Bachelor Thesis (Oliver Holder)

Oliver Holder

August 2020

Technical debt is a wide spread phenomenon in the software world. It occurs when developers make wrong or unhelpful choices, often in the form of short term solutions, which increases the amount of work in the long run. This thesis aims to study machine learning techniques for the prediction of technical debt in issue trackers. Moreover, the use of convolution neural networks allows for the extraction of n-gram phrases that identify the presence of technical debt. Issue tracker repositories were used as data sets. In total 5335 tickets were annotated, used for model training, and had key words extracted. The evaluation of the models shows that convolution neural networks slightly outperform other traditional models in this application.

1 Introduction

Technical debt (TD) is a term used to refer to the cost inccured in software implementations when short term solutions are used as opposed to long term solutions. These short term solutions, or work-arounds, can lead to development artefacts that can cause negative side effects for the software being developed. Technical debt can be introduced within many different areas and levels of a project. Some examples include documentation debt, code debt, test debt, build debt, etc. The analogy to debt is used because these short term work arounds must be paid back in time. Recently there have been many works on autonomously discovering technical debt in software production [4] [12]. These projects aim to facilitate the discovery of technical debt. They also aim to contribute to the understanding of the structure of technical debt, or the modes in which it is introduced. The term 'Technical Debt' is taken to be the debt that accrues in software development when a development team opts for an easy approach to implementation choices that have clear negatives impacts in the long term [5] [7].

From articles such as 'Got Technical Debt?', we know that technical debt is wide spread and common in software development [4]. There is no formal system for handling technical debt, there are however some accepted methods for identifying technical debt [7]. It is often difficult to identify the prevalence of technical debt in any given project. Such information can be obtained from many sources, such as classifying source code, classifying source code comments, or classifying issue tickets in issue trackers. Articles have been written on identifying and classifying technical debt [5][7]. However, classifying these units manually takes a lot of manpower and is often inefficient.

To solve the problem of autonomous identification, machine learning techniques will be implemented. The term 'Machine learning' dates back to 1959 when Arthur Samuel coined the term [6]. Some standard machine learning models include: Naive Bayes classifiers, K nearest neighbours classifiers, Decision tree classifiers, and support vector machines. More recently, since the advent of backwards propagation by Paul Werbos in 1975 [11], neural networks have become the main focus of machine learning. Such networks are usually refered to as the field of deep learning. In this research project, both traditional and deep learning models will be used to attempt to solve the tasks given.

2 Study design

2.1 Project Focus and Scope

The focus of this project will be to create and train multiple machine learning models to analyse the presence/prevalence of technical debt in a given project by the study of said project's 'Issue Tracker' repositories. Five repositories will be taken, corresponding to the software projects: Camel, Hadoop, HBase, Impala, and Thrift. Other studies have been undertaken to study the presence and prevalence of TD using source code comments [4] [3]. The advantage of using issue tracker tickets to analyse prevalence of TD is that source code comments are often informal [4] and therefore less structured, potentially leading to a more complex problem space. Moreover, the conjunction of two separate identification methods that collate data from independent sources could provide a clearer picture of the impact of technical debt in projects and potentially provide insight into the differences of how technical debt is reported in source-code comments and issue tracker tickets. These two methods need not be used exclusively, but rather could be used together to strengthen our knowledge of TD prevalence.

2.2 Questions and Objectives

The goal formulation will take the form recommended in the Encyclopedia of Software Engineering [1]:
The goal of this project will be to analyse machine learning techniques for the purpose of prediction with respect to technical debt from the point of view of developers of software in the context of software engineering. The aims can be broken down like so:

1. What is the best learning approach to automatically identify technical debt?

2. What words are most commonly associated with certain types of technical debt?

3. How much data is required to achieve a high degree of interpretability?

4. What are the most prevalent types of technical debt identifiable in issue trackers?

2.3 Motivation of Aims

1. Technical debt has been shown to be an unavoidable consequence of software development [10]. Therefore, all software development projects would benefit from a resource allowing the autonomous identification of technical debt. This would enable developers to gain an overview into the amount of technical debt in a project, as well as the ability to identify any debt newly entering the project. All this will reduce the amount of work needed to optimise a project from a managerial point of view.

2. A set of indicator words for each type of technical debt will increase the communities understanding of TD as a whole, allowing software engineers and managers to develop strategies to combat the practices that commonly introduce TD.

3. Knowing how much information is needed to achieve a high degree of interpretability will allow future researchers to have an idea of how much data their research will require.

4. Knowing the most common types of technical debt will prompt software developers to dedicate more resources to fixing certain debts in their projects.

2.4 Project structure

2.4.1 Data collection

Issue tracker tickets were collected from 5 project repositories. The collected data set of issues was then refined to exclude noise. Removing irrelevant data from tickets allows the learning algorithm to fit using a less complex data space. Given that the important information for classifying presence and classification of technical debt is more common in the main body of the ticket, other sections such as author, tag, date, and comments were removed to create a refined data-set. Quoted code was also removed by the identifying regular expression:

$$(\{code\}.*?\{code\})|(\{noformat\}.*?\{noformat\})$$

2.4.2 Data annotation

Taking the refined training set from the data collection phase, each ticket will be manually annotated as either technical debt or not technical debt and, further more, classified into different types of TD. This classification will follow an accepted method on the identification of technical debt [7]. The results will be compared and refined until a close consensus is formed regarding the data-set. The method for determining whether an issue ticket is technical debt will be taken from a previously defined method [8].
In-depth descriptions of the types of technical debt annotated will be provided in the data analysis section.

2.4.3 Machine learning

Multiple traditional machine learning models will be created and trained on the self admitted technical debt data set. These machine learning models, including, K nearest neightbor, Naive bayes, Decision tree, and support vector machines, will be used as a baseline for the effectiveness of the deep networks to be designed and trained in the next section of this report.
The main focus of this projects will be the development of a convolutional neural network(CNN) to predict the presence of technical debt. The reason for this is two-fold, firstly, this network has been shown to be effective in other project focusing on technical debt classification [12]. Secondly, CNN implementations allow for the extraction of key words that are heavily weighted by the model in the process of classification. This allows for the compiling of a list of phase indicators for technical debt, as well as for a higher degree of interpretability than other deep learning models.
The annotated data set discussed in data analysis will be split by project, whereby a leave one out cross validation approach will be used to train and validate the learning algorithms. This means for each iteration, one project will be selected to create two test subsets. The first test subset will be used for validation during network training. This subset will be responsible for determining early stopping by f1 score. The second subset will be used to determine the accuracy, precision, and f1 score after the algorithm is finished training. The remaining projects will be used as the training data set.

The benefit of performing this cross validation approach is that it allows the accuracy of the model to be tested over multiple test data sets to gain insight into the overall generalizability of the model across different software projects.

The complexity of the learning model should be related to the complexity of what it is to predict. That is, the complexity or number of layers in the network will depend on the non-linearity of the decision problem.
For some deep learning implementations, natural language processing techniques such as word embeddings will be used.

2.4.4 Key word extraction

The convolutional neural network will allow for uni-gram and bi-gram key word extractions. The comments are extracted by breaking up the model and revealing the output of hidden layers. We use these outputs in conjunction with weights in the network and back-tracing to determine uni-grams and bi-grams with high indication of technical debt.

3 Data analysis

3.1 Data refinement

As previously noted, the data set of collected tickets was defined to omit irrelevant noise such as author, tag, date, or code segments. This allows for the learning algorithm so better refine its predictions without perturbations from noise in the data set. Code segments were removed by identification using the following regular expression:

$$(\{code\}.*?\{code\})|(\{noformat\}.*?\{noformat\})$$

3.2 Manual classification method

The issues collected were classified into one of 8 classes of debt, each class was also partitioned into sub-classes, leading to 23 sub-classes of technical debt.

3.3 Technical debt types

These classifications and sub-classifications are listed. Each of these are given a brief description below.

- Architecture debt
  - Violation of modularity
  - Obsolete technology
  - Other
- Build debt
  - Over declared dependencies
  - Under declared dependencies
  - Other
- Code debt
  - Complex code
  - Dead code
  - Duplicated code

  - Low quality code
  - Multi-thread correctness
  - Slow algorithm
  - Other
- Defect debt
- Design debt
- Documentation debt
  - Out of date documentation
  - Low quality documentation
- Requirements debt
  - Requirement partially implemented
  - Function not satisfied
- Test debt
  - Expensive tests
  - Lack of tests
  - Low coverage

3.3.1 Architecture debt

Architecture debt are problems relating to the structure of a project. These issues are often hard to act on because they are architectural and provide a basis or structure to the project as a whole. Examples of architectural debt are: Using obsolete technology or violations of modularity.

3.3.2 Build debt

These problems relate to problems with compiling and distributing a project. Poorly deployed practices and over or under inclusion of dependencies constitute build debt. Instances of build debt can delay developers from actually working on a task they want to achieve, slowing down the development process.

3.3.3 Code debt

Problems considered as code debt are ones that stem from poor implementation of code. These problems increase the time and effort required to maintain code. Examples of this consist of: magic variables, inefficient algorithms, over complex algorithms, unused code, and many other bad coding practices.

3.3.4 Defect debt

Problems considered defect debt are problems in the application/program that are known but the developers consider low priory and are thus left to be fixed later. It is difficult to give general examples for this case as they it can manifest in a wide variety of issues.

3.3.5 Design debt

Design debt problems are problems that stem from a inefficient or non-optimal design decision. An example of design debt could be using a 'long double' to store a value when only a 'float' is required.

3.3.6 Documentation debt

Documentation debt is caused by lack of documentation or outdated documentation. It is common that developers will change features or implementation of their programs without updating the corresponding documentation.

3.3.7 Requirements debt

Problems considered requirement debt are problems where the software in development either does not implement a feature required form the specification, or only partially implements said feature.

3.3.8 Test debt

Problems considered test debt are problems that occur when developers do not create, update or maintain their testing libraries. Test debt can also occur where tests are inefficient and take unreasonably long to run. This type of debt can lead to an accumulation on unknown inefficiency or bugs in the software.

3.4 Annotation validation

During the annotation stage of this project, subsets of issues were reviewed with another individual with experience in identifying technical debt. These reviews were used to improve the quality of annotation. After the data set was fully annotated, a sub-set of issues were categorized separately by another, these categorisations were then used to determine the degree of agreement between classifications.
Cohen's kappa coefficient κ was used to measure the inter-rater reliability of the classifications . This coefficient is found by the equation: $\kappa = \frac{p_0 - p_e}{1 - p_e}$ Where p_e is the observed agreement and p_0 is the probability of chance agreement. The resulting value of κ will be '1' at maximum agreement, '0' at no agreement, and less than zero at worse than random agreement [2].
The kappa coefficient found for agreement between raters was 0.696, which shows there is considerable agreement between raters.

4 Machine learning

In this section, various modeling methods will be analysed and implemented. These methods will be tested on a previously created data set that is also on the topic of technical debt [3]. Once the most effective model and hyper para-maters are determined, the data set of focus will be used to determine the models f1 score.

4.1 Validation metrics

To validate the learning models, the metric of F1 score will be used. F1 score is a measure of both recall and precision. Precision refers to the true positives divided by total positives, i.e. how accurate the model is when it predicts true. Recall refers to true positives divided by true positives and false negatives, i.e. the proportion of all debt instances that are correctly classified as debt.
F1-score is the harmonic mean of precision and recall.

4.2 Traditional machine learning models

A short overview of the traditional machine learning models used will be given here. These models will be used as comparators for the more complex neural network driven models implemented later.
The information that will be passed to each of these models is a bag of words for the independent variable, and a binary representation of the presence of technical debt as the dependent variable. The bag of words is created from the input text using Sklearns TfidfVectorizer [1]. This turns all sets of strings on input into a vector that the learning models can interpret.
The output is essentially an array that contains a 1 or 0 depending on whether the given text segment exhibits the type of technical debt being predicted.

4.2.1 K Nearest Neighbors

The K Nearest Neighbors algorithm predicts the output of some abstract function by finding the K nearest neighbors in the state space of the dependant variable by using some distance function.

4.2.2 Naive Bayes

A Naive Bayes Classifier is a simple probabilistic classifier that assumes a strong degree of independence between features.

[1] $https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html$

Probability model

Take the input vector $X = (x_1, x_2, x_3, .., x_n)$.
We want to know $P(C_k|x)$.
This is given by $P(C_k|x) = \frac{P(C_k)P(x|C_k)}{P(x)}$
The denominator of this fraction is effectively constant.
The numerator can be rewritten as a joint probability: $P(C_k)P(x|C_k) = P(C_k, x_1, x_2, .., x_n)$
The numerator can be further rewritten using the chain rule:
$P(C_k, x_1, x_2, .., x_n) = P(x_1, x_2, .., x_n, C_k) = P(x_1|x_2, .., x_n, C_k)..P(x_n|C_k)P(C_k)$ Assuming that the input features are conditionally independent, we can reduce the above equation to:
$P(x_1|C_k)...P(x_n|C_k)P(C_k)$
Finally, the probability of class C_k can be expressed succinctly like so:
$P(C_k|X) = \frac{1}{Z}P(C_k)\prod_{i=1}^{n}(x_i|C_k)$

Classifier

Using the above probability model, the classifier can be constructed like so:
y $= argmax_{k\in\{1,..,K\}}(P(C_k)\prod_{i=1}^{n}(p(x_i|C_k))$

4.2.3 Decision Tree

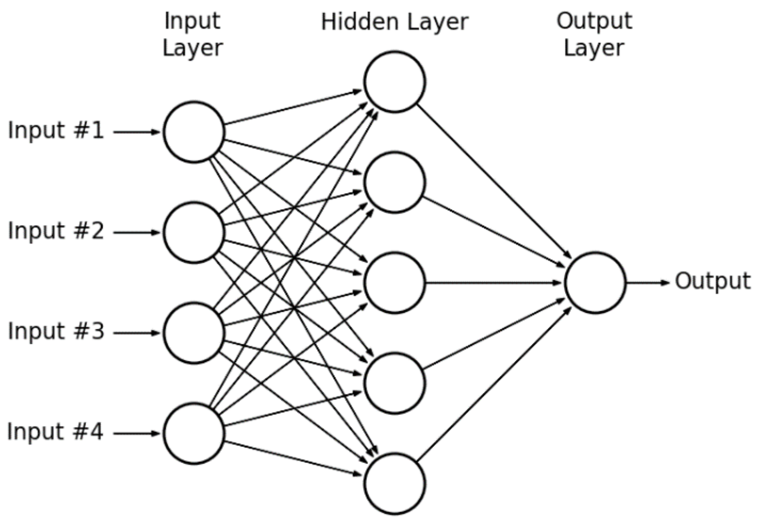
The decision tree classifier works by parsing through a tree based on a feature at each step (or node). The resulting leaves at the bottom of the tree correspond to a categorisation. The tree is constructed by creating nodes that split based on features, this process is continued until a leaf is reached that contains only one type of classification.

4.2.4 Support vector machine

Essentially, an SVM maps its inputs into a state space which is then divided based on classifications. New inputs are then mapped to the same space and the classification is determined by the position.
SVMs can often outperform other simple machine learning models due to its ability to model nonlinear functions on input using the kernel trick, which increases the dimension of the input space.

4.3 Feed-Forward Neural Networks

The simplest deep learning networks are called feed forward networks. These networks, also known as multi-layer perceptions, take a vectored input such as a bag of words. They pass this bag of words vector into layers that weight each feature in the vector (presence of a word). The last layer of the network then implements a more traditional machine learning technique on the weightings given by the earlier layers in the network. The network is trained by modifying the weights attributed to words using backwards propagation on a training set.



[2]

4.4 Word embedding

Before text segments can be used by the deep learning model for prediction, it must first be tokenized, then embedded into a higher dimension. This means each word is converted into a 'n' length vector in which each value in the vector relates to some abstract property of the word. This process can be done starting with random seed values for words, but is more efficient when a corpus of trained values is used. In this project, the wikipedia corpus was used [3].
Word embedding allows for a capturing of multiple dimensions of each word. This is a powerful technique used together with CNNs as their filters can effectively use higher dimensions.

4.5 Convolution Neural Networks

Convolutional networks are networks that implement a convolution layer. The convolution layer applies filters to its input to create feature vectors, allowing

[2] $https : //www.researchgate.net/figure/A-hypothetical-example-of-Multilayer-Perceptron-Network_fig4_303875065$

[3] $https : //fasttext.cc/docs/en/pretrained-vectors.html$

for many different properties of the input to be evaluated. CNNs are adept at recognising patterns that occur locally, which for this application is useful, as local patterns are expected to be more prominent than long distance patterns.
In this project, a convolutional neural network will be the primary focus because it has been shown to be an effective method for identifying technical debt [12] and because of its structure allowing for the extraction of key words, which in turn allows a high degree of interpretability of the model.
Below the structure of the CNN we created will be listed and described.

4.6 CNN Structure

- Embedding Layer -
  Before anything is passed to the embedding layer, an embedding matrix must be constructed from a corpus of words and values [4] in conjunction with the words found in the input sentences. This embedding matrix is used to create the embedding layer which splits individual words into multiple values.

- Reshaping layer -
  The output of the last layer is reshaped to fit the convolution layers. The output of this layer is passed to two parallel branches of the model, the uni-gram and bi-gram branches, respectively.

- Convolution 1D Layer (Uni-gram) This convolution layer takes an input matrix which is determined by the embedding layer (with some values dropped out). This matrix is then dot producted with a set of filters (the size of one word) with some step across the matrix, leading to a set of feature vectors.

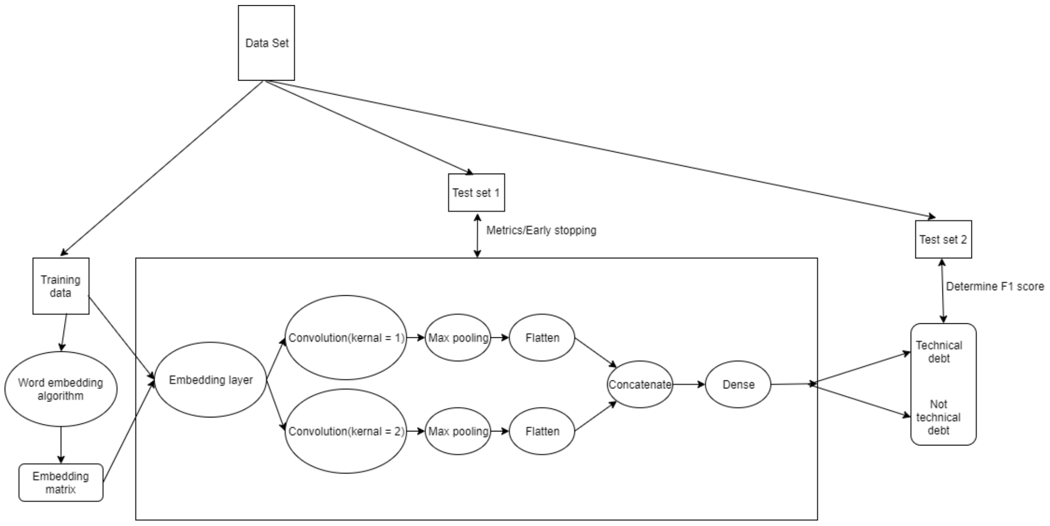
- Max Pooling Layer -
  Max-pooling is where the maximum value of a set of features is extracted. Thus the maximum valued feature is carried to the next flattening layer.

- The flattening layer converts its input into a one dimensional matrix, where each value in the input is maintained.

- The concatenation layer combines the flattened outputs of both the uni and bi-gram branches.

- The output layer, or activation layer, is decided based on a threshold, the value given as its input, and a set of weights.

[4] $https : //fasttext.cc/docs/en/pretrained - vectors.html$



4.7 Tuning hyper parameters

The hyper parameters of the model were initially chosen by reference to similar projects [9] [12] that dealt with CNNs classifying text segments. These values were then further refined by means of analysis using the "CAMEL" project as the test data and the other projects as training data. Each hyper parameter value was used to train the model ten times and the average value of those was taken for comparison. In some cases, the trend is not clear, either because the parameter has little effect, or because of the high variance in the training algorithm.

- Word Embedding Dimension -
  The number of values associated with each word. The initial value used was 300. The optimal value found was 300.
- Kernel sizes -
  This determines the size of the filters in the convolution layers. The values chosen for this were 1 and 2, representing uni-grams and bi-grams respectively.
- Epochs -
  The number of training passes the model makes while training. The initial value used was 15. The optimal value found was 2.
- Batch size -
  The batch size determines the amount of data that is used to train the model for each epoch of training. The value used was 45.

- The number of filters -
  This determines the number of filters applied in the convolution layers. The base value was 300. The optimal value found was 300.

- Sequence(sentence) length -
  This is the max number of words each sentence on the input can contain. Any sentences with less will be padded. The value was chosen by looking for the longest sentences in the training data. A value of 1000 was chosen.

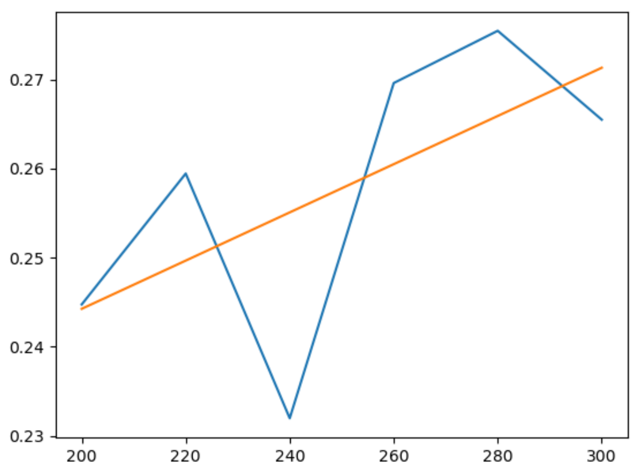
- Strides -
  The size of the step between applying filters in the convolution layers. This value must be (1,1) for key word extraction.

- Pool size -
  This is the size of the pooling region. This value must be $num\_words - filter\_size + 1$ for the key word extraction to work.

- Optimizer -
  The optimizer handles things like how fast the model should adjust its weights during backwards propagation. The initial optimizer used was adam optimizer. The best optimizer found was Nadam.

4.7.1 Word Embedding Dimension

The base value for word embedding was based off of similar projects [9] [12] that dealt with CNNs classifying text segments. This value was set to 300. Unfortunately, the corpus used to create the embedding matrix had only 300 values per word, thus 300 was the maximum usable value. Thus, the possible embedding dimension was searched in intervals of 20.
The trend seems to be a possible correlation between embedding depth and F1 score, thus we choose 300 depth as the optimum value.

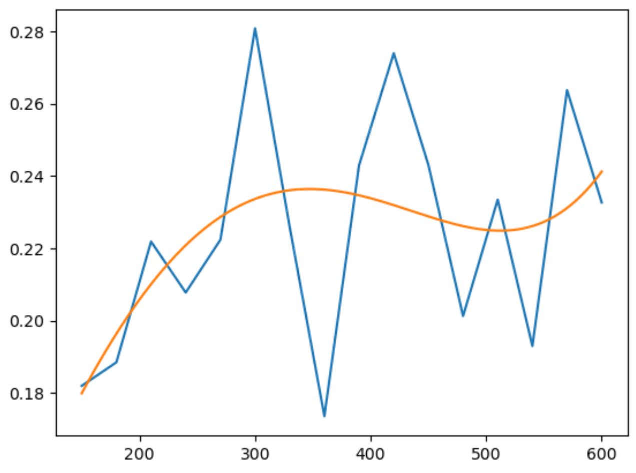


4.7.2 Epochs

Initially, 15 epochs were used. However, the model was very subject to over fitting, and the best results were achieved with only two epochs.

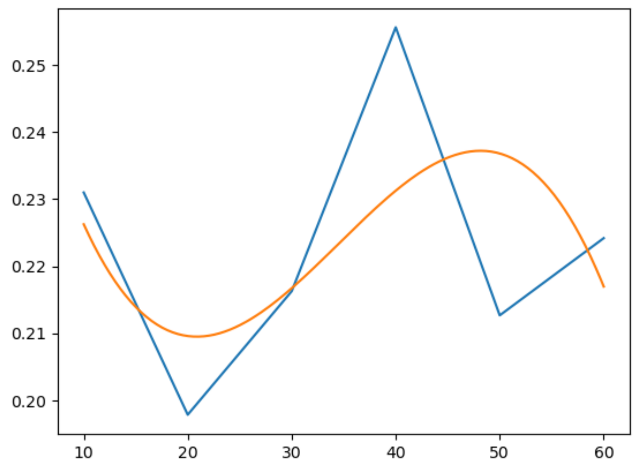
4.7.3 Number of filters

The number of filters showed a positive correlation with f1 score up until around 300 filters where it seems to plateau. Thus 300 was chosen as an optimal value.



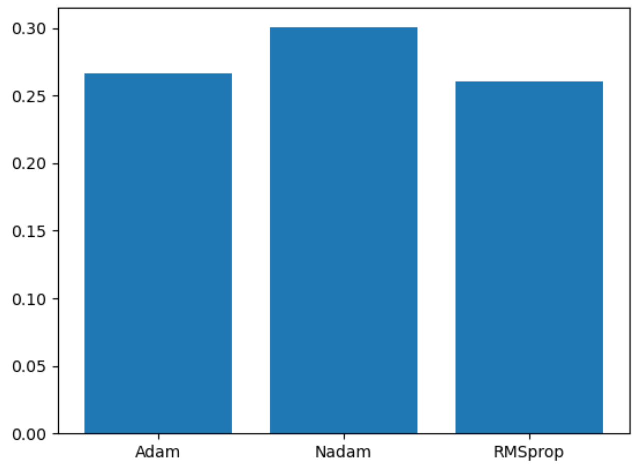
4.7.4 Batch number

The optimum value for batch number appears to be somewhere been 40 and 50. So it was set to 45.



4.7.5 Optimizer

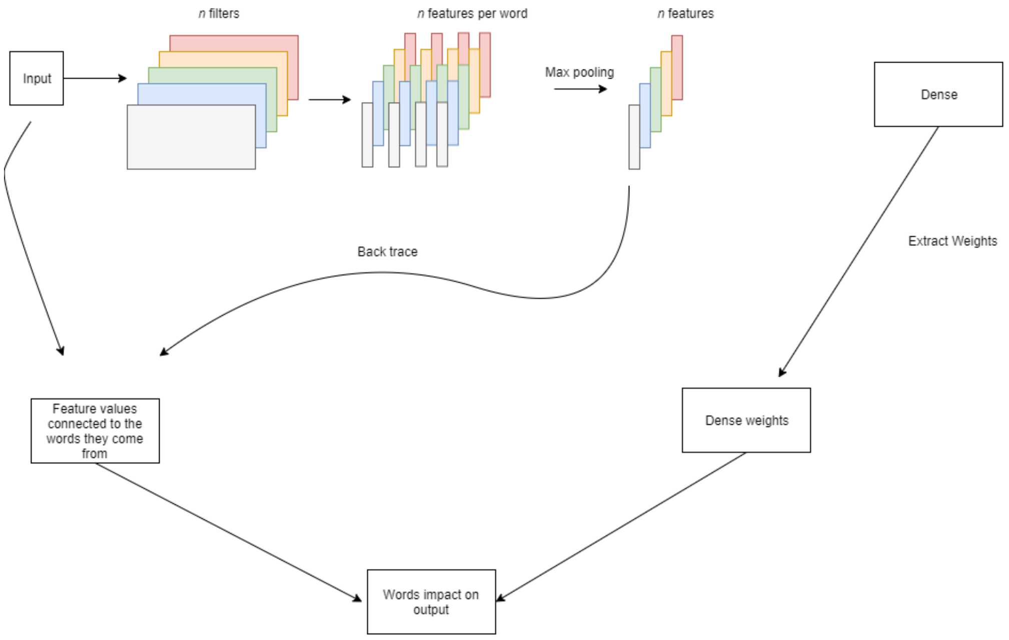
The three optimizers that showed any promise were adam, nadam, and RM-Sprop. Of the three, Nadam gave the best f1 score on average.



5 Key Word Extraction

To extract key words from the trained CNN, first the feature vector must be extracted from the hidden layers on some given input, each value in this feature vector relates to one filter in the convolution layers. These feature values are then used in conjunction with a Sigmoid function to predict the likelihood that this feature implies technical debt.

1. Use trained CNN model to predict comment.
2. Extract output of hidden layers: CNN and max pooling
3. Extract weightings for dense layer
4. Use CNN layer output to match words in input comment with features in max pooling layer.
5. Product max pooling output with dense layer weights to get numeric representation of probability of technical debt.
6. repeat 2-5 for both uni and bi grams.



6 Results

6.1 Model metrics

The results from various learning models will now be compared. The most important variable will be the debt f1 score: This is F1 score where only positive

examples are considered. The reasoning for this is the data set is heavily weighting towards non-debt, thus the weighted(both non debt and debt) F1 score will be less informative.
The source code comment data set, taken from previous studies [4][12], shows that the CNN method we created has about the same effectiveness for identifying the presence of technical debt as decision tree classifiers and support vector machines. The multi-layer perception model fared much worse than the three other models mentioned. Both Naive Bayes and KNN models performed very poorly.
The result for the CNN compares to its effectiveness in other papers [12] in which it is tested on the same data set and achieved around 0.70 on F1 score.

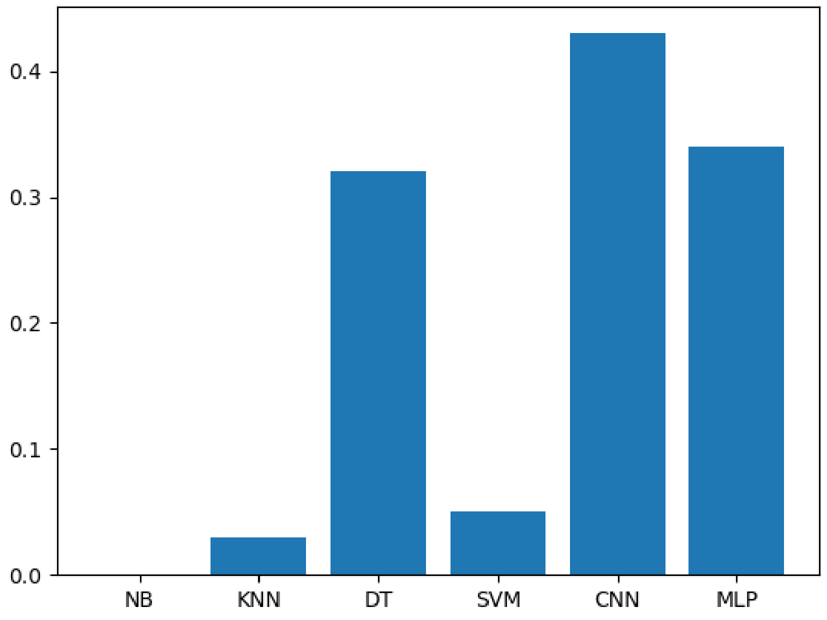
project name	Naive Bayes	KNN	DT	SVM	CNN	MLP
apache-ant-1.7.0(40)	0.37	0.10	0.51	0.53	0.70	0.44
apache-jmeter-2.10(57)	0.33	0.13	0.75	0.74	0.78	0.58
argouml(226)	0.49	0.18	0.78	0.83	0.74	0.67
columba-1.4-src(27)	0.57	0.00	0.80	0.64	0.72	0.62
emf-2.4.1(25)	0.33	0.33	0.36	0.48	0.38	0.24
hibernate-distribution-3.3.2.GA(142)	0.47	0.12	0.78	0.80	0.81	0.67
jEdit-4.2(59)	0.39	0.15	0.53	0.31	0.56	0.55
jfreechart-1.0.19(25)	0.21	0.08	0.63	0.45	0.19	0.31
average (601)	0.32	0.15	0.71	0.70	0.71	0.60

Table 1: Debt F1 scores (Source code comments)

For the issue tickets data set, the best performing models were the CNN at 43%, followed by the decision tree classifier and multi-layer perceptron at 32%. Both KNN and Naive Bayes models performed extremely poorly.
The models perform worse on the tickets data set. This could be caused by multiple factors. Firstly, the consistency of the annotations to the data set could be worse, meaning some of the annotations are wrong. Secondly, the text segments taken from the issue trackers are less succinct than that of the source code comments. This makes sense because source code comments must be short by their nature, whereas issue tickets can be very long and contain much more varied language. Finally, the number of training points in the issue tracker data set is drastically smaller than that of the source code comments data set (5000 to 55000).

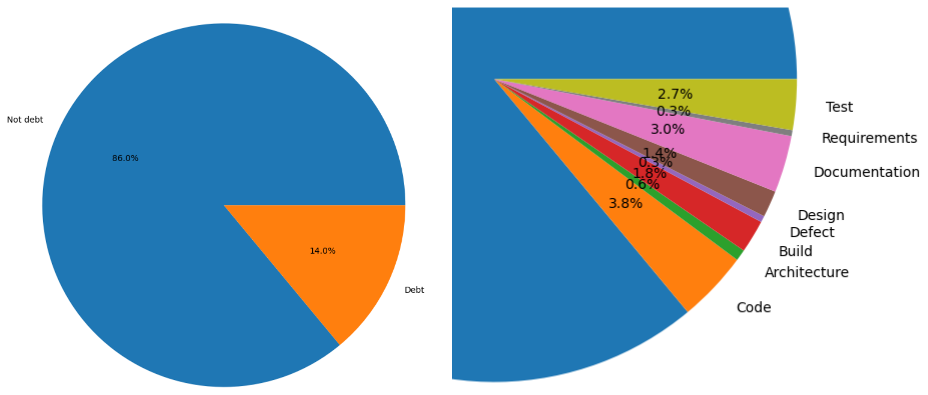
project name	Naive Bayes	KNN	DT	SVM	CNN	MLP
Camel(99)	0.00	0.06	0.28	0.04	0.40	0.41
Hadoop(200)	0.00	0.01	0.34	0.02	0.39	0.28
Hbase(210)	0.00	0.03	0.29	0.07	0.44	0.39
Impala(129)	0.00	0.03	0.37	0.05	0.42	0.28
Thrift(107)	0.00	0.04	0.29	0.07	0.53	0.35
Average(745)	0.00	0.03	0.32	0.05	0.43	0.34

Table 2: Debt F1 scores (Issue tickets)

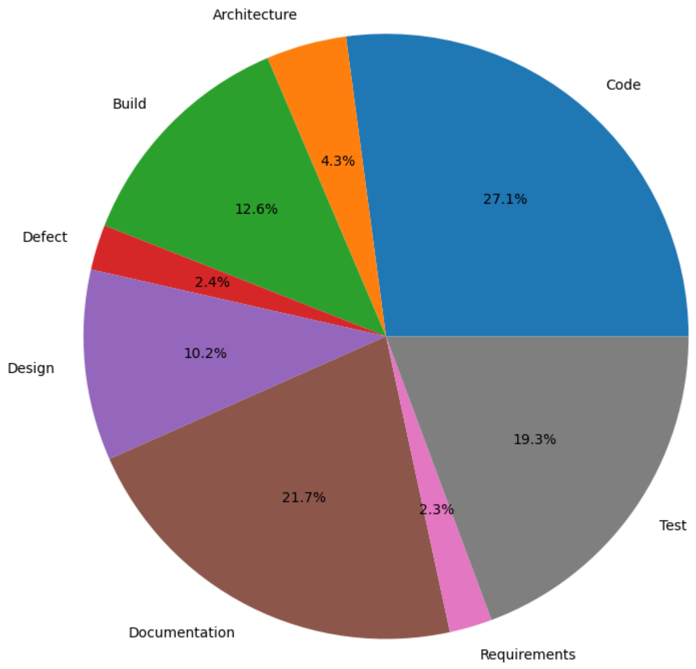


6.2 What are the most prevalent types of technical debt identifiable in issuetrackers?

From an analysis of the annotated data set, we are able to conclude that 14% of issue tracker tickets relate to some type of technical debt.



Of that 14%, Code debt was the most prevalent with a proportion of 27.1%, followed closely by documentation and test debt (21.7% and 19.3% respectively). The least common types of technical debt were requirements debt at 2.3% of positive instances and defect debt at 2.4% of positive instances.



6.3 Key words extraction

The top key words were extracted from our CNN model that predicted all debt, code debt, documentation debt and test debt. These categories were chosen because they represent the largest portion of debt in the issue tickets data set. Some of the words have a clear relation to the type of technical debt the model is predicting. Some of the words appear to be noise that was not removed due to a too small data set.

uni-grams	probability	bi-grams	probability
Wipe	52.11%	Cluster doesn't	54.75%
revisited	52.06%	Tests verify	54.45%
proving	51.92%	Cluster testing	53.91%
unsafe	51.91%	Tests passed	53.77%
diffs	51.87%	Method repeated	53.71%
test	51.82%	Cluster may	53.62%
supplanted	51.81%	Test pass	53.62%
updates	51.80%	Tests make	53.62%
cluster	51.79%	Core test	53.61%
erroneous	51.78%	Generator test	53.56%
complicates	51.77%	Test tool	53.51%
docs	51.76%	Cluster sortby	53.46%
risky	51.69%	Version simultaneously	53.21%

Table 3: Key words indicating technical debt

Words	probability
suspend	71.05%
2033 thrift	70.11%
part thrift	68.77%
run thrift	67.21%
lock pity	65.50%
dangerous atomic	64.92%
nonzero	64.13%
callers	62.75%
reduces amount	61.68%
completely	61.65%
pollute	61.47%
reduce	61.096%
nodes default	60.66%
lookups	60.58%
cut undo	60.19%
somewhat	59.98%
purge method	59.97%
adding explanation	59.96%

Table 4: Key words indicating code debt

Words	probability
docs make	74.27%
verbose	70.91%
everywhere	66.32%
contrib documentation	65.66%
add documentation	62.01%
rev docs	61.98%
go doc	61.90%
documentation separate	60.65%
docs heres	60.28%
update doc	60.19%
hbase72124 doc	60.18%
doc feature	60.13%
design doc	60.08%
documentation isnt	59.92%
official docs	59.72%
impala9467 docs	59.51%
doc clarifying	59.33%
update docs	59.32%

Table 5: Key words indicating documentation debt

Words	probability
test partner	68.06%
test thanks	63.82%
tests pretty	63.49%
tests s3	63.27%
test check	59.89%
test runner	59.00%
test change	58.88%
test checked	58.58%
test	58.48%
test test	58.34%
test lot	58.29%
test deploy	58.06%
hbase52171 automatically	58.05%
hbase2600 test	58.02%
test suite	58.00%
test sure	57.80%
test set	57.80%
update docs	59.32%

Table 6: Key words indicating test debt

7 Discussion

7.1 Threats to Validity

The threats to validity in this project are implementation errors of the learning models and personal bias or inconsistencies in data annotation. The strategy adopted to minimise the risk of implementation errors was to review learning model implementations in similar academic papers [4][9][12]. As for personal bias and inconsistencies in data annotation, the strategy used was to (1) create an agreed upon definition of each type of technical debt, as well as provide a select few examples of each type and (2) to have an independent annotator to classify a subset of the annotated issues independently to the author of this paper and then use this subset to determine 'Cohen's kappa coefficient' which exhibits the inter-rater reliability of the annotations.

7.2 Key Word Comparison Between Issue Tickets and Source Code Comments

The top uni-gram indicators in the source code comments key word extraction were: todo, hack, xxx, sss, wordaround, temporary, fixme, and implement. The top bi-gram indicators were: really ugly, this needs, remove this, not sure, how to, should probably, not used, and temporary solution.
The top indicators extracted from the issue ticket data set have some overlap, for example, the words erroneous, compilcates, pollute, and risky and the words hack, word around, and ugly.
Some key differences between the two sets of indicators is that the source code comments contains many short hand or code specific phrases such as "todo" whereas the issue ticket indicators generally use more complete English.

8 Related Works

The topics broached in this study overlap in many ways with other studies conducted in recent years. In this section a brief overview of these studies and their relation to this thesis will be given.

8.1 Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt

This paper [4] focused on the detection of self admitted technical debt in source code comments along with the extraction of key words from the source code comments. The machine learning technique evaluated in this study was a maximum entropy classifier, which is another name for multinomial logistic regression. This classifier is more in line with the traditional machine learning algorithms viewed in our study and can be said to be less sophisticated.

8.2 Neural Network-based Detection of Self-Admitted Technical Debt: From Performance to Explainability

This study [12] is the closest in terms of content to our topic. It focuses on using a convolutional neural network to predict positive instances of self admitted technical debt in source code comments. It used the CNN to extract n-gram key words. The main difference here being the data set used for training and predictions. This study was used as a baseline for the hyper parameter defining for our CNN. The metrics achieved in this study were used as a comparison for our learning models.

8.3 Using Convolutional Neural Networks to Extract Keywords and Keyphrases About Foodborne Illnesses

This study [9] undertook the analysis of food-borne illnesses using neural networks. More specifically, convolutional neural networks. The CNN was then used to extract key words about food-borne illnesses. Similarly to the paper dicussed above, this paper was used as a basis for hyper parameters in the CNN and its method for key word extraction was a help in our implementation of key word extraction.

9 Conclusion

This work aims at the identification of technical debt in issue trackers using machine learning. The issue tickets were pulled from five repositories of the software projects: Camel, Hadoop, HBase, Impala, and Thrift. These repositories contained a total of around 5000 tickets. These tickets were annotated as one of the nine types: Architecture debt, build debt, code debt, defect debt, design debt, documentation debt, requirements debt, test debt, or not debt. The performance of each machine learning technique was compared by using the F1 metric of positive predictions. The seven compared methods, including, Naive Bayes classifier, K nearest neighbours classifier, decision tree classifier, support vector machines, multi-layer perception, and convolutional neural network, were used to make binary classifications on the annotated data set. A data set from previous study was used in conjunction with the results from that study to verify the techniques were performing as expected. Then the issue tracker data set was used for training and testing. The findings were that the convolutional neural network is not a large improvement over decision tree classifiers in the application of technical debt identification, achieving only 11% more F1 score for the issue tracker data set and the same F1 score for the source code comments data set.
However, the CNN has value in its interpretability. Key words can be extracted and summarised with relative ease. The main cause for trouble with key extraction was the lack of data leading to noise in the data set not being smoothed out. An example of this being one of the bi-grams indicating code debt was "2033

thrift", this is clearly specific to a few tickets and is irrelevant noise. Thus, to achieve a higher degree of interpretability, a much larger data set would be preferable. Many of the words found would be expected indicators of technical debt, such as: unsafe, erroneous, risky, and pollute. These words also have synonyms in words found in similar studies [12].

9.1 Future Work

In future work, this study could be improved in a variety of ways. Firstly, the size of the data set could be increased. Such an increase would allow for more specific predictions from trained models, such as a model that categorizes into all classes on technical debt defined in this project. It would also reduce the noise in extracted key words. This could be achieved by including more projects into the data set. The draw back would be must more manual annotation being required.
Secondly, more annotators could be used to label the data set, with over lapping annotations, to allow for a higher degree of confidence between annotators.
Finally, more convolution layers could be implemented in the model to allow for larger n-gram extractions. This would not take too much work, but would increase the training time of the model.

References

[1] *Encyclopedia of Software Engineering.* CRC Press, 2010.

[2] J. Cohen. A coefficient of agreement for nominal scales. pages 37–46, 1960.

[3] S. E. X. X. e. a. Huang, Q. Identifying self-admitted technical debt in open source projects using text mining. pages 418–457, 2018.

[4] T. N. Maldonado E, Shihab E. Using natural language processing to automatically detect self-admitted technical debt. page 1, 2017.

[5] V. C. T. S. M. R. O. S. Nicolli S. R. Alves, Leilane F. Ribeiro. Towards an ontology on technical debt. page 1, 2014.

[6] A. Samuel. Some studies in machine learning using the game of checkers. 1959.

[7] I. O. Stephany Bellomo, Robert L. Nord and M. Popeck. Got technical debt? surfacing elusive technical debt in issue trackers. page 1, 2016.

[8] I. O. Stephany Bellomo, Robert L. Nord and M. Popeck. Got technical debt? surfacing elusive technical debt in issue trackers. page 331, 2016.

[9] J. Wang. Using convolutional neural networks to extract keywords and keyphrases about foodborne illnesses. page 1, 2018.

[10] G. L. Wehaibi S, Shihab E. Examining the impact of self-admitted technical debt on software quality. in: Proceedings of the 23rd ieee international conference on software analysis, evolution, and reengineering. page 1, 2016.

[11] P. Werbos. The roots of backpropagation : From ordered derivatives to neural networks and political forecasting. 1994.

[12] X. X. D. L. X. W. J. G. XIAOXUE REN, ZHENCHANG XING. Neural network-based detection of self-admitted technical debt: From performance to explainability. 2019.