Towards Automating Code Reviews

Towards Automating Code Reviews

By Muntazir Fadhel,

A Thesis Submitted to the School of Graduate Studies in the Partial Fulfillment of the Requirements for the Degree Master of Applied Science

McMaster University
Master of Applied Science (2019)
Hamilton, Ontario ()

TITLE: Towards Automating Code Reviews
AUTHOR: Muntazir Fadhel (McMaster University)
SUPERVISOR: Dr. Sekerinski Emil
NUMBER OF PAGES: ix, 59

Abstract

Existing software engineering tools have proved useful in automating some aspects of the code review process, from uncovering defects to refactoring code. However, given that software teams still spend large amounts of time performing code reviews despite the use of such tools, much more research remains to be carried out in this area. This dissertation present two major contributions to this field. First, we perform a text classification experiment over thirty thousand GitHub review comments to understand what code reviewers typically discuss in reviews. Next, in an attempt to offer an innovative, data-driven approach to automating code reviews, we leverage probabilistic models of source code and graph embedding techniques to perform human-like code inspections. Our experimental results indicate that the proposed algorithm is able to emulate human-like code inspection behaviour in code reviews with a macro f1-score of 62%, representing an impressive contribution towards the relatively unexplored research domain of automated code reviewing tools.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, Muntazir Fadhel, declare that this thesis titled, "Towards Automating Code Reviews" and the work presented in it are my own.

Chapter 1

Introduction

1.1 Introduction

Software is an integral part of modern society. Health-care, energy, transportation, public safety, and entertainment are only some of the various aspects of life that depend on high quality software. However, high quality software is difficult to build. Engineers must not only produce functional software, but this software must be free of defects, and it must be easily adaptable to requirements that may arise over time. As a result, the software development process is largely considered to be a time consuming one, thereby creating a need for advancements in tools that make the process more automated and sustainable.

The code review is one such area of the software development process that consumes a large amount of time, and has been found to utilize up to 15% of time for developers (Ebert et al. 2019). The formal code review was first defined by Fagan in 1976 (Fagan 1976) as a software inspection practice in which source code is reviewed for correctness, often conducted by multiple reviewers. More recently, code reviews have grown in scope and have shown to have many benefits (Fatima et al. 2018; Bernhart et al. 2010; Zanjani et al. 2016) including the detection of defects early in the development phase, the enforcement of coding standards, and the transfer of knowledge between developers. Yet it is commonly accepted that formal code reviews are time-consuming and expensive (Bernhart et al. 2010; Ouni et al. 2016). Moreover, many software development teams do not perform formal code reviews as a regular task during their development process due to a lack of available time and resources (MacLeod et al. 2018).

A great deal of research has gone into automated tools that assist in performing certain aspects of code reviews including finding defects and assessing code quality. Research in this area has revolved around approaches which leverage the mathematically well-defined nature of programming languages to guide the design of such tools. Moreover, many tools for finding defects and program errors have resulted from the development of powerful abstractions, definitions, algorithms, and proof techniques. On another but related front, the presence of widely available source code through code hosting systems such as GitHub have lead researchers to tackle the development of software engineering tools using data driven approach based on machine learning techniques. The

main goal behind these tools is to use the statistical distribution of millions of available examples to build tools powered by machine learning models that can generalize well from these examples and handle noise.

However, although tooling support for code reviews based off formal and machine learning approaches have made significant progress in recent years, code reviews still add significant overhead to the development process. The extreme difficulty in evaluating and measuring non-functional attributes of code is the primary reason for this. As an example, consider the evaluation of the readability of a piece of code, measured based on how clearly the code exhibits its intention to the reader. Unlike detecting defects in code or assessing formatting conventions, assessing code for readability is not a binary problem, and requires deep insights into the semantic nature of the code in order to be successful.

Given the time consuming nature of code reviews and the large potential for automation in the code review process, this work aims to advance the state of research in code review tools. More specifically, an automated code reviewing algorithm is proposed to emulate human code inspection behaviour in code reviews. The algorithm is comprised of an innovative machine learning algorithm which utilizes both probabilistic language models and graph embeddings that are trained on more than 100,000 GitHub code review comments on GitHub. Because the proposed tool is trained over human code review comments, often in repositories where static analysis tools already exist, it is much more comprehensive than existing tools in terms of the types of issues and improvements it is able to flag.

1.2 Contribution

The main purpose of this dissertation is to introduce a novel algorithm for flagging suspicious code with the goal of reducing the amount of time developers spend performing code reviews.

The main contributions of this dissertation are:

- insight into what types of issues are typically flagged by reviewers on popular Open Source repositories when performing code reviews
- a summary of recent contributions by researchers related to tool advancements for code reviews and the identification of research gaps in this area
- a novel machine learning algorithm that classifies submitted code based on whether or not it would elicit a comment from a human code reviewer along with experimental results demonstrating its effectiveness
- a comparisons of of the proposed algorithm in with existing state-of-the-art static analysis and machine learning tools in terms of their ability to facilitate code reviews

1.3 Organization

This document is organized as follows:

Chapter 2 provides preliminary technical information on key machine learning techniques used throughout this dissertation. Specifically, material on the n-gram language model, Neural Language Model (NLM), Neural Network (NN) and Supervised Learning is presented. An overview of the GitHub software development platform from which much of the source code used in this thesis was sourced from is also provided in this chapter.

Chapter 3 outlines the problem statement this thesis will cover. First, the time consuming nature of code reviews is demonstrated through recently conducted surveys and studies on software development teams. Next, the significance of the problem is argued using the amount of potential savings to organizations that employ code reviews as part of their software development process.

Chapter 4 highlights existing research related to code reviewing in the form of defect finding and code quality tools based off machine learning and traditional static analysis approaches. The strengths and weakness of each approach are briefly discussed and contrasted with the proposed solution.

Chapter 5 conducts an introductory experiment with the goal of providing insight into the nature of discussions that take place in code reviews on GitHub. Additionally, a classifier is developed to categorize code review comments based on the type of critique delivered within the comment.

Chapter 6 expounds on the architecture and design of the proposed algorithm for performing human-like code inspections. Each step of the algorithm is described in detail, along with the rationale and design decisions made.

Chapter 7 presents an evaluation of the proposed algorithm in flagging suspicious code in code reviews based on the way a human code reviewer would. A number of metrics are presented along with a discussion on the generalizability of the study.

Chapter 8 synthesizes the results presented in Chapter 7 and discusses the effectiveness of the proposed tool in comparison to existing software engineering tools developed for maintaining code quality.

Chapter 2

Technical Background

This section presents technical background required for the comprehension of the key contributions presented in this thesis. First, a brief explanation of the GitHub software development platform is outlined. Subsequently, various machine learning and software engineering techniques employed in our solution to the research problem are briefly summarized.

2.1 GitHub and The Software Development Process

The process of building computer software and information systems is dictated by a number of different development methodologies. A software development methodology refers to the framework that is used to plan, manage, and control the process of developing an information system (Electrical and Engineering. 1992). Support for collaborative development is a key aspect of any development methodology, which is why source version control has become a key industry tool to manage increasingly asynchronous and distributed software development efforts. Tools such as Git and GitHub help developers to store the development history, share software artifacts, and collaborate with each other (Owhadi-Kareshk and Nadi 2019).

GitHub is a collaborative code hosting site built on top of the git version control system. GitHub (*GitHub* n.d.) employs a "fork & pull" model in which developers submit pull requests to get their code submissions reviewed and integrated into the project. Highly integrated social features, and the availability of metadata through an accessible Application Programming Interface (API) have made GitHub very attractive for software engineering researchers. Due to accessibility of this data, our research relies primarily on the GitHub platform. To the best of our knowledge, there are no significant differences in the type of code stored on other code hosting platforms, which would make the findings of our study applicable to such platforms as well.

2.1.1 Pull Requests

When a modification needs to be made to a project, either through the implementation of a new feature or a bug fix, a new branch is typically created which encapsulates the desired commits. Commits correspond to the set of file-level changes being introduced by the modification. Additionally, a GitHub pull request is typically opened using a branch, which will allow enable to automatically discover the relevant commits to display and merge (Gousios et al. 2014).

Next, a code review would be performed, in which submitted commits and high level features introduced by the pull request are reviewed and discussed by one or more developers. As a result of this code inspection, pull requests can be updated with new commits or closed as redundant, invalid or duplicated. When pull requests are updated, the contributor creates new commits in the branch, after which the process of reviewing the code is repeated. Finally, the pull request can be merged once the the code review is completed and the quality of code in the pull request is considered high enough to be integrated into the main repository. A typical pull request work flow is illustrated in Figure 2.1.

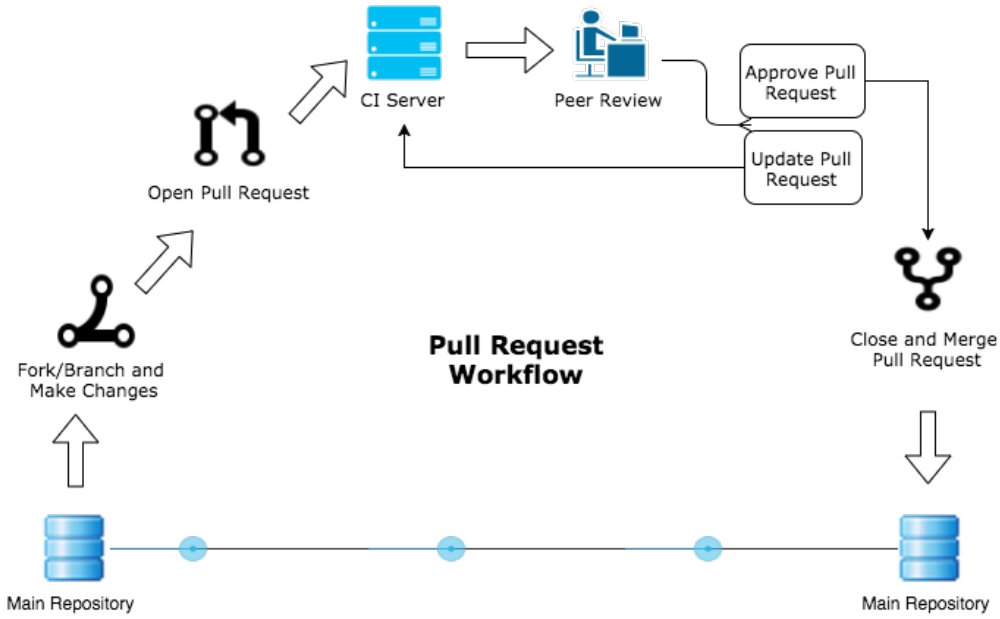

FIGURE 2.1: Typical pull request flow on GitHub

2.1.2 Code Review Comments

As part of the code inspection process, GitHub enables reviewers to leave comments on portions of the unified diff of a GitHub pull request. A unified diff, as illustrated in Figure 2.2, outlines the differences between two files, often between an original file and a

proposed version of the file. In this format, any lines that have undergone modification are depicted beside unchanged lines both before and after. Context to the patch is provided by the inclusion of unchanged lines and serve as a reference to locate the patch's position in a modified file.

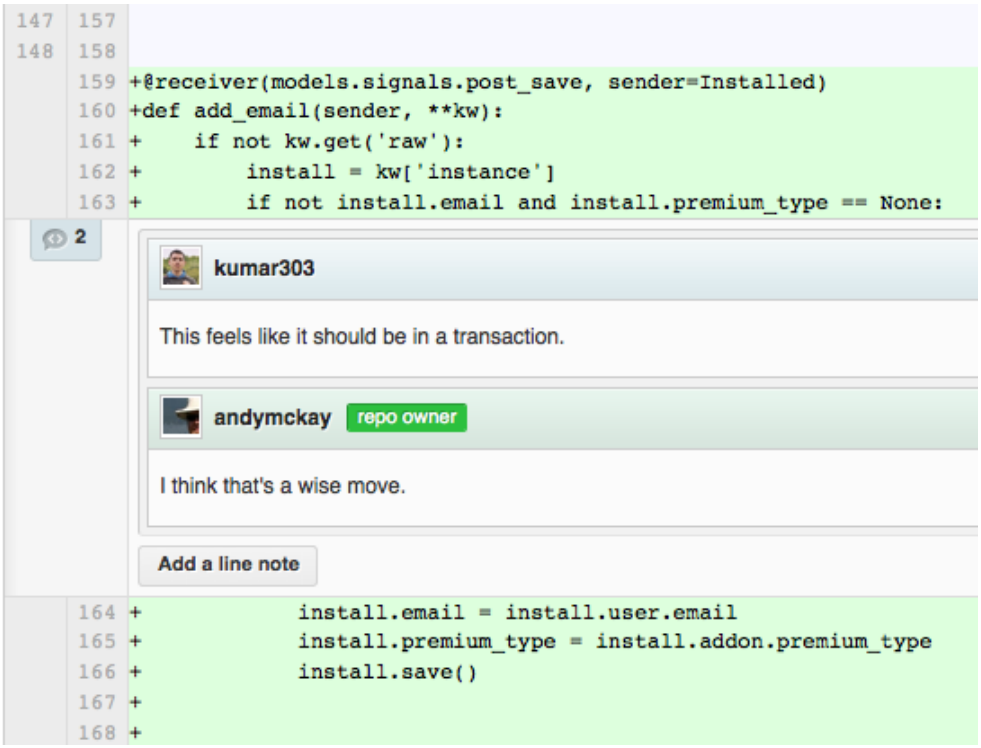

FIGURE 2.2: Sample code review comment left on a unified diff

2.2 Graph Embeddings

Machine Learning models typically operate over numerical data which makes the process of learning source code non-trivial. In the development of an automated code reviewing tool for example, snippets of code cannot simply be "thrown" into a neural network with the the hopes of being able to classify code that is well written as opposed to poorly written, efficient versus inefficient. Rather, these concepts must be represented abstractly using numerical models and data structures in which properties and relationships of interest are preserved from the original model. These abstract representations can then be used in machine learning tasks, where predictions about the original concept are made. The depth and complexity of this problem has inspired a great deal of research dedicated to investigating the representation of real world concepts as data structures suitable for machine learning activities.

2.2.1 Graphs

Graphs can be used to model a wide diversity of real-world scenarios including social networks, chemical compounds, product review systems and source code. Analyzing these graphs provides potential solutions for difficult problems that naturally exhibit graph-like structures, and has recently received significant attention by the research community. Effective graph analytics can benefit many applications, such as node classification, node clustering, node retrieval/recommendation and link prediction (Cai et al. 2018).

Formally, a graph is $G = (V, E)$, where $v \in V$ is a node and $e \in E$ is an edge. G is associated with a node type mapping function $f_v : V \rightarrow T^v$ and an edge type mapping function $f_e : E \rightarrow T^e$. T^v and T^e denote the set of node types and edge types, respectively. Each node $v_i \in V$ belongs to one particular type, i.e., $f_v(v_i) \in T^v$. Similarly, for $e_{ij} \in E, f_e(e_{ij}) \in T^e$.

Abstract Syntax Trees

An Abstract Syntax Tree (AST) is a type of graph in which the abstract syntactic structure of source code written in a programming language is expressed using a tree representation. Each node of the tree represents a code entity found in the actual source code. The syntax is "abstract" because it does not contain every syntactic detail appearing in the code. As an example, grouping parentheses are not explicitly represented in an AST, and a syntactic construct like an if-condition-then expression may be represented by a single node with multiple branches. Figure 2.3 depicts a sample AST.

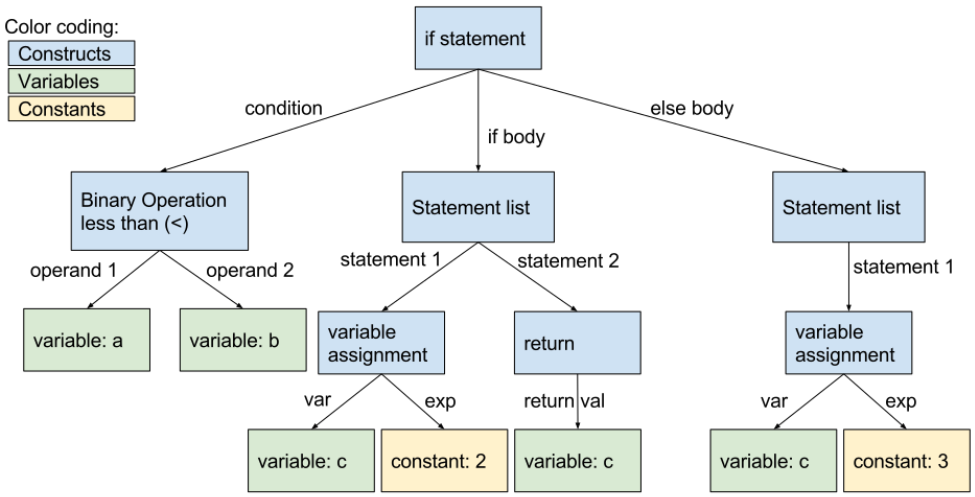

FIGURE 2.3: Sample Abstract Syntax Tree

2.3 Supervised Learning

Whenever a machine is required to perform a function or procedure too complex to develop by hand, learning can be a useful alternative. For example, even while ignoring time and budget constraints, it is not straightforward to develop a computer program to recognize speech. However, although it may be expensive, a simpler approach would start with the collection of a large number of sample speech signals with their annotated content. Next, a supervised learning algorithm can be leveraged to map the input-output relationship present in the training data.

More specifically, a supervised learning algorithm attempts to produce an inferred function from the training data, also referred to as a classifier. This function should predict the correct output value for any valid input object. As a result, the learning algorithm must be able to apply and extend insights from the training data to new, unseen data in a robust way.

The first step in a typical supervised learning machine learning experiment is the observation of a phenomenon or random process which gives rise to a set of training examples of the form:

$$\{(x_1, y_1), ..., (x_N, y_N)\} \qquad x_i \in X, y_i \in Y$$

The output or annotation space Y can either be discrete or real valued in which case we have either a classification or a regression task respectively. We will assume that the input space X is a finite-dimensional real space R^d where d is the number of explanatory variables.

The next step is to model this phenomenon by attempting to make a causal link $f : X \to Y$ between the observed inputs $\{x_i\}_{i=1}^N$ from the input space X and their corresponding observed outputs $\{y_i\}_{i=1}^N$ from the annotation space Y. In a classification task the hypothesis function f is commonly referred to as a decision function whereas in a regression task it is simply called a regression function.

The hypothesis function has two desired properties. First, it must minimize some measure of error over the observed training set to ensure that a causal link is in fact extracted from the observed data. Second, it must also minimize some measure of error over the observed training set to avoid over-fitting the training set with a complex function that is unable to generalize or accurately predict the annotation of a test example.

2.4 Artificial Neural Networks

The inventor of one of the first neurocomputers, Robert Hecht-Nielsen, defined a Neural Network (NN) as "...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response

to external inputs." (Chen and Sheu 1994). To understand how Artificial Neural Network (ANN)s actually work, we analyze the functioning of the neural network inside the human brain at a high level.

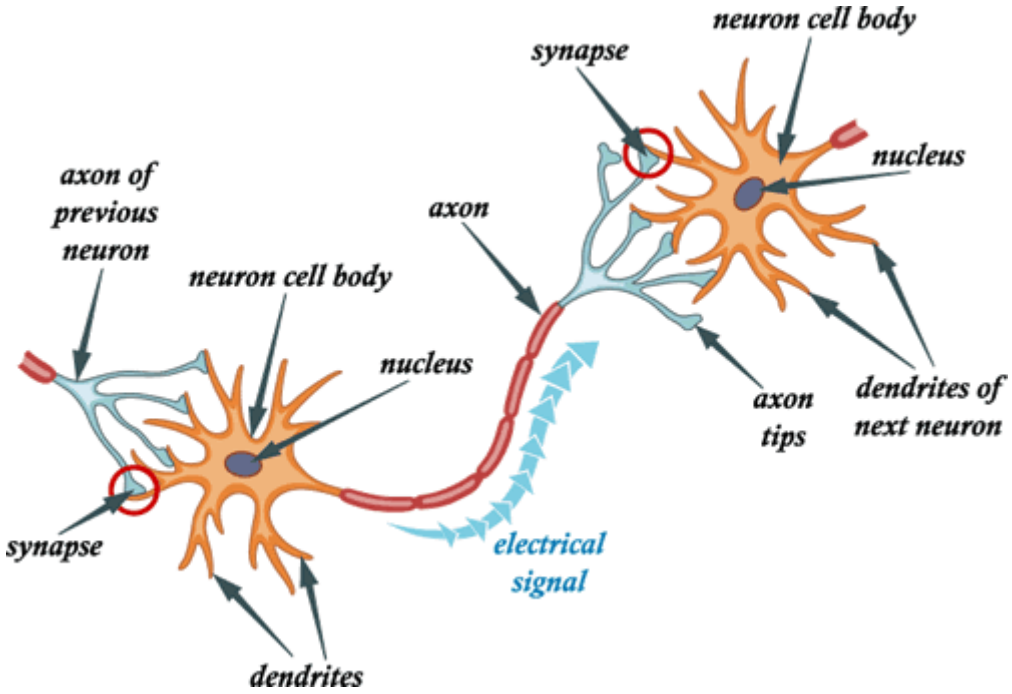

FIGURE 2.4: A visualization of the communication between neurons. Image sourced from simplebiologyy.blogspot.com

In Figure 2.4, the basics of communication between neurons are illustrated. Every neuron has multiple processes called dendrites, which serve as receivers of the information from other cells including neurons, sensory receptors and muscle cells. The received information is transferred as an electrical signal through the cell body to an axon, which is responsible for transferring the information to other cells. The transfer site is called synapse and the transfer itself is handled using chemical processes between axon terminals of the presynaptic neuron and receptors on the end of dendrites of the postsynaptic neuron.

In ANNs, the neural network described above is modeled as a graph, where nodes are the neurons and edges are the synapses (plus dendrites and axons). Every node contains three basic components, which are depicted in Figure 2.5:

- **Weight vector**: vector of synapses, assignment of weights to edges of the graph
- **Transfer function**, which computes the sum of signals multiplied by concrete weights
- **Activation function**: maps the result of the net input to the output of the neuron

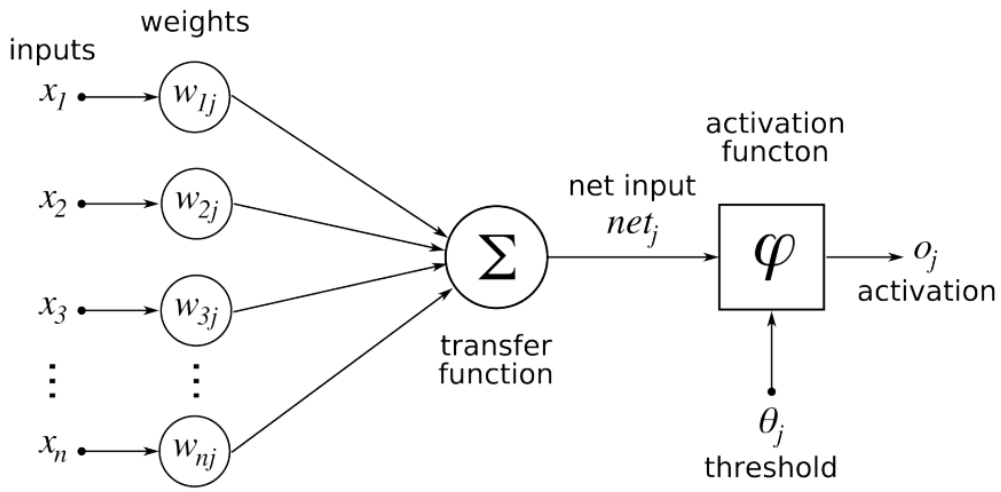

FIGURE 2.5: Artificial neuron model. Image sourced from commons.wikimedia.org

An ANN typically contains multiple layers, of which the first and last layer are referred to as the input and output layer respectively. The input layer is the part of the network in which external input enters the network by means of signals. Neurons of this layer serve to process the external input and transfer it further. After the input layer there may be one or more hidden layers, whose neurons behave like those illustrated in Figure 2.5.

The number of hidden layers and nodes in each layer are some of the parameters that need to be configured based on the particular problem at hand. With an increasing number of hidden nodes, an ANN can learn patterns and relationships present in more complex data, but are more difficult to train and prone to over-fitting as a result. Over-fitting occurs when a NN successfully learns the data it is trained on but cannot generalize for previously unseen data. Lastly, the output of the neurons of the output layer is propagated into the final output of the whole network and represent the results of the computation.

A Feedforward Neural Network (FFNN) is a type of ANN that restricts the travel of signals between neurons to one direction. As a result, there are no feedback loops in the network, and the output of any given layer does not affect the same layer it originated from. Rather, each output of any given layer must connect to the next layer in the network.

Back Propogation

Training a FFNN using a supervised learning strategy closely resembles the manner in which a human might learn to classify unfamiliar objects. At first, the individual does not

know what classifications are correct for a given object and has to learn them. One way to do this would be through the cyclical process of assigning a classification, and using the knowledge of the amount of error in that classification to influence future predictions. The back-propagation (Rumelhart et al. 1986) algorithm operates in a similar fashion. In this method, the goal is to find the parameters of the model (the weights of inputs for neurons) that minimize the error of the output for the data set over a fixed number of iterations. This algorithm requires a training set in which there exists an associated model output for every input. In each training iteration, the algorithm has two major phases:

1. **Forward phase** – After the outputs of all the neurons in the network are computed and compared to the real output, the final error is calculated.
2. **Backward phase** – The calculated error is propagated back through the neural network, and the weights are adjusted towards minimizing the error rate.

The back propagation method leads to minimization of the error function to a minimum, which is not necessary the global one. The speed of the training can be changed in order to prevent the over learning with every single incorrectly recognized input.

2.5 Language Modeling

The goal of Statistical Language Modeling is to build a model that can estimate the distribution of natural language as accurately as possible (Li et al. 2013). Natural language, also referred to as ordinary language, simply refers to a language that has developed naturally in use as opposed to an artificial language which is generated. A high quality language model is considered to be an integral component of many systems for language technology applications, such as speech recognition, machine translation, text summarization, handwriting recognition, and many more (Oualil et al. 2017). The history of language modeling begins in the 20th century when Andrei Markov used language models to model letter sequences in works of Russian literature (Basharin et al. 2004). Additionally, Claude Shannon's models of letter and word sequences constitute another famous application of language models, which he used to illustrate the implications of coding and information theory (Shannon 1951).

A *language model* is formulated as probability distribution $p(s)$ over string s that attempts to reflect how frequently the string s occurs as a sentence (Indurkhya and Damerau 2010). As an example, for a language model describing natural language, it is possible $p(Can) \approx 0.001$ since one out of every thousand sentences likely starts with the word "Can". On the opposite end of the spectrum, it is likely that $p(Octopus\, eats\, beats) \approx 0$ since it is extremely improbable that anyone would speak such a phrase.

The probability of a word sequence $W = w_0, w_1, w_2, ...w_N$ can be decomposed into cascading probabilities using the chain rule (Huang et al. 2001). The overall probability can be written as products of conditional probabilities for each word given its history.

Language models are trained on a set of training corpus to estimate this probability, which is given as below:

$$p(W) = p(w_0, w_1, w_2, w_3 ... w_N) = \prod_{i=1}^{N} p(w_i | w_{i-1} ... w_1, w_0) \tag{2.1}$$

In this section, a technical overview of the two most popular types of language models, n-gram language models and Neural Language Model (NLM)s, is provided.

2.5.1 N-gram Language Models

N-gram language models constitute the most widely used type of language model (Siivola and L. Pellom 2005). Equation 2.1 demonstrated that the probability of a word sequence W can be represented as the products of conditional probabilities for each word given its history. This poses a serious problem as the number of parameters for such a model is extremely large, even when the application under consideration does not consist of a large sized vocabulary. As a result, all combinations of word w_i and history $(w_{i-1}, ...w_1, w_0)$ cannot be stored and calculated in a practical way. Thus, we introduce n-gram models by considering the case $n = 2$; these models are called bigram models. Bigram models are based on the approximation that the probability of a word depends only on the identity of the immediately preceding word, resulting in the following equation (Goodman 2001).

$$p(s) = \prod_{i=1}^{l} p(w_i | w_0 ... w_{i-1}) \approx \prod_{i=1}^{l} p(w_i | w_{i-1}) \tag{2.2}$$

This idea is also referred to as the n-gram history. To make $p(w_i|w_{i-1})$ meaningful for $i = 1$, the beginning of a sentence is often padded with a distinguished token <BOS>, making the first word w_0 equal to <BOS>. In addition, to make the sum of the probabilities of all strings $\sum_s p(s)$ equal to 1, a distinguished token <EOS> is placed at the end of sentences. The insertion of <BOS> and <EOS> tokens ensure that the sum of the probabilities of all strings of a given length is 1, and the sum of of the probabilities of all strings is then infinite. As an example, to calculate $p(Mark\, wrote\, a\, book)$, the following steps would be taken:

$$p(Mark\, wrote\, a\, book) = p(Mark| < BOS >)p(wrote|Mark)p(a|wrote)p(book|a)p(< EOS > |book)$$

To estimate $p(w_i|w_{i-1})$ the number of times the word w_i appears given that the last word is w_{i-1}, the frequency of bigram $w_{i-1}w_i$ is counted and normalized. If we set

$c(w_{i-1}w_i)$ to refer to the number of times the bigram $w_{i-1}w_i$ occurs in the given text, then we have

$$p(w_i|w_{i-1}) = \frac{c(w_{i-1}w_i)}{\sum c(w_{i-1}w_i)} \tag{2.3}$$

The text available for building a model is called *training data*. For n-gram models, millions of tokens typically comprise the training data used to train the model. The estimate for $p(w_i|w_{i-1})$ as given in Equation 2.2 is called the *maximum likelihood* estimate of $p(w_i|w_{i-1})$, as this assignment of probabilities generates the bigram model that associates the greatest probability to the training data of all possible bigram models.

For n-gram models where $n > 2$, the probability of a word is conditioned on the identity of the last $n-1$ words, as opposed to the identity of just the preceding word. Generalizing Equation 2.1 to $n > 2$, we get

$$p(s) = \prod_{i=1}^{l+1} p(w_i|w_{i-n+1}^{i-1}) \tag{2.4}$$

where w_i^j denotes the words $w_i ... w_j$ and where we take w_{n_2} through w_0 to be $< BOS >$ and w_{l+1} to be $< EOS >$. To estimate the probabilities $p(w_i|w_{i-n+1}^{i-1})$, the analogous equation to Equation 2.2 is

$$p(w_i|w_{i-n+1}^{i-1}) = \frac{c(w_{i-n+1}^{i})}{\sum c(w_{i-n+1}^{i})} \tag{2.5}$$

In practice, the largest n that is widely use is $n = 3$, which is referred to as a trigram model. The words w_{i-n+1}^{i-1} preceding the current word w_i also known as the model's *history*, while the value n of an n-gram model is referred to as its order.

As an example, we let our training data S be comprised of three sentences:

1. "Mark wrote Moby Dick"
2. "Mary wrote a different book"
3. "She wrote a book by Claude"

The following steps demonstrate the calculation of $p(Mark\, wrote\, a\, book)$ for the maximum likelihood bi-gram model:

$$p(Mark| < BOS >) = \frac{c(< BOS > \; Mark)}{\sum c(< BOS > \; w)} = \frac{1}{3}$$

$$p(wrote|Mark) = \frac{c(Mark\, wrote)}{\sum c(Mark\, w)} = \frac{1}{1}$$

$$p(a|wrote) = \frac{c(wrote\ A)}{\sum c(wrote\ w)} = \frac{2}{3}$$

$$p(book|a) = \frac{c(a\ book)}{\sum c(a\ w)} = \frac{1}{2}$$

$$p(< EOS > |book) = \frac{c(< EOS >\ book)}{\sum c(book\ w)} = \frac{1}{2}$$

These calculation results in the following equation:

$$p(Mark\ wrote\ a\ book) = \frac{1}{3} \times \frac{1}{1} \times \frac{2}{3} \times \frac{1}{2} \times \frac{1}{2} \approx 0.06$$

Smoothing

Continuing on the example from the previous section, consider the sentence "Claude wrote a book". The probability of such a string would be calculated as:

$$p(wrote|Claude) = \frac{c(Claude\ wrote)}{\sum c(Claude\ w)} = \frac{0}{1} = 0$$

A probability of zero is clearly an underestimate for the probability of "Claude wrote a book", as there is certainly some chance that this sentence occurs. In the context of analyzing code review comments, such behaviour is undesirable since it may often be the case where code sequences contained in a GitHub pull request consist of new tokens that have never been encountered by our probabilistic model of source code, but look similar to *known* sequences. As a result, a zero probability score would not be an accurate probability for such sequences and severely limit the effectiveness of our model.

To address this problem, smoothing techniques are typically used. These techniques seek to adjust the maximum likelihood estimate of probabilities to produce more accurate probabilities. Not only do smoothing methods help prevent zero probabilities from occurring, but they also improve the accuracy of the model as a whole (Indurkhya and Damerau 2010). This is because smoothing has the potential to significantly improve estimation when estimating probabilities from a few counts (Chen and Goodman 1996).

As an example, a common smoothing approach is to assume that each bigram occurs once more than indicated by its actual count, yielding

$$p(w_i|w_{i-1}) = \frac{1 + c(w_{i-1}w_i)}{\sum[1 + c(w_{i-1}w_i)]} = \frac{1 + c(w_{i-1}w_i)}{|V| + \sum c(w_{i-1}w_i)}$$

where V is the collection of all words in the Vocabulary. Using the previous vocabulary outline at the end of Chapter 2.5.1 in which $|V| = 11$, we now have the following for the sentence "Mark wrote a book"

$$
\begin{aligned}
p(Mark\,wrote\,a\,book) &= p(Mark| < BOS >)p(wrote|Mark)p(a|wrote)p(book|a)p(< EOS > |book) \\
&= \frac{2}{4} \times \frac{2}{12} \times \frac{3}{14} \times \frac{2}{13} \times \frac{2}{13} \\
&\approx 0.0001
\end{aligned}
$$

In other words, the sequence "Mark wrote a book" is estimated to occur approximately once every ten thousand sentences. This estimation is much more reasonable than the maximum likelihood estimate of 0.06, or about once every seventeen sentences.

Concluding Thoughts on n-grams

The most significant advantages of models based on n-gram statistics are speed as a result of the storing of probabilities in pre-computed tables, and generality since these models can be applied to any domain effortlessly provided there exists some training data. N-gram models are still the most widely used type of language models because more accurate language models currently in existence are more complex to use and configure, and provide minimal increases in precision.

Despite the wide popularity of the the n-gram language model however, it suffers from a number of issues, the first of which is data sparsity. That is, in order to properly estimate probabilities of all relevant n-grams, sufficient coverage of all possible string sequences is required, which may not be possible in all scenarios. For example, an n-gram language model of order three would require $20000^{31} = 8e^{121}$ free parameters to accurately estimate a moderate vocabulary containing 20,000 words.

Another weakness of the n-gram model lies in the n'th order Markov assumption as outlined in Equation 2.1. Specifically, the predicted word probability is only dependent on the preceding $n-1$ words, thereby ignoring any possibly relevant, longer range context. This is problematic if large amounts of training data is not available, as many of the patterns from the training data cannot be effectively represented by n-grams, which would result in decreased training ability.

Consider the following example:

`THE SKY ABOVE OUR HEADS IS BLUE`

Here, the word BLUE directly depends on the previous word SKY. We can also conclude that there exists a large number of possible variations of words between BLUE and SKY that would not break this relationship. Additionally, this number of variations can increase exponentially as the distance between the two words is increased as well. With this said, an n-gram model with $n = 4$ is unable to efficiently model such common patterns in the language. At first, it may seem like the solution to this problem is

to simply increase the n-gram size; however, this is often not feasible due to the data sparsity problem mentioned earlier, which is further intensified as the n-gram size is increased.

Another weakness of n-gram models lies in their inability to capture the similarity of individual words. For example, consider the statement:

```
MEETING WILL BE ON <DAY OF WEEK>
```

Given that only two or three variations of this sentence are present in the training data, such as

```
MEETING WILL BE ON MONDAY
```

and

```
MEETING WILL BE ON TUESDAY
```

the n-gram models will not be able to assign meaningful probability to unseen but similar sequences such as

```
MEETING WILL BE ON FRIDAY
```

even if the days of the week appeared in the training data frequently enough to discover that there is some similarity among them. In the next section, we introduce Neural Network based language models which are able to overcome these two weaknesses of n-gram language models at the cost of significantly increased model training times.

2.5.2 Neural Language Models

Having briefly discussed the ANN in Section 2.4, we now demonstrate how it is incorporated and used in an NLM. In this section, two popular variants of NLMs are reviewed, Feedforward Neural Language Models (FNLM)s and Recurrent Neural Language Models (RNLM)s.

Feedforward Neural Language Models

In Section 2.5.1, it was demonstrated that the probability of a word sequence can be represented as the products of conditional probabilities for each word given its history. However, the number of parameters for this model is prohibitively large, which is why n-gram models make the approximation that the probability of a word depends only on the identity of a small, immediately preceding set of words. Unlike n-gram language models, an FNLM is based on neural networks; however, they are similar in that they rely on the n-gram approximation to determine word probabilities.

As a result, an FNLM utilizes an FFNN based architecture that accepts a sequence of words with the last word missing as input. Next, the FFNN is trained to output a probability distribution over the all possible words for the empty position. To accomplish

this, a window of fixed length, designed to capture n-grams in the corpus, is slid over the entire corpus. This process enables the FNLM to calculate the probability for each word in the corpus given its context, which in turn allows for the calculation of probabilities over word sequences.

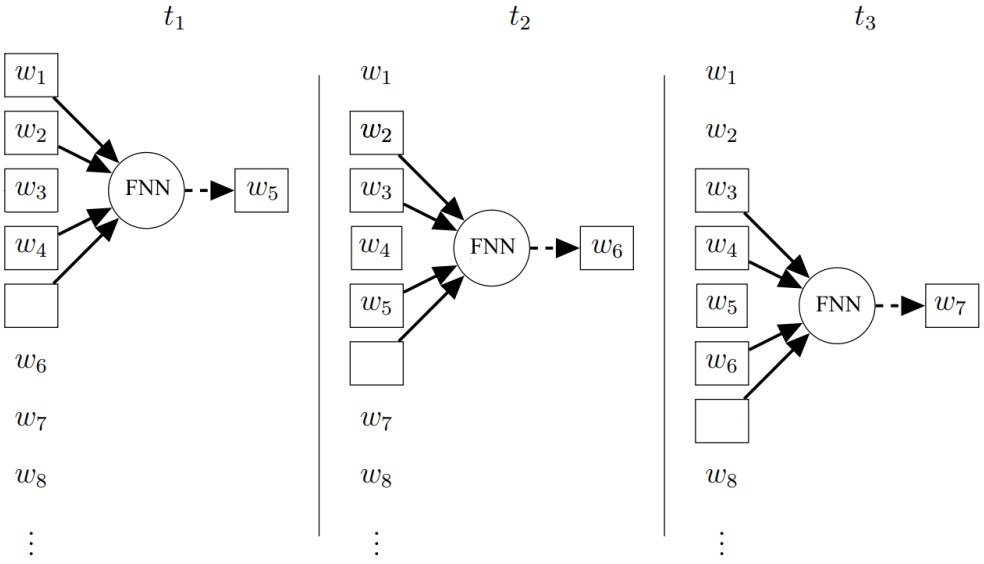

FIGURE 2.6: The learning of the FNLM, with the last word of the n-gram being the predicted one.

However, given that the input format to an ANN is restricted to numerical input, numerical representation of strings is a remaining problem that needs to be addressed. This problem is commonly handled using the one-hot encoding algorithm (Prusa and Khoshgoftaar 2017). In this method, the entire vocabulary is enumerated such that each word is assigned a unique number. Next, a one-hot vector is constructed for each word in the corpus. This vector, R^v, has a size v representing the length of the entire vocabulary, and is filled with zeros, except for the position indicated by the position of the word in the dictionary. In this case, the position is filled with a one. If by $w_{i,j}$ we denote the j-th position in the vector representing the i-th word, then

$$w_{i,j} = \begin{cases} 1, \text{if } j = 1, \\ 0, \ otherwise. \end{cases} \tag{2.6}$$

In the case when unknown words are encountered, a special word is designated as an unknown word token. During the training process, any words that are not in the FNLMs vocabulary are simply replaced by this token.

It is important to note that one-hot encoded vector representations of words are extremely sparse vectors, making the proposed neural language model susceptible to same data sparsity issues faced by n-gram language models. To overcome this problem,

the vectors of the $n-1$ previous words are fed to a linear projection layer, using a shared matrix E. As a result, each word w in the input vocabulary layer V^{in} can be represented using a low-dimension vector, which is also known as word embeddings in literature (Kusner et al. 2015).

$$\mathbf{f}(w) = \mathbf{E}^T r^w = \mathbf{e}^T_{\phi^{in}(w)} \tag{2.7}$$

In Equation 2.7, $\mathbf{e}_i$ represents the i-th row vector of projection matrix $\mathbf{E}$ and $\theta^{in}(w)$ is the index of word w in the input layer. The projection layer matrix E in the input layer is shared among words in history. As a result, the number of model parameters increases linearly with a ratio of the projection layer size with the growth of the n-gram order, instead of increasing exponentially as in standard n-gram language model (Williams et al. 2015). Additionally, a major benefit of the FNLM over the n-gram language model is its ability to represent words as vectors in a continuous space, where the distance between words is measured in the vector space. Words found in a similar context cluster together in vector space, resulting in words like "Monday" and "Tuesday" being close together in the vector space. FNLMs can be trained efficiently using back propagation algorithm (Werbos and John 1974), which is outlined in Section 2.4.

Recurrent Neural Language Models

Although FNLMs are able to overcome the data sparsity issue faced by n-gram models by projecting words into a low-dimension, continuous space, they still adopt a fundamental assumption made by n-grams. This assumption, which considers only $n-1$ words in determining the probability of the next word in a sequence, discards longer context information. Recurrent Neural Language Models RNLMs, based on Recurrent Neural Network (RNN)s, are able to improve upon this situation by capturing long range sequences and do not make this assumption.

In an RNLM, the last word encountered, as opposed to the previous $n-1$ words is presented and utilized as input to the model. However, a recurrent connection between hidden and input layers is maintained. As a result, RNLMs are able to represent the complete, non-truncated, history of all previous seen words at each computation (Williams et al. 2015). More specifically, a one-hot coding vector is used for the input word w_{i-1} in the input layer as in the case of FNLMs; however, an additional continuous vector v_{i-2} is used to capture the entire sequence history through the use of a recurrent connection. This vector is also provided as input to the network, which allows the probability of any given sentence W in a RNLM to be written as

$$
\begin{aligned}
P(W) &= \prod_{i=1}^{N} P(w_i|w_{i-1}, ...w_1, w_0) \\
&= \prod_{i=1}^{K} P(w_i|w_{i-1}, w_0^{i-2}) \\
&= \prod_{i=1}^{K} P(w_i|w_{i-1}, \mathbf{v}_{i-2}) \\
&= \prod_{i=1}^{K} P(w_i|\mathbf{v}_{i-1})
\end{aligned}
$$

In this way, an RNLM is able to solve the short term history issues present in n-gram and FNLMs. However, like FNLMs, a RNLM is notoriously difficult and time consuming to train in comparison to n-gram language models (Williams et al. 2015). As a result, it is common to limit the vocabulary size in an NLM to a subset of the most commonly occurring words in the corpora to reduce training times.

Evaluating Language Model Performance

The most common metric for evaluating a language model is the probability that the model assigns to test data, or the derivative measures of cross-entropy and perplexity (Indurkhya and Damerau 2010). Recall that we can calculate the probability of a sentence $p(s)$ for an n-gram model using Equation 2.4. Then, for a test set T composed of the sentences $(t_1, ..., t_{l_T})$, the probabilities of the test set $p(T)$ can be computed as the product of the probabilities of all sentences in the set:

$$
p(T) = \prod_{i=1}^{l_T} p(t_i) \tag{2.8}
$$

The measure of cross-entropy can be motivated using the well-known relation between prediction and compression (Bell et al. 1989; Cover and Thomas 2006). In particular, given a language model that assigns probability $p(T)$ to a text T, we can derive a compression algorithm that encodes the text T using $-log_2 p(T)$ bits. The cross-entropy $H_p(T)$ of a model $p(w_i|w_{i-n+1}^{i-1})$ on data T is defined as

$$
H_p(T) = -\frac{1}{W_T} log_2 p(T) \tag{2.9}
$$

where W_T is the length of the text T measured in words. Given a repetitive and highly predictable corpus of documents, a good language model captures the regularities in the corpus. Additionally, such a model will not find a new document drawn from the original corpus particularly surprising, or "perplexing". In Natural Language Processing (NLP), this idea is captured by a measure called perplexity $PP_p(T)$ of a model p, which is the reciprocal of the (geometric) average probability assigned by the model to each word in the test set T, and is related to the cross-entropy by the equation

$$PP_p(T) = 2^{H_p(T)} \tag{2.10}$$

Typical perplexities yielded by n-gram models on English text range from about 50 to almost 1000, depending on the type of text. We use the perplexity measurement to understand the naturalness of source code. Code that is highly predictable, and thus highly "natural", is given a lower perplexity score. On the contrary, a language model would associate higher perplexity scores with unpredictable lines of source code.

Chapter 3

Problem Statement

Code reviewing is a an important practice for software quality assurance, which has been widely adopted in both open source and commercial software projects (Tao and Kim 2015; Bavota and Russo 2015; Boehm and Basili 2001; Mäntylä and Lassenius 2009a; Barnett et al. 2015). Additionally, the active participation of developers in code reviews have shown to reduce the number of defects found in software and improve the software quality (Cohen 2006; Mcintosh et al. 2016) overall. The benefits of code reviews can be summarized in the following:

- **Finding Defects**: Code reviews encourages developers to assess the code logic from a different perspective to the code submitter, which can often uncover the presence of bugs.
- **Maintaining Code Quality**: Code reviews facilitate comments or changes about code in terms of readability, commenting, consistency, dead code removal, etc. with respect to project standards. These issues typically are not related to program correctness or bugs.
- **Explore Alternative Solutions**: There are often many solutions to a problem. Code reviews enable developers to discuss why a certain approach was chosen over another one.
- **Knowledge Transfer**: Code reviews distribute the knowledge of the code base among developers, which facilitates the development of higher quality, defect-free code.

However, code reviews also incur cost on the software development projects as they decrease the speed of the overall development process (Pascarella et al. 2018; Bird et al. 2015). This slowdown can be understood through the many responsibilities of a code reviewer. First, developers must gain an understanding of the existing code that is being modified, and of the impact of the modifications being proposed. Additionally, they must also communicate these errors along with possible solutions to the code submitter. Both of these activities require code reviewers to spend a significant amount of time carefully scrutinizing the impact and costs associated with each line of code being introduced and removed in a submission.

Numerous studies have shown that the time spent by a developer reviewing code is non-negligible and may take up to 10-15% of the overall time spent on software development activities (MacLeod et al. 2018). The merge of a code change in the repository can be even further delayed if the reviewers experience difficulties in understanding the change, i.e., they are not certain about its correctness, run-time behavior and impact on the system (Cohen 2006; Bird and Bacchelli 2013; Tao et al. 2012; Sutherland and Venolia 2009; LaToza et al. 2006). Additionally, a large scale study conducted at Microsoft indicated the the biggest challenge faced during code reviews was a lack of time (MacLeod et al. 2018). Furthermore, Codacy recently conducted a survey of 680 companies about their code reviewing practices and found that the rigorous nature of code reviews requires developers to spend five hours per week on average reviewing code (*Codacy* n.d.). Assuming an average developer salary of $80,000 per year, a mid-sized company with 100+ developers would spend over a million dollars per year on code reviews.

Surprisingly, code reviewers still feel that they do not have enough time to complete code reviews properly. This sentiment is further supported by the study conducted by Codacy which revealed that 45% of developers felt a lack of time to be the real obstacle in performing code reviews (*Codacy* n.d.). Additionally, a large scale study at Microsoft found that managing of time constraints was the the third highest ranked challenge facing during code reviews (MacLeod et al. 2018).

Finally, we note that code reviews involve developers who often have differing schedules and priorities. As a result, a large amount of time is often wasted in the back and forth communication between developers at differing times that are convenient for each of them. A study on code inspections by Votta found that 20% of the interval in a "traditional code inspection" is wasted due to scheduling (Votta 1993). Additionally, another study conducted at Microsoft found that receiving timely feedback is the largest challenge faced in doing code reviews (MacLeod et al. 2018). Ironically, this same study indicated that 39% of participants acted as a code reviewer daily in the previous week. Therefore, despite spending a significant time on performing code reviews already, developers feel that they need even more time to complete them satisfactorily.

In summary, while the benefits of code reviews are well established, they are expensive to companies as developers must be compensated for the significant amount of time they spend conducting code reviews. Concurrently, the time-consuming nature of code reviews prevents developers from completing them properly, which prevents organizations from enjoying the entire range of benefits that the code review process has to offer. This results in further costs to organizations in the form of lower software quality and larger post release defects. These two major challenges with performing code reviews highlight the importance of finding ways to reduce the amount of time taken to perform them.

Chapter 4

Related Work

Chapter 3 outlined the objectives of code reviews, in which the activities of uncovering defects and maintaining code quality were listed. To the best of our knowledge, research in computer science has witnessed a much greater emphasis on these two areas in comparison to the other objectives of code reviews. In this section, an overview of this research is presented.

In the development of defect finding and code quality tools, research has been dominated by a theory-first approach that leverages the well-defined properties of programming languages. Software artifacts are analyzed as mathematical objects when evaluating software tools this approach. This approach has a great deal of research in the field of programming languages due to its ability to prove facts and properties of software programs. To this extent, many elaborate abstractions, definitions, algorithms, and proof techniques have been developed, which have led to effective static analysis tools for program analysis, defect finding, and code refactoring (Cousot et al. 2005; Bessey et al. 2010; Clarke et al. 2003).

A comprehensive study was recently carried out in which the capabilities of widely used Java static analysis tools including PMD (*PMD* n.d.), FindBugs (*FindBugs* n.d.), Codacy (*Codacy* n.d.) and SonarQube (*SonarQube* n.d.) were compared. In general, these tools were shown to detect defects of various types including those related to concurrency, maintainability, style, and security. However, it was also shown that the individual effectiveness of such tools in terms of the types and number of defects they are able to uncover varied greatly. This highlights a major weakness of static analysis tools, since teams must maintain multiple tools as part of their software delivery process in order to get the most comprehensive performance. This maintenance process can be time consuming as developers must continuously update such tools to support evolving programming languages as well as heavily configure each to output only those warnings and suggestions of interest.

Even with the best static analysis tools available, it is still unclear how much time they are saving code reviewers. To this end, Singh et al. explored the overlap between review comments on GitHub pull requests and warnings from the PMD static analysis tool (Singh et al. 2017). In an empirical study of 274 comments from 92 pull requests, it was observed that PMD overlapped with nearly 16% of the review comments, indicating a

minor but significant time benefit to the reviewer if static analyzers would have been used prior to pull request submission. Another study applied state-of-the-art bug detectors to a set of almost 600 real world bugs and found that 95% of the bugs were missed (Habib and Pradel 2018). This was primarily the result of the existence of various bug patterns exist, each of which requires a different bug detector. These bug detectors must be manually created, typically by program analysis experts, and require significant fine-tuning to find actual bugs without overwhelming developers with spurious warnings.

The increasing amount of successful, widely used, open source software systems have enabled new data-driven tools which have similar aims to those of static analysis tools. Not only do software systems like GitHub and BitBucket release source code publicly, they also provide meta-data about the users, defect-fixes and revision procedures through widely accessible Application Programming Interface (API)s. In this data-driven approach, the statistical distributional properties estimated over representative software corpora influences the design of development tools. Moreover, thousands of well-written software projects are leveraged to uncover patterns, which are used to enhance existing software development tools and algorithms.

Probabilistic models of source code, designed to estimate a distribution over all possible source files, constitute a data-driven approach rooted in an emerging area of machine learning and statistical Natural Language Processing (NLP). Preliminary work suggests that the probability assigned to source code by probabilistic models can indicate defective code regions (Hindle et al. 2012). More specifically, code containing defects tends to have lower probability than non-defective code. As with all modelling methods, however, probabilistic machine learning models simplify the assumptions regarding the modelled domain. These assumptions make the models suitable for learning large amounts of code, but introduce error. In the case of uncovering defects, probabilistic models tend to consider rare, but correct, code defective (Hindle et al. 2012). Due to such imprecision, it is unclear whether or not incorporating such tools as part of the code review process is worthwhile.

Besides probabilistic models of source code, machine learning techniques over source code embeddings represent another thriving area of research in software engineering tools. The overarching method behind this approach is to represent thousands of code snippets, classified as defective or defect-free, as single fixed-length code vectors. Next, a Neural Network (NN) is trained over the vectors and their associated classifications with the goal of predicting the classification of unseen code snippets. This approach has produced state of the art results in bug detection; however, like tools based on probabilistic models, the drawbacks of this approach prevent their industry adoption. Namely, the training times required for such models are prohibitively large, and their ability to generalize is weak as a result of the use of embeddings generated over entire code snippets which vary greatly in source code.

To reduce training times in this approach, some techniques generate embeddings over an abstract representation of the code. DeepBugs (Pradel and Sen 2018) is one such example which learns embeddings that maps only the identifiers in given code samples to

semantic vector representations using a Word2Vec (Mikolov et al. 2013) NN. DeepBugs was able to achieve an accuracy close to 90% in classifying the generated embeddings based on whether they contained a defect or not. We point out however that very limited types and numbers of defects were utilized in this study, which makes it unclear how useful DeepBugs would be in code reviews. Andrew et al. uses another approach based of embeddings in which vector representations from source code using a one hot encoding scheme are generated based on a vocabulary consisting of the most frequently occurring keywords, separators, identifiers, and literals in the examined corpus (Habib and Pradel 2019). Like DeepBugs, the generated embeddings are classified using a NN as to whether the represent source code that has a defect or not. For certain types of issues, this techniques resulted in accuracies close to 90% whereas other types of issues witnessed accuracies closer to 0%.

While both theory and data-driven based approaches to building software engineering tools have made significant improvements over the last few years, the scope of such tools is often too specific in nature to be used as comprehensive code reviewing tools. As demonstrated in Chapter 3, code reviews consist of many goals outside of simply finding defects and refactoring code. Additionally, despite the existence of such tools, there is not much evidence to suggest that they are being adopted by the industry. This is likely because 75% of issues found during the review do not affect the visible functionality of the software; instead, these issues are typically flagged to improve software evolvability (Mäntylä and Lassenius 2009b). In this work, we develop a data-driven algorithm that uses both probabilistic models of source code and source code embeddings. Unlike earlier work based on similar approaches however, our solution does not suffer from long training times, and is trained over human review comments mined from GitHub that cover a much more comprehensive selection of problems found during code reviews.

Chapter 5

An Analysis of Code Review Comments

The design of any performant model requires a strong understanding of the system and interactions being modelled. In developing a probabilistic model to flag suspicious code, we first try to understand how suspicious code is typically flagged in human code reviews. In this chapter, we analyze thirty thousand GitHub review comments based on the main topic addressed by each comment. As an example, a review comment suggesting an improved variable name might be classified under "naming". A full list of the classifications incorporated by our classifier is presented in Table 5.1. To our knowledge, our experiment is the first of its kind to leverage GitHub review comments for the analysis of code reviews. Moreover, the findings from this experiment will help guide the development of our machine learning algorithm in flagging suspicious code described in Chapter 6. A Jupyter notebook containing the code for the experiment described in this chapter is available on GitHub [1].

5.1 Review Comment Classifications

The list of classifications incorporated by our classifier are summarized in Table 5.1. This list was developed based on a manual survey of two thousand GitHub review comments selected randomly from from the top one thousand most forked repositories on GitHub. The selected categories reflect the most frequently occurring topics encountered in the surveyed review comments. Majority of the categories are related to code level concepts including exception handling and control structures for example; however, certain review comments that did not naturally fall into any existing categories and were unrelated to the overall goal of code reviewing were placed in the "other" category. In situations where a review comment discussed more than one subject, it was given a classification according to the topic it spent the most words discussing.

[1] https://github.com/Zir0-93/What-Code-Reviewers-Talk-About-Blog-Post/blob/master/What-Code-Reviewers-Talk-About.ipynb

Category	Label	Sub-Categories
Readability	1	readability, style, project conventions
Naming	2	source code unit names
Documentation	3	licenses, module documents, code comments
Error Handling	4	exception handling, resource handling
Control Structures	5	loops, if-statements, conditional statements
Visibility	6	member access levels
Efficiency	7	runtime analysis, optimization
Code Organization	8	refactoring
Concurrency	9	thread synchronization, parallelism
Method Design	10	method design and semantics
Class Design	11	class design and semantics
Testing	12	test conditions, test cases
Other	13	comments outside categories 1-12

TABLE 5.1: Review Comment Classifications

5.1.1 Classifier Implementation

Mined code review comments are first preprocessed using a number of steps:

- **Remove formatting characters associated with the Markdown syntax**. Markdown is a lightweight markup language with plain text formatting syntax. It is designed to be easily converted to HTML and many other formats using a tool by the same name. Markdown is often used to format readme files, for writing messages in online discussion forums, and in GitHub code review comments. This preprocessing step is necessary because the additional formatting related characters introduced by the Markdown standard will negatively impact our classifier's ability to recognize identical words.
- **Remove all stop words**. A stop word is a commonly used word, such as "the", "a", "an", or "in" that we would like to ignore. Such words take up valuable processing time even though they are not very relevant to the classification task at hand. They can be removed easily by filtering review comments against an existing list of words considered to be stop words.
- **Convert to numerical feature vector**. The last step of our preprocessing stage is to convert the comment reviews into numerical feature vectors. This is required to make our review comments amenable for machine learning algorithms. The bag of words method is leveraged to accomplish this. Briefly, the bag of words

algorithm represents a sentence using a feature vector according to the following definition:

$$s := (n_1(f_1), n_2(f_2), ..., n_i(f_i)) \tag{5.1}$$

Here, s is a sentence, f_i represents a term and n_i is the frequency of the term f_i in the given sentence. Notice, this is equivalent to a first order n-gram model discussed in Chapter 2.5. As a result, the exact ordering of the terms in each review comment is ignored but the number of occurrences of each term is material. We only retain information on the number of occurrences of each term, known as *term frequency*. Thus, the comment "please rename this variable" is, in this view, identical to the comment "rename this variable please".

At this point, we may view each document as a vector with one component corresponding to each term in the dictionary, together with a weight for each component that is given by the equation above. Dictionary terms that are not found in a document have a weight of zero. The format of this vector is key to the learning and modelling capabilities of our Support Vector Machine (SVM) classifier.

Lastly, we implement our SVM classifier using the SGDClassifier module of the Python scikit library (Buitinck et al. 2013) for convenience and train it using the pre-processed vectors using Stochastic Gradient Descent (SGD). SGD (Robbins and Monro 1951) is an iterative based optimization technique which modifies the SVM parameters on each training iteration to find a local optimum that produces the best results. We set the number of iterations for our estimator at 1000.

Training

An important consideration in any machine learning experiment is the amount of training data to use for testing the classifier. After ensuring that a minimum of hundred review comments for each classification were present in our labeled data set, we experimented with different numbers of review comments to see what gave the best results. Two thousand review comments seemed to give a good accuracy for the classifier as we will see below. We dedicated 80% of our two thousand GitHub review comments to the training set, which are used to train our SVM classifier. The remaining 20% of the review comments will be dedicated to the test set, which are used to test the performance of the developed classifier. As demonstrated below, our developed classifier scored an accuracy of 92.5% on the test data set.

Experimental Results and Discussion

The classifier developed in the previous section is now used to classify over 30,000 GitHub review comments from the top 100 most forked Java repositories on GitHub. In general, repositories with many forks are more likely to maintain more formal code review processes due to their visibility and popularity, and therefore contain more code review comments that can be analyzed for our experiment. The results of the classification exercise are illustrate in Figure 5.1. Many of the classifications are related to each other at a conceptual level. For example, a readability problem may be caused due to poor use of control structures. Therefore, it is important to note that the classifications were based only on what the review comments explicitly discussed. That is to say, if a comment read, "poor readability here", we would expect it to be classified as "Readability". However, if the comment read, "This for loop should be moved before line 2", then "Control Structures" would be its expected classification, despite readability being the motivating factor for the review suggestion. As mentioned previously, in cases where multiple classifications applied, the comment was classified to a category it spend the most words discussing.

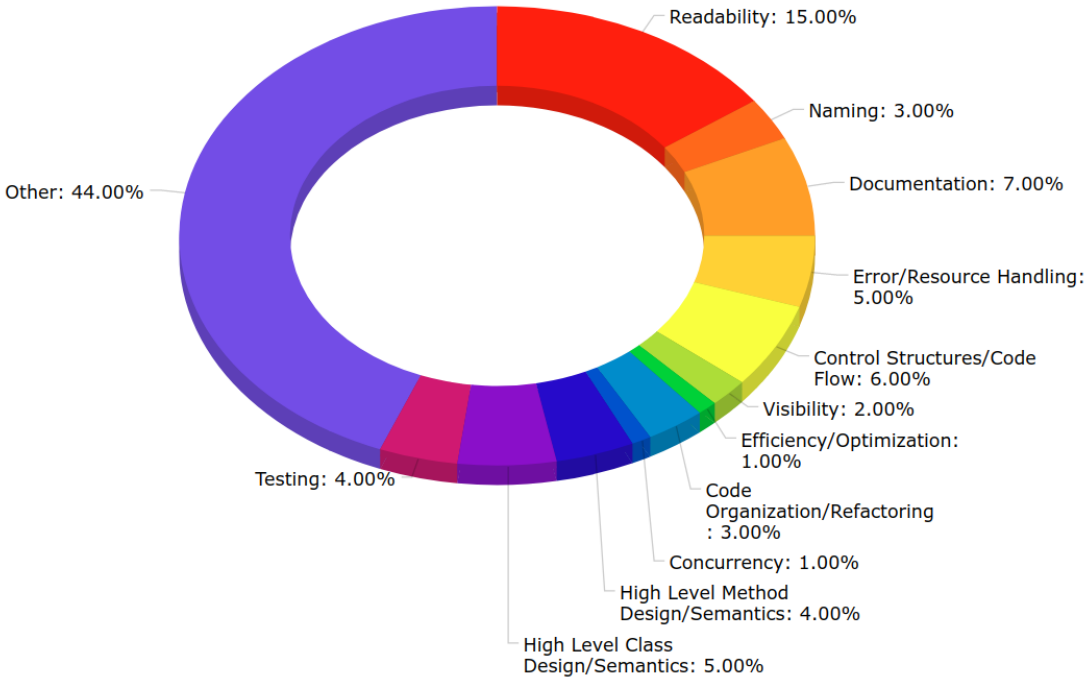

FIGURE 5.1: GitHub code review comment classification results

Interestingly, 15% of review comments were found to discuss readability. A sample comment discussing readability is illustrated in Figure 5.2. This classification deals not

only with formatting, project conventions and style, but also with how easy the code is to follow for a human. While existing static analysis tools are effective in dealing with the former area, they lack the ability to check code effectively in the latter domain. This is supported by the fact that over 80% of the repositories included in our study had some form of static analysis tools checking their code. Additionally, a manual survey of 100 randomly sampled comments classified with readability confirmed this idea as well. One such comment from our study is illustrated below in Figure 5.2. While the approach taken by the code contributor is correct, the code reviewer has suggested an alternate approach that increases the clarity of the code.

FIGURE 5.2: Review comment regarding readability, sourced from https://github.com

Error handling, class and method design were areas according to which a significant percentage of review comments were classified under. This does not come as a surprise, as there is a great deal of variability in the way they are handled between software development teams. Moreover, there is often no single correct way to go about their implementation, resulting in an increased amount of developer discussion, and thus review comments, around them.

The most surprising finding from this study is the fact that 44% of review comments were classified according to the "other" category. This highlights a major limitation of our approach in using review comments alone to understand the subject of typical code review discussions. Without the ability to extract meaningful information from the source code alongside the review comment, it is very difficult to understand what the comment alone is discussing. For example, Figure **??** illustrates a review comment targeted at a Java annotation for a field variable. However, without viewing the source code, it equally possible to suggest that the review comment is targeted at a method

name, class name, or even source code documentation. Moreover, many review comments contain references to source code entities, which also makes their classification a difficult task without the analyzing the source code as well. Lastly, many review comments analyzed consisted of comments serving as a response to previous comments, which do not always reveal information on their own about the original subject of the discussion.

Thus, while the textual content of the analyzed review comments revealed a great deal of information about the topics typically discussed by code reviewers, our classifier suffered from its inability to make use of source code in classifying review comments as well. However, we believe there is sufficient information in our experimental results to proceed with the development of our probabilistic model of source code. This is because while GitHub code review comments are commonly used for many types of discussions, this dissertation is primarily concerned with building an automated code reviewing tool. As such, we are most interested in comments relating to code reviews, which we believe to have sufficiently capture by our classifier. We hope that interested researchers will benefit from and use our findings to further explore this area of research.

Chapter 6

Automated Code Reviewing Algorithm Design

In Chapter 3, the two main responsibilities of a code reviewer were shown to consist of finding defects and areas for improvement in code submissions, and providing solutions and techniques for correcting the code based on those findings. With that said, the goal of this thesis is to develop an algorithm that automates the *first task* of code reviewing. The intended use case for the proposed tool lies in analyzing code submissions for suspicious snippets of code prior to invoking a human code reviewer. Only when these suspicious aspects of the code are addressed would a human reviewer be invoked to look for and provide help in resolving any remaining issues. The benefits of the propose solution are two-fold:

- Costs associated with having a technical resource perform code reviews are expensive due to time. By using the proposed algorithm to flag suspicious snippets of code in much the same way a human would, the amount of time spent by human code reviewers in performing code reviews themselves will decrease.
- Rather than waiting on a human to provide feedback, the automated nature of the tool would allow developers who are submitting code to address any issues right away. This would allow developers to be more efficient with their time, and likely result in code reviews being resolved faster.

However, the aforementioned benefits are only realized to the degree at which proposed solution is able to correctly flag suspicious snippets of code the way way a human code reviewer would. As such, the next few sections will present a detailed description of the algorithm and convincingly demonstrate the effectiveness of the tool through experimental results.

6.1 Overview

The input to the algorithm consists of a set of Java source files that contain the modifications a developer wishes to make to a code repository. For each source file, a language

model, previously trained on hundreds of open source Java projects on GitHub, is used to generate entropies for each source code token. Subsequently, the Abstract Syntax Tree (AST) of each source file is created, and the entropies collected previously are averaged where required and attributed to each of it's corresponding node in the AST. Next, the GraphSage algorithm is used to generate embeddings for each node. Finally, a neural network is trained over the node embeddings to predict whether or not a given node would elicit a human comment in a code review or not. An overview of the process is illustrated in 6.1.

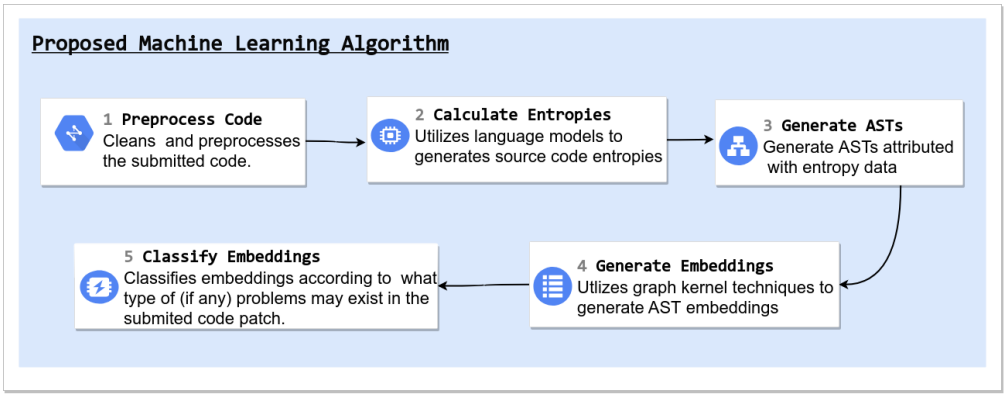

FIGURE 6.1: Proposed algorithm overview

6.2 Preprocessing

In the development of a language model based on source code, the chosen representation of the code itself over which the model will be trained is of has a significant impact on the model's ultimate performance. This process, widely referred to as feature engineering, is a fundamental activity in the development of any machine learning model. The goal of this process is to transform raw data into features that better represent the underlying problem to predictive models, resulting in improved model accuracy. Source code can be represented using a variety of formats including ASTs, call graphs, and byte-code. More importantly however, is that any chosen input representation yields a probabilistic model with unique strengths and weaknesses in comparison to the use of other representation formats.

6.2.1 Syntax Versus Semantics Based Approaches

Recent research exploring involving the use of probabilistic models over source code have uncovered the following two broad approaches:

Syntax Focused In this approach, source files are tokenised at a syntactic level using low-level tokens generated from splitting the source code on white spaces. Existing work has used such methods to build n-gram models to to learn and suggest variable, method, and class naming conventions (Allamanis et al. 2014; Allamanis et al. 2015). For example, after processing the string

```
for (int i = 0; i < n;
```

the model might suggest the tokens

```
i++) {
```

However, building n-gram models at this level is likely to only detect syntactic errors that would typically be caught by a compiler, and requires a large amount of training data to perform well. In contrast, tokenization using Java tokens is another widely used syntax focused approach that can uncover deeper insights than space-based source code tokenization. Java tokens are miniature elements of a program that are meaningful to the compiler, their various types are illustrated in Figure 6.2.

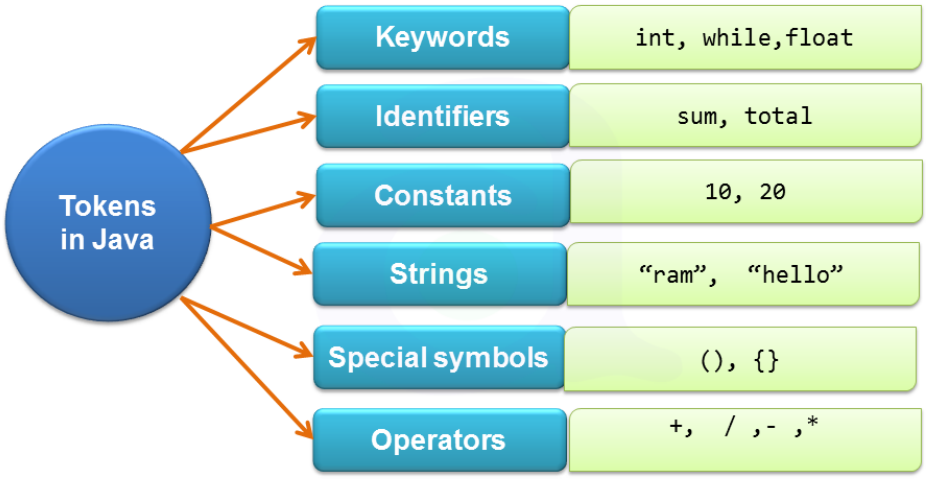

FIGURE 6.2: Sample Java tokens

As an example, the string `car.setSpeed(i + j);` might be tokenized as

car	setSpeed	(	i	+	j	)	;

Because Java tokens are used to tokenise source code in this approach, all the terms in the document are represented in a more fine-grained manner. This allows for probabilistic models to determine probabilities more effectively over snippets of code. Without this pre-processing step, the original line of source code in the above example would be tokenised with the terms:

car.setSpeed(i	+	j

As a result, each of these three terms would be assigned unique probabilities. The problem with this method lies in the calculation of the probability of an unseen string, like

```
car.setSpeed(c)
```

Here, the resulting probability is close to zero because it has never been seen before. However, we would not expect such a probability since it is very similar to the known string,

```
car.setSpeed(i
```

In the proposed approach using java tokens however, the probability of the unseen string is much higher since the sequence of Java tokens "car", ".", "setSpeed", and "(" have been seen before. Therefore, the assigned probability for this snippet of code will be close to 100%, which is what was desired.

Semantics Focused In this approach, emphasis is placed on utilizing the semantics of the code, as opposed to the code's textual content. A common method of semantically representing source code leverages the code's AST representation. In this method, source code is replaced by the AST elements that represent it. As an example, a line of Java source code

```
car.setSpeed(i + j);
```

would be represented using its corresponding AST nodes in a semantics focused preprocessing method, resulting in the following representation.

```
MethodInvocation BinaryOperation MemberReference MemberReference
```

Both syntax and semantics focused approaches have their own strengths and weaknesses and are more suitable for certain applications over others. For example, a statistical language model developed over source code that is preprocessed using the syntax focused approach effectively captures regularities and patterns on the surface level, syntactic nature of source code. Such a model would be useful for assessing source code spacing and indentation, compiler errors, spelling, and so on. However, a semantics focused approach to preprocessing the source code would yield a probabilistic model that is able to capture higher level, semantic regularities across source code. Therefore, such a model would be effective at understanding source code organization, abstract code idioms, and more.

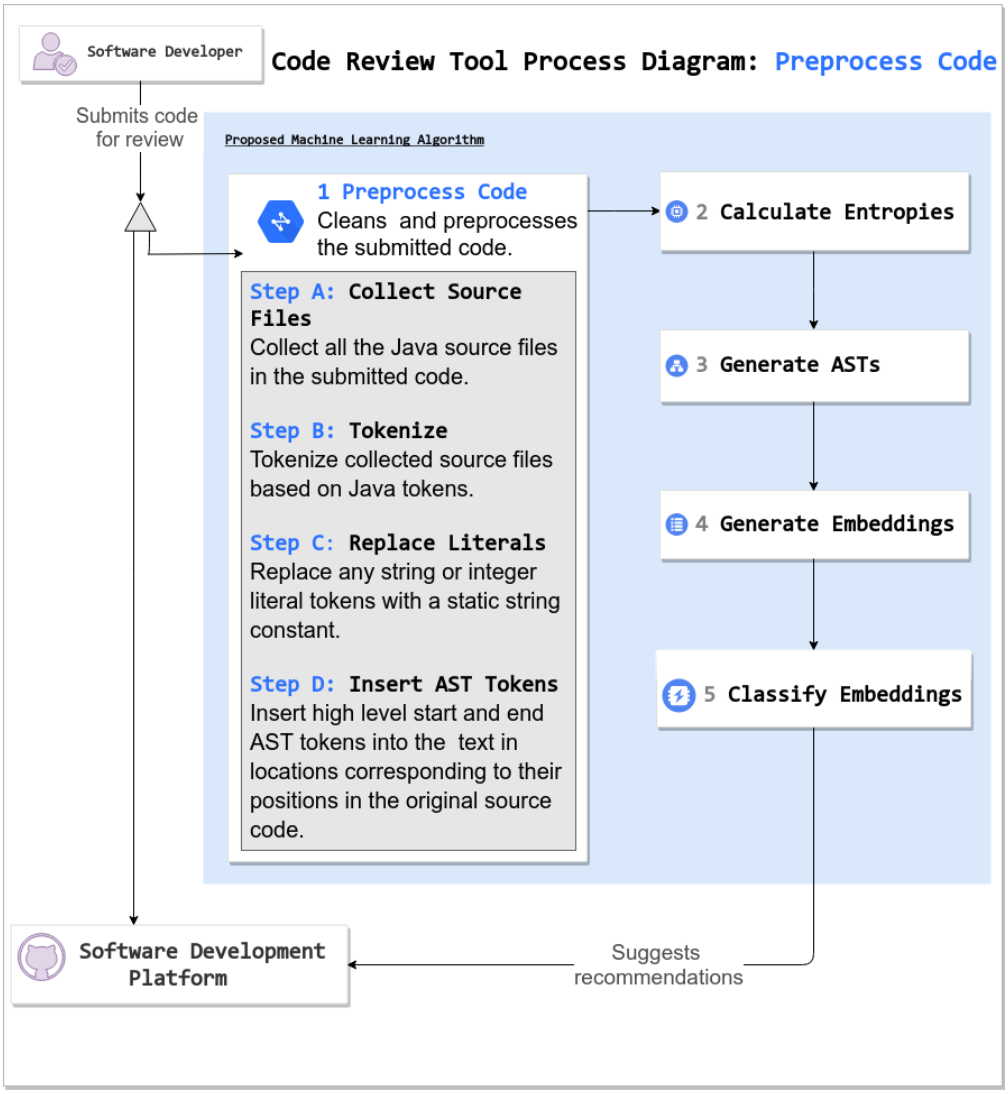

FIGURE 6.3: Preprocessing stage overview

6.2.2 The Combined Approach

In our analysis of GitHub review comments in the previous section, Readability, Class Design, Method Design, and Control Structures constituted the four most discussed topics in code reviews. Because these topics contain major syntactic and semantic elements, a preprocessing technique that incorporates both the syntax and semantic elements of source code is required. Figure 6.3 outlines our preprocessing approach. First, we tokenise the source code based on Java tokens. Next, we insert high level tokens in the preprocessed text output that correspond to the following AST types: class, method, field and local variable declarations, control flow types, and method calls. Nodes and

text corresponding to any other AST types are removed. As a result of this preprocessing approach, the final preprocessed text captures both the structure and context of the code using a succinct semantic representation. As an example, an original snippet of Java source code

```
class Lamp {
    private String fooVar = "cuppycakes";
}
```

would result in the following text after being preprocessed in the proposed combined approach.

```
ClassDeclaration FieldDeclaration private String fooVar = <LITERAL>
```

It also preserves most of the original syntactic representation of the source code as well. Lastly, we also replace any String or Integer literals by a fixed keyword "<LITERAL>" in order to prevent the developed language model from assigning unique probabilities to each String or Integer literal it encounters.

6.3 Entropy Calculation

In this stage of the proposed algorithm, a language model, previously trained over thousands of source files, is used to generate entropies for each token in submitted source files. However, given the various types of language models that can be used, an important decision must be made as to whether an n-gram or Neural Network (NN) based language model is most appropriate for this task.

Chapter 2 introduced both n-gram language models and the Neural Language Model (NLM), and explained that while n-gram models are significant less costly to train in comparison to an NLM, they suffer from issues involving data sparsity and limited historical context. Feedforward Neural Language Models (FNLM) are able to overcome the data sparsity issue by projecting the input layers to lower dimensional vectors; however, these models still make the n-gram approximation and therefore consider only a small subset of words at a time. Recurrent Neural Language Models (RNLM), although the most time-consuming to train, provide a more effective approach for tackling these issues. Therefore, based on the theoretical foundations of the various language models discussed, an RNLM would be the tool of choice for our purpose. Yet, recent literature discussing and evaluating the effectiveness of language models is somewhat divided.

The first experiment comparing the performance of an NLM and n-gram language model was conducted by Bengio (Bisset et al. 1989). The experiment, conducted on two corpora containing more than 15 million words, concluded that the approach based on FNLMs yielded significantly better perplexity than the smoothed trigram, with differences between 10% and 20% in perplexity.

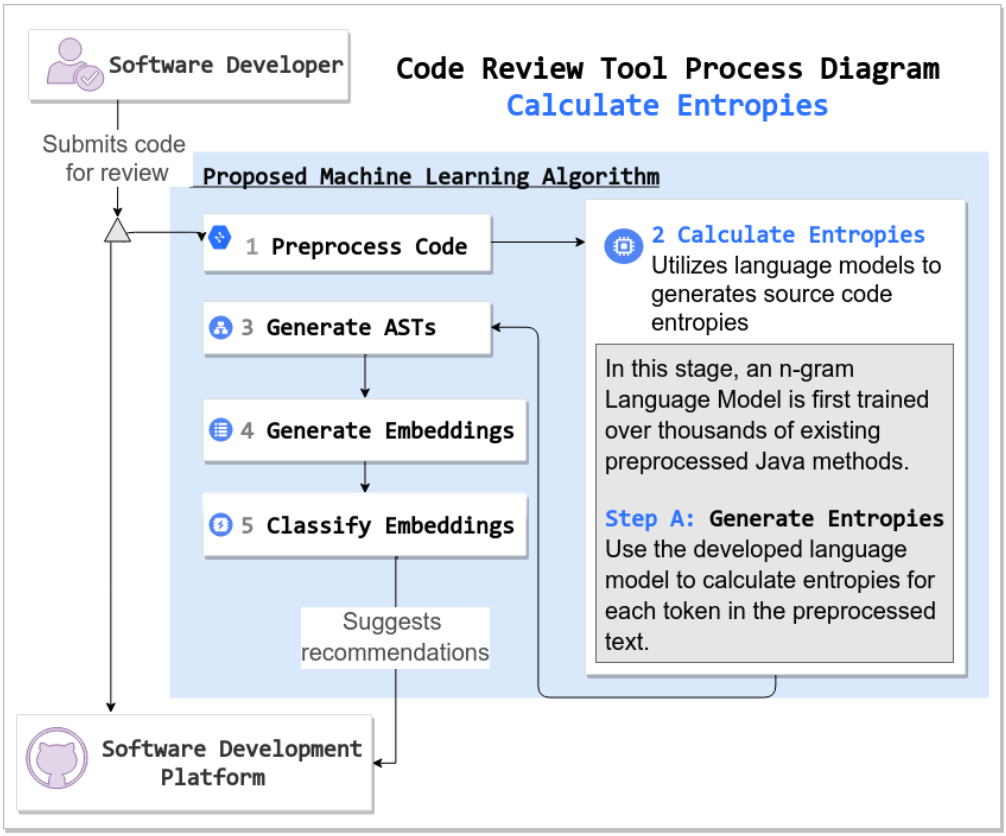

FIGURE 6.4: Entropy calculation stage overview

Mikolov realised that Bengio's method was severely restricted by the need for a Feed-forward Neural Network (FFNN) to use fixed length contexts before training, which meant that a small number of words were required to predict the next one in a sequence. Mikolov's experiment evaluated the performance of an n-gram language model and RNLM over several speech recognition tasks (Mikolov et al. 2010). These tasks were to designed to test the model's ability to recognize and translate spoken language into text. Mikolov found RNLM perplexity scores to be more than 25% lower than than those produced by the n-gram models. This result is further validated by Sundermeyer, who evaluated the effectiveness of RNLM versus FNLM on various speech recognition tasks as well (Sundermeyer et al. 2015). He found that RNLM perplexity scores were 17% lower on average than those produced by a FNLM. Moreover, Tzu-Hsuan verified the long-range dependency of RNLM by comparing the perplexity and the word prediction accuracy with n-gram language models as a function of word position (Tseng et al. 2016). The study showed that the gap between the perplexities of Recurrent Neural Network (RNN) and n-gram increases as the word position increases.

While NLMs are largely considered to be superior to n-gram language models, Hellendorn's study argues that deep neural networks may not be the best choice for developing

language model over source code (Hellendoorn and Devanbu 2017). Additionally, the study develops a nested scope, unlimited vocabulary count-based n-gram model that significantly outperforms all existing token-level models, including "very powerful" ones based on recurrent neural networks. Hellendorn further found that NLMs use a "great many parameters, require extensive configuration and are also often heavily tuned on the task at hand". In the Natural Language Processing (NLP) community, Omer et al. conducted various investigations into word-embeddings and found that state-of-the-art neural-network models perform similarly to simple matrix-factorization models tuned with a few hyper-parameters (Levy and Goldberg 2014). Hellendorn's study makes similar conclusions in which an RNN and Long short-term memory (LSTM) fail to beat a well-calibrated n-gram language model.

Additionally, n-gram language models continue to meet the needs for performance and utility in the development of software engineering tools. Allamanis and Sutton (Allamanis and Sutton 2013) suggest that n-gram language models can be used to analyze the complexity of source code while and Ray et al. (Ray et al. 2016) presents evidence that buggy code tends to have lower probabilities than correct code. Additionally, he is able to show that language models find defects as well as popular tools like FindBugs. Fast et al. (Fast et al. 2014) and Hsiao et al. (Hsiao et al. 2014) learn statistics from large amounts of code to find potentially defective code and perform program analyses. Related to this, Wang et al. develop Bugram (Wang et al. 2016), which leverages n-gram language models to detect uncommon usages of code.

Given all the existing research exploring the effectiveness of both n-gram language models and NLM, we use an n-gram language model for generating source code entropies in the proposed algorithm. This is primarily because we are most interested in finding a correlation between the naturalness of a given piece of code and the probability that a human would find an issue with it. Therefore, any language model that can sufficiently convey naturalness, which both language models have shown to do, will be appropriate for our task. Additionally, since more research is required to convincingly demonstrate the superiority of neural network based language models over well tuned n-gram language models, the ease of use and reliability of n-gram language models make them easier to train and experiment with.

We have also chosen to use an n-gram language model due to the size of the corpora we are dealing with. In NLP, it is extremely common to limit vocabulary to the most frequently occurring words before model estimation. Words outside this vocabulary are treated as unknown words, or omitted entirely. However, this technique significantly reduces model training times, thereby providing increased support for NN based models. The problem with this approach arises when working with source code, which is comprised of an open, rapidly changing vocabulary. This is the main reason as to why many state of the art software engineering tools incorporating language models continue to use n-gram language models. Since developing a NLM over a billion tokens is unfeasible due to the extreme length of training time required with our available hardware, we use the state of the art n-gram language model implementation developed by

Hellendorn (Hellendoorn and Devanbu 2017) for the remainder of this thesis.

6.4 AST Generation

In this stage, the preprocessed source files and entropies generated in previous sections are used to generates ASTs attributed with additional information. First, regular ASTs are created using the source files contained with the submitted patch under review.

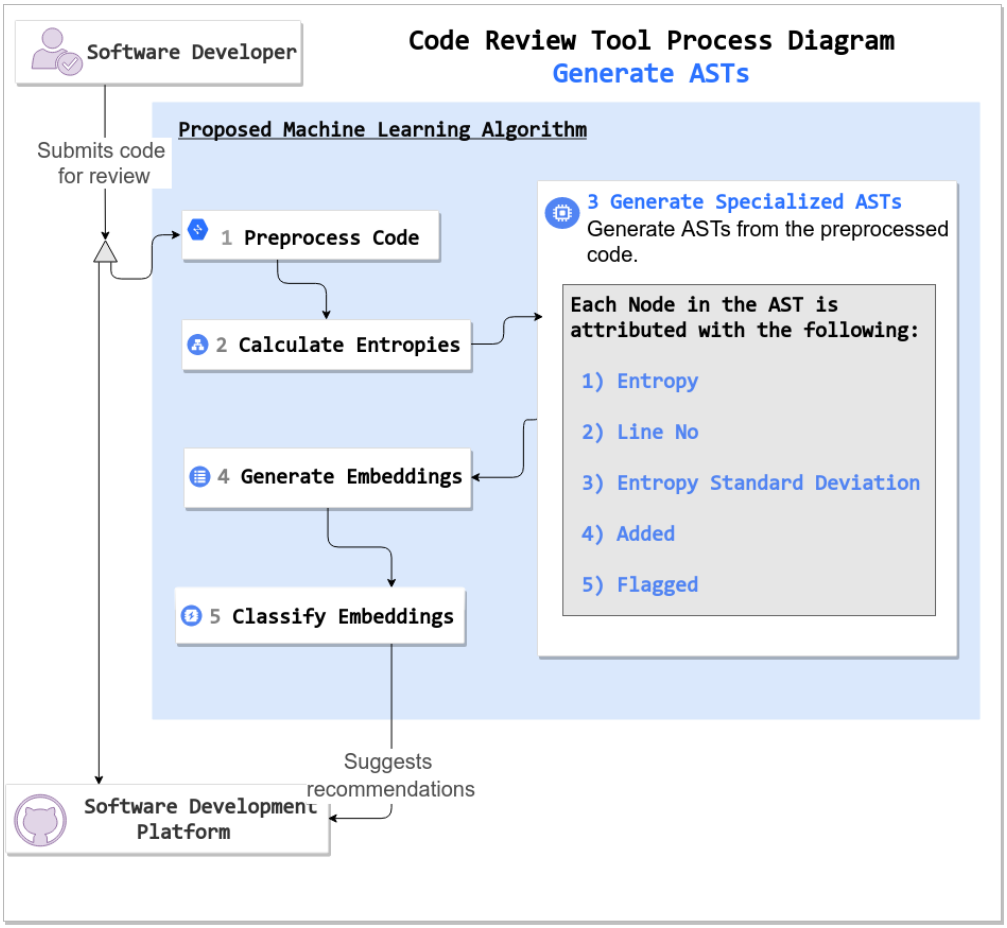

FIGURE 6.5: Specialized AST generation stage overview

The following attributes are appended for each node in the AST:

- **Entropy**: Float value corresponding to the entropy calculated for the syntactic representation of the node. This value is retrieved from entropy data calculated in a previous step.
- **Line**: Integer corresponding to the node's line number in the source file.

- **Entropy Standard Deviation**: Float value corresponding to the standard deviation of the entropies from the syntactic representation of the node. A field declaration node for example, consists of multiple syntactic attributes in the actual source code. Given that each token is associated with its own entropy, this value calculates the standard deviation of all those entropies.
- **Added**: Boolean value corresponding to whether the node correspond to code that was added as a result of the patch under review. This value is determined by seeing whether or not the source code corresponding to the node exists in the original version of the source file or not.
- **Flagged**: A boolean value indicating whether the node is "problematic" or not. This value is set to True when the given node corresponds to the first AST node on a line for which a human code reviewer left a comment on. This method has been used to highlight 'problematic' nodes because the GitHub Application Programming Interface (API) provides only the line number on which code review comments were left on. Given this limitation of our available training data, we assume that the first AST node encountered on such a line is the subject of the comment. Moreover, this is the value our proposed algorithm seeks to predict accurately for any given node.

6.5 Generate Embeddings

In this stage, generated ASTs attributed with additional data are converted into embeddings over which a machine learning model may be trained. In recent years, several Convolutional Neural Network (CNN) architectures for learning over graphs have been proposed (Bruna et al. 2014; Duvenaud et al. 2015; Defferrard et al. 2016; Kipf and Welling 2016; Niepert et al. 2016). However, the majority of these methods cannot be used for large scale graphs or are designed for generating embeddings for entire graphs (Hamilton et al. 2017). However, the approach used in this work is closely related to the Graph Convolutional Networks (GCN) introduced by Kipf et al. (Kipf and Welling 2016; Kingma and Ba 2014) which is designed for semi-supervised learning in a transductive setting and requires that the full graph Laplacian is known during training.

GraphSage

GraphSage (Hamilton et al. 2017) is a framework for inductive node embedding based on GCNs and represents the chosen framework for generating embeddings in the proposed algorithm. GraphSage uses node features including node degrees, text attributes and more to learn an embedding function which generalises to unseen nodes, as opposed to embedding approaches which are based on matrix factorisation. GraphSage is able to learn simultaneously the topological structure of each node's neighbourhood and the

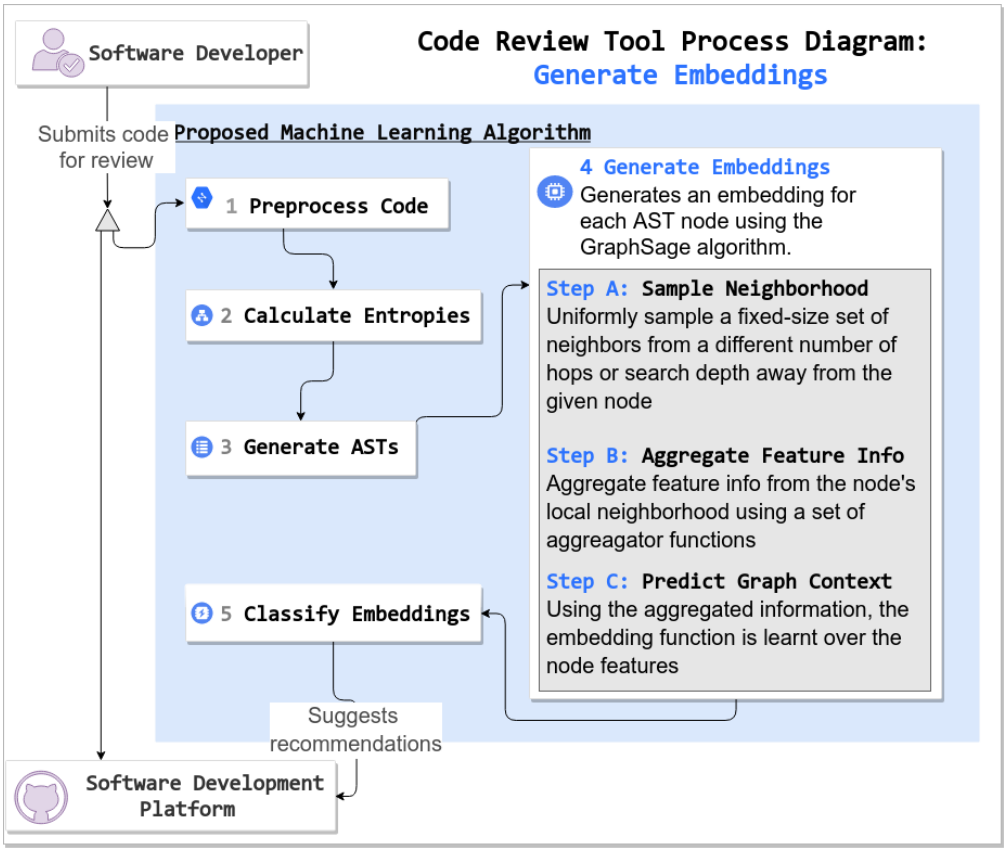

FIGURE 6.6: Embeddings generation stage overview

distribution of node features in its neighbourhood by integrating node features in the learning algorithm.

As illustrated in Figure 6.7, the key contribution behind the GraphSage approach lies in the method used to aggregate feature information from a node's local neighborhood given the available node feature information. That is, the approach leverages a set of aggregator functions which are trained to integrate feature information in a local node neighbourhood rather than to train a distinct embedding vector for each node. Each aggregator function is responsible for aggregating information from a mixed number of neighborhoods away from a given node. At inference time, the trained model is used to generate embeddings for entirely unseen nodes by applying the learned aggregation functions on the input nodes.

Because GraphSage is capable of generating inductive node embeddings and has achieved state-of-the-art performance on various graph embedding tasks and benchmarks, we use the algorithm for generating embeddings for the remainder of this thesis as illustrated in Figure 6.6

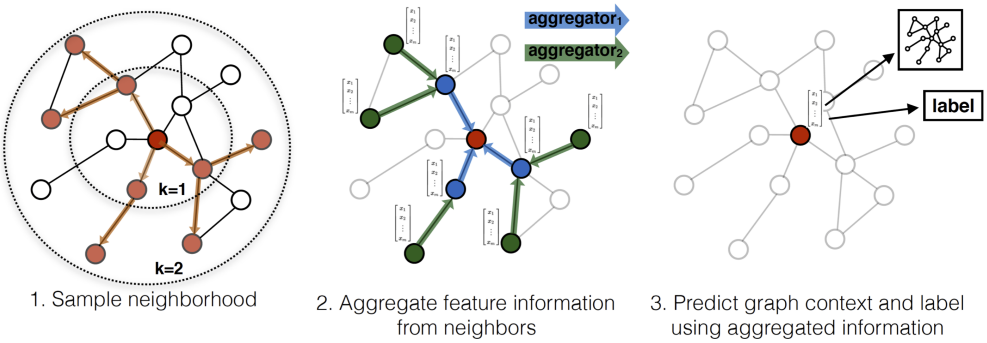

FIGURE 6.7: GraphSage algorithm overview

6.6 Classify Embeddings

The final stage of the proposed algorithm consists of leveraging a NN to learn embeddings that repesent AST nodes with the goal of predicting whether or not a comment was left on a certain node by a human code reviewer. The parameters and configuration of the NN are described in detail in Chapter 7.

Chapter 7

Evaluation and Results

In this chapter, the performance of the proposed algorithm is evaluated over source files extracted from approximately thirty thousand GitHub pull requests, comprising of over thirty five million Abstract Syntax Tree (AST) nodes. To the best of our knowledge, this experiment represents one of the first to utilize GitHub code review comments for the aim of developing more effective code reviewing tools. The experiment described in this section proceeds as a typical machine learning experiment. First, the final Neural Network (NN) outputted by the the proposed algorithm is trained over a portion of the experimental data set, referred to as the training data set. Next, its ability to imitate human-like code inspection behaviour is evaluated on the remaining, unseen portion of the data set, referred to as the test data set.

7.1 Data Set

In measuring the performance of the proposed algorithm in performing human-like code inspections as defined in Chapter 6, a data set containing instances of real code inspections must be used. Therefore, the chosen data set consists of source files mined from pull requests in 1572 repositories on GitHub. Each file consists of multiple lines of code, in which lines over which a review comment was left are labelled accordingly. We also try to work with code review comments that are strictly about code; consequently, any review comments that do not fall under classifications 1-8, and 9 when classified by the model developed in Chapter 5 are excluded from our experiment. Lastly, we note that the chosen GitHub repositories constitute a collection of all the Java GitHub repositories on GitHub with over 250 forks at the time of writing. This number was selected to maximize the following desired attributes of the data set:

1. **Code quality**: We believe that the number of forks can be roughly correlated to the overall quality of a given repository on GitHub. Developers commonly fork repositories on GitHub to either propose changes to the original project, or to use the project as a starting point for a new idea (Jiang et al. 2017). In either case, the original repository must be at a sufficient level of quality for developers to be productive in either task. Additionally, many forks on a repository indicate a high

Attribute	Count
Source Files	59,130
Total AST Nodes	36,873,189
Non-commented AST Nodes	36,833,803
Commented AST Nodes	39,386
Commits	48,438
Lines of Code	28,853,416

TABLE 7.1: Notable statistics of the experimental data set

level of visibility on the project, which makes it more likely that the maintainers have invested a sufficient amount of time in ensuring its comprehensibility. We would like to maximize the code quality of the analyzed repositories in an attempt to focus on code reviewing habits within higher quality projects on GitHub.

2. **Size**: Despite being introduced many years ago, code review comments are still not a widely used feature on GitHub which limits the number of comments from "high quality" repositories that can be used. At the same time however, a typical machine learning model requires thousands if not millions of samples to learn from. Therefore, we would like to maximize the size of our training data as well.

After gathering all the source files, the proposed algorithm detailed in Chapter 6 is applied to the data set. Notable properties of the selected data set are illustrated in Table 7.1. The approximate distribution between AST nodes that were commented on by a human code reviewer and those that were not commented on is approximately 1:1000.

7.2 Algorithmic Implementation Details

Chapter 6.3 discussed the entropy calculation stage of the proposed algorithm, which depends on an existing, pre-trained language model to calculate entropy data for source code found in a code submission. In this experiment, the n-gram implementation by (Hellendoorn and Devanbu 2017) is utilized to develop an n-gram language model with an order of six over the top 200 most starred Java projects on GitHub. This order was chosen for our n-gram model due to its high performance in machine learning experiments leveraging preprocessing techniques similar to the ones used in this study (Wang et al. 2016; Jimenez et al. 2018). The trained model consists of approximately 32,000,000 tokens, and was trained just under two hours on a Linux machine with 64GB of memory and eight Intel Core i7-6700 CPU @ 3.40GHz processors.

Chapter 6.5 discussed the proposed algorithm's use of the GraphSage algorithm to generate embeddings from ASTs. The selection of a number of important parameters specific to the GraphSage algorithm are discussed below:

- **Embedding Size**: An embedding is a dense vector in which high-dimensional information about a node's neighborhood is distilled, often using dimensionality reduction techniques. The size of the embedding can have a significant impact on performance of downstream machine learning systems that depend on it. Smaller embedding sizes tend to be computationally efficient to process, but cannot express as much information as larger embeddings, which are computationally expensive to process. Choosing an appropriate embedding size largely depends on the amount of data that is being embedded. In this experiment, we sample nodes nodes from up to five neighborhoods or hops away from the node being embedded, and have no more than seven features per node. The original GraphSage experiments sampled nodes three hops away from the source node, contained over 500 features per node, and used an embedding size of 32. Based on this experiment and an understanding of embedding sizes used in other node embedding experiments, we found an embedding size of 64 to be sufficiently expressive enough for representing AST nodes in our data set and computationally feasible with available hardware.

- **Hops**: GraphSage generates embeddings for a given node in a graph by sampling the features of nodes at multiple distances away from the node, referred to as hops. Sampling nodes one hop away from the embedding node will only sample nodes in the immediate neighborhood of the node and fail to express any longer range relationships that the embedding node might have with nodes further away from it. A hops value of five is maintained in our experiment, and experimentation with other values is left for future work.

- **Sample Size**: GraphSage samples the features of a number of randomly selected nodes equivalent to the sample size in each neighborhood away from the embedding node. Initial experimental results showed that performance for our task does not increase noticeably after a sample size of 10, which is used for our experimental evaluation.

The configuration of the NN for classifying embeddings as discussed in Chapter 6.6 is the final implementation detail of the proposed algorithm that requires clarification for our experiment. The Keras (keras-team 2019) machine learning python library is utilized to implement the NN as a simple Multilayer perceptron (MLP), which is a class of Artificial Neural Network (ANN)s consisting of an input layer, a hidden layer, and an output layer. We apply a dropout of 0.2 to the input layer and the hidden layer. The binary cross-entropy (Goodfellow et al. 2016) function is used as the NN's loss function along with the Adam optimizer (Kingma and Ba 2014) for 10 epochs with a batch size of 100. Based on our understanding of related work in machine learning research, the chosen NN implementation is considered to be a standard configuration for classification activities over source code, and has been selected to improve the generalizability of our results.

7.3 Evaluation and Results

We now apply the proposed algorithm to the data set described in Table 7.1. On completion of the AST generation phase of the proposed algorithm, 36,873,189 embeddings remained to be classified as expected. Unfortunately, training the final classifier on 80% of the entire available data set would take too long due to its massive size; instead, we would like to train our model over a few million data points. However, naively sampling a few million points would also be problematic due to the highly imbalanced nature of the data set. That is, in a random sample of one million data points from a data set containing a class imbalance of 1:1000, as is the case with the experimental data set, the model would "see" very few data points from the minority class, thereby severely limiting its overall predictive ability over unseen data.

A great deal of research has investigated methods for developing classifiers for highly imbalanced data in machine learning experiments (Datta and Arputharaj 2018; Lipitakis and Lipitakis 2014). Approaches have largely been categorized into data-level and algorithmic in nature. Data level solutions involve re-sampling the data set before applying the classifier, of which undersampling (Hasanin and Khoshgoftaar 2018) and oversampling (Pérez-Ortiz et al. 2016) techniques are most popular. Cost sensitive-learning (Thai-Nghe et al. 2010) is the most popular algorithmic based approach in which the classifier takes missclassification costs into consideration, unlike standard classifiers which treat this cost equally. Therefore, the missclassification cost of the minority class in a highly imbalanced data set can be increased proportionately in order to increase the performance of the classifier. While cost-sensitive learning outperforms random re-sampling, clever re-sampling methods are used more in practices because of their ability to eliminate redundant information for the learning algorithm (Kotsiantis et al. 2005).

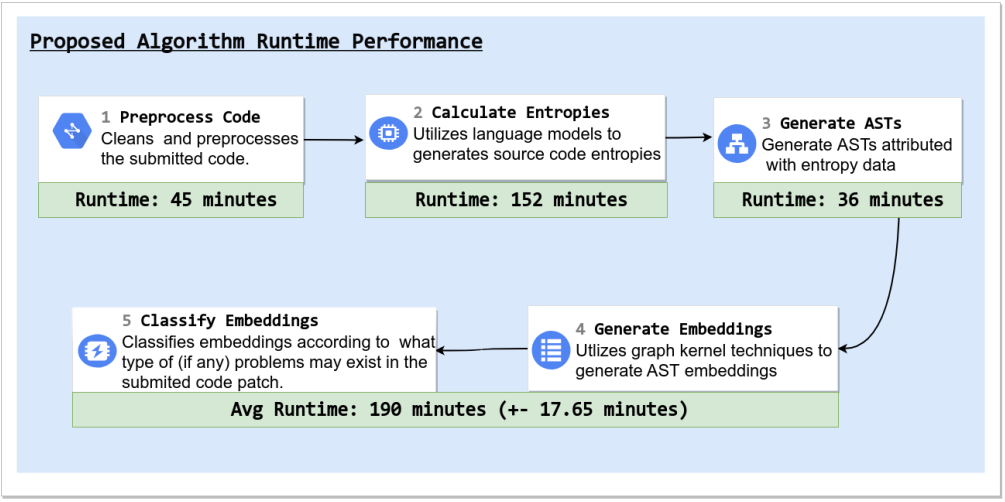

FIGURE 7.1: Runtime performance of the proposed algorithm on the training data set

Class	Precision	Recall	F_1 Score	Support
Non-commented	94.31	84.52	89.14	7,191,250
Commented	23.45	67.80	34.85	7,264
Macro	58.88	76.16	**62.00**	

TABLE 7.2: Classification metrics summarizing the proposed algorithm's performance on the test data set averaged over 10 runs

The experimental approach chosen for our experiment leverages random undersampling to eliminate redundant data and decrease training times, as well as cost sensitive learning. The experimental data set is first split into 80-20 train-test split sets. From the training data set, we form a new data set consisting of all the minority data points representing commented AST nodes in the training set, along with the a random sample of majority data points from the training data set such that the newly formed data set maintains a distribution of 1:100, down from 1:1000, between the minority and majority classes. Finally, our NN Keras implementation is also updated to maintain class weights representing the 1:100 class imbalance for the training phase, effectively incorporating cost-sensitive learning.

After training the NN, the test data set consisting of unseen AST node embeddings is processed and classified. Unlike the data set over which the classifier was *trained* on, this data set maintains the original 1:1000 minority to majority class distribution of the original data set. Due to the inductive nature of GraphSage, and our customized method for dealing with imbalanced data in the training of the classifier. We repeat the entire process of generating embeddings, training, and testing the classifier 10 times. The classification metrics of these runs are averaged and presented in Table 7.2. Additionally, the runtimes of each stage of the proposed algorithm is illustrated in Figure 7.1.

7.4 Generalizability

In this section, a number of external factors that could potentially impact the generalizability of the completed study are discussed. First, we note that the proposed algorithm, from the selection of code review comments to the generation of the n-gram language model, leverages data from highly starred or forked repositories on GitHub. This was done purposefully with the aim of targeting "high quality" repositories; however, this approach limits the generalizability of the experimental results to repositories with fewer forks.

The experimental data utilized in our study was also collected solely from GitHub. While GitHub is one of the most popular code hosting platforms in the world, there may exist a difference between the way code is written on Github in comparison to similar code hosting platforms, or even to the way it is written in commercial settings.

Therefore, we cannot be certain that the proposed algorithm would maintain the same level of effectiveness.

Additionally, we note that not all GitHub code review comments on Github are necessarily made by humans. Many different types of GitHub integrations exist in which automated bots leave code review comments for various purposes such as the static analysis of source code. Moreover, there is an increasing number of static analysis tools which integrate with GitHub to report results as code review comments in pull requests. Without the ability to filter code review comments for those generated by human code reviewers, the proposed algorithm learns non-human like code review inspection behaviours as well. If a major percentage of code review comments were to consist of those made by automated tools, it is possible that the proposed algorithm would simply learn code inspection behaviour inherent in existing tools.

Chapter 8

Discussion

The proposed algorithm performed extremely well and maintained high precision and recall scores on the majority class of the experimental data set representing Abstract Syntax Tree (AST) nodes that did not elicit a human comment. Given the large class imbalance present in the experimental data set, this result is not surprising. However, because the goal of the proposed algorithm is to imitate human like behaviour in flagging suspicious code, we focus our discussion on the classifier's performance over the minority class. Here, the classifier achieved a much lower precision score equal to 23.45%, and a recall score of 67.80%. It can be said therefore, that although over 60% of AST nodes were correctly classified, approximately one in every four nodes were incorrectly classified as well.

There are a number of reasons for decreased experimental results in our study, beginning with a lack of consistency among code reviewers and repositories sampled. Most issues flagged in code reviews deal with aspects of code that are not widely agreed upon, such as software evolvability or module design. While there has been no research conducted to the best of our knowledge that demonstrates this phenomena, we strongly believe that issues flagged in a code review regarding similar aspects of code vary according to project standards and the technical ability, experience, and time available to the code reviewer. As a result, there is great variability in the way code reviews are performed which is not accounted for in the proposed algorithm, resulting in decreased performance.

The main reason for not having accounted for the variability in code reviews was due to a lack of available data, which significantly limited the the experimental performance of the proposed algorithm. The final Neural Network (NN) is trained over embeddings represented as continuous vectors of length 64. Given the relatively large embedding space, the model was only trained on approximately 30,000 embeddings representing nodes which were flagged by human code reviewers. While the proposed algorithm has smaller requirements on training data size in comparison to other machine learning models which are trained over source code, having access to more labelled data representing code flagged by human code reviewers would have likely increased the performance of the model over all.

Finally, we note that human code reviewers themselves may be inconsistent in code reviews, making it difficult to fully interpret the overall effectiveness of the proposed algorithm based on the experimental results alone. Originally, we sought to build a tool that would imitate human code inspection behaviour; As time went on however, it was realized that human code reviewers can regularly be inconsistent according to their own standards. Therefore, it is entirely possible that the proposed algorithm learned widely accepted behaviours and is more *consistent* than human code reviewers which would decrease its experimental performance due to the data set used. As future work, it may be interesting too see the performance of the proposed algorithm over a filtered subset of manually *validated* human code review comments.

As mentioned in Section 4, 75% of issues found during code reviews do not affect the visible functionality of the software; instead, these issues are typically flagged to improve software evolvability. We believe that existing software engineering tools such as defect finders and code refactoring tools have not been developed with this idea in mind. This is the primary reason as to why such tools have assisted only in automating small aspects of code reviews thus far. In contrast, the proposed algorithm is able to find types of defects other tools are not capable of due to two main reasons:

1. **Data Set**: The proposed algorithm trains machine learning models over actual code review comments taken from high quality Java repositories on GitHub.
2. **Approach**: The core belief expressed in the proposed approach is that the relationships between the "naturalness" of elements within source code can be correlated to the way humans view and comment on source code. View Section 6 for further explanation on this idea.

As an example, Figure 8.1 illustrates a sample code review comment in which the reviewer found the throwing of a general exception to be problematic. The proposed tool was able to match the code inspection behaviour of the human code reviewer in this instance. However, over five instances were uncovered in other code reviews in which similar use of a general exception was not considered to be problematic. Interestingly, the proposed algorithm was able to match these behaviours, and did not consider the code to be problematic as well. This example highlights the dynamic, multi-layered ability of the proposed algorithm in reading code the way humans would, as opposed to existing tools that are built in a binary fashion. Thus, while the proposed algorithms mantained a precision score of 23.45%, the level of sophistication of code inspection behaviours it was able to replicate make it an approach with a great deal of potential and worth further exploration in the future.

Lastly, a notable feature of the proposed algorithm is its training time, illustrated in Figure 7.1. First, the algorithm utilizes an n-gram language model which has shown to be less time consuming to train than its NN base counterparts. Additionally, embeddings in the proposed algorithm are generated over AST representations of the source code that do not possess any nodes that corresponding to literal tokens from the code. This

[0]Sourced from https://github.com/ballerina-platform/ballerina-lang/pull/9481

FIGURE 8.1: Code reviewing behaviour that was replicated by the proposed algorithm over a code snippet. Image sourced from https://github.com/ballerina-platform/ballerina-lang/pull/9481

was designed intentionally so that the entropy value attributed with each node in the AST is a calculation based on all the syntactical tokens represented under that AST node. As a result, the amount of time it takes to generate embeddings is greatly reduced as the ASTs under consideration are much smaller than typical ASTs.

Bibliography

Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. (2014). Learning Natural Coding Conventions. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 281–293. ISBN: 978-1-4503-3056-5.

Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. (2015). Suggesting Accurate Method and Class Names. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: ACM, 38–49. ISBN: 978-1-4503-3675-8.

Allamanis, M. and Sutton, C. (2013). Mining Source Code Repositories at Massive Scale Using Language Modeling. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR '13. San Francisco, CA, USA: IEEE Press, 207–216. ISBN: 978-1-4673-2936-1.

Barnett, M., Bird, C., Brunet, J., and Lahiri, S. K. (2015). Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. ICSE '15. Florence, Italy: IEEE Press, 134–144. ISBN: 978-1-4799-1934-5.

Basharin, G. P., Langville, A. N., and Naumov, V. A. (2004). The life and work of A.A. Markov. *Linear Algebra and its Applications* 386. Special Issue on the Conference on the Numerical Solution of Markov Chains 2003, 3–26. ISSN: 0024-3795.

Bavota, G. and Russo, B. (2015). Four eyes are better than two: On the impact of code reviews on software quality. In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 81–90.

Bell, T., Witten, I. H., and Cleary, J. G. (1989). Modeling for Text Compression. *ACM Comput. Surv.* 21(4), 557–591. ISSN: 0360-0300.

Bernhart, M., Mauczka, A., and Grechenig, T. (2010). Adopting Code Reviews for Agile Software Development. In: *2010 Agile Conference*, 44–47.

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. (2010). A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53(2), 66–75. ISSN: 0001-0782.

Bird, C. and Bacchelli, A. (2013). Expectations, Outcomes, and Challenges of Modern Code Review. In: *Proceedings of the International Conference on Software Engineering*. IEEE.

Bird, C., Carnahan, T., and Greiler, M. (2015). Lessons Learned from Building and Deploying a Code Review Analytics Platform. In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. MSR '15. Florence, Italy: IEEE Press, 191–201. ISBN: 978-0-7695-5594-2.

Bisset, D. L., Filho, E., and Fairhurst, M. C. (1989). A comparative study of neural network structures for practical application in a pattern recognition environment. In: *1989 First IEE International Conference on Artificial Neural Networks, (Conf. Publ. No. 313)*, 378–382.

Boehm, B. and Basili, V. R. (2001). Top 10 list [software development]. *Computer* 34(1), 135–137. ISSN: 0018-9162.

Bruna, J., Zaremba, W., Szlam, A., and Lecun, Y. (2014). Spectral networks and locally connected networks on graphs. English (US). In: *International Conference on Learning Representations (ICLR2014), CBLS, April 2014*.

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., and Varoquaux, G. (2013). API design for machine learning software: experiences from the scikit-learn project. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 108–122.

Cai, H., Zheng, V. W., and Chang, K. C. (2018). A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications. *IEEE Transactions on Knowledge and Data Engineering* 30(9), 1616–1637. ISSN: 1041-4347.

Chen, O. T. .-.-. and Sheu, B. J. (1994). Optimization schemes for neural network training. In: *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*. Vol. 2, 817–822 vol.2.

Chen, S. F. and Goodman, J. (1996). An Empirical Study of Smoothing Techniques for Language Modeling. In: *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*. ACL '96. Santa Cruz, California: Association for Computational Linguistics, 310–318.

Clarke, E., Kroening, D., and Yorav, K. (2003). Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In: *Proceedings of the 40th Annual Design Automation Conference*. DAC '03. Anaheim, CA, USA: ACM, 368–371. ISBN: 1-58113-688-9.

Codacy (n.d.).

Cohen, J. (2006). *Best Kept Secrets of Peer Code Review*. Printing Systems. ISBN: 9781599160672.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. (2005). The ASTRÉE analyzer. In: *Programming Languages and Systems, Proceedings of the 14th European Symposium on Programming, volume 3444 of Lecture Notes in Computer Science*. Springer, 21–30.

Cover, T. M. and Thomas, J. A. (2006). *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing)*. New York, NY, USA: Wiley-Interscience. ISBN: 0471241954.

Datta, S. and Arputharaj, A. (2018). An Analysis of Several Machine Learning Algorithms for Imbalanced Classes. In: *2018 5th International Conference on Soft Computing Machine Intelligence (ISCMI)*, 22–27.

Defferrard, M., Bresson, X., and Vandergheynst, P. (2016). Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett. Curran Associates, Inc., 3844–3852.

Duvenaud, D. K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A., and Adams, R. P. (2015). Convolutional Networks on Graphs for Learning Molecular Fingerprints. In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett. Curran Associates, Inc., 2224–2232.

Ebert, F., Castor, F., Novielli, N., and Serebrenik, A. (2019). Confusion in Code Reviews: Reasons, Impacts, and Coping Strategies. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 49–60.

Electrical, I. of and Engineering., E. (1992). *IEEE Standard for Developing Software Life Cycle Processes*. Piscataway, NJ, USA: IEEE Press. ISBN: 1559371706.

Fagan, M. E. (1976). Design and Code Inspections to Reduce Errors in Program Development. *IBM Syst. J.* 15(3), 182–211. ISSN: 0018-8670.

Fast, E., Steffee, D., Wang, L., Brandt, J. R., and Bernstein, M. S. (2014). Emergent, Crowd-scale Programming Practice in the IDE. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '14. Toronto, Ontario, Canada: ACM, 2491–2500. ISBN: 978-1-4503-2473-1.

Fatima, N., Chuprat, S., and Nazir, S. (2018). Challenges and Benefits of Modern Code Review-Systematic Literature Review Protocol. In: *2018 International Conference on Smart Computing and Electronic Enterprise (ICSCEE)*, 1–5.

FindBugs (n.d.).

GitHub (n.d.).

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Goodman, J. (2001). A Bit of Progress in Language Modeling. *Computer Speech Language* 15, 403–434.

Gousios, G., Pinzger, M., and Deursen, A. v. (2014). An Exploratory Study of the Pull-based Software Development Model. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 345–355. ISBN: 978-1-4503-2756-5.

Habib, A. and Pradel, M. (2018). How many of all bugs do we find? a study of static bug detectors. In: 317–328.

Habib, A. and Pradel, M. (2019). Neural Bug Finding: A Study of Opportunities and Challenges. *CoRR* abs/1906.00307.

Hamilton, W. L., Ying, R., and Leskovec, J. (2017). Inductive Representation Learning on Large Graphs. *CoRR* abs/1706.02216.

Hasanin, T. and Khoshgoftaar, T. (2018). The Effects of Random Undersampling with Simulated Class Imbalance for Big Data. In: *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, 70–79.

Hellendoorn, V. and Devanbu, P. (2017). Are Deep Neural Networks the Best Choice for Modeling Source Code? In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 763–773. ISBN: 978-1-4503-5105-8.

Hindle, A., Barr, E. T., Su, Z., Gabel, M., and Devanbu, P. (2012). On the Naturalness of Software. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 837–847. ISBN: 978-1-4673-1067-3.

Hsiao, C.-H., Cafarella, M., and Narayanasamy, S. (2014). Using Web Corpus Statistics for Program Analysis. In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '14. Portland, Oregon, USA: ACM, 49–65. ISBN: 978-1-4503-2585-1.

Huang, X., Acero, A., and Hon, H.-W. (2001). *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR. ISBN: 0130226165.

Indurkhya, N. and Damerau, F. J. (2010). *Handbook of Natural Language Processing*. 2nd. Chapman & Hall/CRC. ISBN: 1420085921, 9781420085921.

Jiang, J., Lo, D., He, J., Xia, X., Kochhar, P. S., and Zhang, L. (2017). Why and how developers fork what from whom in GitHub. *Empirical Software Engineering* 22(1), 547–578.

Jimenez, M., Maxime, C., Le Traon, Y., and Papadakis, M. (2018). On the Impact of Tokenizer and Parameters on N-Gram Based Code Analysis. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 437–448.

keras-team (2019). *Keras*. https://github.com/keras-team/keras.

Kingma, D. P. and Ba, J. (2014). *Adam: A Method for Stochastic Optimization*. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015.

Kipf, T. N. and Welling, M. (2016). Semi-Supervised Classification with Graph Convolutional Networks. *ArXiv* abs/1609.02907.

Kotsiantis, S., Kanellopoulos, D., and Pintelas, P. (2005). Handling imbalanced datasets: A review. *GESTS International Transactions on Computer Science and Engineering* 30, 25–36.

Kusner, M., Sun, Y., Kolkin, N., and Weinberger, K. (2015). From word embeddings to document distances. In: *International conference on machine learning*, 957–966.

LaToza, T. D., Venolia, G., and DeLine, R. (2006). Maintaining Mental Models: A Study of Developer Work Habits. In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, 492–501. ISBN: 1-59593-375-1.

Levy, O. and Goldberg, Y. (2014). Neural Word Embedding as Implicit Matrix Factorization. In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Inc., 2177–2185.

Li, J., Ouazzane, K., Kazemian, H. B., and Afzal, M. S. (2013). Neural Network Approaches for Noisy Language Modeling. *IEEE Transactions on Neural Networks and Learning Systems* 24(11), 1773–1784.

Lipitakis, A. and Lipitakis, E. A. E. C. (2014). On Machine Learning with Imbalanced Data and Research Quality Evaluation Methodologies. In: *2014 International Conference on Computational Science and Computational Intelligence*. Vol. 1, 451–457.

MacLeod, L., Greiler, M., Storey, M., Bird, C., and Czerwonka, J. (2018). Code Reviewing in the Trenches: Challenges and Best Practices. *IEEE Software* 35(4), 34–42. ISSN: 0740-7459.

Mäntylä, M. V. and Lassenius, C. (2009a). What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering* 35(3), 430–448. ISSN: 0098-5589.

Mäntylä, M. V. and Lassenius, C. (2009b). What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering* 35, 430–448.

Mcintosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2016). An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Softw. Engg.* 21(5), 2146–2189. ISSN: 1382-3256.

Mikolov, T., Chen, K., Corrado, G. S., and Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781.

Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In: *INTERSPEECH*. Ed. by T. Kobayashi, K. Hirose, and S. Nakamura. ISCA, 1045–1048.

Niepert, M., Ahmed, M., and Kutzkov, K. (2016). Learning Convolutional Neural Networks for Graphs. In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by M. F. Balcan and K. Q. Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 2014–2023.

Oualil, Y., Singh, M., Greenberg, C., and Klakow, D. (2017). Long-Short Range Context Neural Networks for Language Modeling. *CoRR* abs/1708.06555.

Ouni, A., Kula, R. G., and Inoue, K. (2016). Search-Based Peer Reviewers Recommendation in Modern Code Review. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 367–377.

Owhadi-Kareshk, M. and Nadi, S. (2019). Scalable Software Merging Studies with MergAnser. In: *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR '19. Montreal, Quebec, Canada: IEEE Press, 560–564.

Pascarella, L., Spadini, D., Palomba, F., Bruntink, M., and Bacchelli, A. (2018). Information Needs in Contemporary Code Review. *Proc. ACM Hum.-Comput. Interact.* 2(CSCW), 135:1–135:27. ISSN: 2573-0142.

Pérez-Ortiz, M., Gutiérrez, P. A., Tino, P., and Hervás-Martínez, C. (2016). Oversampling the Minority Class in the Feature Space. *IEEE Transactions on Neural Networks and Learning Systems* 27(9), 1947–1961. ISSN: 2162-237X.

PMD (n.d.).

Pradel, M. and Sen, K. (2018). DeepBugs: A Learning Approach to Name-based Bug Detection. *Proc. ACM Program. Lang.* 2(OOPSLA), 147:1–147:25. ISSN: 2475-1421.

Prusa, J. D. and Khoshgoftaar, T. M. (2017). Improving deep neural network design with new text data representations. *Journal of Big Data* 4(1), 7. ISSN: 2196-1115.

Ray, B., Hellendoorn, V., Godhane, S., Tu, Z., Bacchelli, A., and Devanbu, P. (2016). On the "Naturalness" of Buggy Code. In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 428–439. ISBN: 978-1-4503-3900-1.

Robbins, H. and Monro, S. (1951). A Stochastic Approximation Method. *Ann. Math. Statist.* 22(3), 400–407.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning Representations by Back-propagating Errors. *Nature* 323(6088), 533–536.

Shannon, C. E. (1951). Prediction and entropy of printed English. *The Bell System Technical Journal* 30(1), 50–64. ISSN: 0005-8580.

Siivola, V. and L. Pellom, B. (2005). Growing an n-gram language model. In: 1309–1312.

Singh, D., Sekar, V. R., Stolee, K. T., and Johnson, B. (2017). Evaluating how static analysis tools can reduce code review effort. In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 101–105.

SonarQube (n.d.).

Sundermeyer, M., Ney, H., and Schlüter, R. (2015). From Feedforward to Recurrent LSTM Neural Networks for Language Modeling. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 23(3), 517–529. ISSN: 2329-9290.

Sutherland, A. and Venolia, G. (2009). Can Peer Code Reviews be Exploited for Later Information Needs? In: *Proc. ICSE 2009*. IEEE.

Tao, Y., Dang, Y., Xie, T., Zhang, D., and Kim, S. (2012). How Do Software Engineers Understand Code Changes?: An Exploratory Study in Industry. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. Cary, North Carolina: ACM, 51:1–51:11. ISBN: 978-1-4503-1614-9.

Tao, Y. and Kim, S. (2015). Partitioning Composite Code Changes to Facilitate Code Review. In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (MSR)*. Los Alamitos, CA, USA: IEEE Computer Society, 180–190.

Thai-Nghe, N., Gantner, Z., and Schmidt-Thieme, L. (2010). Cost-sensitive learning methods for imbalanced data. In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*, 1–8.

Tseng, T.-H., Yang, T.-H., and Chen, C.-P. (2016). Verifying the long-range dependency of RNN language models. *2016 International Conference on Asian Language Processing (IALP)*, 75–78.

Votta Jr., L. G. (1993). Does Every Inspection Need a Meeting? In: *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT '93. Los Angeles, California, USA: ACM, 107–114. ISBN: 0-89791-625-5.

Wang, S., Chollak, D., Movshovitz-Attias, D., and Tan, L. (2016). Bugram: Bug Detection with N-gram Language Models. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: ACM, 708–719. ISBN: 978-1-4503-3845-5.

Werbos, P. and John, P. (1974). Beyond regression : new tools for prediction and analysis in the behavioral sciences /.

Williams, W., Prasad, N., Mrva, D., Ash, T., and Robinson, T. (2015). Scaling Recurrent Neural Network Language Models. *CoRR* abs/1502.00512.

Zanjani, M. B., Kagdi, H., and Bird, C. (2016). Automatically Recommending Peer Reviewers in Modern Code Review. *IEEE Transactions on Software Engineering* 42(6), 530–543. ISSN: 0098-5589.