically, we looked at at commit b140c81c1fdcb95364a36965e83850e672f90f13 because the content of this commit seemed to best line up with the contents of their paper. In the provided source code repository, we were able to find out some details on their approach, which the authors did not

[1] `https://server.sociocortex.com/typeDefinitions/1vk4hqzziw3jp/Task`

[2] `https://github.com/sebischair/DocClassification`

	Non-Architectural issues	Architectural Issues			
Projects		**Executive**	**Existence**	**Property**	**Total Per Project**
Cassandra	276	149	249	85	759
Hadoop Common	135	64	171	51	421
Hadoop HDFS	95	27	164	46	332
Hadoop Map Reduce	53	10	44	17	124
Tajo	98	38	69	13	218
Hadoop Yarn	91	7	202	25	325
Total per issue type	748	295	899	237	2179
			1431		

Table 1: Total Amount of issues in the dataset we used after re-classification and additions.

from / **to**	**Architectural (Ban)**	**Architectural (Behavioral)**	**Architectural (Structural**	**Not Architectural**	**Total**
Architectural (Behavioral)	7	64	4	3	78
Architectural (Executive)	0	2	6	0	8
Architectural (Existence)	0	1	0	0	1
Architectural (Structural)	3	4	19	0	26
Not Architectural	22	2	13	148	185
Indeterminable	0	1	1	0	2
Total	32	74	43	151	300

Table 2: Agreement with the original classification done by Bhat et al. For 300 issues. The columns denote the original classification, while the rows denote our new classification. The *Indeterminable* row is for issues where we could not come up with a classification for the issue because of too little information in the issue title and description.

Project	Architectural Issues	Non-Architectural Issues	Total Per Project
Hadoop	263	309	572
Spark	465	433	898
Total	728	742	1470

Table 3: Amount of issues in the dataset of Bhat et al.

clarify in their paper [1]. First, we found that they combined the summary and description of issues through concatenation. Second, we found that the authors used a custom list of stopwords. Third, we also found that the authors used Spark's NGram class[3]. Additionally, we were able to find the code for all of the five classifiers. However, the classifiers´ parameters did not agree with those specified in the work of Bhat et al. [1]. As a result of this step, we understood Bhat´s approach in more details to replicate the classification approach.

2. *Pre-process issues*: In order to train machine learning algorithms, we need to transform issues into feature vectors (i.e. numeric vectors that correspond to issues). We performed the same steps as Bhat et al. paper [1] to transform issues into feature vectors. Formatting was already removed from the issues in the dataset we were given. Hence, we only had to perform the remaining processing steps. First, we concatenated the summary and description for every issue. Second, we transformed the summary and description into a list of words. Third, we converted words to their lower case, and removed stopwords. We experimented with the custom list of stopwords used by Bhat et al. in their provided repository, as well as standard stopwords from the NLTK[4] library. Fourth, we applied stemming (e.g. "developing" becomes "develop") similar to Bhat et al., but using NLTK's implementation of the Porter stemming algorithm. Fifth, we created ngrams from the stemmed text similar to Bhat et al., where we experimented with values of n ranging from 1 to 5. For example, the sentence "Remove individual commit messages" contains the 2-grams "remove individual", "individual commit", "commit messages". In the code provided to us by the authors, we were able to see that the the authors generate n-grams of a specific length (e.g. at n=3), and append these ngrams to the list of words to create the feature vector. We followed the approach taken by Bhat et al. Finally, we transformed the list of words combined with ngrams into vectors of numbers. In accordance with the approach of Bhat et al, we used *term-frequency inverse document frequency* (tf/idf) to do this for detection, and term frequency for classification. We used the sklearn[5] library to compute ngrams, the tf/idf vectors, and *term-frequency* vectors, while Bhat et al. used the Spark library to apply ngrams, tf/idf, and term frequency.

Classifier	Parameters
Support Vector Machine	SVM Type: C-CSV Kernel Type: Linear
Decision Tree	maxDepth: 20 impurity: entropy minWeightFractionPerNode: 0.25 minInfoGain: 1
Logistic Regression	elasticNetParam: 0.8 regParam: 0.001 maxIter: 10
One-vs-Rest	Nested Classifier: Logistic Regression
Naive Bayes	smoothing: 1

Table 4: Parameters used for the classifiers while replicating the work of Bhat et al.

With term frequency, we create a vocabulary of words from all issues in the dataset. Every word in this vocabulary has a unique index in the vector. Thus, each issue can be converted to a vector (with the size of the vocabulary) that reflect words in this issue and their frequency, by counting the number of occurrences of every word in the issue, and set that count as the corresponding value in the vector.

TF/IDF is similar to term frequency, but attempts to account for words which occur more rarely across issues [8]. To compute the TF/IDF value, we multiply the normalized frequency (TF) of a word (given by the term frequency divided by the length of the issue in words), with a measure of how common or rare a word appears among all issues (IDF). We calculated IDF using equation 1 [8].

$$IDF(word) = \log \frac{\#\text{ issues}}{\#\text{ issues containing word}} \quad (1)$$

3. *Implement classifiers*: We experimented with all machine learning classifiers used by Bhat et al. and using the same technologies and parameters (see Table 4). We used the PySpark[6] library to implement the decision tree, logistic regression, one-vs-rest, and naive Bayes classifiers, and we used LibSVM[7] to implement the Support Vector Machine (SVM) classifier. For logistic regression, we used sklearn's polynomial kernel to implement a dot kernel similar to Bhat et al..

[3] https://spark.apache.org/docs/3.1.1/api/python/reference/api/pyspark.ml.feature.NGram.html
[4] https://www.nltk.org/
[5] https://scikit-learn.org/stable/
[6] https://spark.apache.org/docs/latest/api/python/
[7] https://github.com/ocampor/libsvm

7 Develop deep learning approaches

In this section, we explain our proposed deep learning approaches to identify and classify architectural issues. Our proposed deep learning approaches follow the design depicted in Figure 5, which consists of 4 major steps. We explain each of these steps in the following paragraphs.

7.1 Pre-Processing

Jira issues involve some textual contents that can disturb classifiers to identify and classify architectural issues. Examples of these textual contents are source code blocks and formatting tags. Thus, in this step, we identify and process these noisy textual contents.

In details, we performed the following cleanup steps:

- We removed formatting tags from the issue summary and description according to Jira text formatting notation[8].
- We identified tracebacks and logging output in issue descriptions, possibly *not* located within formatting tags, and replaced them with special markers. In order to detect tracebacks and logging output, we used regular expressions to test whether a sentence in an issue description contains a traceback or a logging output. We developed these regular expressions after inspecting a sample of issues from our dataset that contain tracebacks and logging outputs. We did this by word searching for the phrases "DEBUG", "INFO", "WARNING", "WARN", "ERROR", "CRITICAL", "SEVERE", "Error", and "Exception" (case sensitive). Table 5 gives an overview of the regular expressions we used, along side some the example output they were designed to capture.
- We removed dates from the text, and replaced them with them with a marker. The numbers in the dates we checked for could be separated either by dots or slashes. We removed dates with the year in the front and at the end.
- We removed IP addresses from the text, and replaced them with markers. We also checked for IP addresses where part of the address was obscured by "xx", such as the address `xx.xx.xx.142:51010`.
- We removed web links, as well as links to attachments, and user profiles. Links to other issues and repositories were replaced by special marker words. Remaining links were replaced with generic web link markers. We also searched for links without formatting by looking for words separated by dots, ending with a top level domain name (e.g. .com, .org, and .edu).

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

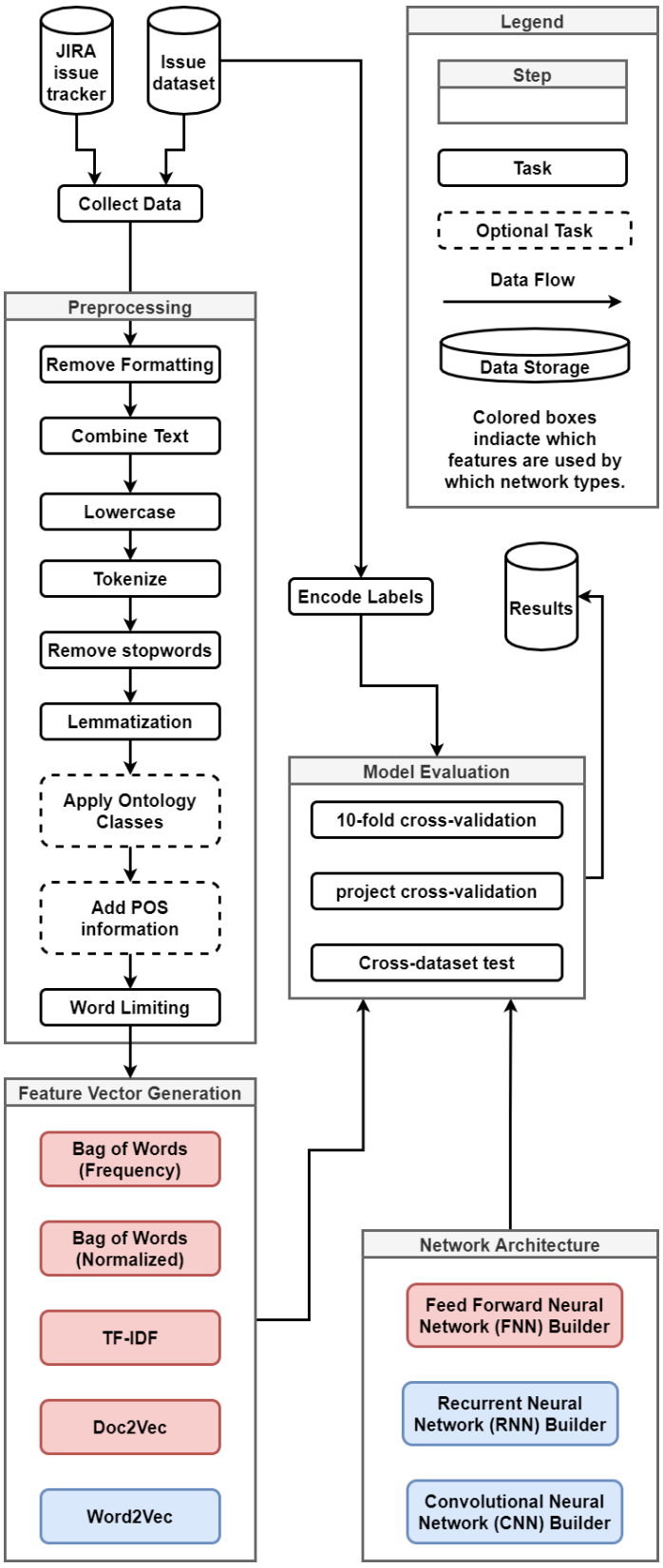

Figure 5: Steps of the deep learning pipeline we used.

- We removed the contents of *code* blocks, and replaced them with placeholders to allow classifiers to use this information to identify architectural issues. Moreover, we replaced *noformat* blocks that contain program outputs, such as tracebacks or the result of SQL queries with marker words. We used the regular expressions for detecting tracebacks and logging output mentioned before to check the content of the code and noformat blocks. We replaced the blocks with special markers when we encountered logging output or exceptions. If not, the blocks were replaced with generic code block and noformat block markers. We also experimented with keeping the content of the blocks (but replacing class and package names with markers), but this did not result in any major difference in classification performance.

- We removed version numbers (e.g. 2.7.0, 3.6.x) from the text and replaced these with markers.

- We removed file size indicators (e.g. 1.2MB, 1.5GB), float literals (e.g. 100.0f), and Amazon instance type names (e.g. c4.large) from the text.

- We removed source code within textual content, these come in between brackets {{ }} (e.g. {{`System.currentTimeMillis() + ttl(20 years) > Integer.MAX_VALUE`}}). However, we replaced the names of methods, classes, and components with special markers, using the regular expression `[a-zA-Z_\.:#]+(\(.*\))?`, because developers use class names to refer to architectural components [12]. This regular expression was designed to account for classes and method names containing `.`, `#` (e.g. `SnapshotManager#createSnapshot`), and `:` (e.g. `ColumnFamilyStore::getSSTables`), and possibly followed by brackets. Such names were replaced with marker words indicating the presence of a class, method, or package. We tested for class names by testing for UpperCamelCase. We tested for methods/fields by testing for lowerCamelCase. We tested for packages by checking whether the string we found was a substring of any known Hadoop, Cassandra, or Java standard library package. If no checks succeeded, we classified the string as method/field.

  We also searched for class or component names not located within inline formatting tags. We search for such class names in two different ways. First, we searched for words combined together with dots. These could be potential class names, prefixed by their location or package. We excluded a number of different construct from the result of this search. We tried to detect file names by looking whether the part after the last dot was a file extension. We also included a list of common abbreviations to ignore. We also removed version numbers. The previous removal of version numbers, file size indicators, float literals, Amazon instance type names and missed web links also helped in eliminating false matches. The second way we searched was by looking for words spelled in lowerCamelCase or UpperCamelCase. The results of this second search were manually examined to come up with a list of technologies and abbreviations to exclude. Next, we classified the remaining search results as either a class, method/field, or package names, as described earlier.

- We removed formatting tags (e.g. {`panel`}, {`nopanel`}, `hn.`, `bq.`, `*strong*`), but kept the textual content in between formatting tags to allow classifiers to use their textual contents for classifications. Moreover, we removed images. We also did not remove `-delete-` formatting, because we found that removing this also removed other information between `-` symbols.

- We removed list markup. Lists are lines starting with `-`, `#`, or `*`. Additionally, we removed manually formed numbered lists which use the `1)` or `1.` format. We removed the list syntax, but kept the actual text.

- We removed file paths without any additional formatting, and replaced them with markers. Here, file paths were detected by looking for words combined using `/` symbols, optionally starting with one. Optionally, the first symbol can be a dot, to account for relative paths. We discarded items where the part after the last slash was a number, because this was most likely not an actual file path. An example of this is `/10`, which is most likely not a file but does follow the format.

  We also tried to detect and extract class and method names in the text without a package name. We did this by testing for words written in UpperCamelCase and lowerCamelCase. The results of this search were inspected manually, and we created a list of technologies and abbreviations which should not be classified as class names. The remaining UpperCamelCase names were replaced with a class marker, and the lowerCamelCase names were replaced with a method/field marker.

- We removed numbers and punctuation from the sentences. The main purpose of this was that we found that this resulted in better results when using NLTK's `tokenize_sent` function to split the text into sentences.

Second, we processed the textual content of issue descriptions and summaries after cleanup. We concatenated the textual content of the summary and description similar to Bhat et al. ([1]), and converted all words to lowercase. Then, we removed stop words using the list of English stop words from NLTK. Next, we used lemmatization in combination with part of speech analysis to eliminate inflected form of words. Finally, we limited the number of words from an issue summary and description to 400 words, because most issues do not exceed 400 words in their issue description (see figure 6). Limiting the number of words is required, because few issues have large descriptions that would enforce large

Regex:
`\s*at [$#]?\w+([\.$#]\w+)*\(.*\)\s*`
Example Target Text:
`at org.apache.hadoop.ipc.Client.call(Client.java:1337)`

Regex:
`\s*Caused by: \w+(\.\w+)*: .*\s*`
`Caused by: java.nio.channels.ClosedByInterruptException: null`

Regex:
`\s*(\w+\.)*\w+(Error|Exception)(: (\w+\.)*\w+(Error|Exception))*\s*`
Example Target Text:
`org.apache.cassandra.io.FSReadError: java.nio.channels.ClosedByInterruptException`

Regex:
`\s*\d\d/\d\d/\d\d \d\d:\d\d:\d\d (DEBUG|INFO|WARNING|WARN|ERROR|CRITICAL|SEVERE) .*\s`
Example Target Text:
`07/05/20 20:45:38 INFO mapred.JobClient: Running job: job_0029`

Regex:
`\s*\[(DEBUG|INFO|WARNING|WARN|ERROR|CRITICAL|SEVERE)\] .*\s*`
Example Target Text:
`[INFO] +- org.apache.cassandra:cassandra-all:jar:4.0-alpha3:provided`

Regex:
`\s*\d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d,\d\d\d (DEBUG|INFO|WARNING|WARN|ERROR|CRITICAL|SEVERE) .*\s*`
Example Target Text:
`2015-09-13 15:59:57,884 ERROR org.mortbay.log: Error for /query_exec`

Regex:
`\s*(DEBUG|INFO|WARNING|WARN|ERROR|CRITICAL|SEVERE) .*? \d\d\d\d-\d\d-\d\d \d\d:\d\d:\d\d,\d\d\d .*\s*`
Example Target Text:
`ERROR [main] 2016-07-20 15:34:50,176 CassandraDaemon.java:698 - Exception encountered during startup`

Regex:
`\s*[A-Z][a-z]{,3} \d\d?, \d\d\d\d \d\d?:\d\d?:\d\d? (AM|PM) .*?\s*`
Example Target Text:
`Apr 3, 2013 7:46:12 AM com.google.inject.servlet.GuiceFilter setPipeline`

Regex:
`\s*(DEBUG|INFO|WARNING|WARN|ERROR|CRITICAL|SEVERE): .*\s*`
Example Target Text:
`INFO: Initiating Jersey application, version 'Jersey: 1.8 06/24/2011 12:17 PM'`

Regex:
`\s*(DEBUG|INFO|WARNING|ERROR|CRITICAL|SEVERE) - .*\s*`
Example Target Text:
`ERROR - Error in ThreadPoolExecutor`

Table 5: Overview of the regular expressions used to capture logging output and exceptions in the text of issues.

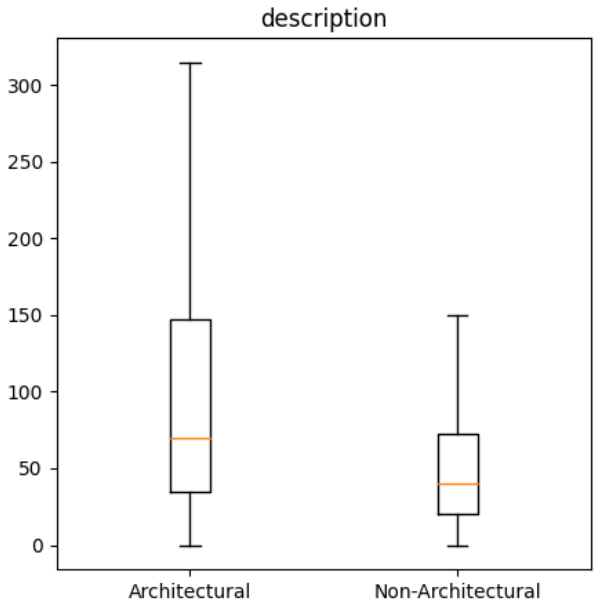

Figure 6: The number of words in the description of the issues.

feature vectors with negative impact on the classifiers and less benefit on the accuracy of classification.

7.1.1 Ontology Classes

As an additional possible step before generating feature vectors, we included a possibility to simplify the text using ontology classes. We replaced specific words in the text with the name of their respective ontology classes. We hoped that this would possibly lead to patterns which help to improve the accuracy of the deep learning models. We started out with the simple ontology classes and lexical triggers from [13]. We then selected potential candidate words from the issues in our dataset to add to these classes. We only looked for candidate words in the issue title and description. We looked for these words in two different ways. First of all, we counted how often all words occurred across all issues, and we included all words for 50 or more occurrences in our list of potential candidates. We also used a modified version of TF/IDF. We took the word counts from the previous step, and we multiplied this with the IDF of every word. We included all words with a score of 200 of greater in our list of candidates. The two authors determined independently which words should be added to the existing classes. For cases where agreement could not be reached, the primary supervisor made the final decision. Lists of the ontology classes and lexical triggers, as well as some example entries, are given in tables 6 and 7.

7.1.2 Part-of-Speech Tagging

One optional pre-processing step we added is part-of-speech (POS) tagging. When doing this, we obtain the parts of speech for the words in the text using NLTK (at the moment that stopwords are still included). Then (after all other pre-processing steps), we concatenate the words with their POS. For instance, if "support" has POS tag "verb", then this becomes "support_verb". After pre-processing all the words in this manner, we feed these concatenated words into the feature generators.

7.2 Generating Feature Vectors

In order to train deep learning models, we need to transform text in issues into representative numerical values. We experimented with a number of approaches to generate numerical feature vectors from the lists of words in issues. We will describe these approaches below.

- *Bag of words (frequency)*: This is the same as term frequency as mentioned in section 6.
- *Bag of words (normalized)*: To consider the different sizes of the issues, we divide the frequency of a word by the total number of words in an issue. Thus, the occurrences of words have less values in large issues compared to small issues.
- *Bag of words using term frequency inverse document frequency (TF-IDF)*: This is the same as TF/IDF as mentioned in section 6.
- *Ontology Features*: In stead of counting the occurrences of all separate words, we also experimented with counting the amount of words from any given ontology class (see section 7.1.1 - Ontology Classes), and using this as a feature. For instance, if there are 3 words from the Technology Decision ontology class, the corresponding field in feature vector will be 3. In addition to the ontology classes and lexical triggers, we also included fields for the amount of occurrences of marker words we placed while removing the formatting from issues.
- *Word2Vec*: Bag of words techniques consider each word separately, independent of its relationship to other words in the same issue. Word2Vec represents words as vectors, such that words that are similar also have similar vectors [7]. We experimented with pre-trained vectors from Stack Overflow[9], which provide vectors for words that appear in Stack Overflow posts. Moreover, we created our own vectors from issue summaries and descriptions. To create these vectors, we performed word embedding training on all issues from vari-

[9] https://github.com/vefstathiou/SO_word2vec

Class Name	Example Words
Component Element Names	classes, endpoints, event, field, fields, files, interface, job, jobs, procedure
Component Names	beans, consumer, consumers, context, host, machines, producers, queue, records, thread
Connector Data Names	calls, information, lists, map, message, messages, results, socket, structures, updates
Connector Names	calling, checking, communication, connect, connectivity, consume, implements, linking, read, sharing
Pattern Names	REST, RESTful, blocking, filter, interceptor, remote-procedure, repository, routing, shadow, structured
Quality Attribute Names	debuggable, dependable, effectiveness, flexible, integrate, mobility, operability, reliable, securability, usable
Technology Names	Java, Java JMS, ProtoRPC, codeigniter-2, grails, ironpython, sencha-touch-2, ssrs-2008, stanford-nlp, xamarin.android

Table 6: The ontology classes, with up to ten example words per class.

ous projects[10] using the Word2Vec embedding from the Gensim library[11].

- *Doc2Vec*: The Doc2Vec model training is similar to the Word2Vec training [6]. Both approaches train word vectors based on the context of the words. The main difference is that with the Doc2Vec training, an additional document vector is added for the training. This document vector is essentially a memory vector for the context of the document. Therefore at the end of the training, this document vector is a vector that represents the context of the document and is called the Doc2Vec vector. For a visual representation of the difference between Word2Vec and Doc2Vec training, refer to the paper [6]. This paper also contains the details about the Doc2Vec training. This embedding also needs to be trained on the various projects[10] using the Doc2Vec embedding from the Gensim library[6][12].

- *Properties of issues*: In addition to looking at the text of issues, we also looked at other properties of issues. We explain the properties we experimented with in the following section. That section also contains the issue property model optimization, i.e. what issue properties yield the best performance.

Determining useful issue properties

For the issue properties deep learning model, we extracted many issue properties. These properties can be found in table 8. In order to test how effective each property is for the detection or classification of issues, we tested each property individually first. We did this for two models initially: one model without hidden layers and one model with a hidden layer of size 16.

The results for the detection task can be found in tables 9 and 10. For both models we see impressive performance for the issuetype property. We also see that most of the properties do contain information useful for detecting architectural issues. These properties have a f-score above the random guessing threshold of 0.5757. Furthermore, for most properties we see a performance improvement when using a hidden layer. Therefore we expect more improvement when doing a hyper-parameter optimization for the hidden layer of this model.

Tables 11 and 12 contain the results for the classification task. We see bad performance for all properties. Only very few properties perform above the random guessing f-score value of 0.2461. Issue properties therefore do not seem to be suitable for classifying issues. However, we did experiment with combinations of issue properties to find out if this can improve the performance.

Determining useful combinations of issue properties

We also tested which combinations of issue properties yield good performance. Since an exhaustive search for the optimal combination of issue properties is infeasible (due to the many combinations), we applied another approach. This approach starts with the best performing individual issue property. Then each iteration, we will add the next best issue property to the set and test the performance. We do this until all issue properties are in the set. Again we did this for two models. One without a hidden layer and one with a hidden layer of size 16.

The tables 13 and 14 show the issue detection performance of the combinations for both models. We start with the best performing issue property from the previous section

[10]The word2vec and doc2vec embeddings were trained on 16054 Hadoop issues, 17597 Cassandra issues, 2183 Tajo issues, 15821 HDFS issues, 7070 MapReduce issues, and 10912 Yarn issues. The embeddings for the tests with the Bhat dataset also included 39293 Spark issues. These issues were all issues available for the mentioned projects on the 4th of July 2022.

[11]`https://radimrehurek.com/gensim/models/word2vec.html`

[12]`https://radimrehurek.com/gensim/models/doc2vec.html`

Class Name	Example Words
Alternative	alternative, choice, direction, option, solution, tendency, trend, way
Appropriate	adequate, appropriate, convenient, convenient, desirable, favorable, proper, suitable, suited
Better	better, defeat, exceed, outdo, outgo, outmatch, outperform, outstrip, surmount, surpass
BigNumber	all, hundreds, hundreds, lots, many, much, multiple, piles, thousands, tons
Choose	choose, favor, go, opt, pick, select, take, use, utilise, utilize
Complex	complex, complicated
Constraint	constraint, limitation, restraint, restriction
Depend	build, count on, depend, implement, rely
Difference	compare, difference, dissimilarity, distinction, distinctness
Evaluate	appraise, assess, evaluate
Fast	efficient, fast, quick, rapid, robust
Forced	forced, have to
Good	brilliant, cool, fine, good, magnificent, nice, superb, top, well
Hard	backbreaking, difficult, effort, hard, laborious, much work, overhead, punishing, tough, unmanageable
Important	critical, crucial, eminent, high, important, significant
Integrate	complement, fit, hook, incorporate, integrate, play, run, work
Learn	acquire, learn, study, teach
Need	ask, demand, desire, hope, like, need, plan, require, want, wish
Problem	hurdle, issue, obstacle, problem, snag, trouble
Programming Activities	acquire, compress, deploy, enable, expand, fix, hardcode, open, set, switch
ProsCons	benefit, cons, deficiency, disadvantage, favor, favour, flaw, fragility, prononess, strength
Recommend	advice, advise, commend, encourage, favor, motivate, proceed, propose, recommend, suggest
Recommendation	guidance, recommendation, suggestion
Requirement	condition, criteria, demand, essential, necessary, necessity, request, requirement, requisite, worry
Search	consider, explore, look, research, search, seek
Simple	comfortable, easy, effective, flexible, lightweight, painless, productive, simple, uncomplicated, unproblematic
Slow	heavy, slow, slowly
Stick	adhere, avert, avoid, bind, bond, evade, impel, stay, stick
Support	allow, offer, provide, supply, support
Versus	against, contrary, contrast, counter, differ, opposed, opposing, opposition, versus, vs

Table 7: The classes of lexical triggers, with up to ten example words per class.

Property	Description
n_attachments	Number of files added as attachment to the issue
n_comments	Number of comments left on the issue
len_comments	Total length of the comments, in words
n_issue links	Number of links to other issues
n_sub-tasks	Number of sub-tasks (child-issues) this issue has
n_votes	Number of votes on the issue
n_watches	Number of people "watching" this issue
len_description	Length of the issue description, in words
len_summary	Length of the issue summary, in words
components	Components of the software the issue affects
n_components	Number of components of the issue
labels	Labels added to the issue (e.g. "beginner issue")
n_labels	Number of labels added to the issue
priority	Priority of the issue (e.g. "Major")
resolution	Final resolution of the issue (e.g. "Unresolved")
status	Current status of the issue (e.g. "Open")
issuetype	Type of the issue (e.g. "Bug")
parent	Flag indicating whether this issue has a parent (i.e. this issue is a subtask)

Table 8: Potentially useful issue properties.

and each iteration we extend it with the next best performing properties, hence the naming scheme 'previous + *next best property*'. For the model without hidden layer we see a major improvement when combining issuetype with issue priority. We also still see that issuetype individually performs really good compared to the combinations. A few combinations achieve 78% f-score, which is quite good but still worse than the issuetype only and issuetype with priority.

The results for the classification task can be found in tables 15 and 16. For the combinations we do see an improvement compared to the individual attribute results. Many of them score above the random guessing threshold of 0.2461 f-score. However, issue properties still do not seem to be useful for classifying issues.

Based on both the result of the individual issue properties and the combinations of issue properties, we selected three combinations for detection and two combinations for classification for which we do the hyperparameter optimization.

For detection we selected issuetype for the hyperparameter optimization. This issue property has by far the best performance for detection on its own and it performs on par with the best combinations we tested. For detection we also included the set with issuetype and priority. This set also had similar performance. We also do a hyperparameter optimization for a larger set with good performance: (issuetype, n_issuelinks, n_attachments, priority, len_summary, n_watches, n_components, parent, len_description, status, resolution, components, labels, n_labels, n_votes).

The classification performance of the individual properties were as good or worse than random guessing performance and therefore we do not perform a hyper-parameter optimization on these. Combinations of properties were slightly better and therefore we only did hyperparameter optimizations for the best combinations of properties of each model: (n_comments, n_watches, n_issuelinks, components, n_attachments, issuetype, priority, len_description, len_summary, n_components, parent, labels, status, resolution, n_labels, n_votes, n_subtasks) and (components, n_issuelinks, issuetype, n_watches, n_comments, n_attachments, priority).

7.3 Deep learning network architectures

We experiment with three kinds of deep learning models: Feed forward neural networks (FNN), Convolutional neural networks (CNN), and Recurrent neural networks (RNN). We will first briefly elaborate on the high-level architecture of these networks, then explain how we determined the specific hyper-parameters in the next section, and finally discuss how we combined models together in an attempt to obtain even better models. We implemented the classifiers using Keras[13].

- *Feed Forward Neural Network* (FNN): Feed forward neural networks are the simplest neural networks [10].

[13] `https://keras.io/`

Issue Properties	Precision	Recall	F1-score	Imp. over Random
labels	0.7260	0.2873	0.3239	0.54x
n_votes	0.7396	0.3505	0.4094	0.68x
parent	0.6708	0.4047	0.4657	0.78x
n_subtasks	0.7168	0.5343	0.4804	0.80x
components	0.7216	0.4438	0.5380	0.90x
len_summary	0.5284	0.5647	0.5402	0.90x
n_labels	0.6660	0.6050	0.5776	0.96x
n_attachments	0.6842	0.5843	0.5832	0.97x
status	0.6637	0.5847	0.5845	0.97x
n_comments	0.5396	0.7036	0.6001	1.00x
resolution	0.6822	0.7120	0.6184	1.03x
n_components	0.6504	0.6336	0.6366	1.06x
len_comments	0.6004	0.7706	0.6513	1.08x
len_description	0.6624	0.7577	0.6663	1.11x
n_watches	0.6929	0.7405	0.6757	1.12x
n_issuelinks	0.6559	0.8266	0.7083	1.18x
priority	0.6825	0.7898	0.7263	1.21x
issuetype	0.7653	0.7586	0.7564	1.26x

Table 9: Individual issue property detection performance, for a model without hidden layer

Issue Properties	Precision	Recall	F1-score	Imp. over Random
len_comments	0.1968	0.2986	0.2372	0.39x
n_comments	0.3911	0.4441	0.3793	0.63x
n_subtasks	0.7965	0.3852	0.3863	0.64x
n_votes	0.7200	0.4489	0.4690	0.78x
n_labels	0.6734	0.4761	0.4820	0.80x
labels	0.7154	0.5363	0.4969	0.83x
components	0.7221	0.3814	0.4985	0.83x
resolution	0.7233	0.5673	0.5231	0.87x
status	0.6633	0.6315	0.5818	0.97x
len_description	0.5742	0.7544	0.6299	1.05x
parent	0.6464	0.7736	0.6722	1.12x
n_components	0.6567	0.7647	0.6957	1.16x
n_watches	0.7038	0.7824	0.7093	1.18x
len_summary	0.5912	0.8910	0.7105	1.18x
priority	0.6951	0.7414	0.7126	1.19x
n_attachments	0.6661	0.8838	0.7489	1.25x
n_issuelinks	0.6785	0.8953	0.7653	1.27x
issuetype	0.7532	0.8455	0.7947	1.32x

Table 10: Individual issue property detection performance, for a model with a hidden layer of size 16

Issue Properties	Precision	Recall	F1-score	Imp. over Random
n_subtasks	0.1287	0.2519	0.1192	0.48
n_votes	0.1329	0.2470	0.1402	0.57
n_labels	0.1205	0.2481	0.1426	0.58
len_description	0.1724	0.2486	0.1527	0.62
labels	0.2569	0.2575	0.1539	0.63
status	0.1564	0.2354	0.1615	0.66
resolution	0.2255	0.2651	0.1620	0.66
parent	0.1344	0.2624	0.1699	0.69
len_summary	0.1635	0.2576	0.1708	0.69
len_comments	0.1601	0.2734	0.1770	0.72
n_components	0.1734	0.2642	0.1889	0.77
priority	0.2157	0.2470	0.1965	0.80
n_attachments	0.1812	0.2849	0.1968	0.80
n_comments	0.1639	0.2970	0.1980	0.80
n_watches	0.2000	0.3012	0.2051	0.83
issuetype	0.2125	0.2608	0.2151	0.87
n_issuelinks	0.2198	0.2949	0.2308	0.94
components	0.2709	0.2731	0.2400	0.98

Table 11: Individual issue property classification performance, for a model without hidden layer

Issue Properties	Precision	Recall	F1-score	Imp. over Random
len_comments	0.1210	0.2608	0.1255	0.51
n_subtasks	0.1793	0.2602	0.1332	0.54
n_votes	0.1666	0.2674	0.1490	0.61
n_labels	0.1338	0.2570	0.1543	0.63
resolution	0.2497	0.2603	0.1544	0.63
status	0.1571	0.2391	0.1589	0.65
labels	0.2544	0.2562	0.1621	0.66
parent	0.1280	0.2617	0.1633	0.66
n_components	0.1374	0.2436	0.1709	0.69
len_summary	0.1425	0.2496	0.1726	0.70
len_description	0.1635	0.2583	0.1783	0.72
priority	0.2493	0.2518	0.2052	0.83
issuetype	0.2262	0.2608	0.2209	0.90
n_attachments	0.2262	0.2914	0.2281	0.93
components	0.2721	0.2634	0.2424	0.98
n_issuelinks	0.2435	0.2928	0.2469	1.00
n_watches	0.2383	0.3124	0.2531	1.03
n_comments	0.2640	0.3469	0.2677	1.09

Table 12: Individual issue property classification performance, for a model with a hidden layer of size 16

Issue Properties	Precision	Recall	F1-score	Imp. over Random
issuetype	0.7653	0.7586	0.7564	1.26x
previous + priority	0.7634	0.8315	0.7953	1.32x
previous + n_issuelinks	0.7888	0.7484	0.7675	1.28x
previous + n_watches	0.7612	0.7401	0.7444	1.24x
previous + len_description	0.7838	0.7266	0.7482	1.25x
previous + len_comments	0.8112	0.6401	0.6839	1.14x
previous + n_components	0.6697	0.6229	0.6012	1.00x
previous + resolution	0.7727	0.7247	0.7347	1.22x
previous + n_comments	0.7537	0.7726	0.7417	1.23x
previous + status	0.7590	0.7598	0.7411	1.23x
previous + n_attachments	0.6712	0.6507	0.6463	1.08x
previous + n_labels	0.7807	0.7493	0.7440	1.24x
previous + len_summary	0.7504	0.7767	0.7412	1.23x
previous + components	0.8169	0.7277	0.7626	1.27x
previous + n_subtasks	0.7990	0.7072	0.7430	1.24x
previous + parent	0.8008	0.7098	0.7457	1.24x
previous + n_votes	0.8182	0.6806	0.7429	1.24x
previous + labels	0.7997	0.7221	0.7518	1.25x

Table 13: Combinations of issue properties detection performance, for a model without hidden layer

Issue Properties	Precision	Recall	F1-score	Imp. over Random
issuetype	0.7532	0.8455	0.7947	1.32x
previous + n_issuelinks	0.7745	0.7624	0.7673	1.28x
previous + n_attachments	0.7712	0.7771	0.7723	1.29x
previous + priority	0.7651	0.8035	0.7809	1.30x
previous + len_summary	0.7606	0.7996	0.7780	1.29x
previous + n_watches	0.7675	0.7840	0.7745	1.29x
previous + n_components	0.7650	0.7994	0.7808	1.30x
previous + parent	0.7740	0.7896	0.7813	1.30x
previous + len_description	0.7850	0.7854	0.7831	1.30x
previous + status	0.7972	0.7232	0.7578	1.26x
previous + resolution	0.7957	0.7394	0.7648	1.27x
previous + components	0.7989	0.7457	0.7706	1.28x
previous + labels	0.8120	0.7470	0.7775	1.29x
previous + n_labels	0.8055	0.7562	0.7791	1.30x
previous + n_votes	0.8068	0.7581	0.7813	1.30x
previous + n_subtasks	0.8110	0.7471	0.7772	1.29x
previous + n_comments	0.8023	0.7547	0.7770	1.29x
previous + len_comments	0.8447	0.4071	0.4649	0.77x

Table 14: Combinations of issue properties detection performance, for a model with a hidden layer of size 16

Issue Properties	Precision	Recall	F1-score	Imp. over Random
n_components	0.1734	0.2642	0.1889	0.77
previous + n_issuelinks	0.2796	0.2894	0.2660	1.08
previous + issuetype	0.2998	0.2949	0.2922	1.19
previous + n_watches	0.3281	0.3246	0.3168	1.29
previous + n_comments	0.3030	0.3138	0.2956	1.20
previous + n_attachments	0.3134	0.3221	0.3057	1.24
previous + priority	0.3227	0.3267	0.3185	1.29
previous + n_components	0.3120	0.3178	0.3076	1.25
previous + len_comments	0.3257	0.3193	0.3011	1.22
previous + len_summary	0.2954	0.2949	0.2683	1.09
previous + parent	0.3152	0.3181	0.2794	1.14
previous + resolution	0.3188	0.3129	0.2918	1.19
previous + status	0.3173	0.3184	0.2913	1.18
previous + labels	0.3124	0.3036	0.2576	1.05
previous + len_description	0.2667	0.2950	0.2450	1.00
previous + n_labels	0.2535	0.2859	0.2176	0.88
previous + n_votes	0.2341	0.2724	0.2004	0.81
previous + n_subtasks	0.2946	0.3319	0.2787	1.13

Table 15: Combinations of issue properties classification performance, for a model without hidden layer

Issue Properties	Precision	Recall	F1-score	Imp. over Random
n_comments	0.2640	0.3469	0.2677	1.09
previous + n_watches	0.2403	0.3319	0.2583	1.05
previous + n_issuelinks	0.3344	0.3475	0.3032	1.23
previous + components	0.3056	0.3286	0.3044	1.24
previous + n_attachments	0.3128	0.3163	0.3056	1.24
previous + issuetype	0.3281	0.3363	0.3235	1.31
previous + priority	0.3169	0.3144	0.3083	1.25
previous + len_description	0.3065	0.3136	0.3023	1.23
previous + len_summary	0.3274	0.3324	0.3232	1.31
previous + n_components	0.3100	0.3205	0.3080	1.25
previous + parent	0.3227	0.3283	0.3199	1.30
previous + labels	0.3335	0.3392	0.3306	1.34
previous + status	0.3286	0.3369	0.3265	1.33
previous + resolution	0.3381	0.3420	0.3352	1.36
previous + n_labels	0.3411	0.3417	0.3347	1.36
previous + n_votes	0.3395	0.3367	0.3315	1.35
previous + n_subtasks	0.3437	0.3432	0.3378	1.37
previous + len_comments	0.1975	0.2710	0.1783	0.72

Table 16: Combinations of issue properties classification performance, for a model with a hidden layer of size 16

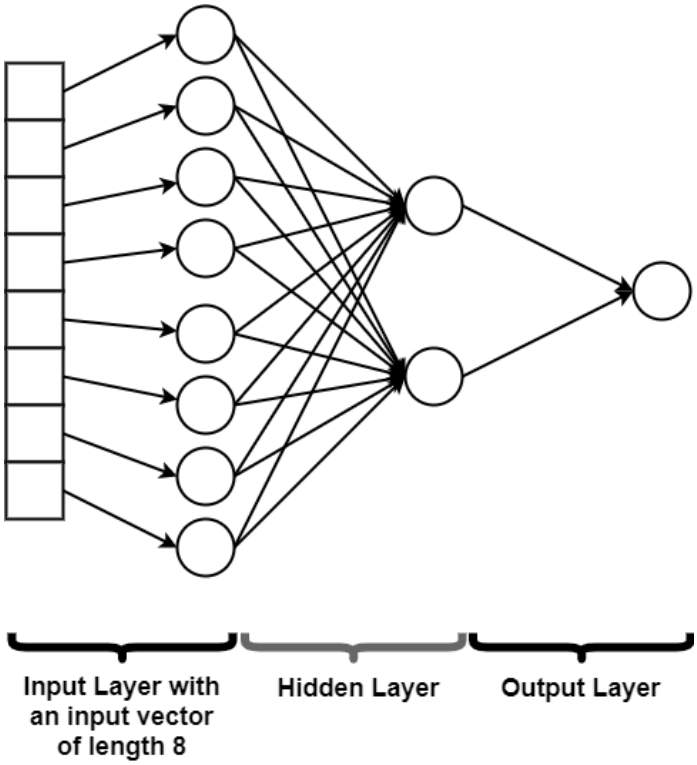

Figure 7: Example of a feed forward neural network architecture. This is not the final architecture we used. The hyper-parameters in this drawing are small for simplicity. Our actual architectures can be found in table 17

They consist of a number of layers, where all neurons in one layer are connected to all neurons in the next layer. The connections between neurons hold weights that need to be adjusted through training to improve the performance for the classification tasks. The first layer in the FNN accepts the input and the final layer produces the outputs [10]. An example of a network with such an architecture is given in figure 7. We will further explain the exact architectures we used in table 17. For FNNs, we will list what hidden layers we used.

- *Convolutional Neural Network* (CNN): FNN do not consider the relationships between words, because each input word is specified separately. CNNs try to resolve this problem by performing convolution operations on closely related words to create different perspectives on the relationships between words. To achieve this, CNNs use two different types of layers: convolutions and pooling layers. Convolution layers transform input words into other dimensions with a focus on a specific number of related words, while pooling layers condense the input into smaller vectors to produce a specific result.

  The architecture for our CNNs consists of a number of convolutions operating in parallel, all followed by a max pooling layer. Next, the outputs of all these pooling layers are combined in a concatenation layer, followed by a flattening layer. The concatenation layer combines the outputs of all incoming layers. The output of every incoming layer is a tensor, and the concatenation layer combines these by concatenating them together. At this point, we have tensors of shape $1 \times 1 \times n$. The flattening layer gets rid of the additional dimensions, resulting in a vector of length n. The output of the flattening layer feeds into the output layer. This type of architecture is commonly used for text classification and has been successfully used for other purposes, such as identifying Self-Admitted Technical Debt (SATD) [9, 7].

  In table 17, we explain the exact hyper-parameters we used for our CNN models. We will give the number of filters, the amount of parallel convolutions, and the sizes of the kernels. We will also give the size of the vectors in the Word2Vec embedding we used.

- *Recurrent Neural Network*: RNNs can be distinguished by the fact that they have a form of memory [7]. In FNNs, the input remains isolated. However, in RNNs, previous inputs can be taken into account for the current input. RNNs are therefore able to cope with sequential data, such as time series and also natural text. Hence, we experimented with RNN on our dataset, since classifying our data is a form of natural language processing (NLP). An example network is given in figure 9. In this network, the RNN part is the bidirectional layer consisting of so-called LSTM units. Optionally, the network can have a hidden (dense) layer after the bidirectional layer.

  In table 17, we explain the exact hyper-parameters we used for our RNN models. We will give the number of LSTM units in the bidirectional layer, and the layout of the hidden layers after the bidirectional layer. We will also give the size of the vectors in the Word2Vec embedding we used.

We tested different combinations of neural networks and input encodings. Table 17 lists all combinations we investigated, including a mnemonic used to reference the models in the remaining part of the paper.

All the networks use the same structure for the outputs. For detection, we used a simple Boolean as the label. This means that the output of the network is also given by a layer with a single neuron. We use a sigmoid activation function for this neuron, because we want to have the values 0 and 1 as output.

For classification, we used a one-hot encoding. This means that the label is represented as a tuple of length four, with one element set to 1. This element determines the class an issue belongs to. For example, a 1 in the last position means that an issue is non-architectural, while a 1 in the first position means that it contains an executive ADD. The neural network uses four output neurons. Of these neurons, one must be 1 and the others must be 0. This can be achieved using an argmax function. However, because neural net-

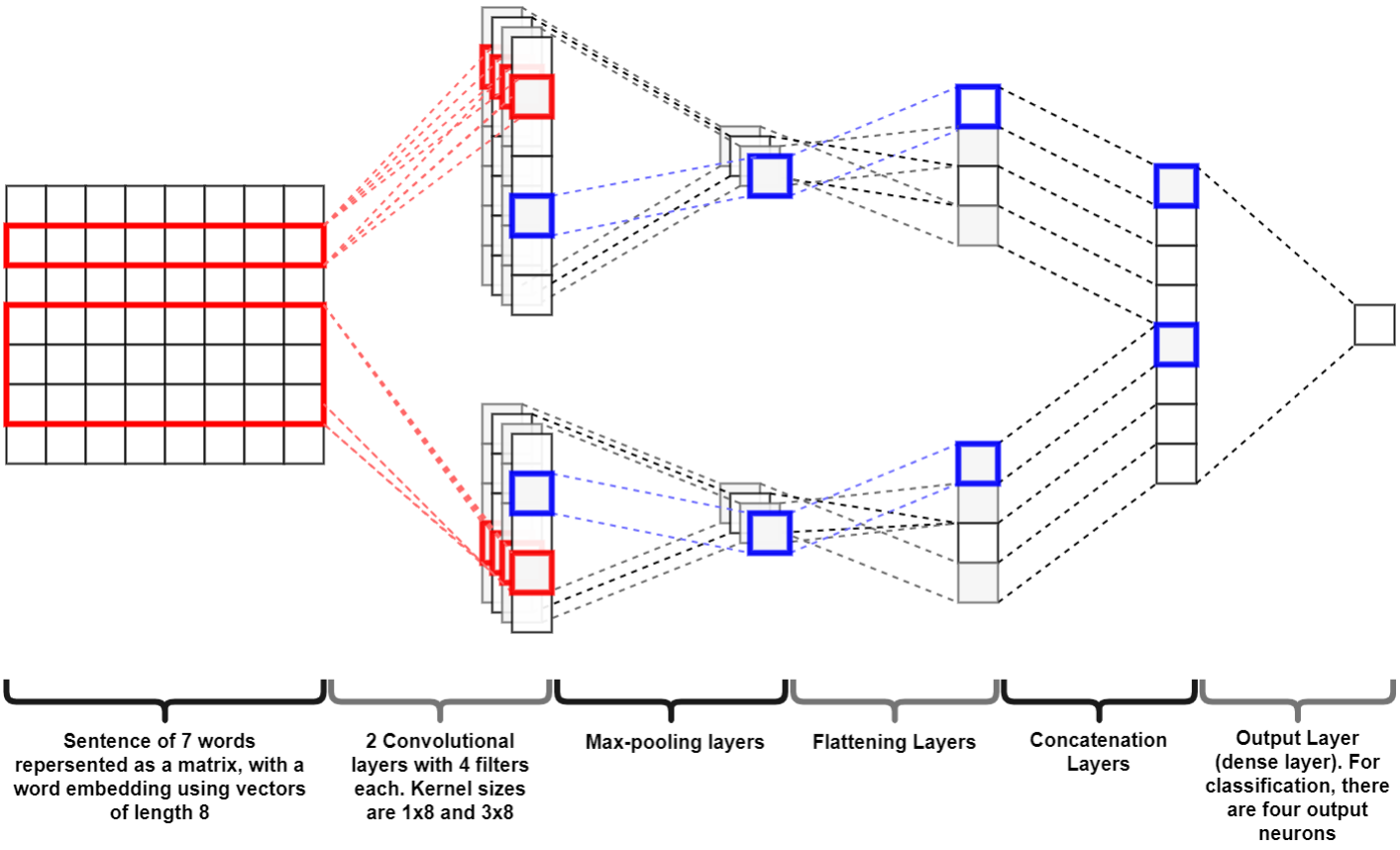

Figure 8: Example of a convolutional neural network architecture with two convolutions. This is not the final architecture we used. The hyper-parameters in this drawing are small for simplicity. Our CNN architecture can be found in table 17.

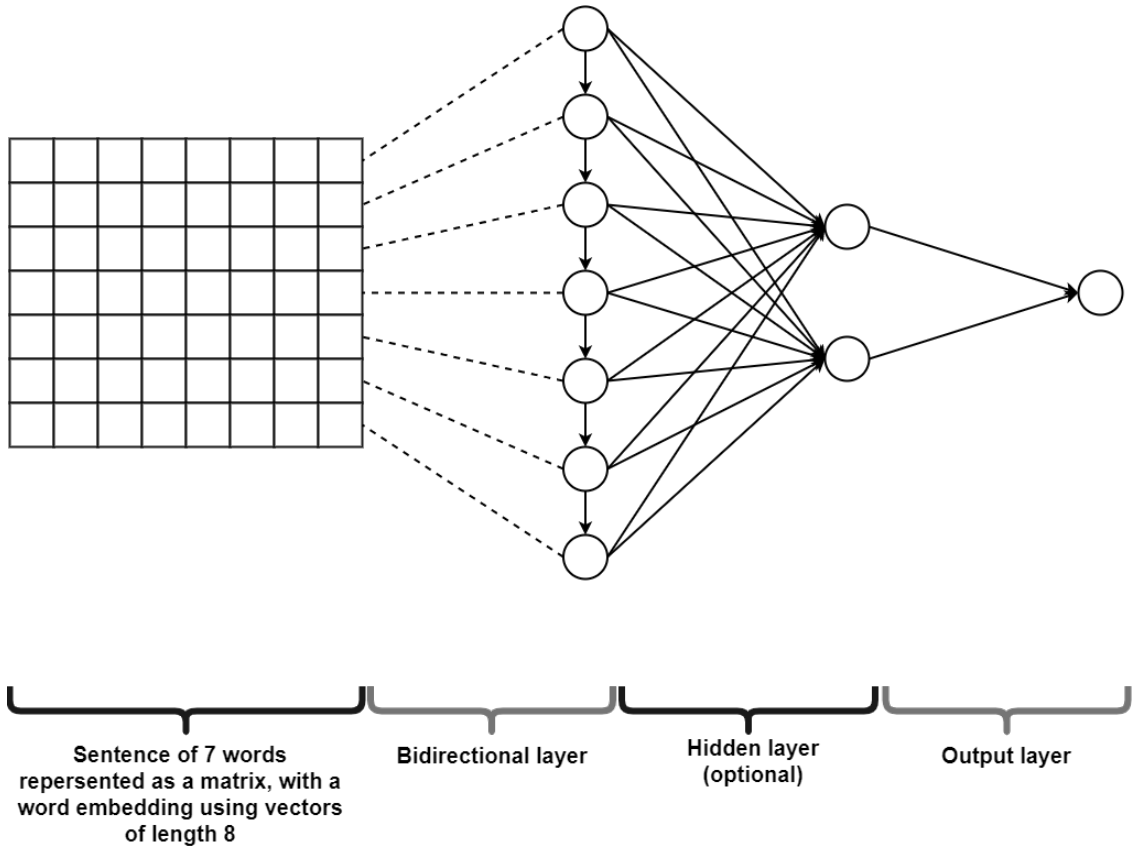

Figure 9: Example of a recurrent neural network architecture. This is not the final architecture we used. The hyper-parameters in this drawing are small for simplicity. Our RNN architecture can be found in table 17

Model Name	**Input Encoding**	**Model Type**	**Architecture Detection**	**Architecture Classification**
BOWFreq	Bag of words (frequency)	FNN	One hidden layer of size 2	Two hidden layers. The first one of size 32, the second one of size 16.
BOWNorm	Bag of words (normalized)	FNN	Two hidden layers, both of size 32.	Two hidden layers. The first one of size 32, the second one of size 16.
TFIDF	TF-IDF	FNN	Two hidden layers. The first one of size 64, the second one of size 2.	Two hidden layers. The first one of size 256, the second one of size 128.
Doc2vec	Doc2Vec	FNN	One hidden layer of size 64; Input vectors of length 25	One hidden layer of size 256; Input vectors of length 100
CNN	Word2Vec	CNN	One convolution with kernel size 75 and 32 filters; Input vectors of length 25	One convolution with kernel size 50 and 64 filters; Input vectors of length 10
RNN	Word2Vec	RNN	Bidirectional layer size of 128, followed by dense layer of size 4; Input vectors of length 25	Bidirectional layer size of 128, no dense layer; Input vectors of length 300
Issue Properties	Issue Properties	FNN	Two hidden layers. The first one of size 16, the second one of size 4.	Two hidden layers. The first one of size 16, the second one of size 8.
Ontology-Features	Ontology Features	FNN	Two hidden layers. The first one of size 128, the second one of size 16.	Two hidden layers. The first one of size 64, the second one of size 32.

Table 17: Combinations of input encodings and model architectures we tested.

works require differentiable activation functions, we use the softmax activation function.

7.3.1 Optimizing Classifiers

One important step in creating the classifiers, is making sure that their architectures are optimized. This means that we must optimize the so-called hyper-parameters which dictate the shape of the network. The way in which we optimize these hyper-parameters, is by training the neural networks with different hyper-parameters, evaluating their accuracy, and picking the hyper-parameters resulting in the best performance.

For FNNs, we optimized the amount of hidden layers and the size of those layers. We varied between one or two hidden layers, and with layer sizes 2, 4, 8, 16, 32, 64, 128, 256, 512. We inspected those results and performed smaller steps if the results indicated that the larger layers performed better. We did an exhaustive search of the entire hyper-parameter space spanned by these values.

For the CNNs, we did not perform an exhaustive search because of the large amount of possible combinations. Instead, we first optimized the size of the kernel by testing a number of models with a single convolution for a number of kernel sizes, ranging from 1 to 10 and a few larger kernels with sizes 25, 50, 75, and 100. Once we found a size that seemed to give good results, we performed an exhaustive search with combinations of convolutions with kernels close to the size we found. After optimizing the kernel size, we optimized the number of filters and the size of the fully connected layer. We tried 2, 4, 8, 16, 32, 64, 128, 256, and 512 for both, with a more find-grained search performed when appropriate. For this part, we did perform an exhaustive search over all combinations.

RNN models make use of so-called bidirectional layers. We tried the sizes 1, 2, 4, 8, 16, 32, 64, and 128. Then we selected the best performing layer size and started experimenting with a follow-up dense layer. For this we tried the sizes 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512. In the end we selected the combination which gave the best results.

For the models that use either the word2vec or doc2vec embeddings (see table 17), we also optimized the vector length. We trained embeddings with sizes 5, 10, 25, 50, 100, 200, and 300. The idea behind this is that we have a much smaller vocabulary than state of the art embeddings such as Google News[14]. Therefore it can be sufficient to have a smaller vector size and thus reduce training time, while still keeping good performance.

The complete results of this hyper-parameter optimization are not presented in this work. In stead, we will only discuss the final resulting architectures. The results of the hyper-parameter optimization can be found in section 10. The final architectures can be found in table 17.

[14] `https://code.google.com/archive/p/word2vec/`

7.3.2 Combining Classifiers

In order to further improve the accuracy of our classifiers, we tried to combine them. We tried various combinations. Specifically, we looked at combining all the best performing text models, combining all best performing text models with issue properties and the ontology features, and we combined every best performing text model with issue properties and ontology features separately. We tested every combination in three different ways: using model concatenation, using voting, and using stacking.

When concatenating models, we take the basic deep learning models we want to combine and combine all their outputs together in a *concatenation layer*, which effectively combines all outputs together into a single vector [7]. This concatenation layer then feeds into one or more hidden layers, which feed into the final output layer. When doing concatenation, all models are thus combined into one single network [7].

This is different for voting and stacking. For both methods, the base classifiers are trained independently. For stacking, the outputs of the base classifiers are then combined into vectors. These vectors are used as the input for a new neural network (a FNN). In stacking, the classifiers are thus trained separately, and the final classification is then made using their outputs by a final *meta classifiers* [11].

When doing voting, we also train the base classifiers separately and take their outputs. However, we do not combine them using a new classifier. In stead, we classify based on the labels with the most *votes* from the base classifiers [11]. Ties are resolved by looking at the highest sum of the predicted probabilities (un-rounded outputs).

Examples of all these ways of combining classifiers are given in figure 10.

8 Training and evaluating classifiers

For all classifiers we implemented, both the Bhat and deep learning classifiers, we used 10-fold cross validation. We split the issues into 10 equally sized subsets. For every subset, we conduct a stratified sampling, where we ensure that the proportions of issues from any project with any label are the same as in the full dataset. For example, the proportion of executive issues from the Hadoop project in any fold is equal to the proportion of executive Hadoop issues in the full dataset. For the Bhat classifiers, we used 9 folds for training and 1 for testing. For deep learning, , where we used 8 folds for training, 1 for validation, and 1 for testing.

In order to test RQ2, we implemented a special variant of 10-fold cross-validation. We test RQ2 by training on one dataset (e.g. our dataset) and testing with the other (e.g. the dataset of Bhat et al.). We split the training set once again into 10 folds. 9 folds are used for training, and 1 for validation. The test set remains the same.

Additionally, we also implemented project cross-validation to answer RQ3. When doing this, one of the

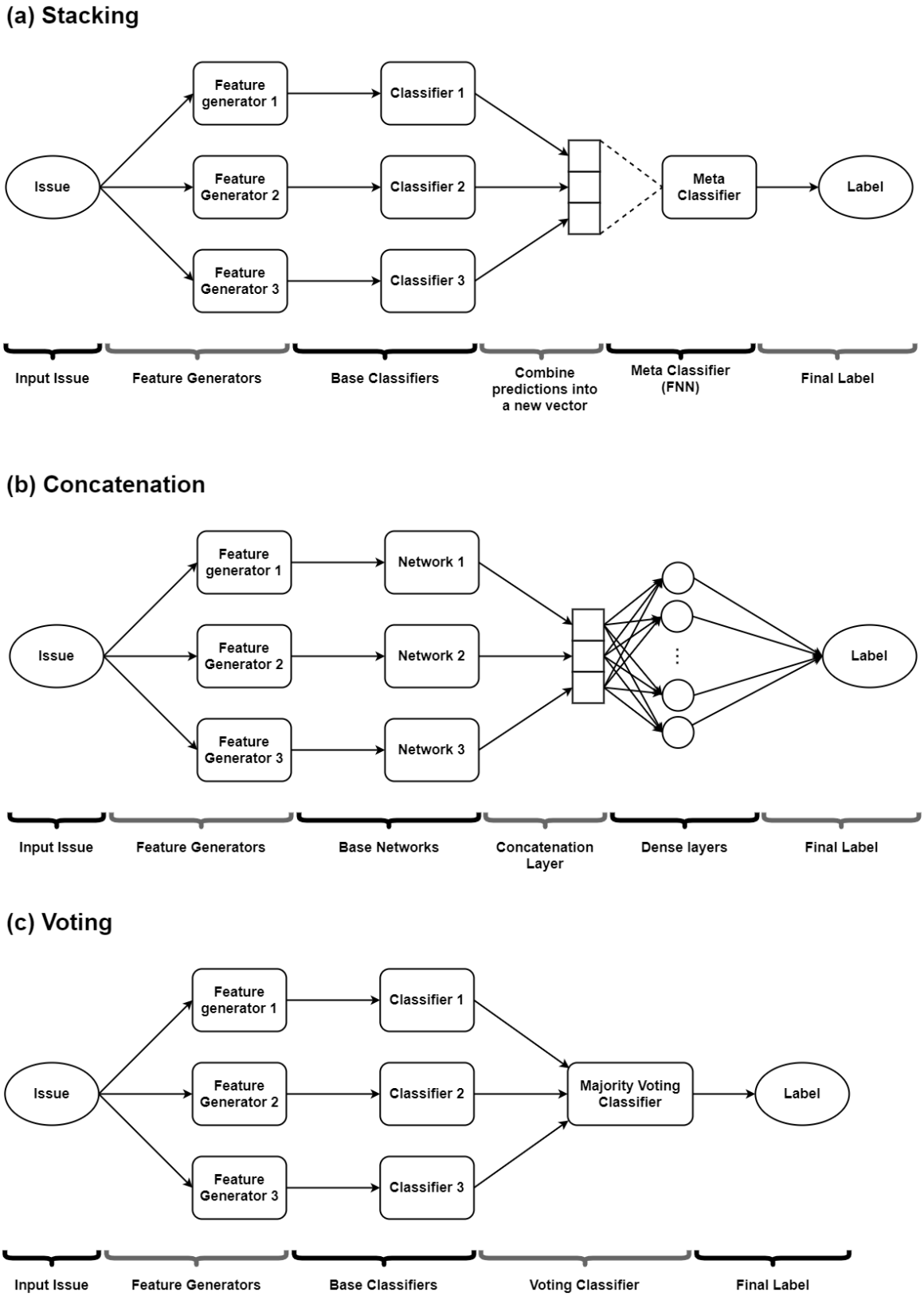

Figure 10: Schematic representation of (a) a stacking classifier, (b) a classifier obtained by concatenating models, (c) a majority voting classifier.

projects (e.g. HADOOP) is taken from the dataset and used as the test set. The other projects are used to construct the training and validation sets. For all evaluation methods mentioned here, we computed the average accuracy over all training/testing sessions.

8.1 Deep Learning

In order to avoid class-imbalance, we used class weights and class limits for deep learning. Specifically, we used class weights for the detection task. Class weights tell the deep learning algorithm to attach more or less value to samples from some given class. We computed class weights according to equation 2. Here, W_L is the weight of class L, S_L the amount of samples with label L, S the total amount of samples, and C the amount of classes. The S/C term is there to normalize the loss to the same magnitude as if no class weights were used.

$$W_L = \frac{1}{S_L}\frac{S}{C} \tag{2}$$

For the classification task, we used class limits. Class limits specify a limit to the amount of samples from any given class. We used class limits because we found that even with class-weights, we obtained models which (almost) exclusively outputted the existence label, because there are many more existence issues than any other type. We use the class limit 237.

The goal of the deep learning algorithm is to minimize the so-called loss function. This is done by updating the weights and other trainable parameters in the network. How these parameters are updated is managed by the optimizer. For the CNNs, we used the SGD (stochastic gradient descent) optimizer with a momentum parameter of 0.25, and we used Hinge loss function. For other models, we used the Adam optimizer in combination with cross-entropy. We have decided on these optimizers and loss functions because they result in the most stable (non-fluctuating) results during training compared to other optimizers and loss functions. We let the models train for a maximum of 1000 epochs, with a possibility to stop earlier if the validation f-score did not improve any more for 20 epochs in a row. Hence, the validation is used to determine when the training should stop. We used a batch size equal to the size of the dataset. The amount of epochs determines how many times the algorithm will process the entire dataset in order to update the weights. The batch size determines how many samples are processed before updating the weights. The average of the gradients of all samples in a batch is used for the update step.

For every epoch, we calculate the accuracy of the classifiers on the test set. Specifically, we calculate the true positive count, false positive count, true negative count, false negative count, accuracy (eq. 3), precision (eq. 4), recall (eq. 5), f-score (eq. 6), training loss, validation loss, loss on the test set, and the actual predictions. When doing classification, we also record class-specific precision, recall, and f-score. When doing classification, we compute the precision, recall, and f-score as the arithmetic mean of the corresponding class-specific metrics. All metrics, except the training and validation loss, are computed using the test set. By collecting all these results, we were able to thoroughly investigate the training process whenever that was necessary.

$$\text{accuracy} = \frac{tp + tn}{tp + tn + fp + fn} \tag{3}$$

$$\text{precision} = \frac{tp}{tp + fp} \tag{4}$$

$$\text{recall} = \frac{tn}{tn + fp} \tag{5}$$

$$\text{f-score} = \frac{2 * \text{precision} * \text{recall}}{\text{precision} + \text{recall}} \tag{6}$$

8.2 Machine Learning

To avoid effects from class imbalance for the machine learning task, we followed the approach taken by Bhat et al. They used class limits [1]. When working with their dataset, we used the limit 790 for detection. For our own dataset, we used 750 for detection and 237 for classification.

For the machine learning methods developed by Bhat et al., we can only record metrics at the end of the training process. We recorded the accuracy, average F-score, and class-specific precision, recall, and f-score.

For every test we did with the approach of Bhat et al., we will list the results corresponding to the test with the best value of n for the n-grams, together with that value of n.

9 Analyze Important Keywords for the Deep Learning Models

To answer RQ4, we have to be able to determine what keywords are important for the deep learning algorithm. This is difficult to do in general, but this can be done for our CNN models. In order to do this, we use an approach based on the work done in [9], with some slight modifications. The basic idea behind this approach is as follows:

1. We train a CNN model. Suppose that the outputs of the concatenation layer are given by $X = (x_1, x_2, \ldots, x_n)^T$. Then, the outputs of the final output layer are given by $WX + B$, where W is a matrix of learned weights and B is the bias.

2. We feed input vectors into the CNN model one-by-one, and we obtain the activation X in the concatenation layers. One important observation to make is that, due to the architecture of the neural network, each entry x_i in the vector X corresponds to exactly one filter in the convolution layer. Now, for each individual x_i, we

compute the probability $p(L \mid x_i)$ using W and B. Here, L is the *true* label of the vector we gave as input to the model. When using the softmax activation function, this probability is given by equation 7, where ℓ is the number of classes.

$$P(L \mid x_i) = \frac{\exp(w_{ji}x_i)}{\sum_{k=1}^{\ell} \exp(w_{ki}x_i)} \tag{7}$$

3. If $p(L \mid x_i) > 1/2$, then we say that the value x_i corresponds to a candidate keyword. We extract the corresponding words by obtaining the activation in the convolutional layer corresponding to the value x_i. We then look for the index in the input where the convolution gave the response x_i, and we look up the corresponding words in the input vector.

We made the following modifications to the original approach defined in [9]:

- For detection, we use a single output neuron with a sigmoid activation function, instead of two neurons with a softmax activation function. Hence, we estimate $p(L \mid x_i)$ as $p(L \mid x_i) = \text{sigmoid}(w_i x_i)$.
- We do not merge the overlapping h-grams into longer key-phrases. In stead, we collect the h-grams resulting from the extraction process directly.

We collected keywords for both the detection and classification tasks. We identified unique and common keywords, and investigated these.

One important thing to note is that during our hyper-parameter optimization, we found that very large kernel sizes result in the best performance. However, such large kernel sizes result in keywords which occur very infrequently in the dataset, making them difficult to analyze. This is because the length of the identified key phrases is equal to the size of the kernel [9]. In stead, we performed this keyword extraction with models with kernel sizes 1, 2, and 3. These models still perform similarly to their optimized counterparts, but the resulting keywords are easier to analyze.

10 Hyperparameter Optimization

In this section, we will elaborate on the results of the hyper-parameter optimization. In general, we will choose the hyper-parameters that give the best result. The exception to this rule is when there seems to be a pattern in the data which the maximum value does not follow. In this case, a particularly lucky outlier run is more likely.

The hyper-parameter optimization was done with a slightly older version of the dataset (before the final addition to the dataset described in section 4). Because of this, the f-scores presented in this section may be somewhat higher than those in the remainder of the papar.

10.1 BOW (frequency)

For the BOW frequency model, we optimized the amount of hidden layers and the size of the hidden layers. The results for detection can be found in figure 11, and the results for classification can be found in figure 12.

For detection, there is no clear layer size around we get consistently good results. We do observe that in general, we already obtain good results for very small layer sizes. Because of this, we choose the hyperparameters corresponding to the maximum: one hidden layer of size 2.

For classification, we observe that most good performing runs seems to lie around the point (32, 16) (i.e. the first hidden layer has size 32, the second one size 16). Because of this, we choose for this architecture. The run with the model with two layers of size 64 has better performance, but that result is more of an outlier compared to the model with the layer of sizes 32 and 16. Hence, we did not use the (64, 64) model. The (32, 16) model has the added benefit of being simpler, which is desirable.

10.2 BOW (normalized)

For the BOW Normalized we did a hyper-parameter optimization in the same fashion as for the frequency variant. The results for detection are given in figure 13. We once again do not see really clear patterns, but the maximum of (32, 32) does seem to be surrounded by other well-performing models. As such, we choose to use the model with two hidden layers of size 32 for detection.

The results for classification are displayed in figure 14. For this model, we also went for the best performing model, which is the model with one hidden layer of size 32, followed by a hidden layer of size 16.

10.3 TF/IDF

The results for the detection hyper-parameter optimization for TF/IDF can be found in figure 15. We can see that there seem to be multiple "peeks" where the performance is good: (2, 1), (32, 16), (64, 2), and (128, 128). The latter three models are all surrounded by other well-performing models. We discarded (128, 128) as an option, because the other two models could achieve similar performance with simpler models. The choice between the latter two models was difficult, but we ended up using the model where the first hidden layer contains 64 neurons and the second one 2. We had two reasons for this: first of all, this model had slightly better performance than the (32, 16) model. Additionally, this model can be considered simpler. This is becaues there are $2 * 64 = 128$ connections between the two hidden layers, while there are $32 * 16 = 512$ connections between the hidden layers for the (32, 16) model.

The results for classification can be found in figure 16. We can see a clear peak of well-performing models around the (256, 128) model. Hence, we chose the model with a first

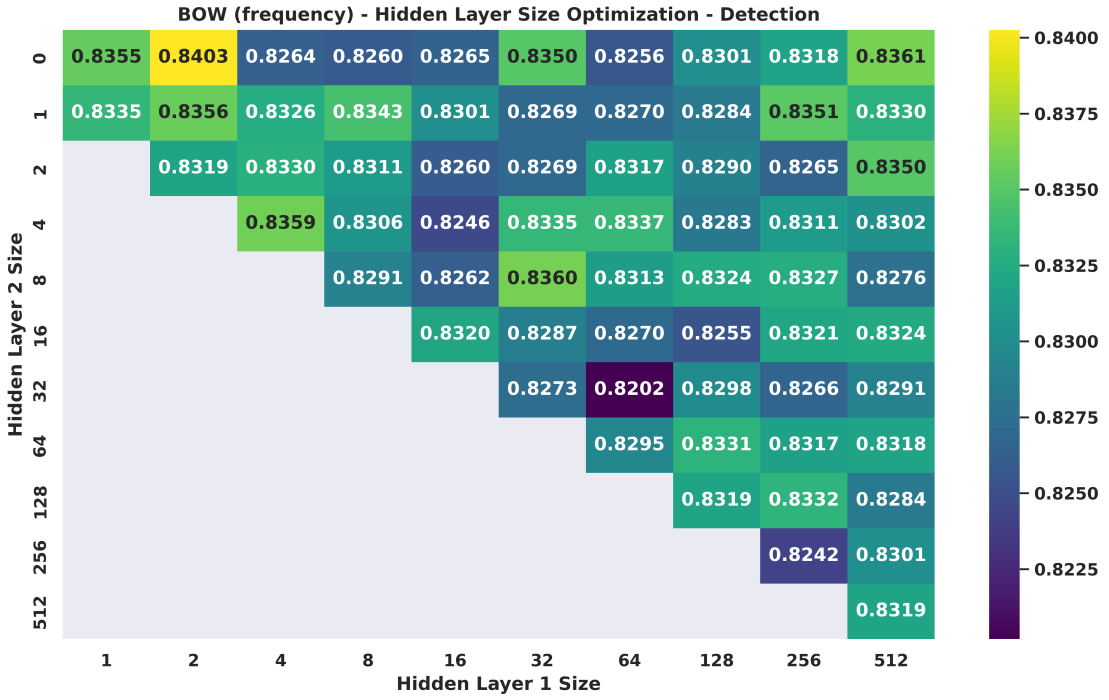

Figure 11: Results of the hyper-parameter optimization for the BOW frequency model, for the detection task.

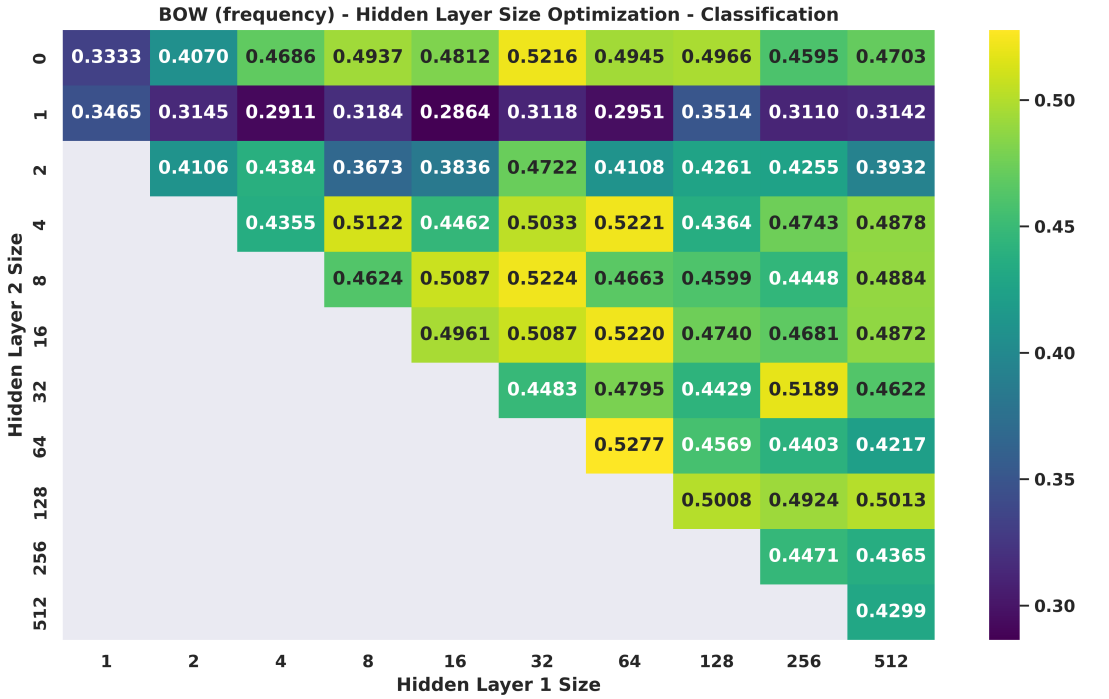

Figure 12: Results of the hyper-parameter optimization for the BOW frequency model, for the classification task.

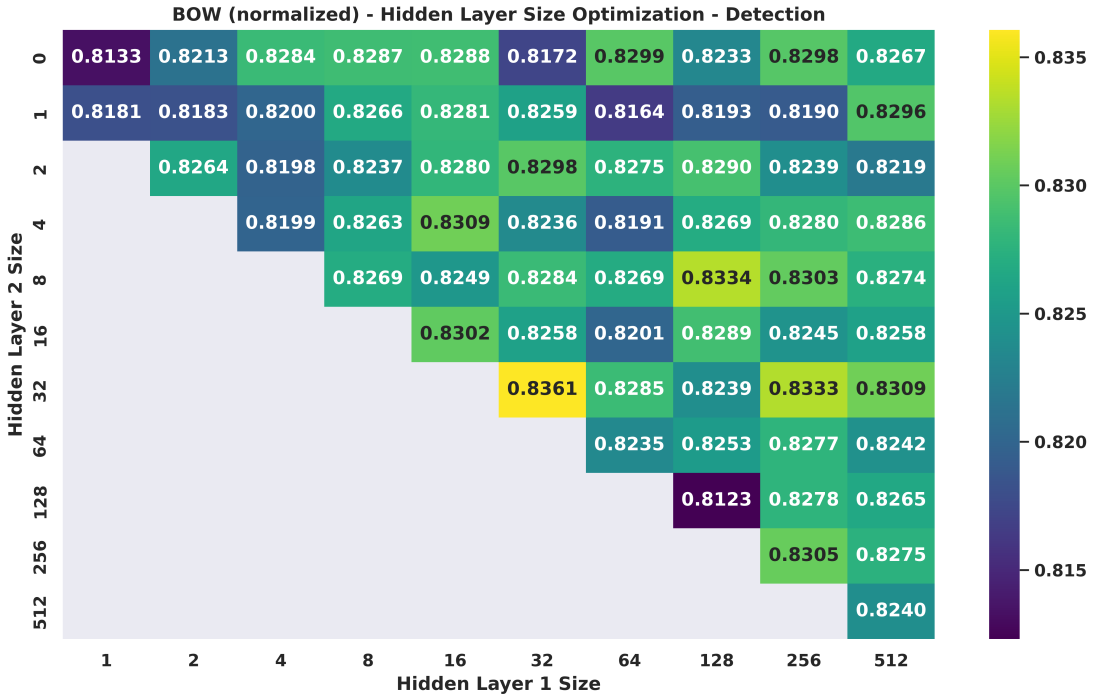

Figure 13: Results of the hyper-parameter optimization for the BOW normalized model, for the detection task.

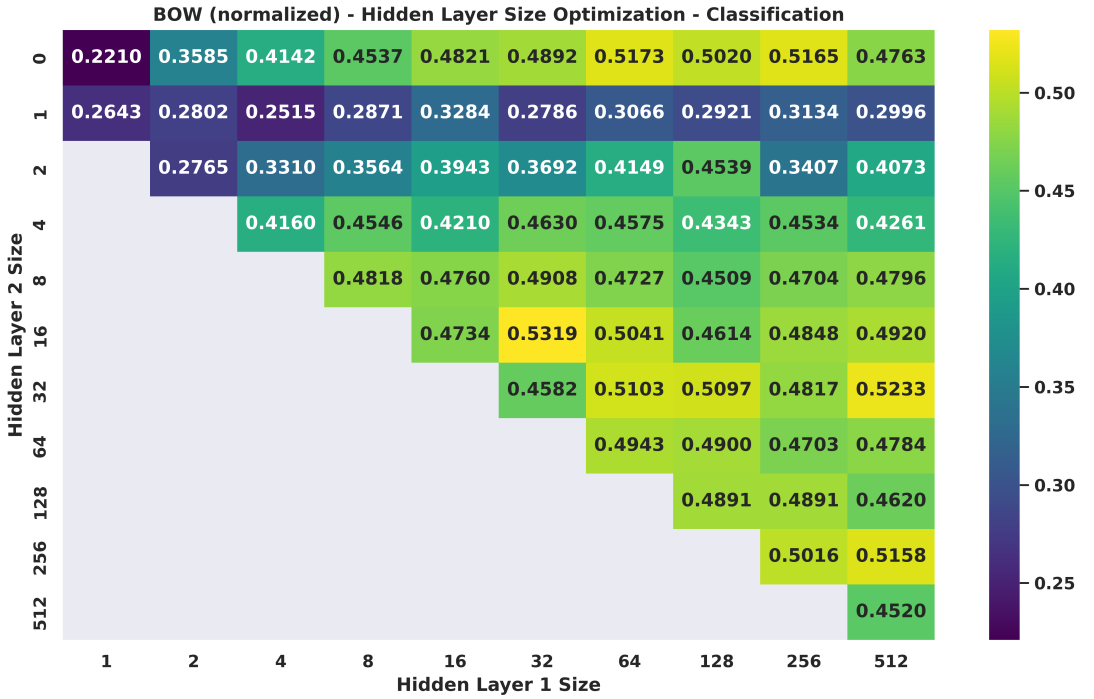

Figure 14: Results of the hyper-parameter optimization for the BOW normalized model, for the classification task.

hidden layer of size 256 and the second layer of size 128 as the best model.

10.4 Issue Properties

In figures 17 and 18, the results of the hyper-parameter optimization for the issue properties model can be found – for detection and classification, respectively. We did not observe any clear patterns in the data, so we went for the simplest high-performing models. This means that for detection, we chose the model with one hidden layer of size 16, followed by a hidden layer of size 4.

For classification, we chose the model with hidden layers of size 16 and 8. We did not choose the model with two hidden layers of size 256, because this model is considerable more complicated and offers little to no benefits.

10.5 Ontology Features

In figure 19, we find the results of the detection hyper-parameter optimization for the ontology features model. After we get to a first layer of size 128, there are many well performing models. We went with the model with a first hidden layer of size 128, and a second hidden layer of size 16. We chose this model because a) it has the best performance b) models with similar hyper-parameters perform really well, c) it is one of the simplest well-performing models.

The results for the classification hyper-parameter optimization can be found in figure 20. For this case, we also went for the best model: two hidden layers, of size 64 and 32, respectively.

10.6 Doc2Vec

The vector size benchmark for the Doc2Vec detection model (figure 21) shows that a vector size of 25 has the best performance. Vector sizes smaller than 25 show much worse performance and vector sizes larger than 25 either show no benefit or even a decrease in performance. Therefore we used vector size 25 in the next benchmarks for the Doc2Vec detection model.

For the hidden layer size benchmark (figure 22) we see that larger hidden layers tend to perform better than smaller layers. Also combinations of larger layers show good performance. However, we see that single layers perform as well or better than combinations of layers. Since we should not make the models unnecessarily complex and because larger layers and combinations of layers slow down the training of the model, we opted for the smallest good performing single layer, which is 64. This layer turns out to have the best performance, but this can also be the result of slight variations in the results. However, it is clear that a hidden layer of 64 is able to obtain good performance and hence we selected this size for the Doc2Vec detection model.

The vector size benchmark for the Doc2Vec classification model (figure 23) shows that larger vector sizes achieve higher performance, up till a vector size of 100. Vector size larger than 100 show a decrease in performance. Therefore we selected a vector size of 100 for the Doc2Vec classification model.

For the Doc2Vec classification model we again see that larger layers have better performance (figure 24). There are multiple peak performances: (256), (16, 16), (128, 16), (256, 128), and (512, 64). As the results contain quite some fluctuations, we selected the simplest good performing layer, which is a hidden layer size of 256.

10.7 CNN

Figure 25 shows that a vector size of 25 performs better than smaller vector sizes for the CNN detection model. It also shows that vector sizes larger than 25 do not seem to improve upon that result. Therefore we decided to use a vector size of 25, as this reduces training and testing time and also reduces memory usage while not decreasing the performance.

In figure 26 we see that that larger kernel sizes (¿ 9) perform better compared to smaller vector sizes (¡ 9) for the CNN detection model. Do note that these results contain a bit of variation and therefore it is not clear if there is an optimal kernel size. However, since we see the gradual improvement when the kernel size gets larger, we opted for a kernel size of 75. We also experimented with combining some of the best kernel sizes, but this did not yield a substantial improvement.

The results in figure 27 suggests that more filters improve the performance. Furthermore we see that 8, 16 and 32 filters perform good. When taking both conclusions into consideration, we opted for a filter size of 32 for the CNN detection model.

For the CNN classification model we see that all vector sizes except 5 perform quite well (figure 28). Vector size 10 seems to perform the best, but this could be due to some variation in the results. Since a vector size of 10 seems to be sufficient for this task, we opted for this vector size in order to reduce training and testing time and memory usage while maintaining good performance.

For the CNN classification model we see that larger kernel sizes seem to yield better results, with a small exception for kernel size 2 (figure 29). Since kernel size 75 scores the best of all, we chose that for the CNN classification model. We also experimented with multiple kernel sizes, but this did not improve the performance.

In figure 30 we see a clear optimum for 64 filters. Hence we chose to use 64 filters for the CNN classification model.

10.8 RNN

The vector size benchmark for the RNN detection model (figure 31) shows that there is no significant difference in the performance between the vector sizes. Since a vector size of 25 obtained the best result, we opted for that size. Another

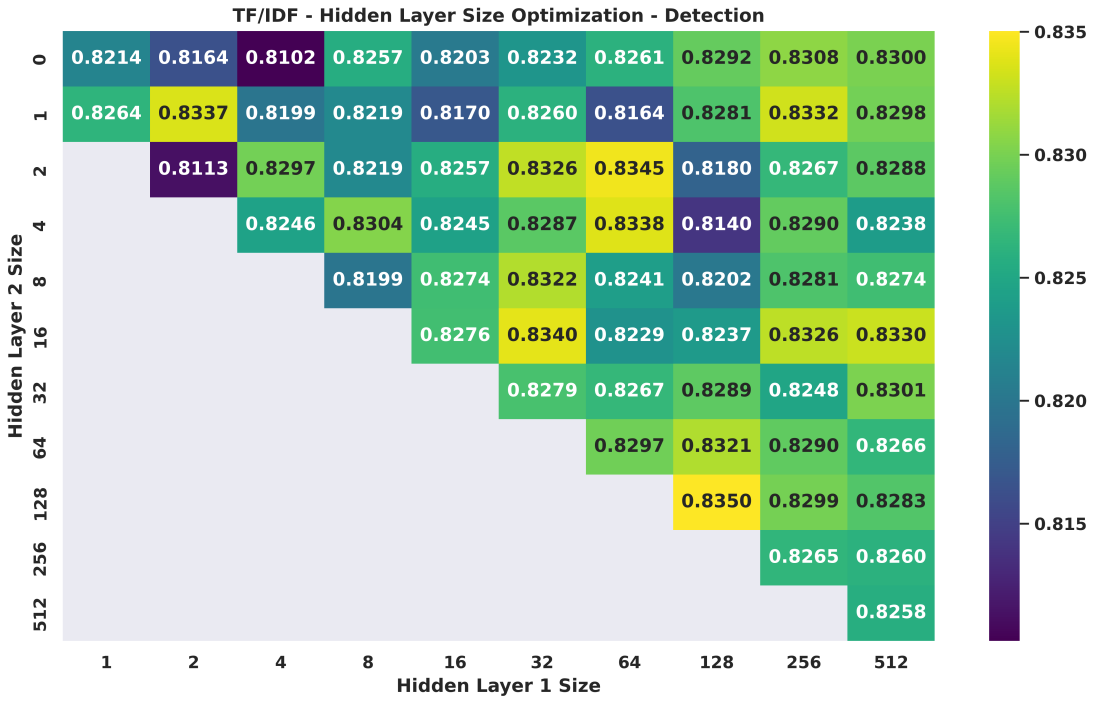

Figure 15: Results of the hyper-parameter optimization for the TF/IDF model, for the detection task.

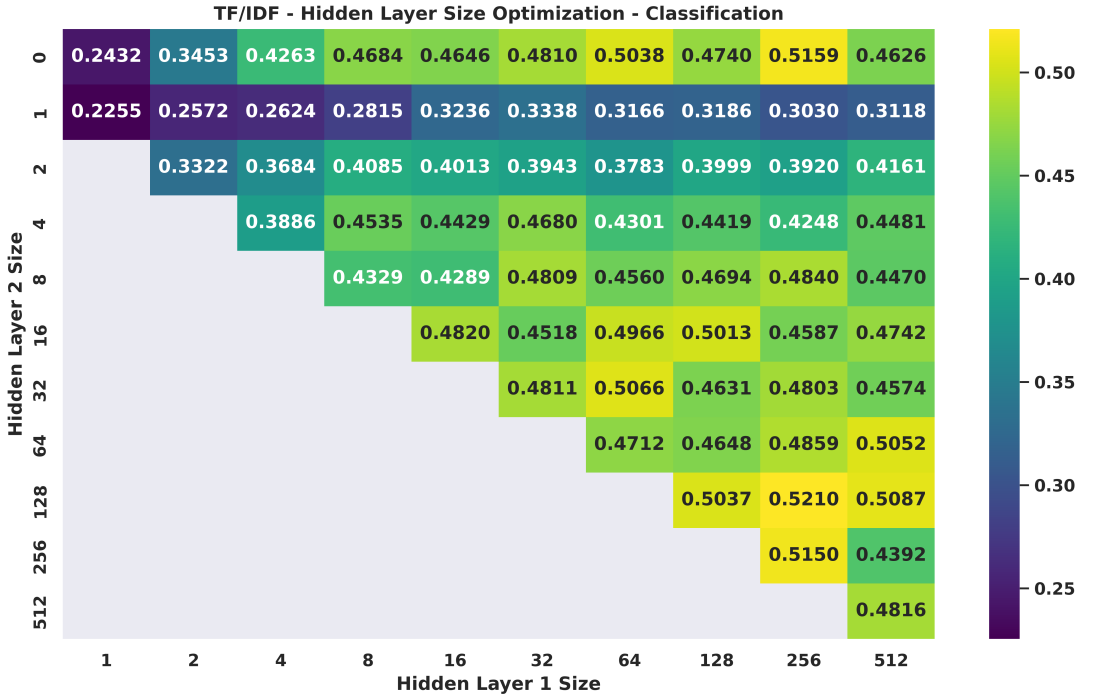

Figure 16: Results of the hyper-parameter optimization for the TF/IDF, for the classification task.

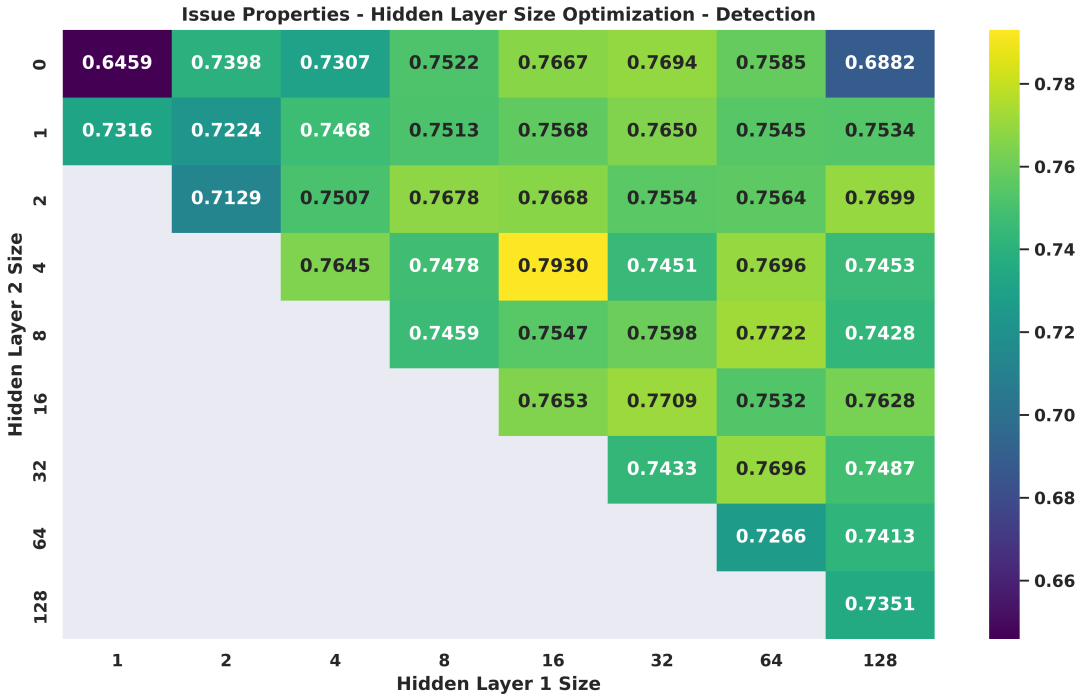

Figure 17: Results of the hyper-parameter optimization for the issue properties model, for the detection task.

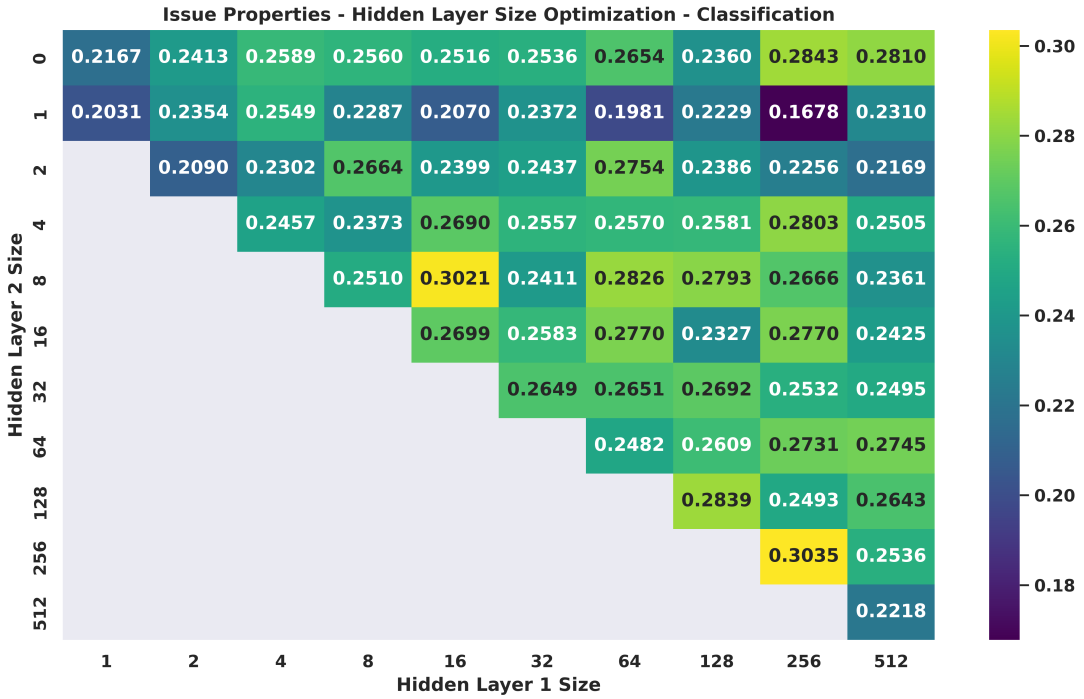

Figure 18: Results of the hyper-parameter optimization for the issue properties model, for the classification task.

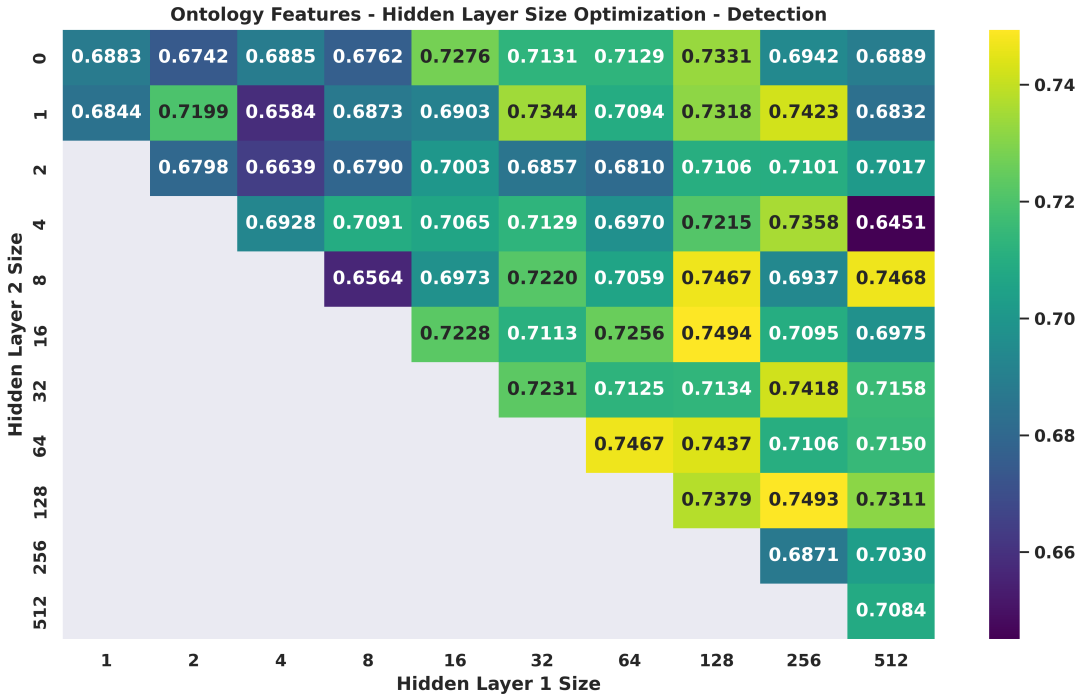

Figure 19: Results of the hyper-parameter optimization for the ontology features model, for the detection task.

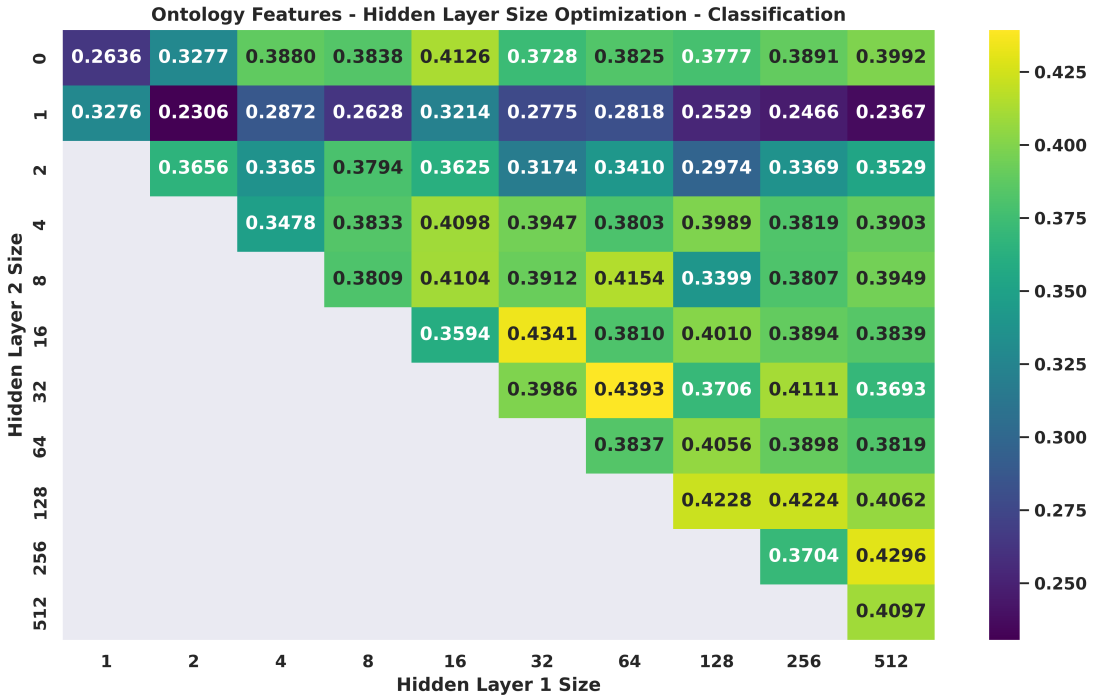

Figure 20: Results of the hyper-parameter optimization for the ontology features model, for the classification task.

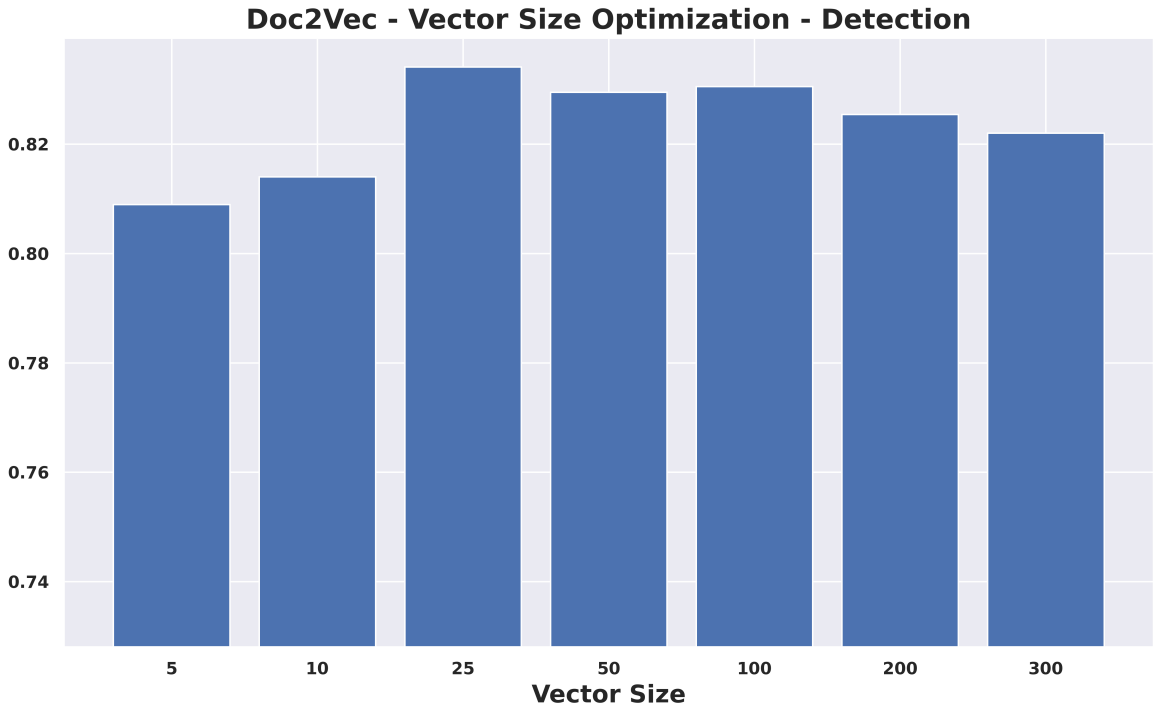

Figure 21: Doc2Vec detection vector size benchmark

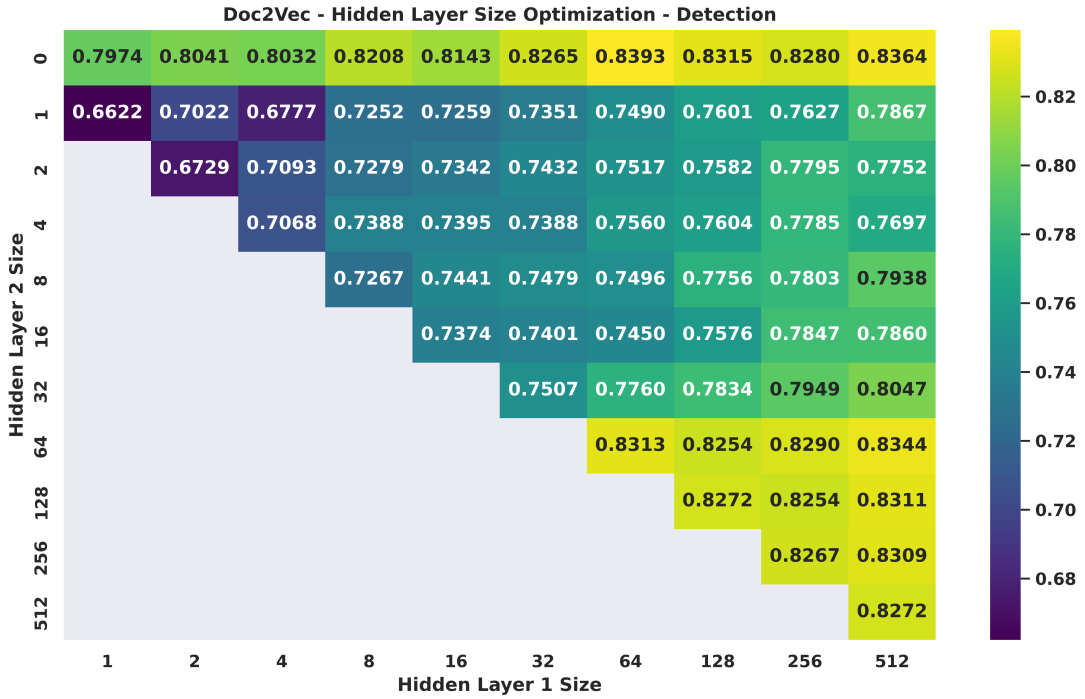

Figure 22: Doc2Vec detection hidden layer size benchmark

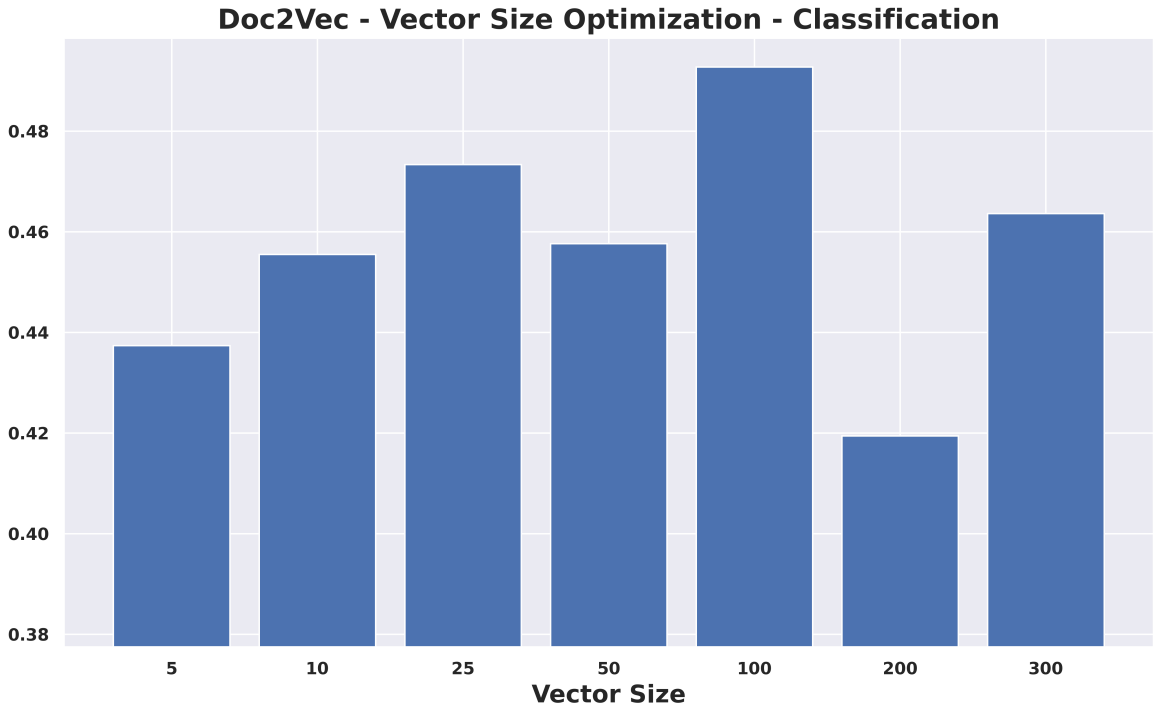

Figure 23: Doc2Vec classification vector size benchmark

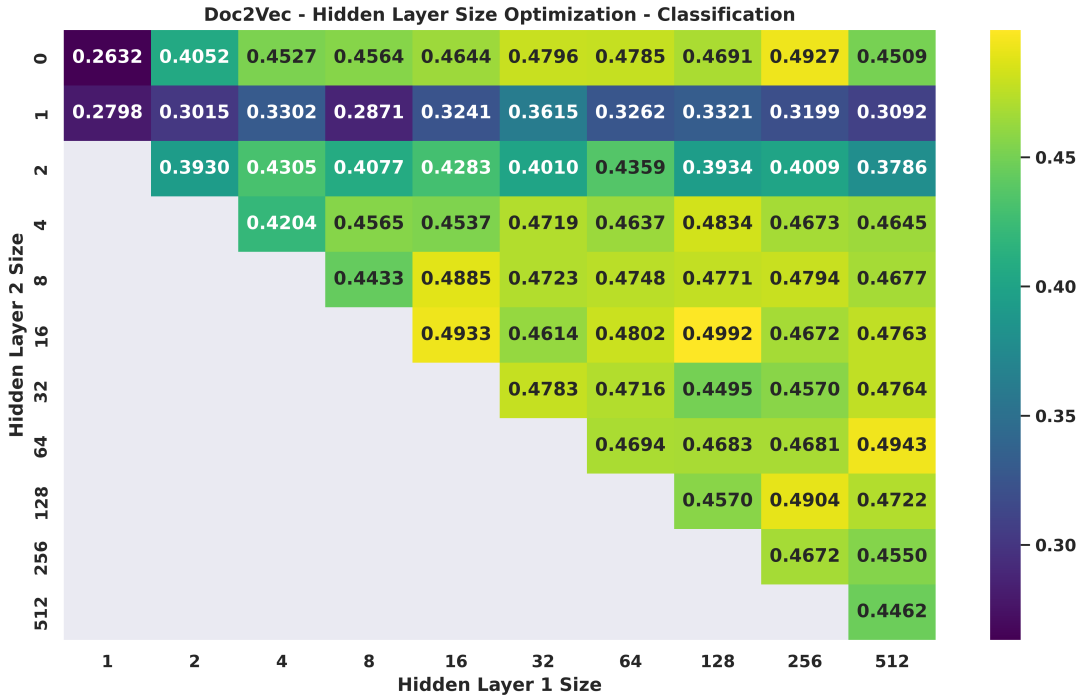

Figure 24: Doc2Vec classification hidden layer size benchmark

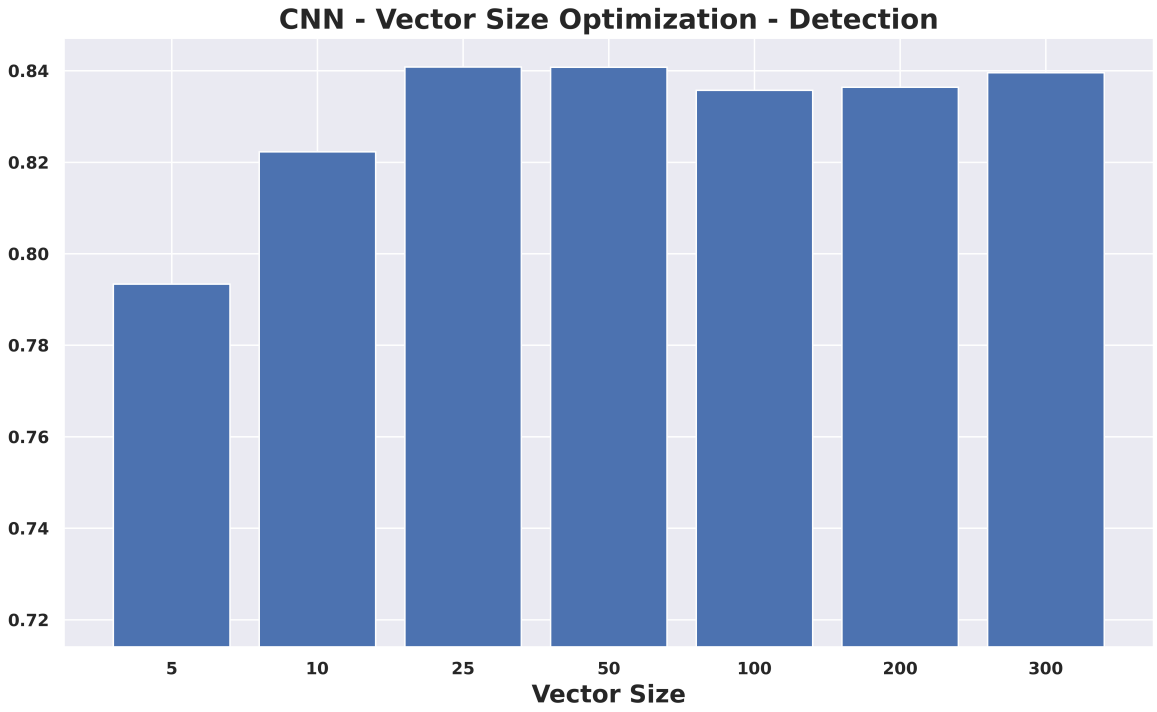

Figure 25: CNN detection vector size benchmark

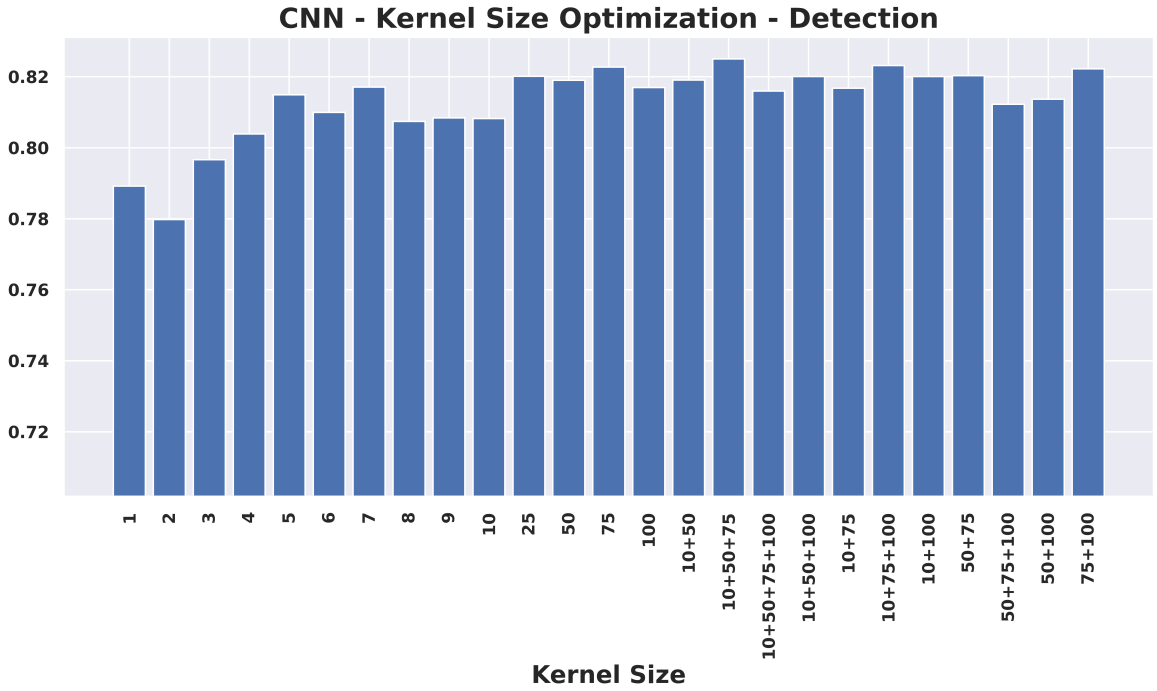

Figure 26: CNN detection kernel size benchmark

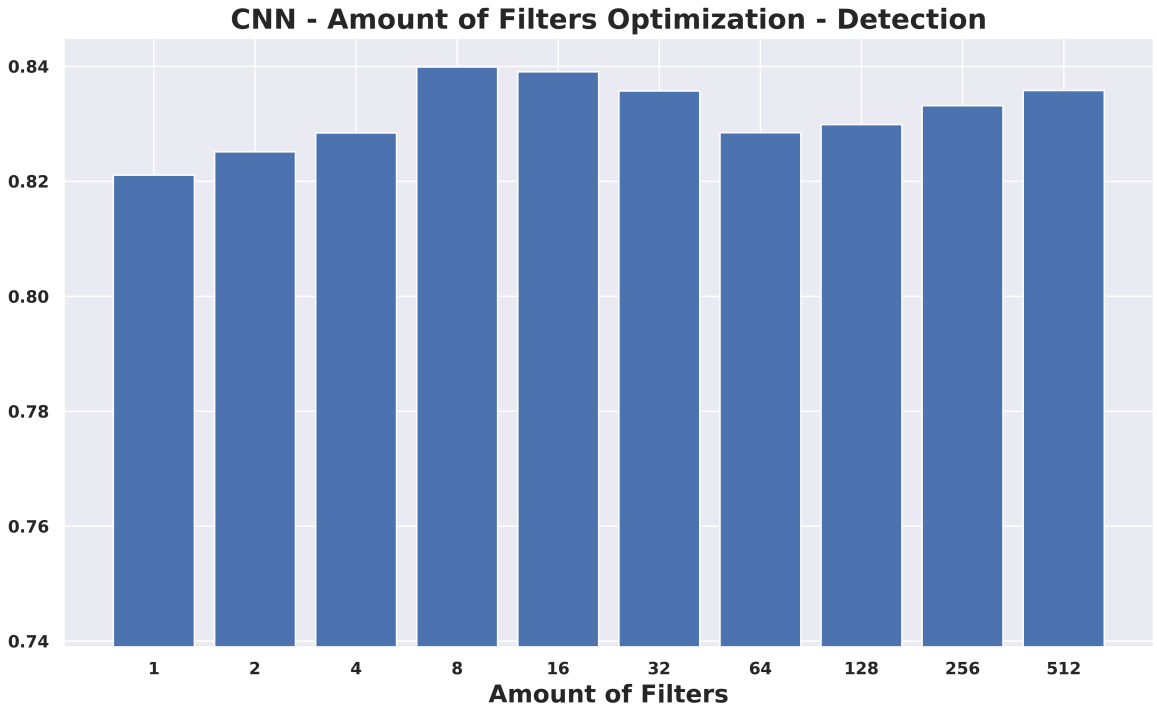

Figure 27: CNN detection number of filters benchmark

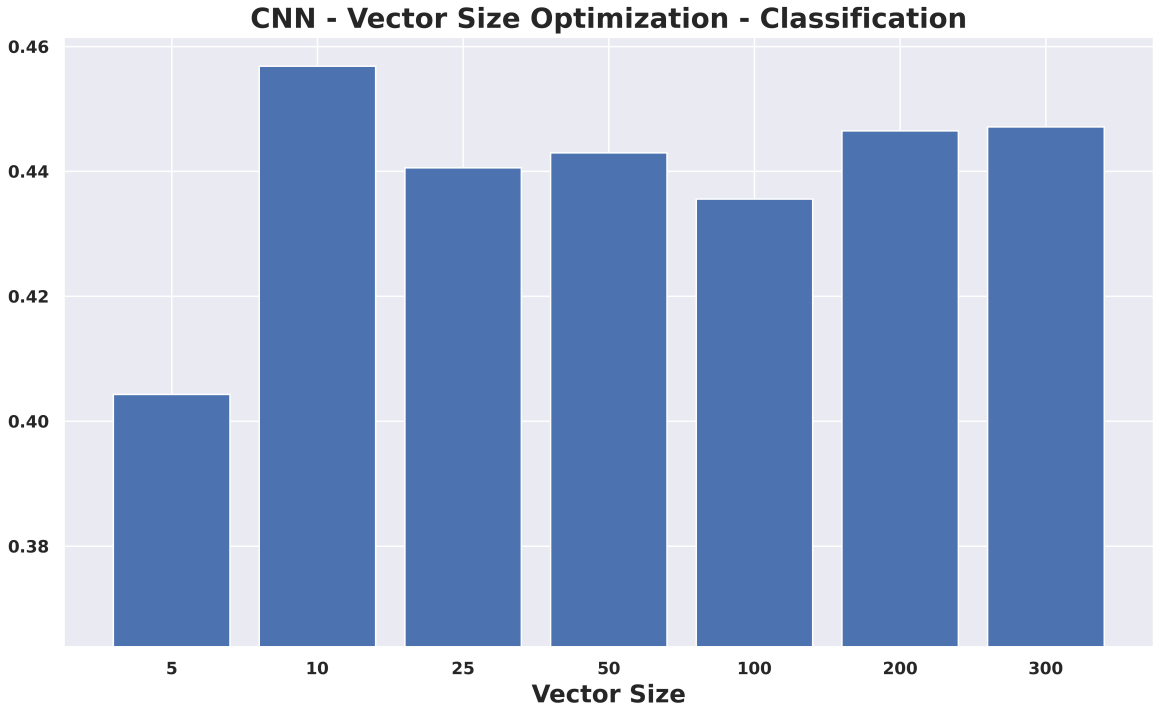

Figure 28: CNN classification vector size benchmark

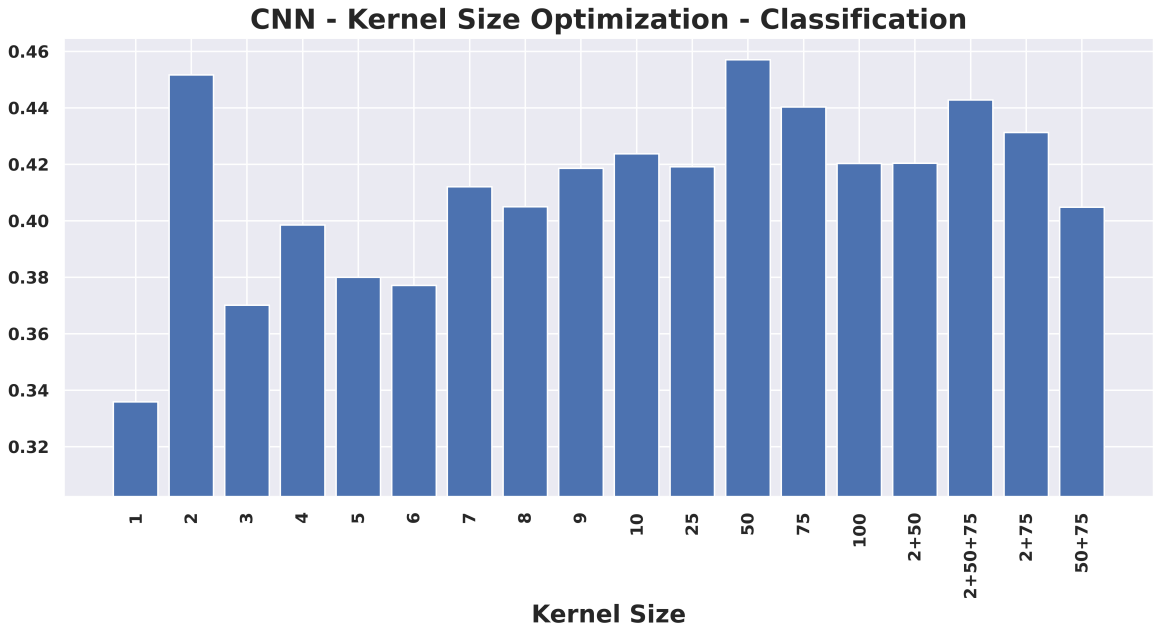

Figure 29: CNN classification kernel size benchmark

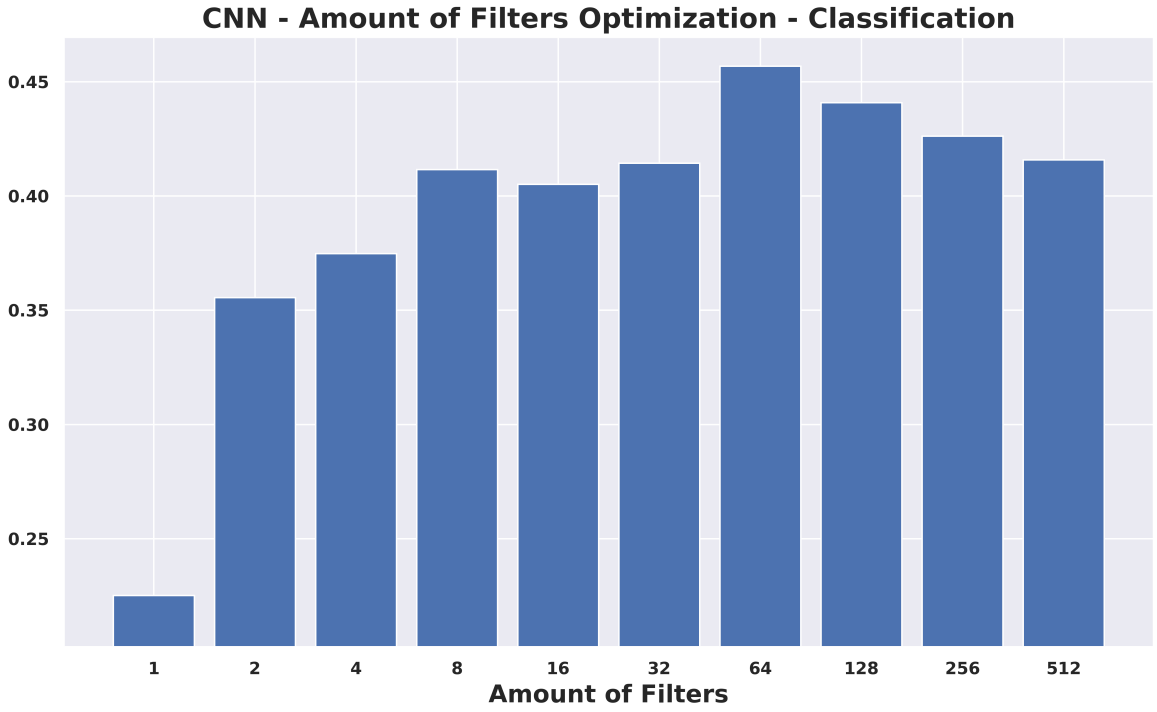

Figure 30: CNN classification number of filters benchmark

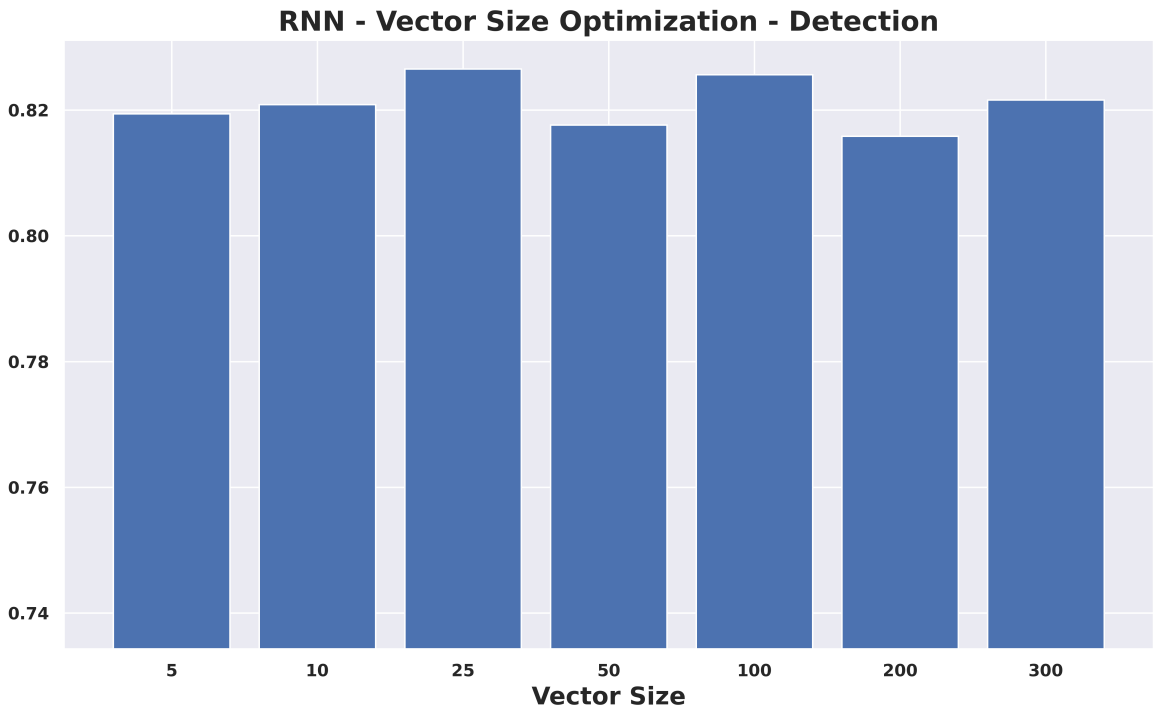

Figure 31: RNN detection vector size benchmark

benefit is that the size is quite small, which reduces training and testing time and also reduces the memory usage.

Figure 32 shows that with bidirectional layer sizes smaller than 16 a lot of performance is lost for the RNN detection model. For larger layer sizes we see a slight improvement for a layer size of 128. Therefore we opted for 128 for the RNN detection model.

When we look at an additional hidden layer after the bidirectional layer, we see that there is a slight benefit for small bidirectional layer sizes (figure 33). A layer size of 4 seems to be giving the best performance and therefore we chose this layer size for the RNN detection model.

For the RNN classification model we see more difference between the different vector sizes (figure 34). Up till vector size 25 we see much improvement regarding performance. After this there does not seem to be much performance benefit, except for a vector size of 300. Therefore we opted for vector size 300.

Increasing the bidirectional layer size up till and including 64 seems to gain performance for the RNN classification model (figure 35). Larger layer sizes do not seem to be an improvement. Hence we chose a bidirectional layer of 64 for the RNN classification model.

For the RNN classification model we see no additional benefit from a hidden layer after the bidirectional layer (figure 36). Therefore we chose to abandon this hidden layer at all. This keeps the model simpler and it also reduces training time and memory usage.

Set number	Issue property combination
1	issuetype
2	issuetype, priority
3	issuetype, n_issuelinks, n_attachments, priority, len_summary, n_watches, n_components, parent, len_description, status, resolution, components, labels, n_labels, n_votes

Table 18: Selected combinations of issue properties for the detection task

10.9 Issue properties

In section 7.2 we determined which issue property combinations we wanted to do hyper-parameter optimizations. For the detection task these are the sets described in table 18. For these models we optimized a single hidden layer. The results of this optimization can be found in figures 37, 38, and 39.

For most combinations we see that the performance increases for larger layer sizes. For set 1 the optimal layer size was 128, for set 2 it was 16, and for set 3 also 128. In the end our best result was 0.8082 f-score for set 1 with a hidden layer of size 128. We used this set and hidden layer for answering the detection task questions for RQ1, RQ2, and RQ3.

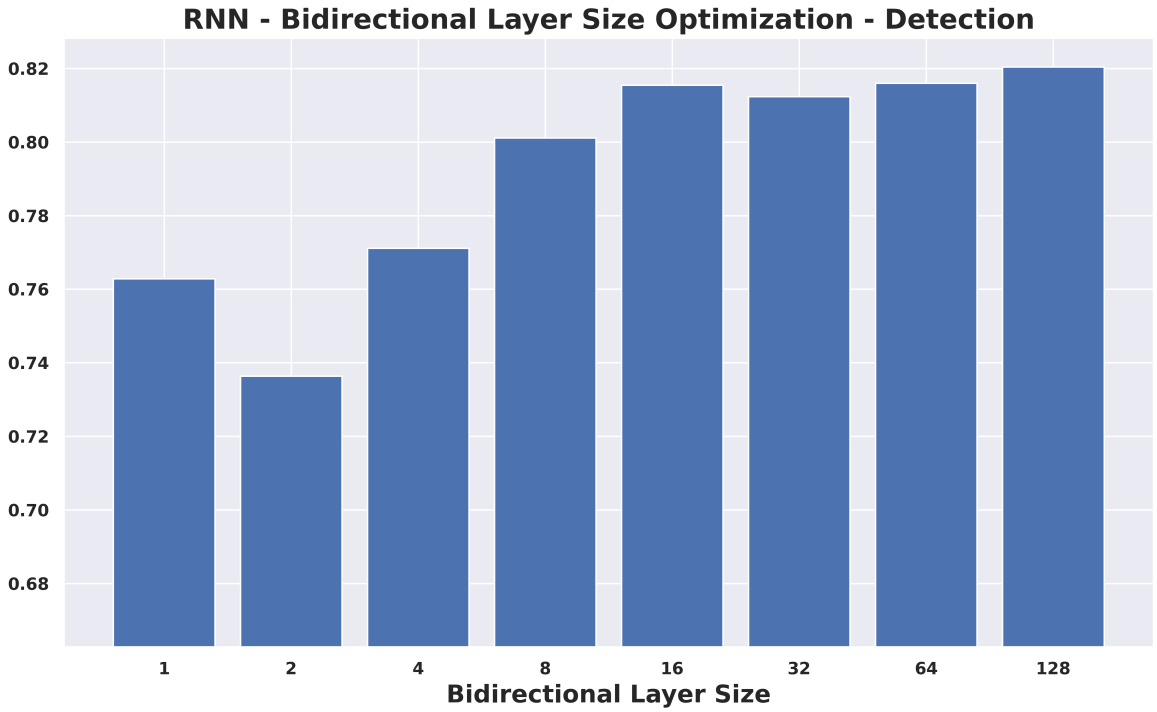

Figure 32: RNN detection bidirectional layer size benchmark

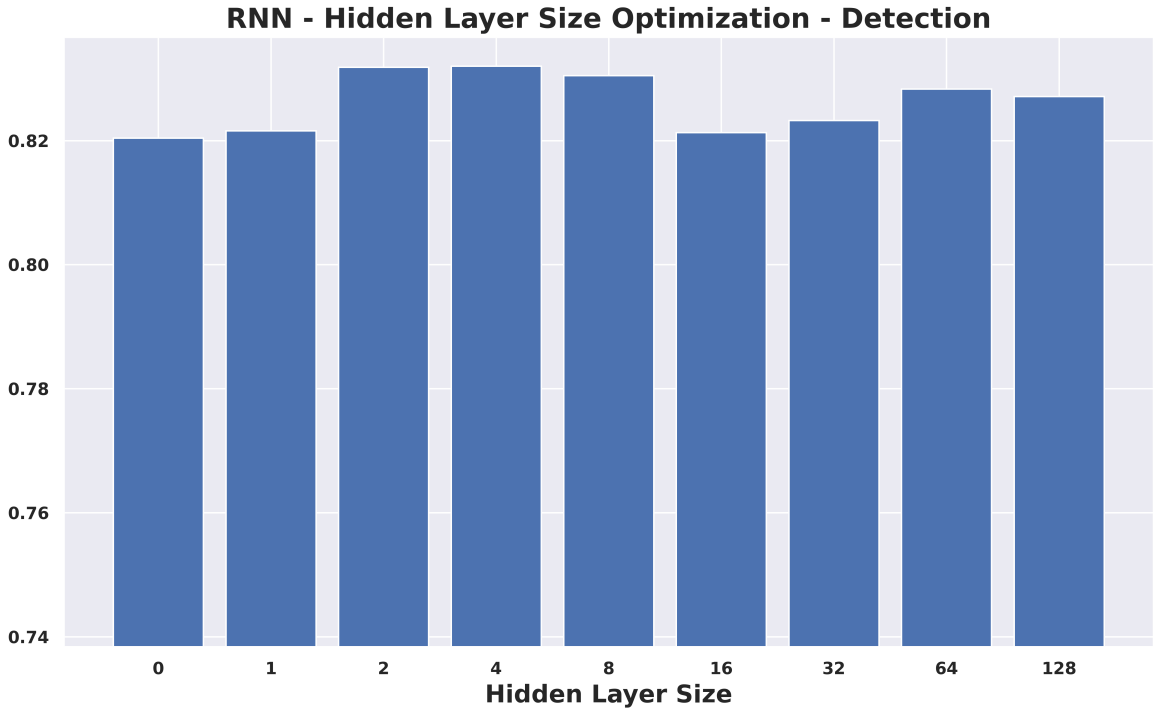

Figure 33: RNN detection hidden layer size benchmark

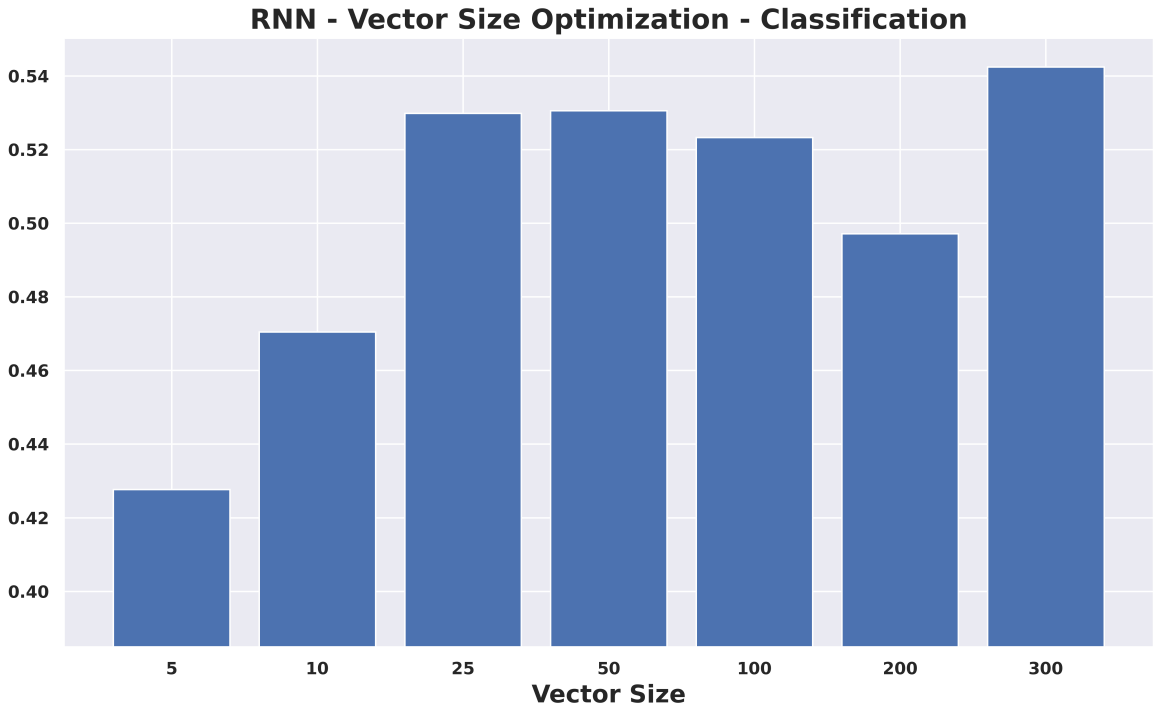

Figure 34: RNN classification vector size benchmark

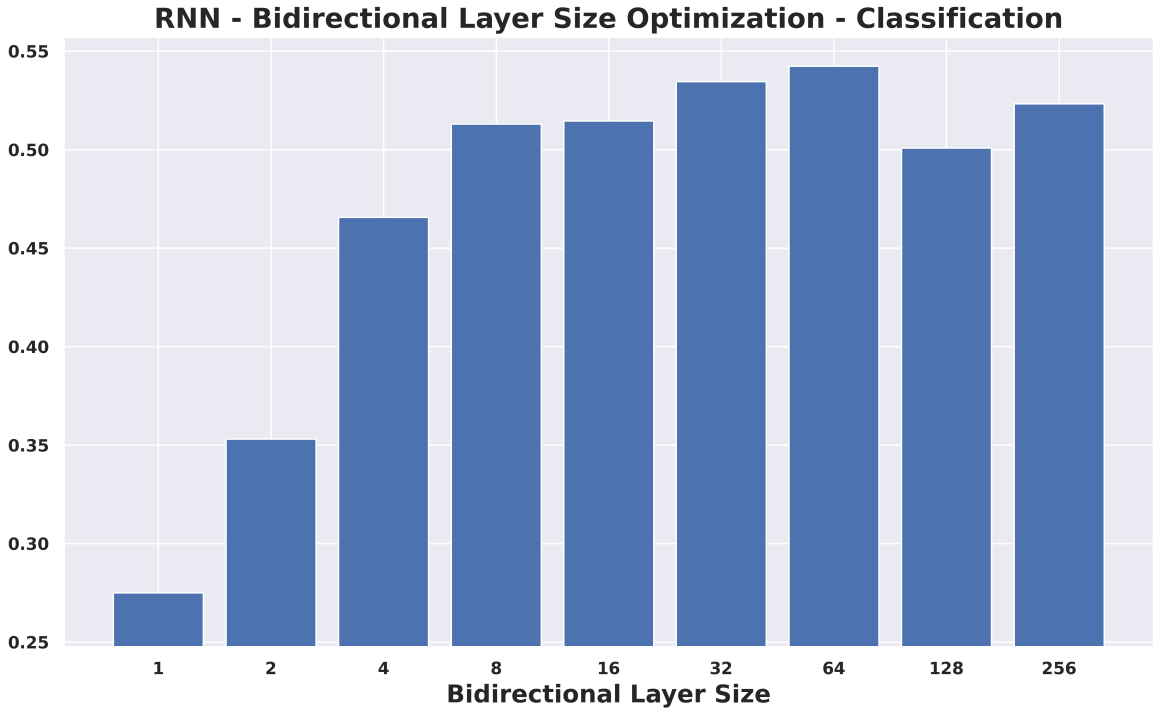

Figure 35: RNN classification bidirectional layer size benchmark

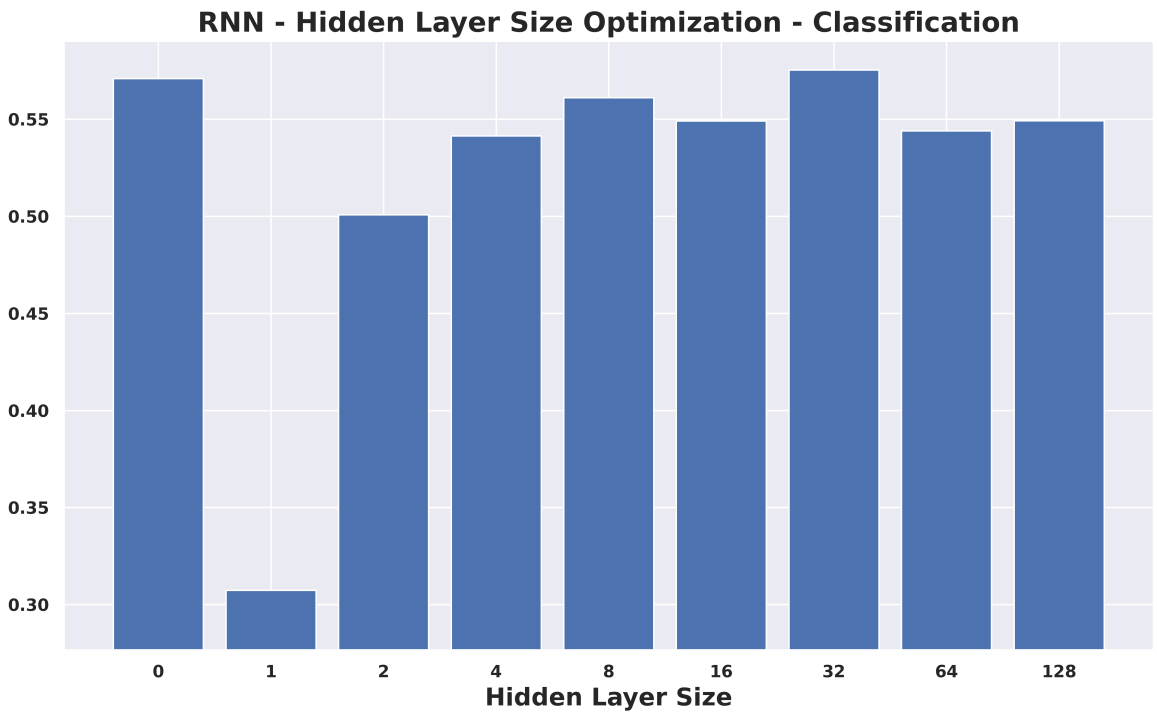

Figure 36: RNN classification hidden layer size benchmark

Figure 37: Issue properties set 1 hidden layer size benchmark for the detection task

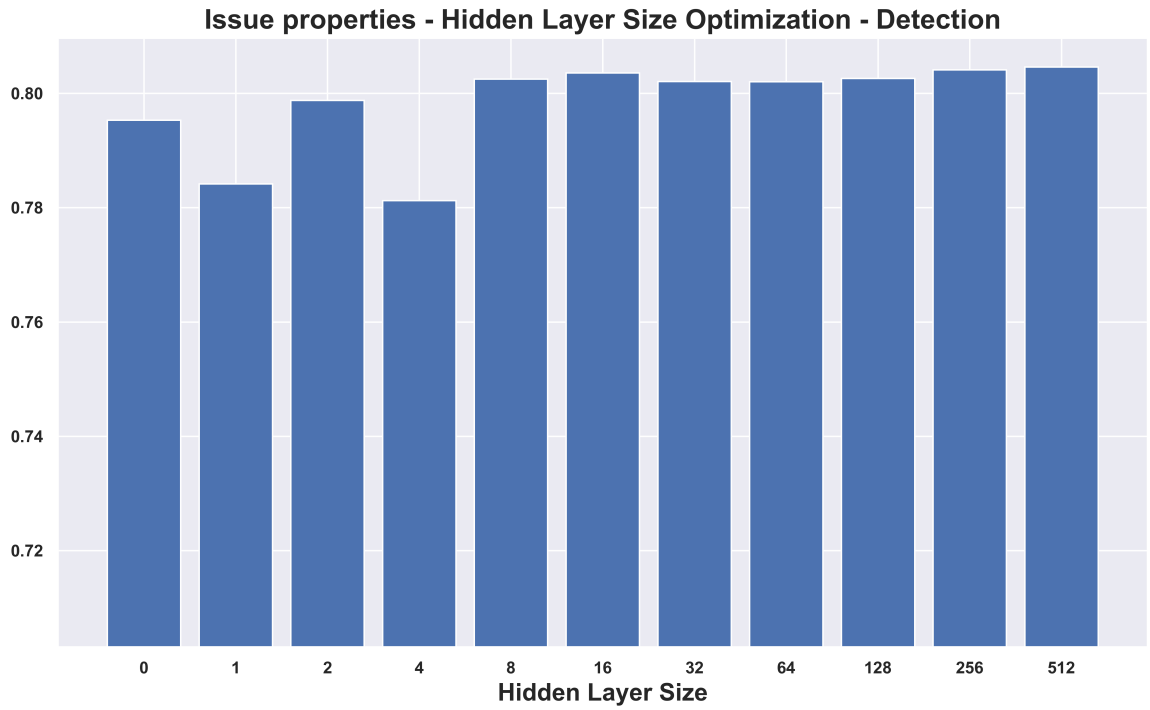

Figure 38: Issue properties set 2 hidden layer size benchmark for the detection task

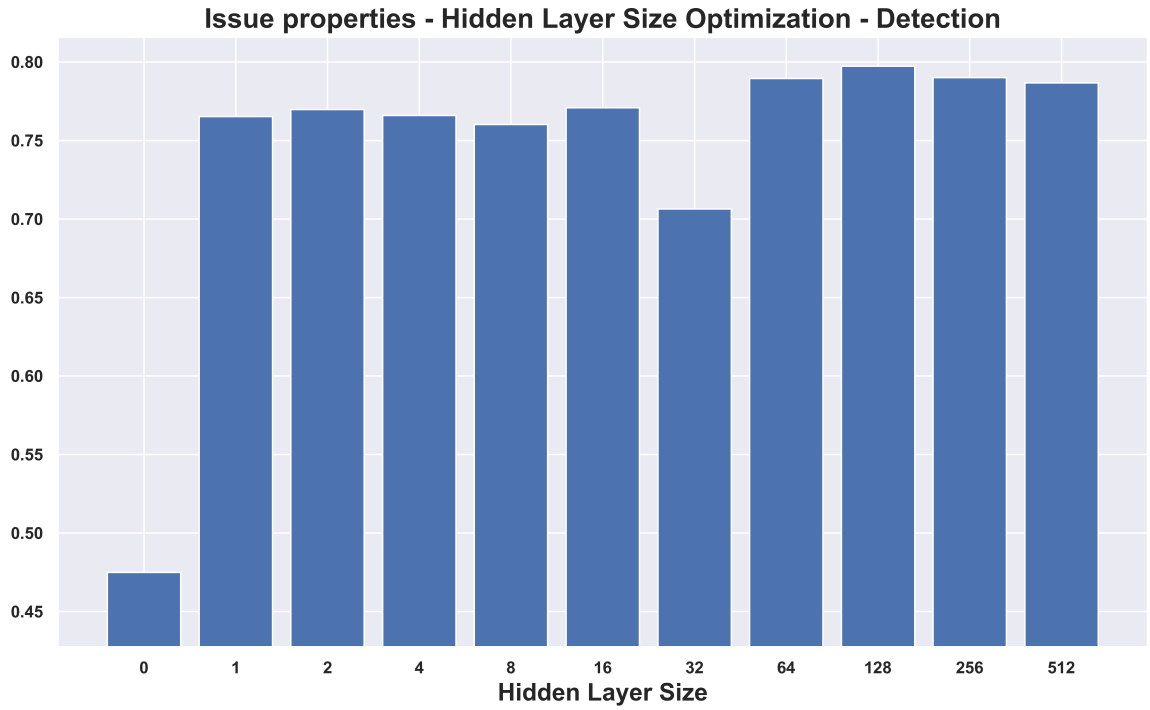

Figure 39: Issue properties set 3 hidden layer size benchmark for the detection task

Set number	Issue property combination
1	components, n_issuelinks, issuetype, n_watches, n_comments, n_attachments, priority
2	n_comments, n_watches, n_issuelinks, components, n_attachments, issuetype, priority, len_description, len_summary, n_components, parent, labels, status, resolution, n_labels, n_votes, n_subtasks

Table 19: Selected combinations of issue properties for the classification task

For the classification task we also determined which combinations of properties we would use for the hyper-parameter optimization. These sets are described in table 19. We again optimized a single hidden layer for these combinations. Figures 40 and 41 show the results of these optimizations.

For set 1 we see the best performance with a hidden layer of size 64. We also see that the performance without a hidden layer is quite good. However, the f-score can be slightly improved by adding this hidden layer of size 64.

We obtained the highest classification f-score of 0.3431 with set 2 using a hidden layer of size 8. Again we see that the model without a hidden layer already had quite good performance. However, again we can slightly improve upon this by adding the hidden layer of size 8. Since this configuration achieved the highest f-score, we will use this for answering the classification questions of RQ1 and RQ3.

11 Optimizing Text Pre-Processing and Features Generation

11.1 Description of Experiments

After optimizing the hyper-parameters, we optimized the text pre-processing and the feature generation. In this section, we first describe what parameters we wanted to optimize, and how we did this. Next, we will present the results of this optimization.

First, we will give a list of the things we can tweak and optimize:

1. We can simplify words using either stemming, lemmatization, or neither of these two – meaning that we do not remove inflected forms. (e.g. lemmatization would change "implementing" to "implement", but this would then not be done)
2. We can optionally annotate words in the issue text with part of speech information.
3. We can apply ontology classes to the text, meaning that we replace belonging to ontology classes with the class names. Additionally, we can do this with and without lexical triggers
4. For the models which make use of word2vec embeddings, we can train the embedding either on all issues from all the projects from which the dataset consists, of use a pre-trained Stack-Overflow word embedding.
5. We can either remove code and noformat blocks in the text and replace them with markers, or we can keep their content (but with class names within the content still replaced with markers)

These tests are in turn relevant for the following models:

- TFIDF, BOWFreq, BOWNorm, Doc2Vec, CNN, and RNN are all affected by (1), (2), (3), and (5)
- CNN and RNN are afffected by (4)
- Ontology Features are affected by (5)
- Issue Property features are affected by none

Performing an exhaustive search requires performing more than 400 different experiments for all possible combinations. This was considered both unpractical and infeasible. In stead, we performed the experiments in different steps. Specifically, we performed the following experiments, in order:

1. We tested (1) for all models. For CNN and RNN, we experimented with both types embeddings (point 4).
2. We tested all combinations of (3) and (5), leading to a total of 6 experiments per model per task. For CNN and RNN, we once again experiment with both types of embeddings (point 4)
3. We tested POS tagging (point 2) with all the text based models (i.e. TFIDF, BOWFreq, BOWNorm, Doc2Vec, CNN, and RNN)

11.2 Results

In this section, we describe the results of optimizing the pre-processing of the text and the feature generation. The experiments we did in this section were described previously in section 11.1.

11.2.1 Removing Inflected Forms

We tested the influence of using stemming and lemmatization for the removal of inflected forms of words. We also experiment with using neither of the two and keeping inflected forms as-is. The results for the detection task are shown in table 20. The main conclusion we draw based on this table, is that there is very little difference between the different approaches. In fact, for most cases, the difference is so small that it can be explained by the randomness inherent to the deep learning process. Because lemmatization was consistently among the top performing combinations for all models, we simply used lemmatization for all models.

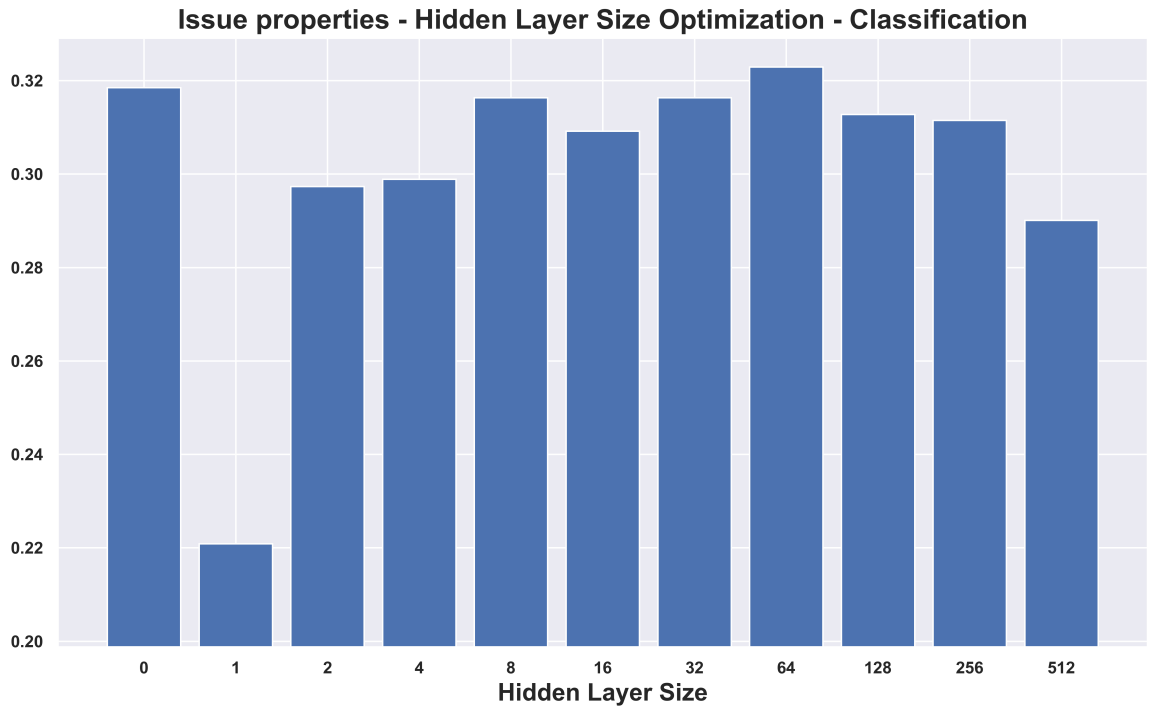

Figure 40: Issue properties set 1 hidden layer size benchmark for the classification task

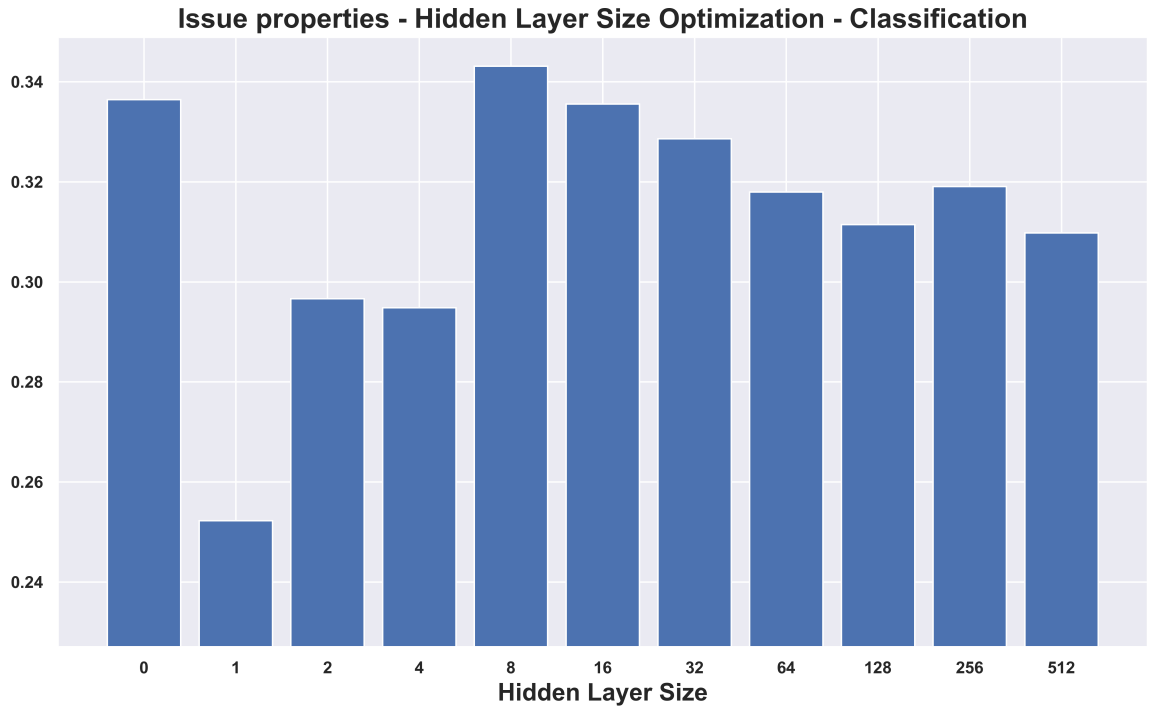

Figure 41: Issue properties set 2 hidden layer size benchmark for the classification task

The results for classification are shown in table 21. For similar reasons, we decided to always go with lemmatization here.

There is one exception to this rule. The Stack Overflow embedding (SO) was trained without stemming or lemmatization, so we did not perform any of those either when working with the Stack Overflow embedding. A surprising result from table 21 is that RNN SO with lemmatization performs considerable better than the two other options, which is not in line with our expectations. However, we still decided not to use lemmatization because this was consistent with how the embedding was trained.

11.2.2 Ontology and Formatting

We tested the effect of ontology classes and the handling of formatting (replacing code and noformat blocks with markers versus keeping their content). The results for detection can be found in table 22, and the results for classification can be found in table 23. We observe that there is very little difference between the results. However, in general no ontology classes and the removal of code and noformat blocks seems to yield (one of) the best results (with the only noteworthy exception being RNN classification). Hence, we went with this option. This option also has the benefit of being the simplest, and being more generalizable. Keeping formatting may introduce language-specific bias, and ontology classes might be incomplete or may have to be updated for new projects.

A similar thing holds for the ontology features. The best results seem to be obtained when replacing code and noformat blocks with markers.

11.2.3 Part of Speech

In tables 24 and 25, we can find the results for testing the models with part-of-speech tagging. The first table contains the results for detection, the second one contains the result for classification. When comparing with the best results from tables 22 and 23, we see no improvement by adding this pre-processing step. Hence, we did not use part-of-speech tagging anymore after this test.

12 RQ1 - Accuracy of machine learning approaches

12.1 Base Models - Detection

Table 27 shows the results for the detection task on our dataset. For these results we used the model configurations described in table 26. In general we see that deep learning massively outperforms machine learning on our dataset. Using deep learning we obtained a 9.09% higher f-score than traditional machine learning. We also have a general note about the Decision Tree model. This model sometimes produces only positive predictions or only negative predictions. This leads to either a very high recall (1.0) or a very low precision (0.0).

CNN performs best for the detection task on our dataset. This is closely matched by other models such as BOW (frequency), CNN (SO), RNN, RNN (SO), and TF/IDF. BOW (normalized) performs a bit worse than BOW (frequency), but since those models have a very similar word embedding and use the same model, we recommend using BOW (frequency) instead of BOW (normalized) for this task.

We also see that Doc2Vec has poor performance. On top of that, training the Doc2Vec embedding requires a lot of time compared to the other embeddings. Hence we do not recommend using this model and embedding for detecting architectural issues.

The ontology feature model performs similarly to the Doc2Vec model. As it takes much effort to obtain the ontology classes, this approach is not worth it.

The issue properties model on the other hand has good performance. It performs slightly worse than the best text models, but with 80.82% f-score it is certainly usable. This indicates that issue properties distinguish architectural issues from non-architectural issues.

We also tested the text models using ontology classes and lexical triggers. This however did not yield better performance, while it took effort to obtain these classes and triggers. Therefore this approach is not worth it.

We also tested the detection performance of the model on the Bhat dataset (table 28). Whereas we saw a clear advantage for deep learning on our dataset, the performance of deep learning and machine learning is similar for the Bhat dataset.

The best deep learning model is RNN with the Stack Overflow embedding and for machine learning the best model is SVM. They are both able to achieve approximately 88% f-score. These models are closely matched by the issue properties model and logistic regression respectively. Especially the issue properties model performance is noticeable. Again we can conclude that issues with certain issue properties are more likely to be architectural or non-architectural. Furthermore we also see good performance for the RNN model with a word embedding trained on issue text and the naive bayes model.

There are also two models that should not be used for this task. These are the Doc2Vec and the ontology features models. These have a major performance loss compared to the best performing models: 19% and 23% worse f-scores respectively.

The models we did not mention have a performance loss in the range of 2%-5% f-score. All these models have decent performance, but are lacking compared to the best models.

The best performing text model that makes use of the ontology classes and lexical triggers performs quite bad for the Bhat dataset. It performs more than 3% worse than the other CNN models. Hence we do not recommend this approach for detecting architectural issues.

Model	Sub-Type	Precision	Recall	F1-score	Imp. over Random
BOW (frequency)	Lemmatization	0.7701	0.8965	0.8273	1.44x
	No Transform	0.7590	0.8980	0.8211	1.43x
	Stemming	0.7722	0.8463	0.8065	1.40x
BOW (normalized)	Lemmatization	0.7639	0.8756	0.8155	1.42x
	No Transform	0.7475	0.9106	0.8187	1.42x
	Stemming	0.7544	0.8916	0.8161	1.42x
CNN	Lemmatization	0.7771	0.8993	0.8332	1.45x
	No Transform	0.7486	0.9254	0.8250	1.43x
	Stemming	0.7411	0.9085	0.8153	1.42x
CNN (SO)	Lemmatization	0.7482	0.9191	0.8235	1.43x
	No Transform	0.7626	0.9049	0.8267	1.44x
	Stemming	0.6873	0.9658	0.8021	1.39x
Doc2Vec	Lemmatization	0.8374	0.6765	0.7464	1.30x
	No Transform	0.8119	0.6695	0.7297	1.27x
	Stemming	0.8229	0.6340	0.7128	1.24x
RNN	Lemmatization	0.7819	0.8706	0.8183	1.42x
	No Transform	0.7938	0.8561	0.8234	1.43x
	Stemming	0.7909	0.8554	0.8210	1.43x
RNN (SO)	Lemmatization	0.7801	0.8853	0.8272	1.44x
	No Transform	0.7734	0.8911	0.8272	1.44x
	Stemming	0.7538	0.8882	0.8142	1.41x
TF/IDF	Lemmatization	0.7636	0.9008	0.8262	1.44x
	No Transform	0.7615	0.8973	0.8234	1.43x
	Stemming	0.7605	0.9000	0.8235	1.43x
Random		0.6589	0.5115	0.5757	

Table 20: Test with lemmatization versus stemming vs neither for the detection task.

Model	Sub-Type	Precision	Recall	F1-Score	Imp. over Random
BOW (frequency)	Lemmatization	0.5212	0.5142	0.5132	2.09
	No Transform	0.5201	0.5166	0.5111	2.08
	Stemming	0.5261	0.5157	0.5162	2.10
BOW (normalized)	Lemmatization	0.5337	0.5228	0.5140	2.09
	No Transform	0.5187	0.5033	0.4958	2.01
	Stemming	0.5145	0.5075	0.5032	2.04
CNN	Lemmatization	0.4840	0.4849	0.4816	1.96
	No Transform	0.4844	0.4863	0.4821	1.96
	Stemming	0.4663	0.4666	0.4607	1.87
CNN (SO)	Lemmatization	0.4913	0.4798	0.4705	1.91
	No Transform	0.5154	0.5077	0.4933	1.97
	Stemming	0.4232	0.4220	0.4035	1.64
Doc2Vec	Lemmatization	0.4932	0.4915	0.4892	1.99
	No Transform	0.4778	0.4744	0.4692	1.91
	Stemming	0.4745	0.4735	0.4682	1.90
RNN	Lemmatization	0.5682	0.5693	0.5657	2.30
	No Transform	0.5772	0.5728	0.5700	2.32
	Stemming	0.5721	0.5703	0.5688	2.31
RNN (SO)	Lemmatization	0.5800	0.5749	0.5743	2.33
	No Transform	0.5419	0.5368	0.5366	2.18
	Stemming	0.5087	0.5100	0.5059	2.06
TF/IDF	Lemmatization	0.5256	0.5180	0.5004	2.03
	No Transform	0.5278	0.5119	0.4950	2.01
	Stemming	0.5149	0.5060	0.4944	2.01
Random		0.2605	0.2543	0.2461	

Table 21: Test with lemmatization versus stemming vs neither for the classification task. See table 51 in section A for class-specific metrics.

Model	Sub-Type	Precision	Recall	F1-score	Imp. over Random
BOW (frequency)	No Ontology, Formatting Markers	0.7701	0.0.8965	0.8273	1.44x
	No Ontology, Keep Formatting	0.7617	0.8869	0.8180	1.42x
	Ontology w/ Lexical Triggers, Formatting Markers	0.7525	0.9028	0.8192	1.42x
	Ontology w/ Lexical Triggers, Keep Formatting	0.7698	0.8714	0.8165	1.42x
	Ontology, Formatting Markers	0.7502	0.8880	0.8112	1.41x
	Ontology, Keep Formatting	0.7752	0.8791	0.8224	1.43x
BOW (normalized)	No Ontology, Formatting Markers	0.7639	0.8756	0.8155	1.42x
	No Ontology, Keep Formatting	0.7483	0.9014	0.8151	1.42x
	Ontology w/ Lexical Triggers, Formatting Markers	0.7284	0.9246	0.8134	1.41x
	Ontology w/ Lexical Triggers, Keep Formatting	0.7511	0.8896	0.8133	1.41x
	Ontology, Formatting Markers	0.7321	0.9188	0.8126	1.41x
	Ontology, Keep Formatting	0.7471	0.9036	0.8155	1.42x
CNN	No Ontology, Formatting Markers	0.7771	0.8993	0.8332	1.45x
	No Ontology, Keep Formatting	0.7485	0.9030	0.8173	1.42x
	Ontology w/ Lexical Triggers, Formatting Markers	0.7573	0.9063	0.8232	1.43x
	Ontology w/ Lexical Triggers, Keep Formatting	0.7411	0.9245	0.8214	1.43x
	Ontology, Formatting Markers	0.7675	0.8931	0.8245	1.43x
	Ontology, Keep Formatting	0.7599	0.9112	0.8272	1.44x
CNN (SO)	No Ontology, Formatting Markers	0.7482	0.9191	0.8235	1.43x
	No Ontology, Keep Formatting	0.7373	0.9222	0.8174	1.42x
	Ontology w/ Lexical Triggers, Formatting Markers	0.7155	0.9267	0.8043	1.40x
	Ontology w/ Lexical Triggers, Keep Formatting	0.7244	0.9209	0.8087	1.40x
	Ontology, Formatting Markers	0.7342	0.9239	0.8146	1.41x
	Ontology, Keep Formatting	0.7658	0.8888	0.8219	1.43x
Doc2Vec	No Ontology, Formatting Markers	0.8374	0.6765	0.7464	1.30x
	No Ontology, Keep Formatting	0.8123	0.6918	0.7450	1.29x
	Ontology w/ Lexical Triggers, Formatting Markers	0.8245	0.6681	0.7347	1.28x
	Ontology w/ Lexical Triggers, Keep Formatting	0.8356	0.6547	0.7319	1.27x
	Ontology, Formatting Markers	0.8403	0.6569	0.7360	1.28x
	Ontology, Keep Formatting	0.8137	0.6506	0.7202	1.25x
RNN	No Ontology, Formatting Markers	0.7895	0.8671	0.8211	1.43x
	No Ontology, Keep Formatting	0.7917	0.8686	0.8236	1.43x
	Ontology w/ Lexical Triggers, Formatting Markers	0.7710	0.8867	0.8188	1.42x
	Ontology w/ Lexical Triggers, Keep Formatting	0.7573	0.8910	0.8147	1.42x
	Ontology, Formatting Markers	0.7827	0.8435	0.8059	1.40x
	Ontology, Keep Formatting	0.7986	0.8469	0.8187	1.42x
RNN (SO)	No Ontology, Formatting Markers	0.7801	0.8853	0.8272	1.44x
	No Ontology, Keep Formatting	0.7261	0.9210	0.8077	1.40x
	Ontology w/ Lexical Triggers, Formatting Markers	0.7480	0.8643	0.7985	1.39x
	Ontology w/ Lexical Triggers, Keep Formatting	0.7196	0.9197	0.8041	1.40x
	Ontology, Formatting Markers	0.7507	0.8817	0.8086	1.40x
	Ontology, Keep Formatting	0.7185	0.9176	0.8025	1.39x
TF/IDF	No Ontology, Formatting Markers	0.7636	0.9008	0.8262	1.44x
	No Ontology, Keep Formatting	0.7615	0.8784	0.8147	1.42x
	Ontology w/ Lexical Triggers, Formatting Markers	0.7462	0.9098	0.8192	1.42x
	Ontology w/ Lexical Triggers, Keep Formatting	0.7567	0.9057	0.8231	1.43x
	Ontology, Formatting Markers	0.7600	0.8889	0.8189	1.42x
	Ontology, Keep Formatting	0.7741	0.8806	0.8228	1.43x
Ontology Features	Formatting Markers	0.7585	0.7156	0.7310	1.27x
	Keep Formatting	0.7616	0.6623	0.7048	1.22x
Random		0.6589	0.5115	0.5757	

Table 22: Results of testing the effects of ontology classes and formatting handling, for the detection task.

Model	Sub-Type	Precision	Recall	F1-Score	Imp. over Random
BOW (frequency)	No Ontology, Formatting Markers	0.5389	0.5297	0.5292	2.15
	No Ontology, Keep Formatting	0.5360	0.5278	0.5254	2.13
	Ontology w/ Lexical Triggers, Formatting Markers	0.5187	0.5150	0.5099	2.07
	Ontology w/ Lexical Triggers, Keep Formatting	0.5121	0.5091	0.5013	2.04
	Ontology, Formatting Markers	0.5320	0.5238	0.5192	2.11
	Ontology, Keep Formatting	0.5098	0.5019	0.4983	2.02
BOW (normalized)	No Ontology, Formatting Markers	0.5337	0.5228	0.5140	2.09
	No Ontology, Keep Formatting	0.5246	0.5113	0.5048	2.05
	Ontology w/ Lexical Triggers, Formatting Markers	0.5003	0.4878	0.4804	1.95
	Ontology w/ Lexical Triggers, Keep Formatting	0.4866	0.4779	0.4689	1.91
	Ontology, Formatting Markers	0.5194	0.5098	0.5050	2.05
	Ontology, Keep Formatting	0.5008	0.4939	0.4850	1.97
CNN	No Ontology, Formatting Markers	0.4840	0.4849	0.4816	1.96
	No Ontology, Keep Formatting	0.4847	0.4868	0.4811	1.95
	Ontology w/ Lexical Triggers, Formatting Markers	0.4780	0.4765	0.4735	1.92
	Ontology w/ Lexical Triggers, Keep Formatting	0.4650	0.4684	0.4621	1.88
	Ontology, Formatting Markers	0.4811	0.4820	0.4774	1.94
	Ontology, Keep Formatting	0.4772	0.4798	0.4738	1.93
CNN (SO)	No Ontology, Formatting Markers	0.5154	0.5077	0.4933	2.00
	No Ontology, Keep Formatting	0.5016	0.4882	0.4774	1.94
	Ontology w/ Lexical Triggers, Formatting Markers	0.4802	0.4667	0.4609	1.87
	Ontology w/ Lexical Triggers, Keep Formatting	0.4698	0.4516	0.4447	1.81
	Ontology, Formatting Markers	0.4855	0.4753	0.4661	1.89
	Ontology, Keep Formatting	0.4907	0.4762	0.4669	1.90
Doc2Vec	No Ontology, Formatting Markers	0.4932	0.4915	0.4892	1.99
	No Ontology, Keep Formatting	0.4795	0.4761	0.4729	1.92
	Ontology w/ Lexical Triggers, Formatting Markers	0.4751	0.4729	0.4693	1.91
	Ontology w/ Lexical Triggers, Keep Formatting	0.4655	0.4601	0.4585	1.86
	Ontology, Formatting Markers	0.4778	0.4682	0.4668	1.90
	Ontology, Keep Formatting	0.4591	0.4580	0.4545	1.85
RNN	No Ontology, Formatting Markers	0.5682	0.5693	0.5657	2.30
	No Ontology, Keep Formatting	0.5482	0.5424	0.5420	2.20
	Ontology w/ Lexical Triggers, Formatting Markers	0.5757	0.5716	0.5688	2.31
	Ontology w/ Lexical Triggers, Keep Formatting	0.5837	0.5784	0.5772	2.35
	Ontology, Formatting Markers	0.5585	0.5562	0.5528	2.25
	Ontology, Keep Formatting	0.5423	0.5391	0.5360	2.15
RNN (SO)	No Ontology, Formatting Markers	0.5800	0.5749	0.5743	2.33
	No Ontology, Keep Formatting	0.5823	0.5793	0.5754	2.34
	Ontology w/ Lexical Triggers, Formatting Markers	0.5060	0.5057	0.5019	2.04
	Ontology w/ Lexical Triggers, Keep Formatting	0.5077	0.5036	0.4988	2.03
	Ontology, Formatting Markers	0.5190	0.5171	0.5139	2.09
	Ontology, Keep Formatting	0.5171	0.5172	0.5133	2.09
TF/IDF	No Ontology, Formatting Markers	0.5256	0.5180	0.5004	2.03
	No Ontology, Keep Formatting	0.5070	0.5024	0.4856	1.97
	Ontology w/ Lexical Triggers, Formatting Markers	0.4877	0.4796	0.4669	1.90
	Ontology w/ Lexical Triggers, Keep Formatting	0.4689	0.4635	0.4522	1.84
	Ontology, Formatting Markers	0.5013	0.4941	0.4785	1.94
	Ontology, Keep Formatting	0.4908	0.4830	0.4687	1.90
Ontology Features	Formatting Markers	0.4289	0.4256	0.4132	1.68
	Keep Formatting	0.4133	0.4130	0.3994	1.62
Random		0.2737	0.2749	0.2713	

Table 23: Results of testing the effects of ontology classes and formatting handling, for the classification task. See table 52 in section A for class-specific metrics.

Model	Sub-Type	Precision	Recall	F1-score	Imp. over Random
BOW (frequency)		0.7701	0.9008	0.8286	1.44x
BOW (normalized)		0.7452	0.9161	0.8197	1.42x
CNN		0.6567	1.0000	0.7928	1.38x
Doc2Vec		0.6581	0.9723	0.7842	1.36x
RNN		0.6567	1.0000	0.7928	1.38x
TF/IDF		0.7598	0.8944	0.8212	1.43x

Table 24: Results of part-of-speech tagging for the detection task.

Model	Sub-Type	Precision	Recall	F1-Score	Imp. over Random
BOW (frequency)		0.5300	0.5294	0.5249	2.13
BOW (normalized)		0.5397	0.5189	0.5115	2.08
CNN		0.4711	0.4706	0.4674	1.90
Doc2Vec		0.4797	0.4760	0.4722	1.92
RNN		0.5535	0.5496	0.5473	2.22
TF/IDF		0.5178	0.5057	0.4883	1.98

Table 25: Results of part-of-speech tagging for the classification task. See table 53 in section A for class-specific metrics.

Model	Configuration
BOW (frequency)	One hidden layer with size 2; adam optimizer; crossentropy loss; lemmatization; no ontology; formatting markers
BOW (normalized)	Two hidden layers with sizes 32 and 32; adam optimizer; crossentropy loss; lemmatization; no ontology; formatting markers
CNN	One convolution with size 75 and 32 filters; Word2Vec vector size of 25; sgd (momentum=0.25) optimizer; hinge loss; lemmatization; no ontology; formatting markers
CNN SO	One convolution with size 75 and 32 filters; Word2Vec vector size of 200; sgd (momentum=0.25) optimizer; hinge loss; no lemmatization, no stemming; no ontology; formatting markers
Doc2Vec	One hidden layer with size 64; sgd (momentum=0.25) optimizer; hinge loss; lemmatization; no ontology; formatting markers
RNN	One bidirectional layer with size 64, followed by a hidden layer with size 4; Word2Vec vector size of 25; sgd (momentum=0.25) optimizer; hinge loss; lemmatization; no ontology; formatting markers
RNN SO	One bidirectional layer with size 64, followed by a hidden layer with size 4; Word2Vec vector size of 200; sgd (momentum=0.25) optimizer; hinge loss; no lemmatization, no stemming; no ontology; formatting markers
TF/IDF	Two hidden layers with size 64 and 2; adam optimizer; crossentropy loss; lemmatization; no ontology; formatting markers
Issue Properties	One hidden layer with size 1; adam optimizer; crossentropy loss; issue properties: issuetype
Ontology Features	Two hidden layers with sizes 128 and 16; adam optimizer; crossentropy loss; lemmatization; formatting markers

Table 26: Model configurations used for the detection task

Model	Precision	Recall	F1-score	Imp. over Random
BOW (frequency)	0.7701	0.8965	0.8273	1.44x
BOW (normalized)	0.7639	0.8756	0.8155	1.42x
CNN	0.7771	0.8993	0.8332	1.45x
CNN (SO)	0.7626	0.9049	0.8267	1.44x
Doc2Vec	0.8374	0.6765	0.7464	1.30x
RNN	0.7917	0.8686	0.8236	1.43x
RNN (SO)	0.7734	0.8911	0.8272	1.44x
TF/IDF	0.7636	0.9008	0.8262	1.44x
Issue Properties	0.7504	0.8769	0.8082	1.40x
Ontology Features	0.7585	0.7156	0.7310	1.27x
Best Model (CNN) + Ontology	0.7573	0.9063	0.8232	1.43x
Support Vector Machine ($n = 4$)	0.7381	0.7480	0.7423	1.29x
Decision Tree ($n = 1$)	0.5007	1.0000	0.6673	1.16x
Logistic Regression ($n = 2$)	0.7381	0.7253	0.7310	1.27x
One-vs-Rest ($n = 1$)	0.6815	0.6640	0.6719	1.17x
Naive Bayes ($n = 1$)	0.6274	0.9080	0.7420	1.29x
Random	0.6589	0.5115	0.5757	

Table 27: Best performing base models for detecting architectural issues on our dataset

Model	Precision	Recall	F1-score	Imp. over Random
BOW (frequency)	0.8168	0.8646	0.8389	1.65x
BOW (normalized)	0.8259	0.8567	0.8387	1.65x
CNN	0.8110	0.8529	0.8298	1.63x
CNN (SO)	0.7976	0.8722	0.8320	1.64x
Doc2Vec	0.4987	0.9519	0.6542	1.29x
Issue Properties	0.8086	0.9675	0.8803	1.73x
Ontology Features	0.5746	0.8724	0.6913	1.36x
RNN	0.8583	0.8788	0.8676	1.71x
RNN (SO)	0.8674	0.9011	0.8832	1.74x
TF/IDF	0.7910	0.8347	0.8104	1.59x
Best Model (CNN) + Ontology	0.7841	0.8150	0.7987	1.57x
Support Vector Machine ($n = 4$)	0.8668	0.8985	0.8816	1.73x
Decision Tree ($n = 1$)	0.0000	0.0000	0.0000	0.00x
Logistic Regression ($n = 4$)	0.8680	0.8841	0.8754	1.72x
One-vs-Rest ($n = 2$)	0.7962	0.8984	0.8435	1.66x
Naive Bayes ($n = 2$)	0.8835	0.8347	0.8572	1.69x
Random	0.4903	0.5305	0.5083	

Table 28: Best performing base models for detecting architectural issues on the Bhat dataset

12.2 Combined Models - Detection

We also experimented with combining models using concatenation, stacking and voting. We combined most of the base models with each other, except for the BOW normalized and the TF/IDF models. The models are very similar to BOW frequency, since they all use similar features and also a similar FNN model. However, BOW frequency consistently had the best performance of the three models. Hence we decided to not include BOW normalized and TFIDF in the results (table 29).

The best combined model is BOW frequency, CNN, and RNN combined using stacking. It achieved an f-score of 83.26%, which is slightly lower than the best performing base model (83.32% f-score). For this task we also see that stacking consistently outperforms voting and concatenation. Furthermore we also see that different combinations produce very similar results, most model combinations achieve 82% f-score using stacking.

We also performed a test on the Bhat dataset with the best performing combined model on our dataset (table 30). This model has a higher f-score of about 1.5% compared to the best base model (RNN SO).

Since combining models does not yield much better performance, requires a lot of programming effort, requires a lot of training time, and requires good hardware for training, we do not recommend using combinations of models for detecting architectural issues unless the absolute best performance is needed.

12.3 Base Models - Classification

Table 32 shows the results of the classification task on our dataset. Compared to the detection task, we see a lot more variation between the models.

The two best performing models are both RNN models. The RNN model with the SO embedding achieved 57.43% f-score, and the RNN model with the embedding trained on issue texts achieved 56.57% f-score. The next best model is a machine learning model: SVM. This model achieved an f-score of 54.96%, which is almost 2.5% worse than the best deep learning model. Therefore deep learning seems to have an advantage for classifying architectural issues compared to machine learning.

Most other models show poor performance, except for BOW frequency and to some extent BOW normalized. Since BOW frequency is a much more lightweight model compared to RNN, this option can be considered as well. Still however, this model loses almost 5% performance compared to RNN.

For classification we also tested the text models using ontology classes and lexical triggers. This performed slightly worse than the best model without ontology classes. Since this approach also does not yield better performance, we recommend not using these ontology classes and lexical triggers.

12.4 Combined Models - Classification

For classifying issues on our dataset, we also experimented with combining models (table 33). We again did not include BOW normalized and TF/IDF in the results for the same reason as explained earlier. We see that both stacking and voting are able to achieve high f-scores of about 56%. However, all of these models include an RNN model. Therefore we suspect that this is the main driver of the high f-score. Besides, the performance is worse compared to the base RNN models. Hence we see no reason for using combinations of models over the base RNN model for classifying issues.

13 RQ2 - Cross Dataset Generalizability

Table 34 contains the results of the cross-dataset performance of the different models. For this benchmark, the models were trained on one dataset and tested on the other dataset. With this benchmark we can determine how well the performance translates to different datasets, so we can gain insight in the generalizability of the different models.

The best model is TF/IDF. It achieves 88.69% f-score when training on our dataset and testing on the Bhat dataset and it achieves 86.60% when training on the Bhat dataset and testing on our dataset. Other well performing models are the other BOW models (frequency and normalized), both RNN models, and both CNN models. We also see good performance for the issue properties model, which indicates that for both datasets similar issue property values determine whether an issue is architectural or not.

Also for this test we see poor performance for the Doc2Vec and ontology features models. These two models are massively outperformed by the previously mentioned models and therefore we do not recommend using these for detecting architectural issues.

Similar conclusions can be made for the machine learning models. The best machine learning models perform slightly better than the worst deep learning model. This indicates that the performance of machine learning does not translate well to other dataset. Some of the deep learning models on the other hand are able to do this very effectively.

14 RQ3 - Cross Project Generalizability

14.1 Detection

In table 35 the results are shown for the cross-project detection performance for the different models. With these results we gain insight how well the architectural issue detection performance translates to other projects. To do this, we tested on a single project and trained on all the other projects in the dataset. By doing this for every project and

Model	Sub-Type	Precision	Recall	F1-score	Imp. over Random
BOW (frequency) + CNN + RNN	Concatenation	0.7731	0.8793	0.8192	1.42x
	Stacking	0.7661	0.9155	0.8326	1.45x
	Voting	0.7994	0.8448	0.8211	1.43x
BOW (frequency) + CNN + RNN + Issue Properties + Ontology Features	Concatenation	0.7563	0.8680	0.8034	1.40x
	Stacking	0.7522	0.9329	0.8323	1.45x
	Voting	0.8235	0.8323	0.8276	1.44x
BOW (frequency) + Issue Properties	Concatenation	0.7722	0.8033	0.7833	1.36x
	Stacking	0.7138	0.9268	0.8043	1.40x
	Voting	0.8495	0.6109	0.7101	1.23x
BOW (frequency) + Issue Properties + Ontology Features	Concatenation	0.7523	0.8670	0.8008	1.39x
	Stacking	0.7330	0.9204	0.8148	1.42x
	Voting	0.8157	0.7617	0.7873	1.37x
BOW (frequency) + Ontology Features	Concatenation	0.7230	0.8784	0.7875	1.37x
	Stacking	0.7120	0.9356	0.8076	1.40x
	Voting	0.8518	0.4501	0.5885	1.02x
CNN + Issue Properties	Concatenation	0.6914	0.8301	0.7406	1.29x
	Stacking	0.7573	0.9029	0.8225	1.43x
	Voting	0.8565	0.6605	0.7457	1.30x
CNN + Issue Properties + Ontology Features	Concatenation	0.7176	0.8889	0.7855	1.36x
	Stacking	0.7561	0.9203	0.8294	1.44x
	Voting	0.8169	0.7680	0.7915	1.37x
CNN + Ontology Features	Concatenation	0.6878	0.9070	0.7787	1.35x
	Stacking	0.7530	0.9049	0.8206	1.43x
	Voting	0.8407	0.4855	0.6145	1.07x
Issue Properties + Ontology Features	Concatenation	0.7014	0.8426	0.7605	1.32x
	Stacking	0.7400	0.9239	0.8210	1.43x
	Voting	0.8552	0.4619	0.5991	1.04x
RNN + Issue Properties	Concatenation	0.7261	0.9162	0.8067	1.40x
	Stacking	0.7775	0.8883	0.8276	1.44x
	Voting	0.8713	0.6528	0.7456	1.30x
RNN + Issue Properties + Ontology Features	Concatenation	0.7588	0.8379	0.7924	1.38x
	Stacking	0.7691	0.9063	0.8295	1.44x
	Voting	0.8303	0.7541	0.7890	1.37x
RNN + Ontology Features	Concatenation	0.7490	0.8773	0.8027	1.39x
	Stacking	0.7803	0.8868	0.8280	1.44x
	Voting	0.8627	0.4759	0.6123	1.06x
Random		0.6589	0.5115	0.5757	

Table 29: Combined models for detecting architectural issues on our dataset

Model	Sub-Type	Precision	Recall	F1-score	Imp. over Random
BOW (frequency) + CNN + RNN	Stacking	0.8327	0.9793	0.8998	1.77x
Random		0.4903	0.5305	0.5083	

Table 30: Combined models for detecting architectural issues on the Bhat dataset

Model	Configuration
BOW (frequency)	Two hidden layers with sizes 64 and 64; adam optimizer; crossentropy loss; lemmatization; no ontology; formatting markers
BOW (normalized)	Two hidden layers with sizes 32 and 16; adam optimizer; crossentropy loss; lemmatization; no ontology; formatting markers
CNN	One convolution with size 50 and 64 filters; Word2Vec vector size of 10; sgd (momentum=0.25) optimizer; hinge loss; lemmatization; no ontology; formatting markers
CNN SO	One convolution with size 50 and 64 filters; Word2Vec vector size of 200; sgd (momentum=0.25) optimizer; hinge loss; no lemmatization, no stemming; no ontology; formatting markers
Doc2Vec	One hidden layer with size 256; sgd (momentum=0.25) optimizer; hinge loss; lemmatization; no ontology; formatting markers
RNN	One bidirectional layer with size 128; Word2Vec vector size of 300; sgd (momentum=0.25) optimizer; hinge loss; lemmatization; no ontology; formatting markers
RNN SO	One bidirectional layer with size 128; Word2Vec vector size of 200; sgd (momentum=0.25) optimizer; hinge loss; no lemmatization, no stemming; no ontology; formatting markers
TF/IDF	Two hidden layers with size 256 and 128; adam optimizer; crossentropy loss; lemmatization; no ontology; formatting markers
Issue Properties	One hidden layer with size 8; adam optimizer; crossentropy loss; issue properties: n_comments, n_watches, n_issuelinks, components, n_attachments, issuetype, priority, len_description, len_summary, n_components, parent, labels, status, resolution, n_labels, n_votes, n_subtasks
Ontology Features	Two hidden layers with sizes 64 and 32; adam optimizer; crossentropy loss; lemmatization; formatting markers

Table 31: Model configurations used for the classification task

Model	Precision	Recall	F1-score	Imp. over Random
BOW (frequency)	0.5389	0.5297	0.5292	2.15x
BOW (normalized)	0.5337	0.5228	0.5140	2.09x
CNN	0.4840	0.4849	0.4816	1.96x
CNN (SO)	0.5154	0.5077	0.4933	2.00x
Doc2Vec	0.4932	0.4915	0.4892	1.99x
RNN	0.5682	0.5693	0.5657	2.30x
RNN (SO)	0.5800	0.5749	0.5743	2.32x
TF/IDF	0.5256	0.5180	0.5004	2.03x
Issue Properties	0.3529	0.3515	0.3431	1.39x
Ontology Features	0.4289	0.4256	0.4132	1.68x
Best Model (RNN) + Ontology	0.5757	0.5716	0.5688	2.31x
Support Vector Machine ($n = 3$)	0.5562	0.5525	0.5496	2.23x
Decision Tree ($n = 1$)	0.0627	0.2500	0.1006	0.41x
One-vs-Rest ($n = 1$)	0.4334	0.4374	0.4270	1.74x
Naive Bayes ($n = 1$)	0.5623	0.4982	0.4772	1.98x
Random	0.2605	0.2543	0.2461	

Table 32: Best performing base models for classifying architectural issues. See See table 54 in section A for class-specific metrics.

Model	Sub-Type	Precision	Recall	F1-Score	Imp. over Random
BOW (frequency) + Issue Properties	Concatenation	0.4007	0.3961	0.3656	1.49x
	Stacking	0.5402	0.5369	0.5332	2.17x
	Voting	0.5317	0.5278	0.5268	2.14x
BOW (frequency) + Issue Properties + Ontology Features	Concatenation	0.3731	0.3707	0.3588	1.46x
	Stacking	0.5311	0.5292	0.5258	2.14x
	Voting	0.5126	0.5164	0.5087	2.07x
BOW (frequency) + Ontology Features	Concatenation	0.3618	0.3612	0.3393	1.38x
	Stacking	0.5308	0.5300	0.5266	2.14x
	Voting	0.5340	0.5301	0.5274	2.14x
BOW (frequency) + RNN	Concatenation	0.3890	0.4161	0.3799	1.54x
	Stacking	0.5061	0.4677	0.4560	1.85x
	Voting	0.5715	0.5667	0.5659	2.30x
Issue Properties + Ontology Features	Concatenation	0.2704	0.3352	0.2731	1.11x
	Stacking	0.2429	0.2951	0.2451	1.00x
	Voting	0.4526	0.4499	0.4388	1.78x
RNN + Issue Properties	Concatenation	0.4125	0.4280	0.4030	1.64x
	Stacking	0.5695	0.5661	0.5646	2.29x
	Voting	0.5635	0.5595	0.5581	2.27x
RNN + Issue Properties + Ontology Features	Concatenation	0.3709	0.3779	0.3679	1.49x
	Stacking	0.5557	0.5535	0.5488	2.23x
	Voting	0.5390	0.5382	0.5323	2.16x
RNN + Ontology Features	Concatenation	0.4134	0.4335	0.4089	1.66x
	Stacking	0.5598	0.5570	0.5549	2.25x
	Voting	0.5672	0.5612	0.5586	2.27x
Random		0.2605	0.2543	0.2461	

Table 33: Combined models for classifying architectural issues. See table 55 in section A for class-specific metrics.

Model	Test Set	Precision	Recall	F1-score
BOW(frequency)	Bhat	0.8744	0.8705	0.8724
	Our	0.8547	0.8449	0.8498
BOW(normalized)	Bhat	0.8634	0.8731	0.8682
	Our	0.8490	0.8668	0.8577
CNN	Bhat	0.8008	0.9134	0.8526
	Our	0.7781	0.9050	0.8345
CNN(SO)	Bhat	0.7849	0.9347	0.8518
	Our	0.7539	0.9306	0.8290
Doc2Vec	Bhat	0.8052	0.6633	0.7271
	Our	0.7646	0.7157	0.7348
Issue Properties	Bhat	0.7683	0.9265	0.8400
	Our	0.7691	0.9228	0.8389
Ontology Features	Bhat	0.7481	0.6124	0.6717
	Our	0.7239	0.6587	0.6785
RNN	Bhat	0.8143	0.8797	0.8441
	Our	0.8135	0.8683	0.8376
RNN(SO)	Bhat	0.8337	0.9091	0.8689
	Our	0.7983	0.9081	0.8450
TF/IDF	Bhat	0.8755	0.8986	0.8869
	Our	0.8377	0.8962	0.8660
Support Vector Machine ($n = 2$)	Bhat	0.7974	0.4818	0.6696
	Our	0.6360	0.4345	0.7049
Decision Tree (ML)	Bhat	0.4948	1.0000	0.3276
	Our	0.3433	1.0000	0.1754
Logistic Regression (ML)	Bhat	0.5109	0.6393	0.5118
	Our	0.7122	0.3275	0.6897
One-vs-Rest (ML)	Bhat	0.6867	0.4766	0.6242
	Our	0.5707	0.4639	0.6876
Naive Bayes (ML)	Bhat	0.6471	0.8190	0.6844
	Our	0.5897	0.5361	0.7092

Table 34: Model performance for detecting architectural issues cross-dataset

taking the average performance, we obtain the cross-project score for each model. We did these tests for both datasets.

Again we see unmatched performance for RNN. The embedding trained on issue texts shows the best result, closely followed by the embedding trained on Stack Overflow posts.

One surprising result is the issue properties model. The architectural and non-architectural issue properties of the issues in the Bhat dataset seem to be more distinguishable than the issue properties of the issues in our dataset. Hence the performance of this model on the Bhat dataset is much higher.

We also see strong performance for both CNN models on both datasets. However, it does lack some performance for the Bhat dataset compared to the RNN models. We also see decent performance from the BOW frequency, BOW normalized, and ontology features models, but the performance is considerably lower compared to RNN. For the Doc2Vec model we see similar performance for both datasets, but the results are marginally worse than the ones for CNN and RNN.

For the machine learning models we see decent performance on the Bhat dataset, but poor performance on our dataset. In the end, the best machine learning model is outperformed by 3% on the Bhat dataset and 8.5% on our dataset compared to the best deep learning model. Hence we recommend using deep learning for this task, as the performance of deep learning translates better to new projects.

14.2 Classification

We performed a similar test for classification on our dataset (table 36).

For the classification task we see that the best deep learning model and best machine learning model have similar performance, about 50% f-score. For deep learning, only the RNN models obtain this level of performance and for machine learning only SVM is able to achieve that. The next best performing models are BOW frequency and BOW normalized, but they are still lacking approximately 6% f-score.

After that, we see a group of poorly performing models, namely CNN, CNN SO, Doc2Vec, ontology features, TF/IDF, one-vs-rest, and naive bayes. All of these models are lacking about 10%-12% f-score compared to the best models. The issue properties model is the worst model, with only 29.79% f-score and is not really usable for the classification task. This indicates that classification between different projects using issue properties is not an option.

15 RQ4

In this section, we will cover research question 4. We collected the keywords using the approach described in section 9. For every class, we collected keywords in two ways: we identified the keywords which occurred the most often, and we identified the keywords which were associated with the highest probabilities for some class. The amount of issues we selected per table was determined by 1) making sure that we included all keywords $\geq$ the threshold selected, and 2) making sure the table fits on 1 page.

15.1 Detection Keywords

For detection, the keywords are give tables 37, 38, 39, 40, and 41. The first two tables contain the keywords indicative of architectural issues, the next two tables those for the non-architectural label, and the final table contains a list of common tables. Tables 37 and 39 contain keywords selected based on frequency (≥ 4 and ≥ 3 occurrences, respectively). Tables 38 and 40 contain keywords selected based on probability (≥ 0.7 and ≥ 0.675, respectively). The keywords in table 41 were selected based on frequency (≥ 15).

In table 37, we can make several observations about the keywords. First of all, there are many phrases involving the word "would" (e.g. "would nice", "would allow", "would great", "would good", "similar would nice"). Apparently, such phrases occur often as keyword for architectural issues. However, we also observe that such phrases do not occur much in table 38.

Tables 37 and 38 also list a number of quality attributes, such as "pluggable interface", "consistency", "bad latency", and "trunk performance performance". The classifier hence feels that quality attributes may be important to identify an architectural issue. Quality attributes, or words closely related to them, seem to be more common in table 38. these, we see multiple phrases about latency, performance, and consistency – but also security, in the form of key phrases containing the word "security" or "authentication".

We also observe some key phrases hinting towards the addition of components or functionality (e.g. "useful feature" and "need support"). There are also a number of keywords about dependencies (e.g. "dependency versionnumber").

Finally, we can also observe that there are a lot of keywords in tables 37 and 38 that somehow relate to components. For instance, there are simple words such as "ui" and "manager node", but also more project specific phrases such as "hdfs client". Such words seem to be somewhat more common in tables 37 and 38 than in the tables for non-architectural keywords (i.e. tables 39 and 40). Those two tables still contain component names (e.g. "mapreduce client"), but such phrases seem to be more rare.

One of the main things we noted in tables 39 and 40, is that the keywords associated with non-architectural issues are strongly associated with software failure, and often contain method or class names. Table 39 for instance contains the keywords "formattedtraceback", "unformattedtraceback", and "formattedloggingoutput". These are all markers we used where we removed formatting blocks. All these keywords are related to program output, and, more noteworthy, program failure (in the form of exceptions). There are also many longer key phrases in tables 39 and 40 containing these 3 markers. Additionally, table 39 also contains some other error related keywords such as "find-

Model	Dataset	Precision	Recall	F1-score
BOW(frequency)	Bhat	0.7439	0.7254	0.7305
	Our	0.7892	0.8041	0.7956
BOW(normalized)	Bhat	0.7615	0.7001	0.7293
	Our	0.7727	0.8162	0.7935
CNN	Bhat	0.7525	0.7950	0.7692
	Our	0.7643	0.8671	0.8115
CNN(SO)	Bhat	0.7146	0.8451	0.7731
	Our	0.7259	0.9178	0.8062
Doc2Vec	Bhat	0.7426	0.7902	0.7572
	Our	0.8108	0.6938	0.7432
Issue Properties	Bhat	0.7944	0.9620	0.8696
	Our	0.7647	0.7897	0.7475
Ontology Features	Bhat	0.5367	0.7258	0.6155
	Our	0.7643	0.6634	0.6972
RNN	Bhat	0.8588	0.8096	0.8318
	Our	0.7456	0.9087	0.8149
RNN(SO)	Bhat	0.8150	0.8060	0.8105
	Our	0.7036	0.9520	0.8023
TF/IDF	Bhat	0.7256	0.6706	0.6968
	Our	0.7654	0.8474	0.8035
Support Vector Machine ($n = 2$) (ML)	Bhat	0.8405	0.7597	0.7961
Support Vector Machine ($n = 2$) (ML)	Our	0.7454	0.6937	0.7094
Decision Tree ($n = 1$) (ML)	Bhat	0.2416	0.5000	0.3258
Decision Tree ($n = 3$) (ML)	Our	0.2157	0.5000	0.3011
Logistic Regression ($n = 2$) (ML)	Bhat	0.8079	0.7620	0.7746
Logistic Regression ($n = 3$) (ML)	Our	0.7046	0.6887	0.6941
One-vs-Rest ($n = 2$) (ML)	Bhat	0.8253	0.6460	0.7234
One-vs-Rest ($n = 1$) (ML)	Our	0.6271	0.6443	0.6339
Naive Bayes ($n = 2$) (ML)	Bhat	0.7801	0.8290	0.8037
Naive Bayes ($n = 2$) (ML)	Our	0.5967	0.9476	0.7293

Table 35: Cross-project performance for detecting architectural issues

Model	Precision	Recall	F1-Score
BOW (frequency)	0.4555	0.4486	0.4463
BOW (normalized)	0.4622	0.4408	0.4319
CNN	0.4332	0.4054	0.3993
CNN (SO)	0.4603	0.4312	0.4073
Doc2Vec	0.4214	0.4222	0.4092
Issue Properties	0.3367	0.3243	0.2979
Ontology Features	0.4070	0.3980	0.3777
RNN	0.5068	0.4967	0.4933
RNN (SO)	0.5241	0.5109	0.5009
TF/IDF	0.4505	0.4247	0.4077
Support Vector Machine ($n = 3$) (ML)	0.4869	0.4934	0.5066
Decision Tree ($n = 4$) (ML)	0.0383	0.2500	0.0443
One-vs-Rest ($n = 1$) (ML)	0.4018	0.4025	0.4084
Naive Bayes ($n = 1$) (ML)	0.4469	0.4049	0.3866

Table 36: Cross-project performance for classifying architectural issues. See table 56 in section A for class-specific metrics.

bugs", "race condition", "bug structuredcodeblock", and "bootstrap error". Additionally, there are many class name markers present among the keywords in table 39.

Similarly, table 40 also contains a number of words expressing software failure. Examples here include "name cause error", "state exception", and "case problem log".

15.2 Classification Keywords

In this section, we will describe the keywords obtained for the classification process. For every class, we have a table constructed using frequencies and a table constructed using probabilities. Respectively, we have: tables 42 and 43 for the executive class, tables 44 and 45 for the property class, tables 46 and 47 for the existence class, and tables 48 and 49 for the non-architectural class. Additionally, table 50 contains a list of common keywords, which were present for at least two classes.

We start with table 42. There are two types of keywords which seem quite logical for existence issues. The first type is keywords containing the word "upgrade" (e.g. "upgrade", "upgrade guaba", "need upgrade"). This seems logical because updating software or dependencies may have architectural implications. The other type of keyword, is keywords containing the name of other software or technologies. Examples include "html", "http", "parquet", "oauth", "ssl", "rfc", "jna", and "avro arpc".

In table 43, such phrases containing software names or the word "upgrade" are almost non-existent (one note-worth example being the phrase "new available"). In stead, there are other phases related to the use of external software. For instance, there are many phrases containing the marker word "versionnumber" (e.g. "versionnumber used example", "available versionnumber"). More common phrases are phrases implying the use of some other software for some purpose, or the intend to support some functionality. Examples of this include "able support", "authentication via", "nfs interface support", and "driver need functionality".

We now move to keywords relevant for coming to a property classification. Table 44 does not seem to contain many words related to property issue, aside from a view. One example is the phrase "improve performance". There is also "connection timeout", which might be related to performance or quality of service. When also looking at table 45, we can see more patterns. Key phrases containing the word "cache" are really common, possibly because caches are tightly related to performance. We also see that in both tables, but especially in 45, we can find words related to encryption and security (e.g. "encrypt", "encryption authentication", "authentication rpc layer".

Additionally, in table 45, we see phrases relating to quality attirbutes. Examples of such phrases include "quickly performance per", "secure environment design", "pluggable security feature", "efficiency namenode memory", and "extremely quickly performance".

One observation we cannot explain, is that table 45 also contains many words referring to memory. For instance, we have the keywords "large memory", "row memory", "memory jvm", "reasonably large memory", and "memory requirement support". This phrase occurred often enough that we noticed it during our analysis, but we could not clearly identify why the machine learning model thought this phrase was important.

We now move to the keywords the classifier deemed indicative of existence decisions, in tables 46 and 47. We had a bit more difficulty finding keywords obvious keywords pointing the existence issues, although we did find some interesting patterns. First of all, in table 46, we noticed a number of words possibly related to the removal of code or functionality: "remove dependency", "disable", and "deleted". However, we do not encounter such words in table 47.

We do encounter the word "cluster" many times in table 47 (e.g. "sub cluster federation", "node large cluster", and "user cluster"). One other thing we noticed, is that there are many words which are based of the word "fail": "failure case functional", "failed specific", "fails run", "failure case", and "failed run". It is not clear why these words are apparently so important for coming to an existence classification – Especially since we established based on tables 39 and 40 that failure is very strongly related to non-architectural issues. One possible explanation is that fixing issues is somehow related to existence decisions; there is also a number of keywords associated with fixing faults in software: "repair many", "try address", "fix find", and 'repair many small".

We now arrive at the final class: Non-Architectural issues. The relevant tables are 48 and 49. The most important we once again make is that markers for formatting output (e.g. "unformattedtraceback", and words such as "exception", 'unable open", "ticket error", and "race condition guarantee" are important keywords. We want to contrast this with the existence class. For the existence class, words derived from "fail" or "failure" were important, whereas these words for non-architectural are closer to actual exceptions, or are actually exceptions.

15.3 Common Keywords

Finally, we look at the common keywords in tables 41 and 50. We felt that most of these keywords are somewhat generic, in the sense that they can easily be seen to be related to coding or version control systems in general. Additionally, almost all common phrases consist of a single word – this is the most likely to happen, because in some sense there are less possible phrases of length 1 than of length > 1. There also seems to be considerable overlap between the words in tables 41 and 50.

23	order	similar would nice
config	min	share
22	memtable	service user
would nice	match	separate thread
17	manager	sense would
discussion	linux	run command
13	info	remote
would allow	file system	reduce common
list	erasure	reader
datanodes	add simpleclassname	proxy user
12	**5**	protocol rpc
version versionnumber	view	pluggable interface
11	useful feature	please consider
scheduler	tool	partition key
logic	tajo client	ohc versionnumber
10	storage format	nodetool repair
xml	sense	need support
web	r	n
thrift	progress	multi
page	primary	memory namenode
introduce	ozone	mapreduce api
9	mount	manager node
unknown	make possible	management
ui	make easy	make sense
make sure	machine	little
line	lookup	like noformatblock
library	long running	jvm dtest
history	leak	jira introduce
format	join	jdbc jar
8	group	information simpleclassname
web ui	give	index would
wait	future	increase
row	attached	hdfs project
placement	argument	hdfs client
place	apis yarn	hadoop see
nice	analysis	hadoop hdfs project
directly	alternative	family
already	**4**	expose cql
ability	would make sense	encryption
7	versionnumber would	edits
step	versionnumber fixes	driver writing
scheduling	variable	directory path
retry	username	dependency versionnumber
related	use netty	datanodes directly
recovery	timeline data	current version
open	ticket make	cpu load
http	system hdfs	consistency
efficient	support user	connect
edit	successfully	complexity
6	streaming repair	comparators
would great	stop	client project
would good	start simpleclassname	bind
transfer	standby	attached patch
short	ssl	advantage
serialization	simpleclassname table	across cluster
rack	simpleclassname class	

Table 37: Keywords with frequency ≥ 4 for the Architectural class (Detection).

0.78 - 0.77	hadoop would give	tajo hive
log thread	http connection	production analysis important
0.77 - 0.76	application capacity scheduler	version version
consistency latency	storage add	implementation username
span data compaction	partition table	mapreduce job execution
0.76 - 0.75	c library	info display
method container	**0.72 - 0.71**	java get problem
src contrib fuse	library non issue	support log
make sure data	want buffer memory	add dependency
inlinecodesample cache instance	make sure node	able send
one page data	cpu disk	cpu usage
0.75 - 0.74	protocol make	priority priority available
handle rm	low level data	cache thread
support cql	process block compaction	manual testing junit
return service	least job significantly	sec fast
run classname test	weblink would require	sub acl priority
compaction pointer	run stream data	intensive job cpu
web interface	cell	hadoop jira significant
rate thread	ui filepath	run oom heap
0.74 - 0.73	row index	underlying container
src githublink ui	data log latency	data column
ubuntu versionnumber problem	deal token	quorum read
intel r	resource take	default service
support c	rw mount	java memory something
authentication data	dfs per node	compaction block
see point technology	token many security	though progress
weblink building linux	per disk machine	hard run
filepath datacenter	sends r	x token
current rpc allow	native function	bit place due
ip per dc	http current data	web service
fuse rw	support latency	query select
src data	usage classname row	hadoop_home ant
0.73 - 0.72	storage option	consistency behavior
management status	sec time	hdfs data directory
version security	checksum	behavior client
level design high	driver understand	would cause
close transaction log	turn container	max request useful
state classname tree	across k	trunk performance performance
plugin technology	across thread	home able
issue response	performance c	table min
version library	update slf	see cassandra allow
buffer write mapreduce	network however take	consistency active
detail related data	module jdbc	object method path
available maven	**0.71 - 0.70**	per queue cluster
protocol via http	count column	server expected column
make make	node make	environment variable build
scheduler running	enable authentication native	versionnumber info
make cql	client side encryption	state make easy
david r	scheduler version	separate maven
cluster capacity current	compaction result	fix analysis
reduce make	jdbc driver	bad latency
fix compaction compression	client interface use	version work
example store data	capacity throughput big	r time coordinator
every message	cassandra many good	
libhdfs library logging	handle block	

Table 38: Keywords with probability ≥ 0.7 for the Architectural class (Detection).

24	org	private
formattedtraceback	npe	principal
8	metadata simpleclassname	preset image
pom	mapreduce clover dependency	must
7	mapreduce client core	length configurable
unformattedtraceback	mapreduce client	ivy
formattedloggingoutput	make compile	internal
6	loading auth	implementation simpleclassname
compile	like simpleclassname	handler formattedloggingoutput
5	jvm dtests	filepath hadoop
tombstone	general weblink	fail disk
findbugs	filepath unformattedtraceback	ed
enum	encoding cache	descriptor
auth	client module	db cli
4	cache formattedtraceback	column names cql
writes formattedtraceback	bug structuredcodeblock	colon create
validity simplemethodorvariablename	also mapreduce	code simpleclassname
title tajo	**3**	code refactor
system environment	would fine	clover integration broken
src java	without classname	clover dependency
simpleclassname v	within serializable	client core
simpleclassname package	stage enum	client code
shutdown formattedtraceback	specify version	classpath
serializable	simplemethodorvariablename parent	checking
secondary column	scan	case
recycling design	review	buffer weblink
rc	return udt	broken trunk
race condition	remove limitation	bootstrap error
processing	recently	attach
prefix methodorvariablename	rather	add filepath
parent	public	
output formattedloggingoutput	proper package	

Table 39: Keywords with frequency ≥ 3 for the Non-Architectural class (Detection).

0.79 - 0.78	log see	prefix mode
src test unchanged	kill client	linux hadoop may
0.77 - 0.76	core j	like stack
name cause error	way metric	control task
mapred mvn	capacity queue b	src test
handle service	zk running heap	rest api result
0.76 - 0.75	hive dependency	basis make
src contrib thriftfs	sec per	cql client require
0.75 - 0.74	page per kb	methodorvariablename classname proposed
g data	methodorvariablename queue b	mapred package
run heap ram	refine client throw	client v
filepath auto	standard interface like	cluster wide gb
clean maven	response response correct	version asm
api v	future jira	storage currently
src core contain	per application basis	owner request
0.74 - 0.73	simpleclassname task	table name cause
handle event	node heartbeat many	cassandra jvm
python cql	state exception	simpleclassname write data
artifacts inlinecodesample build	see many like	size state store
io ipc log	hadoop may involve	org apache cassandra
org apache	**0.70 - 0.69**	port gridmix data
usergroupinformation method log	cql see	arbitrary number principal
0.73 - 0.72	src split	distributed cache allow
namespace usage	directory link	application run
option expose	static void	b return
lib slf	namenode port gridmix	gb per node
statement structuredcodeblock result	core j affect	manner code jira
versionnumber slf	driver visible issue	see weblink explanation
task cause	timeline service	store technology
0.72 - 0.71	driver schema	methodorvariablename true
replica return	methodorvariablename	tombstone index
	methodorvariablename throw	
data set gb	stress versionnumber	generator script
common lib	result response	protocol buffer
client side dfs	remove ant	lib directory
case reading data	project ivy	response execute contains
methodorvariablename event complete	case problem log	response response
mode case insensitive	mr code	g heap
0.71 - 0.70	queue b	sec per application
work yarn	forward backward add	affect current build
raid jvm	**0.69 - 0.675**	
_ timed	weblink cassandra version	

Table 40: Keywords with probability ≥ 0.675 for the Non-Architectural class (Detection).

148	class	long	**19**
would	**41**	level	userprofilelink
135	user	following	running
methodorvariablename	problem	cache	run
105	**40**	**26**	resource
simpleclassname	currently	x	mechanism
100	**38**	system	local
support	protocol	separate	large
98	patch	main	githublink
add	cluster	good	failure
96	**37**	every	fail
versionnumber	provide	**25**	bit
95	**35**	option	**18**
weblink	yarn	**24**	store
90	per	size	solution
hadoop	default	repair	property
87	core	project	path
data	**34**	native	one
76	propose	**23**	handle
client	**33**	write	call
62	thread	type	branch
version	make	structuredcodeblock	**17**
61	**32**	range	use
simplemethodorvariablename	mapreduce	method	technology
54	error	get	table
code	dependency	approach	sstables
52	configuration	**22**	side
time	**31**	read	query
cassandra	upgrade	new	inlinecodesample
50	**30**	move	directory
node	way	interface	**16**
column	set	hdfs	technology names
49	server	disk	stream
jira	performance	**21**	last
implementation	package	update	hard
48	java	task	cause
issue	implement	something	capacity
create	**29**	return	byte
47	value	let us	**15**
multiple	storage	compaction	without
46	rpc	change	take
see	need	c	response
block	many	**20**	request
45	let	trunk	namenode
filepath	file	tajo	message
allow	common	state	maven
44	**28**	setting	latency
classname	useful	network	current
43	start	id	configurable
number	like	exception	build
name	index	container	access
42	**27**	connection	
remove	single	api	

Table 41: Keywords with frequency ≥ 15 which occur for both the Architectural and Non-Architectural class (Detection).

54	http	java driver
upgrade	dev weblink	fit hadoop
26	average	context
thrift	apache license	c use
21	**7**	broken design
c	upgrade netty	accept oauth
13	upgrade guava	**5**
html	unit html	version versionnumber
11	since parquet	upgrade guava versionnumber
upgrade thrift	netty methodorvariablename	support thrift
rfc	move maven	schemas
10	jna	library
lzop	dependency filepath	java driver versionnumber
9	data_join package	implementation hdfs
parquet	classpath	functionality
license	boundary version	filepath file
8	accept	driver
tokens	**6**	avro rpc
ssl	versionnumber structuredcodeblock	
oauth	trunk branch	
mr	need upgrade	

Table 42: Keywords with frequency ≥ 5 for the Executive class (Classification).

0.70 - 0.69 secure io support	side read request package package compile scope dependencies partition secondary release weblink dist	versionnumber env however jackson version use netty twitter implement connect server certificate
0.69 - 0.68 container heartbeat request		
0.64 - 0.63 chain client interface use upload design list access data	**0.52 - 0.51** issue problem current maven pom	**0.46 - 0.45** probably open new bulk loading interface authentication via state could make functionality java order useful handling maven nfs interface support across multiple connection eg hadoop client simplemethodorvariablename nfs following analysis
0.62 - 0.61 inlinecodesample project client server protocol	**0.51 - 0.50** able support simpleclassname thread per	
0.60 - 0.59 classname tajo index etc	**0.50 - 0.49** encryption sse server better checksum driver need functionality list partition key currently various configured different jdbc make sql	
0.59 - 0.58 permit large cluster		
0.58 - 0.57 path dependent tie client side		**0.45 - 0.44** everyone operation would data structure maven pom use simpleclassname hadoop driver java protocol track nfs versionnumber currently tajo layer data writing
0.57 - 0.56 classname certificate key counter versionnumber branch config dependency	**0.49 - 0.48** necessary implement project compile scope classname tasks jar hadoop fails inlinecodesample date time available versionnumber protocol particularly http	
0.56 - 0.55 switch rpc use		
0.55 - 0.54 new available versionnumber upgrade	**0.48 - 0.47** use open work avoid users	
0.54 - 0.53 like use versionnumber used example	**0.47 - 0.46** parquet weblink support methodorvariablename hadoop	
0.53 - 0.52 structure structuredcodeblock node		

Table 43: Keywords with probability ≥ 0.44 for the Executive class (Classification).

15 datanodes	go cassandra comment	length key cache
14 comment	**7** us encrypt improve performance connection timeout	**5** xml versionnumber good thread safe take static session methodorvariablename call encrypt coupling buffer cache
10 old		
9 tmp	**6** yarn nm would make updates solve problem page cache logs less	
8 usage nm native memory machine java implementation		

Table 44: Keywords with frequency ≥ 5 for the Property class (Classification).

0.76 - 0.75	one configure	service would nice
key management	instantiation issue track	manual refresh command
resource especially resource	artifacts many need	user service
0.75 - 0.74	**0.57 - 0.56**	job user
token authentication	process high	yarn order make
0.73 - 0.72	**0.56 - 0.55**	first time
token access	variable class	per second
end network protocol	create multi	memory jvm
0.70 - 0.69	multiple parallel	take time
quickly performance per	cache hdfs	**0.48 - 0.47**
cache improve	overall system performance	improve namenode
affect network bandwidth	**0.55 - 0.54**	java fix involves
secure environment design	behind processing rest	formattedloggingoutput similar
join machine part	implementation thread safe	directory aware
configuration specify	new device	high hot much
0.69 - 0.68	order make interface	static instance
encryption authentication	sstables lot classname	**0.47 - 0.46**
index lookup cache	**0.54 - 0.53**	large image
0.67 - 0.66	table time load	make environment
authentication rpc layer	filepath implement	native coupling way
cache load	wrong use	could find
0.66 - 0.65	config file	cache easily
cache different	configuration information	cassandra cli etc
0.65 - 0.64	java code literal	versionnumber faster see
multiple per	**0.53 - 0.52**	rack may
specific client side	like hadoop	reporting api external
0.64 - 0.63	like query cache	cache wide
java nice way	**0.52 - 0.51**	make much useful
0.63 - 0.62	java gc	set target
heap query	large memory	large cache
problem scenario list	hdfs hadoop yarn	log subdirectory
team performance issue	like implement	edit human
calculation heartbeat allocation	**0.51 - 0.50**	situation large
pre request model	vm allow us	reasonably large memory
0.62 - 0.61	row memory	notify big
pluggable security feature	order make	handling integrate
class per	set environment	end network
cache size	authentication cluster provide	**0.46 - 0.45**
0.61 - 0.60	instead multiple per	extremely quickly performance
datanodes fill storage	issue flush might	memory requirement support
resource utilization	**0.50 - 0.49**	patch hdfs would
ipc support following	versionnumber nodes rf	take long
0.60 - 0.59	take run	time scenario
new level	optimal cpu	class thread safe
load scale	slow many	read write
compaction compression	kerberos ticket	restart service
following time per	like understand	goal keep much
spread across entire	job cause	classname weblink
0.59 - 0.58	efficiency namenode memory	every key
like make	**0.49 - 0.48**	
even contention per	problem many scheduler	
path convenient	cache single	
try provide	every release	
0.58 - 0.57	pluggable interface run	
storage layer	generation	

Table 45: Keywords with probability ≥ 0.45 for the Property class (Classification).

15 range	handle client bin tsql	similar ring
12 column	**6** state store	remove dependency reduce common
11 dc	purpose per file	disable deleted
8 multiple logging mapreduce api	methodorvariablename run java source interval	cluster hadoop active rm
7 really useful	**5** submit	

Table 46: Keywords with frequency ≥ 5 for the Existence class (Classification).

0.71 - 0.70	methodorvariablename make	**0.47 - 0.46**
mismatch multiple range	range name	failure case
cpu network disk	large scale	local configuration
0.69 - 0.68	**0.53 - 0.52**	specify hdfs
shared cluster feature	server interpret json	example replace
node large cluster	compaction small	specific application version
0.68 - 0.67	repair many	nm launch container
sub cluster federation	script bin tsql	try address
blacklist mechanism track	**0.52 - 0.51**	final status
container cpu	like hdfs	fix find
0.65 - 0.64	restart yarn	delete option
join order join	parent user	**0.46 - 0.45**
0.64 - 0.63	like add	slow startup delay
completed dns pipeline	**0.51 - 0.50**	run problem
0.63 - 0.62	service request feature	history storage
admin server update	hdfs r	failed rm
0.61 - 0.60	could choose	generic ui display
used read	file add	tombstone list
conf file	useful unit use	filepath patch columns
0.60 - 0.59	**0.50 - 0.49**	versionnumber used datetime
could specify	objective find best	job already
dedicate connection likely	configuration file	docker support
0.59 - 0.58	user cluster	appended data would
store r	root queue	repair many small
writen recovery read	state store r	reproduce case
path cql	node large	encourage implement
0.57 - 0.56	mechanism yarn address	**0.45 - 0.44**
range name slice	**0.49 - 0.48**	multiple one
app specific	failed specific	protocol push know
versionnumber add help	request r	per host
0.56 - 0.55	end multiple	unavailable open
classname related	store write	hdfs cluster readable
cluster coordination engine	multiple excess	interface join enumeration
0.55 - 0.54	busy cluster	recover running
easier mock implementation	r r	side perform following
partition match	properly binary	
0.54 - 0.53	**0.48 - 0.47**	
failure case functional	fails run	
storage handling	could use	
since client already	currently mount	
via checkpointing jira	store internal	
fair scheduler	operation bulk	

Table 47: Keywords with probability ≥ 0.44 for the Existence class (Classification).

23 unformattedtraceback	around argument	become
12 formattedloggingoutput compile	**7** title tajo reproduce structuredcodeblock	**5** startup set default see title
11 type	methodorvariablename formattedtraceback make private	reason separate query history
10 int history	hadoop built **6** world	patch githublink owner inlinecodesample inlinecodesample
8 per connection make compile instance	trace title supported formattedloggingoutput filepath unformattedtraceback	count zero builder around

Table 48: Keywords with frequency ≥ 5 for the Non-Architectural class (Classification).

0.76 - 0.75 use network interface	keyspace one	would greatly kerberos simultaneously let
0.70 - 0.69 responsible kerberos ticket	**0.53 - 0.52** debug logging	empty group ticket error
0.67 - 0.66 node partial implementation	**0.52 - 0.51** different eg type per	log tailing **0.46 - 0.45** fails following
0.66 - 0.65 key search	**0.51 - 0.50** hdfs mapred yarn	columns case message_id
0.64 - 0.63 case sensitive protocol	**0.50 - 0.49** due issue	lot data even **0.45 - 0.44**
0.62 - 0.61 race condition guarantee	exception thrown methodorvariablename specify support	unformattedtraceback running describe userprofilelink somewhere quite
0.60 - 0.59 path retrieve	responses unformattedtraceback ticket error handling	configured use even client
0.58 - 0.57 unable open enumeration server current	server principal hdfs client fails following false make	network interface ip version set
0.57 - 0.56 exception unformattedloggingoutput methodorvariablename javadoc may require	**0.49 - 0.48** latency long unit test	
0.56 - 0.55 operator ticket focus lead potential memory key message_id	readrepairchance versionnumber like pig artifact patch warning startup user	
0.55 - 0.54 single network kerberos principal need provide	**0.48 - 0.47** object created storage pgsql version patch committed	
0.54 - 0.53 already local	**0.47 - 0.46** container along	

Table 49: Keywords with probability ≥ 0.44 for the Non-Architectural class (Classification).

123	**40**	multiple	release
cassandra	new	job	port
119	**39**	improve	page
weblink	like	**25**	option
109	**37**	use	mapreduce
would	filepath	due	large
105	**36**	**24**	group
methodorvariablename	support	native	githublink
101	remove	maven	**17**
versionnumber	example	gc	trunk
98	**35**	**23**	ticket
time	connection	provide	though
83	**34**	disk	primary
data	user	currently	object
80	set	client	map
cluster	memory	**22**	look
77	**33**	tajo	introduce
issue	see	system	discussion
hadoop	**32**	rm	common
70	useful	return	commit
hdfs	key	report	cli
63	heap	netty	across
make	dependency	method	**16**
java	configuration	**21**	x
add	classname	work	track
62	**31**	level	side
cache	list	let	security
58	interface	**20**	scheduler
performance	code	value	non
57	**30**	timeout	native protocol
version	rpc	technology	high
54	**29**	structuredcodeblock	format
jira	package	run	fix
implementation	following	resource	build
50	**28**	even	authentication
state	since	apache	**15**
api	query	**19**	without
49	problem	yarn	update
default	number	technology names	service
47	implement	table	rest
per	design	store	queue
file	current	policy	partition
46	**27**	logging	may
node	userprofilelink	branch	formattedtraceback
44	server	**18**	enable
simpleclassname	please	task	call
43	**26**	start	
simplemethodorvariablename	request	specify	
patch	protocol	schema	

Table 50: Keywords with frequency ≥ 15 which occur in at least two of the classes used in classification.

16 Conclusion

This section contains an overview of the conclusions we can make for each research question.

RQ1: How accurate are deep learning approaches to identify and classify architectural issues?

Using deep learning (CNN) we were able to detect architectural issues in our dataset with a 83.32% f-score. This is approximately 9% higher than the best machine learning model (SVM).

For the Bhat dataset, deep learning (RNN SO and issue properties) and machine learning (SVM) achieved similar performance: 88% f-score.

Combining models (BOW frequency + CNN + RNN) yielded no performance benefit for our dataset, but for the Bhat dataset it achieved an f-score of 89.98% (1.7% improvement over RNN SO and SVM).

For the best performance we recommend using the combined model for detecting architectural issues. However, since this is a complex model and takes long to train, we recommend RNN if this is not needed. RNN also performs outstanding on the classification of issues. If an even more lightweight model is required, we recommend using CNN or BOW frequency as these have both good performance for detecting architectural issues and classifying issues. Machine learning lacks performance on our dataset and therefore we do not recommend using them.

RQ2: How would the training data-set of architectural issues impact the generalizability of deep learning approaches to identify and classify architectural issues?

We determined that the performance of TF/IDF translates the best to another dataset. Other well performing models are the other BOW models (frequency and normlaized), both RNN models, and both CNN models.

The performance of the machine learning models did not seem to translate to other datasets effectively. On average, the performance was 18% worse than the best deep learning model (RNN SO). Therefore we recommend using one of the mentioned deep learning models instead.

RQ3: How generalizable are deep learning approaches to identify and classify architectural issues from different projects?

We see the most consistent performance for the RNN model for detecting architectural issues across projects.

The issue properties model performs well on the Bhat dataset, but it lacks performance on our dataset compared to RNN.

A more lightweight alternative is the CNN model. However, this model does lose quite a bit of performance compared to RNN.

The best machine learning model (naive bayes) only performs well on the Bhat dataset, and again lacks performance on our dataset. Hence we recommend using deep learning for the detection task.

For the classification task we see similar performance for deep learning (RNN SO) and machine learning (SVM).

A more lightweight alternative to RNN is BOW frequency. However, it loses almost 5.5% performance compared to RNN. Therefore we recommend using RNN for the classification task if possible.

RQ4: What are the keywords used by deep learning approaches to identify and classify architectural issues?

For detection, keywords expression software failure are the most important pattern we found among the keywords for the Non-Architectural classification. Especially marker words for formatting related to exceptions were commonly part of key phrases. For the Architectural label, we found a wider variety of keywords – quality attributes, addition of components, components names. Additionally, phrases containing the word "would" combined with some positive phrase (e.g. "would nice") were common keywords for architectural issues.

For classification, we have four different classes. For executive issues, common keywords were related to technology names. We also found many keywords containing the word "upgrade" or "versionnumber". We also found some phrases implicitly hinting at the use of external software (e.g. "authentication via")
For Property issues, we found quality attributes and words related to quality attributes, as could be expected. There were only three major quality attributes, though: "consistency", "performance", and "security". A significant number of the other keywords were somehow related to one of these. For instance, "cache" and "latency", related to "performance", are common keywords. For "security", we often found phrases involving "authentication" or "encryption". One thing we cannot explain is that we found many keywords involving the word "memory".
For Existence issues, key phrases containing the word "cluster" were encountered often. This may have something to do that all projects the issues in the dataset were taken from, use or support clusters. Other noteworthy keywords for existence issues are phrases containing "fail" or "failure", and key phrases expressing the intent to fix a fault, such as "repair".
Finally, we looked at keywords for the Non-Architectural class in the detection task. Words connected to software failure were still important keywords. However, there is a clear distinction with the existence class. Whereas the existence class had keywords containing the words "fail" or "failure", keywords for Non-Architectural tended to focus more on exceptions and the formatting markers associated with exceptions.

17 Threats to Validity

In this section, we discuss a number of threats to validity.

- *Inconsistency of Deep Learning Results*: at times, the performance of the deep learning process fluctuated somewhat due to the stochastic nature of deep learning approach. This could lead to some variety in our results. We tried to look for patterns or tendencies in our results. If some results was inconsistent with the patterns we had observed up until that point, we repeated experiments multiple times to check whether our inconsistent result was unfortunately a really "lucky" or "unlucky" run. Additionally,

- *Inexperience with Deep Learning*: before working on this research, the two authors were relatively unfamiliar with deep learning. To attempt to avoid pitfalls, steps were thoroughly researched before they were performed. Additionally, the two authors had guidance from their supervisor and one of their PhD students who was more experience in deep learning (see section 19)

- *Wrong Labels in the Dataset*: one potential flaw in this research, is issues in the dataset being assigned the wrong labels. For instance, as explained previously in section 5, we did not find large agreement with Bhat et al. on the classification on architectural issues. Additionally, as explained in section 4, we re-classified part of our own dataset we used after the primary supervisor found reason for possible wrong classification. This all suggests that identifying and classifying architectural issues is hard for humans. This leads to the possibility that there are issues with the wrong label in the dataset, which can confuse or hurt the performance of the classifiers.

- *Apache-only projects*: using the cross-project validation we verified that testing on a project that does not occur in the training set, deep learning is able to achieve good performance on those projects. However, all the projects are Apache projects and many of them are sub projects from Hadoop. Therefore we cannot guarantee that the detection and classification of architectural issues using deep learning will perform well on non-Apache projects.

- *Java-only projects*: another result of the Apache projects is that all projects use Java as their main programming language. Hence, it is not clear if the performance will be the same on non-Jave projects.

- *All issues from Apache Jira*: all issues were obtained from the Jira issue tracker. Especially for the issue properties model, we cannot guarantee that it works for issues from other issue trackers. For the text models the performance should translate to other issue trackers, but we cannot guarantee this.

18 Future Work

Future work could include overcoming the limitations and threats to validity of this research. First of all, we can extend the dataset using issues from non-Apache projects, non-Java projects and issues from other issue trackers than Jira.

Furthermore we could apply the BERT model on the detection and classification task. This model is the current state of the art, but requires a lot of computational power. Therefore it would be good to find out if this computationally heavy model is worth using for these tasks.

Similarly to our keyword extraction analysis, we could do an issue property analysis. With this analysis we could identify which values of the issue properties determine whether an issue is classified as architectural or non-architectural.

19 Acknowledgements

- We would like to thank our supervisor Mohamed Soliman for coaching us during the time we worked on this project, and for providing feedback, ideas, and direction.

- We would like to thank Yikun Li for providing tips on improving the stability of the training in the deep learning process, checking the reasonableness of our models, and for providing information and a basic script for keyword extraction.

A Class-specific Metrics for Classification Tests

References

[1] 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), pp. 138–154. ISSN: 16113349. DOI: 10.1007/978-3-319-65831-5{_}10/COVER/. URL: https://link.springer.com/chapter/10.1007/978-3-319-65831-5_10.

[2] Rafael Capilla et al. "10 years of software architecture knowledge management: Practice and future". In: *Journal of Systems and Software* 116 (June 2016), pp. 191–205. ISSN: 0164-1212. DOI: 10.1016/J.JSS.2015.08.054.

[3] Said Faroghi. "Mining architectural Knowledge in issue tracking systems". Bachelor's Thesis. Rijksuniversiteit Groningen, Feb. 2022.

[4] Philippe Kruchten. "An ontology of architectural design decisions in software intensive systems". In: *2nd Groningen workshop on software variability*. 2004, pp. 54–61.

[5] 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), pp. 43–58. ISSN: 03029743. DOI: 10.1007/11921998{_}8/COVER/. URL: https://link.springer.com/chapter/10.1007/11921998_8.

[6] Quoc Le and Tomas Mikolov. "Distributed Representations of Sentences and Documents". In: *31st International Conference on Machine Learning, ICML 2014* 4 (May 2014), pp. 2931–2939. DOI: 10.48550/arxiv.1405.4053. URL: https://arxiv.org/abs/1405.4053v2.

[7] Shervin Minaee et al. "Deep Learning-Based Text Classification: A Comprehensive Review". In: *ACM Computing Surveys* 54.3 (June 2021). ISSN: 15577341. DOI: 10.1145/3439726.

[8] Juan Ramos. "Using tf-idf to determine word relevance in document queries". In: *Proceedings of the first instructional conference on machine learning* (2003), pp. 29–48. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.1424&rep=rep1&type=pdf.

[9] 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 (Mar. 2019), p. 1. ISSN: 15577392. DOI: 10.1145/3324916. URL: https://ink.library.smu.edu.sg/sis_research/4476.

[10] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural Networks* 61 (Jan. 2015), pp. 85–117. ISSN: 0893-6080. DOI: 10.1016/J.NEUNET.2014.09.003.

[11] Georgios Sigletos et al. "Combining Information Extraction Systems Using Voting and Stacked Generalization". In: *Journal of Machine Learning Research* 6.11 (2005).

[12] 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), pp. 117–133. ISSN: 16113349. DOI: 10.1007/978-3-030-86044-8{_}8.

[13] Mohamed Soliman, Matthias Galster, and Matthias Riebisch. "Developing an Ontology for Architecture Knowledge from Developer Communities". In: *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017* (May 2017), pp. 89–92. DOI: 10.1109/ICSA.2017.31.

		Executive			Existence			Property			Non-Architectural			Average			
Model	**Sub-Type**	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	**Imp. over Random**
BOW (frequency)	Lemmatization	0.4382	0.4676	0.4498	0.6702	0.6090	0.6363	0.5056	0.5471	0.5217	0.4707	0.4331	0.4452	0.5212	0.5142	0.5132	2.09
	No Transform	0.4530	0.3760	0.4051	0.6357	0.5766	0.6004	0.4950	0.5434	0.5111	0.4966	0.5705	0.5277	0.5201	0.5166	0.5111	2.08
	Stemming	0.4482	0.5185	0.4779	0.6844	0.5996	0.6347	0.5154	0.4810	0.4944	0.4563	0.4636	0.4576	0.5261	0.5157	0.5162	2.10
BOW (normalized)	Lemmatization	0.4373	0.4881	0.4594	0.6708	0.6584	0.6631	0.4988	0.6508	0.5640	0.5279	0.2939	0.3694	0.5337	0.5228	0.5140	2.09
	No Transform	0.4331	0.5018	0.4616	0.6581	0.5908	0.6185	0.4681	0.6257	0.5340	0.5156	0.2948	0.3692	0.5187	0.5033	0.4958	2.01
	Stemming	0.4146	0.4448	0.4252	0.6107	0.6239	0.6154	0.4980	0.5952	0.5406	0.5345	0.3662	0.4315	0.5145	0.5075	0.5032	2.04
CNN	Lemmatization	0.4088	0.3883	0.3968	0.5109	0.5649	0.5360	0.4868	0.5023	0.4909	0.5296	0.4840	0.5027	0.4840	0.4849	0.4816	1.96
	No Transform	0.4064	0.3803	0.3895	0.5512	0.5616	0.5541	0.4748	0.4897	0.4787	0.5052	0.5138	0.5061	0.4844	0.4863	0.4821	1.96
	Stemming	0.3755	0.3555	0.3596	0.5111	0.5868	0.5440	0.4861	0.4614	0.4647	0.4923	0.4628	0.4745	0.4663	0.4666	0.4607	1.87
CNN (SO)	Lemmatization	0.4029	0.4330	0.4006	0.5250	0.5988	0.5514	0.4919	0.4231	0.4416	0.5456	0.4645	0.4883	0.4913	0.4798	0.4705	1.91
	No Transform	0.4545	0.4540	0.4415	0.5323	0.6408	0.5778	0.5067	0.3910	0.4290	0.5126	0.4946	0.4929	0.5015	0.4951	0.4853	1.97
	Stemming	0.3656	0.2910	0.2973	0.4794	0.5623	0.5108	0.4595	0.4027	0.4106	0.3881	0.4323	0.3954	0.4232	0.4220	0.4035	1.64
Doc2Vec	Lemmatization	0.4505	0.4130	0.4291	0.5429	0.5480	0.5444	0.5234	0.4767	0.4958	0.4560	0.5283	0.4875	0.4932	0.4915	0.4892	1.99
	No Transform	0.4238	0.3781	0.3952	0.5295	0.5538	0.5386	0.4950	0.4561	0.4626	0.4629	0.5095	0.4803	0.4778	0.4744	0.4692	1.91
	Stemming	0.4234	0.3798	0.3962	0.5481	0.5572	0.5463	0.4601	0.4290	0.4398	0.4666	0.5279	0.4905	0.4745	0.4735	0.4682	1.90
RNN	Lemmatization	0.4890	0.4730	0.4781	0.6462	0.7130	0.6768	0.5441	0.5194	0.5287	0.5934	0.5717	0.5793	0.5682	0.5693	0.5657	2.30
	No Transform	0.4967	0.4939	0.4887	0.6457	0.6678	0.6507	0.5673	0.5484	0.5531	0.5992	0.5812	0.5876	0.5772	0.5728	0.5700	2.32
	Stemming	0.4550	0.4417	0.4463	0.6797	0.6838	0.6787	0.5843	0.5740	0.5774	0.5695	0.5819	0.5730	0.5721	0.5703	0.5688	2.31
RNN (SO)	Lemmatization	0.5454	0.4855	0.5099	0.6279	0.6417	0.6326	0.6020	0.6254	0.6108	0.5448	0.5469	0.5439	0.5800	0.5749	0.5743	2.33
	No Transform	0.4658	0.4485	0.4552	0.6201	0.6158	0.6138	0.5929	0.5986	0.5939	0.4888	0.4842	0.4836	0.5419	0.5368	0.5366	2.18
	Stemming	0.4265	0.3958	0.4084	0.6120	0.6247	0.6162	0.5106	0.5130	0.5089	0.4857	0.5063	0.4901	0.5087	0.5100	0.5059	2.06
TF/IDF	Lemmatization	0.4429	0.5095	0.4712	0.6032	0.6888	0.6401	0.5018	0.6334	0.5570	0.5546	0.2405	0.3332	0.5256	0.5180	0.5004	2.03
	No Transform	0.4467	0.5102	0.4747	0.5987	0.6801	0.6335	0.4862	0.6257	0.5439	0.5795	0.2316	0.3279	0.5278	0.5119	0.4950	2.01
	Stemming	0.4139	0.5082	0.4525	0.6341	0.6719	0.6501	0.4714	0.5573	0.5065	0.5403	0.2867	0.3686	0.5149	0.5060	0.4944	2.01
Random		0.3200	0.2812	0.2823	0.2299	0.2625	0.2282	0.2677	0.2529	0.2543	0.2245	0.2207	0.2196	0.2605	0.2543	0.2461	

Table 51: Class-specific precision, recall, and F1-score metrics for the test we did with stemming and lemmatization, for the classification task.

		Executive			Existence			Property			Non-Architectural			Average			
Model	**Sub-Type**	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	**Imp. over Random**
BOW (frequency)	No Ontology, Formatting Markers	0.4386	0.4846	0.4571	0.7067	0.6337	0.6651	0.5262	0.5666	0.5419	0.4839	0.4337	0.4525	0.5389	0.5297	0.5292	2.15
	No Ontology, Keep Formatting	0.4311	0.4434	0.4323	0.6951	0.6297	0.6567	0.5098	0.5790	0.5374	0.5081	0.4589	0.4751	0.5360	0.5278	0.5254	2.13
	Ontology w/ Lexical Triggers, Formatting Markers	0.4423	0.4695	0.4495	0.6057	0.6264	0.6106	0.5050	0.5503	0.5236	0.5219	0.4138	0.4560	0.5187	0.5150	0.5099	2.07
	Ontology w/ Lexical Triggers, Keep Formatting	0.4462	0.4521	0.4447	0.5942	0.6398	0.6117	0.4852	0.5748	0.5208	0.5230	0.3698	0.4278	0.5121	0.5091	0.5013	2.04
	Ontology, Formatting Markers	0.4181	0.4891	0.4478	0.6367	0.6304	0.6303	0.5325	0.5833	0.5526	0.5406	0.3924	0.4461	0.5320	0.5238	0.5192	2.11
	Ontology, Keep Formatting	0.3878	0.4425	0.4106	0.6060	0.6252	0.6123	0.5327	0.5404	0.5304	0.5129	0.3994	0.4399	0.5098	0.5019	0.4983	2.02
BOW (normalized)	No Ontology, Formatting Markers	0.4373	0.4881	0.4594	0.6708	0.6584	0.6631	0.4988	0.6508	0.5640	0.5279	0.2939	0.3694	0.5337	0.5228	0.5140	2.09
	No Ontology, Keep Formatting	0.4399	0.5151	0.4721	0.6639	0.6153	0.6352	0.4789	0.6158	0.5374	0.5156	0.2990	0.3744	0.5246	0.5113	0.5048	2.05
	Ontology w/ Lexical Triggers, Formatting Markers	0.4098	0.4646	0.4308	0.6117	0.5796	0.5876	0.4617	0.5871	0.5146	0.5177	0.3196	0.3888	0.5003	0.4878	0.4804	1.95
	Ontology w/ Lexical Triggers, Keep Formatting	0.3823	0.4356	0.4012	0.5815	0.5704	0.5703	0.4658	0.5828	0.5153	0.5170	0.3229	0.3887	0.4866	0.4779	0.4689	1.91
	Ontology, Formatting Markers	0.4387	0.4768	0.4532	0.6118	0.6002	0.6032	0.4967	0.6167	0.5490	0.5305	0.3456	0.4145	0.5194	0.5098	0.5050	2.05
	Ontology, Keep Formatting	0.4164	0.4556	0.4310	0.6091	0.5916	0.5967	0.4723	0.6207	0.5352	0.5055	0.3077	0.3773	0.5008	0.4939	0.4850	1.97
CNN	No Ontology, Formatting Markers	0.4088	0.3883	0.3968	0.5109	0.5649	0.5360	0.4868	0.5023	0.4909	0.5296	0.4840	0.5027	0.4840	0.4849	0.4816	1.96
	No Ontology, Keep Formatting	0.3995	0.3659	0.3776	0.5746	0.6365	0.6028	0.4544	0.4857	0.4664	0.5104	0.4590	0.4775	0.4847	0.4868	0.4811	1.95
	Ontology w/ Lexical Triggers, Formatting Markers	0.3814	0.3416	0.3577	0.5314	0.5612	0.5428	0.4844	0.5020	0.4899	0.5150	0.5013	0.5035	0.4780	0.4765	0.4735	1.92
	Ontology w/ Lexical Triggers, Keep Formatting	0.4023	0.3838	0.3902	0.5025	0.5271	0.5104	0.4522	0.4685	0.4534	0.5030	0.4943	0.4944	0.4650	0.4684	0.4621	1.88
	Ontology, Formatting Markers	0.4105	0.3762	0.3893	0.5259	0.5494	0.5353	0.4783	0.5109	0.4875	0.5095	0.4916	0.4976	0.4811	0.4820	0.4774	1.94
	Ontology, Keep Formatting	0.4043	0.4031	0.4012	0.5425	0.6016	0.5684	0.4769	0.4770	0.4692	0.4851	0.4376	0.4565	0.4772	0.4798	0.4738	1.93
CNN (SO)	No Ontology, Formatting Markers	0.4780	0.3790	0.4103	0.5380	0.6497	0.5828	0.5375	0.4928	0.4838	0.5079	0.5092	0.4961	0.5154	0.5077	0.4933	2.00
	No Ontology, Keep Formatting	0.4566	0.3288	0.3670	0.5286	0.5901	0.5482	0.4900	0.5739	0.5144	0.5313	0.4598	0.4799	0.5016	0.4882	0.4774	1.94
	Ontology w/ Lexical Triggers, Formatting Markers	0.3741	0.4366	0.4012	0.5328	0.5661	0.5432	0.5319	0.4552	0.4708	0.4819	0.4090	0.4282	0.4802	0.4667	0.4609	1.87
	Ontology w/ Lexical Triggers, Keep Formatting	0.3691	0.4212	0.3810	0.4937	0.4803	0.4769	0.5268	0.4405	0.4585	0.4895	0.4643	0.4625	0.4698	0.4516	0.4447	1.81
	Ontology, Formatting Markers	0.4227	0.3454	0.3648	0.5170	0.5951	0.5483	0.5316	0.4879	0.4846	0.4707	0.4730	0.4667	0.4855	0.4753	0.4661	1.89
	Ontology, Keep Formatting	0.3785	0.4403	0.3938	0.5433	0.5830	0.5570	0.4765	0.4135	0.4321	0.5645	0.4681	0.4847	0.4907	0.4762	0.4669	1.90
Doc2Vec	No Ontology, Formatting Markers	0.4505	0.4130	0.4291	0.5429	0.5480	0.5444	0.5234	0.4767	0.4958	0.4560	0.5283	0.4875	0.4932	0.4915	0.4892	1.99
	No Ontology, Keep Formatting	0.4255	0.4158	0.4187	0.5217	0.5416	0.5286	0.5396	0.4647	0.4933	0.4313	0.4825	0.4511	0.4795	0.4761	0.4729	1.92
	Ontology w/ Lexical Triggers, Formatting Markers	0.4227	0.4052	0.4111	0.5200	0.5536	0.5348	0.5039	0.4292	0.4578	0.4537	0.5035	0.4736	0.4751	0.4729	0.4693	1.91
	Ontology w/ Lexical Triggers, Keep Formatting	0.4539	0.4264	0.4374	0.4750	0.4816	0.4742	0.5194	0.4546	0.4807	0.4136	0.4779	0.4418	0.4655	0.4601	0.4585	1.86
	Ontology, Formatting Markers	0.4057	0.3796	0.3900	0.4966	0.5440	0.5158	0.5580	0.4634	0.4999	0.4510	0.4859	0.4616	0.4778	0.4682	0.4668	1.90
	Ontology, Keep Formatting	0.4468	0.3968	0.4169	0.4638	0.4853	0.4707	0.4759	0.4638	0.4663	0.4498	0.4862	0.4641	0.4591	0.4580	0.4545	1.85
RNN	No Ontology, Formatting Markers	0.4890	0.4730	0.4781	0.6462	0.7130	0.6768	0.5441	0.5194	0.5287	0.5934	0.5717	0.5793	0.5682	0.5693	0.5657	2.30
	No Ontology, Keep Formatting	0.4403	0.4594	0.4475	0.6089	0.6096	0.6050	0.5357	0.5526	0.5422	0.6078	0.5482	0.5731	0.5482	0.5424	0.5420	2.20
	Ontology w/ Lexical Triggers, Formatting Markers	0.5230	0.4634	0.4880	0.5792	0.6421	0.6043	0.6057	0.5861	0.5897	0.5950	0.5950	0.5932	0.5757	0.5716	0.5688	2.31
	Ontology w/ Lexical Triggers, Keep Formatting	0.5040	0.5025	0.5007	0.6403	0.5960	0.6112	0.5912	0.6040	0.5945	0.5994	0.6112	0.6023	0.5837	0.5784	0.5772	2.35
	Ontology, Formatting Markers	0.4798	0.4391	0.4541	0.5778	0.5713	0.5688	0.5816	0.6205	0.5980	0.5947	0.5938	0.5902	0.5585	0.5562	0.5528	2.25
	Ontology, Keep Formatting	0.4551	0.4085	0.4271	0.5949	0.5666	0.5752	0.5512	0.6080	0.5766	0.5679	0.5732	0.5652	0.5423	0.5391	0.5360	2.18
RNN (SO)	No Ontology, Formatting Markers	0.5454	0.4855	0.5099	0.6279	0.6417	0.6326	0.6020	0.6254	0.6108	0.5448	0.5469	0.5439	0.5800	0.5749	0.5743	2.33
	No Ontology, Keep Formatting	0.5376	0.4756	0.5003	0.6611	0.6527	0.6513	0.5890	0.6337	0.6055	0.5415	0.5552	0.5443	0.5823	0.5793	0.5754	2.34
	Ontology w/ Lexical Triggers, Formatting Markers	0.4538	0.4269	0.4374	0.5614	0.5882	0.5703	0.5242	0.5478	0.5328	0.4847	0.4598	0.4672	0.5060	0.5057	0.5019	2.04
	Ontology w/ Lexical Triggers, Keep Formatting	0.4345	0.4227	0.4205	0.5975	0.5673	0.5750	0.5009	0.5267	0.5068	0.4978	0.4978	0.4927	0.5077	0.5036	0.4988	2.03
	Ontology, Formatting Markers	0.4713	0.4344	0.4476	0.5876	0.6038	0.5922	0.4809	0.5074	0.4920	0.5361	0.5226	0.5237	0.5190	0.5171	0.5139	2.09
	Ontology, Keep Formatting	0.4535	0.4189	0.4322	0.5941	0.6209	0.6034	0.5186	0.5359	0.5241	0.5020	0.4931	0.4937	0.5171	0.5172	0.5133	2.09
TF/IDF	No Ontology, Formatting Markers	0.4429	0.5095	0.4712	0.6032	0.6888	0.6401	0.5018	0.6334	0.5570	0.5546	0.2405	0.3332	0.5256	0.5180	0.5004	2.03
	No Ontology, Keep Formatting	0.4364	0.5225	0.4719	0.6004	0.6801	0.6335	0.4850	0.5790	0.5260	0.5064	0.2278	0.3111	0.5070	0.5024	0.4856	1.97
	Ontology w/ Lexical Triggers, Formatting Markers	0.4177	0.5192	0.4609	0.5601	0.5963	0.5723	0.4691	0.5534	0.5069	0.5041	0.2494	0.3275	0.4877	0.4796	0.4669	1.90
	Ontology w/ Lexical Triggers, Keep Formatting	0.4013	0.4731	0.4302	0.5402	0.5917	0.5603	0.4456	0.5291	0.4826	0.4888	0.2602	0.3356	0.4689	0.4635	0.4522	1.84
	Ontology, Formatting Markers	0.4296	0.5185	0.4670	0.5757	0.6346	0.6004	0.4802	0.5880	0.5261	0.5197	0.2354	0.3205	0.5013	0.4941	0.4785	1.94
	Ontology, Keep Formatting	0.3988	0.4630	0.4269	0.5716	0.6252	0.5929	0.4712	0.5960	0.5248	0.5216	0.2479	0.3302	0.4908	0.4830	0.4687	1.90
Ontology Features	Ontology, Formatting Markers	0.3621	0.2528	0.2908	0.4331	0.4549	0.4358	0.5066	0.3998	0.4409	0.4138	0.5950	0.4854	0.4289	0.4256	0.4132	1.68
	Ontology, Keep Formatting	0.3438	0.2319	0.2733	0.4089	0.4420	0.4188	0.5060	0.3996	0.4376	0.3946	0.5783	0.4677	0.4133	0.4130	0.3994	1.62
Random		0.3200	0.2812	0.2823	0.2299	0.2625	0.2282	0.2677	0.2529	0.2543	0.2245	0.2207	0.2196	0.2605	0.2543	0.2461	

Table 52: Class-specific precision, recall, and F1-score metrics for the test we did with formatting handling and ontology classes, for the classification task.

	Executive			Existence			Property			Non-Architectural			Average			
Model	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	**Imp. over Random**
BOW (frequency)	0.4483	0.4220	0.4309	0.6931	0.6629	0.6745	0.5091	0.5614	0.5272	0.4694	0.4713	0.4669	0.5300	0.5294	0.5249	2.13
BOW (normalized)	0.4628	0.5192	0.4878	0.6898	0.6330	0.6577	0.4552	0.6336	0.5280	0.5511	0.2897	0.3727	0.5397	0.5189	0.5115	2.08
CNN	0.3971	0.3765	0.3853	0.5146	0.5484	0.5286	0.4814	0.5105	0.4921	0.4911	0.4469	0.4636	0.4711	0.4706	0.4674	1.90
Doc2Vec	0.4462	0.4045	0.4158	0.5091	0.5357	0.5207	0.5111	0.4866	0.4907	0.4522	0.4772	0.4615	0.4797	0.4760	0.4722	1.92
RNN	0.4529	0.4846	0.4657	0.5788	0.6208	0.5955	0.5928	0.5536	0.5681	0.5896	0.5393	0.5598	0.5535	0.5496	0.5473	2.22
TF/IDF	0.4442	0.5060	0.4718	0.6035	0.6805	0.6368	0.4721	0.6132	0.5314	0.5512	0.2233	0.3133	0.5178	0.5057	0.4883	1.98
Random	0.3200	0.2812	0.2823	0.2299	0.2625	0.2282	0.2677	0.2529	0.2543	0.2245	0.2207	0.2196	0.2605	0.2543	0.2461	

Table 53: Class-specific precision, recall, and F1-score metrics for the test we did with part of speech tagging, for the classification task.

	Executive			Existence			Property			Non-Architectural			Average			
Model	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	**Imp. over Random**
BOW (frequency)	0.4386	0.4846	0.4571	0.7067	0.6337	0.6651	0.5262	0.5666	0.5419	0.4839	0.4337	0.4525	0.5389	0.5297	0.5292	2.15
BOW (normalized)	0.4373	0.4881	0.4594	0.6708	0.6584	0.6631	0.4988	0.6508	0.5640	0.5279	0.2939	0.3694	0.5337	0.5228	0.5140	2.09
CNN	0.4088	0.3883	0.3968	0.5109	0.5649	0.5360	0.4868	0.5023	0.4909	0.5296	0.4840	0.5027	0.4840	0.4849	0.4816	1.96
CNN (SO)	0.4780	0.3790	0.4103	0.5380	0.6497	0.5828	0.5375	0.4928	0.4838	0.5079	0.5092	0.4961	0.5154	0.5077	0.4933	2.00
Doc2Vec	0.4505	0.4130	0.4291	0.5429	0.5480	0.5444	0.5234	0.4767	0.4958	0.4560	0.5283	0.4875	0.4932	0.4915	0.4892	1.99
RNN	0.4890	0.4730	0.4781	0.6462	0.7130	0.6768	0.5441	0.5194	0.5287	0.5934	0.5717	0.5793	0.5682	0.5693	0.5657	2.30
RNN (SO)	0.5454	0.4855	0.5099	0.6279	0.6417	0.6326	0.6020	0.6254	0.6108	0.5448	0.5469	0.5439	0.5800	0.5749	0.5743	2.33
TF/IDF	0.4429	0.5095	0.4712	0.6032	0.6888	0.6401	0.5018	0.6334	0.5570	0.5546	0.2405	0.3332	0.5256	0.5180	0.5004	2.03
Issue Properties	0.3216	0.2355	0.2692	0.3317	0.4258	0.3691	0.3258	0.2621	0.2806	0.4324	0.4827	0.4536	0.3529	0.3515	0.3431	1.39
Ontology Features	0.3621	0.2528	0.2908	0.4331	0.4549	0.4358	0.5066	0.3998	0.4409	0.4138	0.5950	0.4854	0.4289	0.4256	0.4132	1.68
Support Vector Machine ($n = 3$)	0.5684	0.6833	0.6150	0.5265	0.4875	0.5051	0.5757	0.5310	0.5513	0.5543	0.5083	0.5270	0.5562	0.5525	0.5496	2.23
Decision Tree ($n = 1$)	0.2508	1.0000	0.4010	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0627	0.2500	0.1006	0.41
One-vs-Rest ($n = 1$)	0.4896	0.6708	0.5631	0.4113	0.3583	0.3788	0.3764	0.2995	0.3319	0.4562	0.4208	0.4326	0.4334	0.4374	0.4270	1.74
Naive Bayes ($n = 1$)	0.6735	0.5458	0.5975	0.4530	0.4458	0.4448	0.4160	0.7886	0.5437	0.7067	0.2125	0.3232	0.5623	0.4982	0.4772	1.98
Random	0.3200	0.2812	0.2823	0.2299	0.2625	0.2282	0.2677	0.2529	0.2543	0.2245	0.2207	0.2196	0.2605	0.2543	0.2461	

Table 54: Class-specific precision, recall, and F1-score metrics for the best version of every classifier for the classification task.

		Executive			Existence			Property			Non-Architectural			Average			
Model	**Sub-Type**	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	**Imp. over Random**
BOW (frequency) + Issue Properties	Concatenation	0.3070	0.4963	0.3647	0.5238	0.3831	0.4208	0.3533	0.3626	0.3323	0.4188	0.3422	0.3445	0.4007	0.3961	0.3656	1.49
	Stacking	0.4500	0.4342	0.4368	0.6992	0.6707	0.6815	0.5301	0.5797	0.5481	0.4816	0.4632	0.4663	0.5402	0.5369	0.5332	2.17
	Voting	0.4334	0.4519	0.4384	0.6768	0.6504	0.6613	0.5365	0.5500	0.5388	0.4802	0.4588	0.4687	0.5317	0.5278	0.5268	2.14
BOW (frequency) + Issue Properties + Ontology Features	Concatenation	0.3759	0.4145	0.3841	0.4422	0.3791	0.3958	0.3167	0.3864	0.3421	0.3574	0.3026	0.3131	0.3731	0.3707	0.3588	1.46
	Stacking	0.4266	0.3980	0.4095	0.7023	0.6556	0.6741	0.5326	0.5661	0.5456	0.4629	0.4969	0.4739	0.5311	0.5292	0.5258	2.14
	Voting	0.4400	0.3544	0.3889	0.5909	0.6440	0.6120	0.5245	0.5109	0.5125	0.4948	0.5562	0.5215	0.5126	0.5164	0.5087	2.07
BOW (frequency) + Ontology Features	Concatenation	0.3362	0.2460	0.2632	0.3964	0.4949	0.4308	0.3688	0.2935	0.3036	0.3459	0.4103	0.3597	0.3618	0.3612	0.3393	1.38
	Stacking	0.4396	0.4276	0.4302	0.6863	0.6553	0.6677	0.5040	0.5656	0.5309	0.4935	0.4713	0.4776	0.5308	0.5300	0.5266	2.14
	Voting	0.4480	0.4267	0.4302	0.6950	0.6638	0.6768	0.5027	0.5291	0.5099	0.4904	0.5007	0.4927	0.5340	0.5301	0.5274	2.14
BOW (frequency) + RNN	Concatenation	0.3042	0.3175	0.2901	0.4525	0.4513	0.4331	0.3796	0.4904	0.4262	0.4197	0.4053	0.3702	0.3890	0.4161	0.3799	1.54
	Stacking	0.5748	0.2866	0.3613	0.4620	0.6673	0.5420	0.3874	0.5041	0.4340	0.6001	0.4128	0.4865	0.5061	0.4677	0.4560	1.85
	Voting	0.4769	0.4808	0.4767	0.6903	0.6640	0.6738	0.5708	0.5826	0.5734	0.5478	0.5393	0.5396	0.5715	0.5667	0.5659	2.30
Issue Properties + Ontology Features	Concatenation	0.2615	0.2321	0.2007	0.3241	0.4845	0.3814	0.2722	0.3866	0.3052	0.2239	0.2377	0.2052	0.2704	0.3352	0.2731	1.11
	Stacking	0.2876	0.4596	0.3528	0.1921	0.1257	0.1468	0.1443	0.0337	0.0542	0.3477	0.5614	0.4267	0.2429	0.2951	0.2451	1.00
	Voting	0.3852	0.2691	0.3115	0.4520	0.5347	0.4861	0.5389	0.4214	0.4648	0.4343	0.5744	0.4928	0.4526	0.4499	0.4388	1.78
RNN + Issue Properties	Concatenation	0.3476	0.2098	0.2513	0.5004	0.6113	0.5390	0.3757	0.3940	0.3762	0.4261	0.4969	0.4456	0.4125	0.4280	0.4030	1.64
	Stacking	0.4608	0.4502	0.4532	0.6514	0.6323	0.6386	0.5740	0.6085	0.5869	0.5920	0.5734	0.5797	0.5695	0.5661	0.5646	2.29
	Voting	0.4467	0.4709	0.4543	0.6569	0.6459	0.6475	0.5672	0.5489	0.5553	0.5832	0.5724	0.5754	0.5635	0.5595	0.5581	2.27
RNN + Issue Properties + Ontology Features	Concatenation	0.2911	0.2833	0.2801	0.4196	0.4642	0.4341	0.3402	0.3687	0.3472	0.4329	0.3954	0.4103	0.3709	0.3779	0.3679	1.49
	Stacking	0.4410	0.4210	0.4260	0.6413	0.6671	0.6494	0.5512	0.5277	0.5354	0.5892	0.5984	0.5845	0.5557	0.5535	0.5488	2.23
	Voting	0.4567	0.3791	0.4108	0.5633	0.6556	0.6022	0.5684	0.4810	0.5184	0.5675	0.6370	0.5979	0.5390	0.5382	0.5323	2.16
RNN + Ontology Features	Concatenation	0.3552	0.2552	0.2913	0.4524	0.5221	0.4778	0.4547	0.5667	0.4951	0.3913	0.3901	0.3714	0.4134	0.4335	0.4089	1.66
	Stacking	0.4513	0.4507	0.4481	0.6336	0.6667	0.6475	0.5621	0.5788	0.5680	0.5923	0.5317	0.5559	0.5598	0.5570	0.5549	2.25
	Voting	0.4730	0.4418	0.4539	0.5950	0.6415	0.6133	0.5911	0.6002	0.5889	0.6099	0.5611	0.5782	0.5672	0.5612	0.5586	2.27
Random		0.3200	0.2812	0.2823	0.2299	0.2625	0.2282	0.2677	0.2529	0.2543	0.2245	0.2207	0.2196	0.2605	0.2543	0.2461	

Table 55: Class-specific precision, recall, and F1-score metrics for the test we did with combined models for the classification task.

	Executive			Existence			Property			Non-Architectural			Average		
Model	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score	Precision	Recall	F1-Score
BOW (frequency)	0.4071	0.3672	0.3829	0.5036	0.4882	0.4871	0.4606	0.4824	0.4641	0.4506	0.4566	0.4508	0.4555	0.4486	0.4463
BOW (normalized)	0.3759	0.3675	0.3675	0.5036	0.4647	0.4725	0.4217	0.6106	0.4941	0.5477	0.3206	0.3935	0.4622	0.4408	0.4319
CNN	0.3681	0.3123	0.3262	0.3863	0.3700	0.3521	0.4644	0.5150	0.4805	0.5141	0.4243	0.4383	0.4332	0.4054	0.3993
CNN (SO)	0.4211	0.3393	0.3130	0.4108	0.4924	0.4155	0.4985	0.4782	0.4501	0.5108	0.4150	0.4507	0.4603	0.4312	0.4073
Doc2Vec	0.4215	0.3931	0.4035	0.3424	0.4622	0.3745	0.4578	0.3582	0.3923	0.4639	0.4752	0.4666	0.4214	0.4222	0.4092
Issue Properties	0.2294	0.1724	0.1726	0.1437	0.4118	0.1984	0.2389	0.2741	0.2126	0.3957	0.4171	0.3927	0.2519	0.3189	0.2441
Ontology Features	0.3950	0.2475	0.2895	0.3517	0.3854	0.3304	0.4971	0.4034	0.4393	0.3843	0.5558	0.4517	0.4070	0.3980	0.3777
RNN	0.4422	0.3726	0.3971	0.4592	0.4860	0.4623	0.5547	0.5475	0.5487	0.5713	0.5809	0.5652	0.5068	0.4967	0.4933
RNN (SO)	0.5032	0.3988	0.4291	0.4485	0.5484	0.4802	0.5918	0.5874	0.5739	0.5528	0.5092	0.5205	0.5241	0.5109	0.5009
TF/IDF	0.3994	0.4085	0.4018	0.4129	0.4771	0.4322	0.4281	0.5819	0.4862	0.5615	0.2311	0.3105	0.4505	0.4247	0.4077
Support Vector Machine ($n = 3$)	0.4618	0.5390	0.4850	0.4809	0.3650	0.4115	0.5151	0.4887	0.4994	0.4900	0.5808	0.5253	0.4869	0.4934	0.5066
Decision Tree ($n = 4$)	0.0572	0.5000	0.1012	0.0959	0.5000	0.1608	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0383	0.2500	0.0443
One-vs-Rest ($n = 1$)	0.3857	0.5372	0.4367	0.3754	0.3859	0.3714	0.4048	0.3476	0.3682	0.4415	0.3392	0.3607	0.4018	0.4025	0.4084
Naive Bayes ($n = 1$)	0.4562	0.3812	0.3932	0.3167	0.3708	0.3186	0.4139	0.7098	0.5131	0.6010	0.1577	0.2459	0.4469	0.4049	0.3866

Table 56: Class-specific precision, recall, and F1-score metrics for the test we did with cross-project classification