NeuroLog: Predicting logging statements via a Graph Neural Network

Georgios Nikolopoulos
University College London
London, UK
Athens, Greece
nikolopoulosgwork@gmail.com

ABSTRACT

Logging is one of the most important programming practices, as it enables developers to perform post-mortem and runtime analysis of their programs. A logging statement within source code has two components: the logging severity, which indicates the criticality of the statement, and the logging message, which contains any relevant runtime information the developer wishes to store. Over the last few years, research has explored *where to log* considerably, both to make recommendations for logging locations and to understand how and why developers log. *What to log* has also been explored, but to a smaller extent. Research has mostly focused on creating recommendations for already existing logging statements, with very little work attempting to outright predict the severity level or message of a logging statement. To help developers with *what to log* in a newly formed logging statement and to advance logging related research, we introduce NeuroLog. A corpus containing 20.066 logging statements is collected and analyzed. Almost 9.000 graphs are then used in combination with a Graph Neural Network for two purposes: NeuroLog trains a model that can predict the severity level of a logging statement, achieving a 97.03% accuracy on previously unseen data. Additionally, NeuroLog attempts to train a model to predict the logging message of a statement. Ultimately, the model is not successful, but much can be learned from this failure, with recommendations being made for future work.

ACKNOWLEDGMENTS

Dr. Earl Barr and Dr. Miltos Alamanis for their guidance, without which NeuroLog would have never been created.

My family, my friends and Anastasia, their support is invaluable.

1. INTRODUCTION

Logging is the de facto programming practice of choice for developers to utilize when the system runtime information of an application must be collected. Logging is done by adding logging statements to the application's source code, which outputs any required information to a database, file or the console. A logging library, such as Log4J or LogBack [21, 22], is usually utilized to assist developers by handling most of the low-level requirements of logging. Logging is used primarily for post-mortem analysis in case of issues [40] or for the analysis of the runtime information of an application.

A logging statement can be broken down to two components. The first one is the log verbosity level, which indicates the log's severity by "tagging" it with an appropriate level. Level names vary depending on the programming language and logging library used, but Java has seen the proliferation of the following levels: trace, debug, info, warning, error and fatal [35]. Each level represents a different criticality level, starting with trace and debug, which are used for mundane reporting and reaching error and fatal, which are used when serious issues and errors occur. Levels are particularly useful, as they allow developers to filter the log output based on a level in order to expedite the analysis of run-time information of an application.

The second component of a logging statement is that of the log message. The message contains runtime information that the developer wishes to log and is created by using static text, most often in combination with runtime variables or method calls. The message, in tandem with its severity level, paints a clear picture of the runtime status of an application, containing all required information to allow for fast filtering and analysis.

Prior research has extensively focused on the analysis of logging statements for the purpose of anomaly detection, security monitoring and other useful goals [8, 23]. Research has also examined avenues in logging improvement, with examples such as Zhu et al.'s pioneering work on where to log [40], LogEnchancer [36] and ErrLog [34]. However, most research is a few years old, and with a few exceptions, almost no work utilizes the power of neural networks to achieve its goals. Additionally, the subject of predicting *what to log* has not been explored, with the exception of Anu et al. [3] who uses contextual features and combines them with a machine learning model to assist with logging severity. Furthermore, recent research has explored the application of machine learning methods for variable selection, as well as predicting data types and variable names [1, 15, 2, 25]. This research employs the power of a Graph Neural Network (GNN), which utilizes a graph-based input to train a machine learning (ML) model for a specific purpose . The research first converts source code into a graph-based representation that contains the code's unique features. The graphs are then used with a GNN for various purposes. However, none of this recent research has been applied to logging prediction.

In this thesis, we explore these gaps in research by introducing NeuroLog, a set of tools written in Python that allow for the analysis of Java source code in order to make recommendations for the logging level and message of a statement. NeuroLog uses cutting edge techniques, such as the graph representation of source code [1, 7, 2] in combination with the strength of a Graph Neural Network[1, 27, 11] to generate and train a model that achieves an 97.03% accuracy when attempting to predict severity levels. Unfortunately, NeuroLog is unable to predict the logging message, but much insight can be gained by analyzing the causes of this failure. The following is a summary of Neurolog's sequential developmental and research steps:

Step 1) Convert Java source code into a graph that contains nodes, which represent the source code and directional edges that contain information on the relationship between nodes (ex. NEXT_TOKEN, CHILD, etc.). These graph files are built using Google's protocol buffer technology, are directly inspired by the work of Alamanis et al. [1] and are created with Andre Rice's features plugin [26]. The source code of a few different open-source Java projects has been converted by Andre Rice and stored into a corpus for the University of Cambridge to use in one of its modules. This corpus was shared to us by Miltos Alamanis and utilized over the next development steps. Java is used mainly because it is one of the most statically typed languages, meaning that the generated graph will contain a significant amount of information within it.

Step 2) Analyze the graph files to detect any logging statements contained within the corpus. This is done by examining the graph's content (i.e. the source code) and looking for specific keywords that indicate a call to one of the most popular logging libraries, such as "logger", "log" or "LOG". The logging statement's severity, message and other useful information is then extracted and saved. Using this keyword based approach, 8954 statements were discovered and stored.

Step 3) Creation of a new corpus containing modified graph files in preparation for ML training. Each graph file that contains one or more logging statements is modified following a specific algorithm. The result of this modification is a complete concealment of the logging statement in addition to several other ML-beneficial graph transformations. For an in-depth explanation of the algorithm and its reasoning, please see Chapter 4.1.

Step 4) Training of the Graph Neural Network. A Graph Neural Network is the best fit for the problem at hand, as it has been optimized to work with graph-based inputs. NeuroLog's training is handled via Microsoft's ptgnn tool, a PyTorch-based GNN training library [42]. For speed and efficiency, the actual training is done with Azure Machine Learning, using a state-of-the-art GPU cluster. The training first requires the conversion of the modified corpus to a specific input for ptgnn to parse. Following that, several experimentation and optimization training sessions are conducted, resulting in a trained model that can predict the logging level of a statement with 97.03% accuracy.

Furthermore, NeuroLog attempts to predict the logging message of a statement by following these development steps, in addition to the steps described above:

Step 1) Tokenization training. In order to train a tokenizer that can tokenize a log message for the message prediction model training, a second dataset of logging statements is collected through the analysis of several open-source Java projects. The two datasets are then combined to form a text only (i.e. non-graph and not usable for ML training) dataset that is 20.066 statements in size. This dataset can effectively be used to explore and understand the practices that developers utilize.

Step 2) Modification of ptgnn. ptgnn's source code is modified to optimize it for the task at hand by adding support for the Bilingual Evaluation Understudy Score (BLEU). Finally, much like the severity model training, ptgnn is used to try and create a model to predict the logging messages. Unfortunately, this proved to be unsuccessful due to a few possible reasons, analyzed in Chapter 4.3.

In order to allow for ease of reproduction in addition to enabling future researchers to utilize and evolve NeuroLog, the source code has been uploaded to GitHub [9]. Additionally, the original graph corpus, the intermediate log search results and the final, modified corpus in combination with the trained ML models have been uploaded to OneDrive [24], where they may be freely downloaded.

The thesis is organized as follows: Chapter 2 covers and summarizes related research. Chapter 3 discusses the collection of the logging statement dataset and analyzes it to gain insight on how and why developers log. Chapter 4 performs an in-depth analysis of all the sequential development steps taken for the creation of NeuroLog. Chapter 5 presents and analyzes the results obtained from NeuroLog's severity and message predictions models. Chapter 6 analyzes possible threats to validity and finally, Chapter 7 presents future avenues for researchers to explore.

2. RELATED WORK

2.1 Background

Stemming from computer science, software engineering (SE) research has become an important field over the last few decades, due to its impact on academia as well as the software development world [31]. More recently, SE research has begun exploring the analysis of source code, most often with the intention of creating algorithms, tools and techniques that expand the field. Additionally, this exploration has had the effect of providing insight into the coding practices of developers, which allows for recommendations to be made on them.

Over the last few years, a subfield of SE research that has seen extensive research growth is logging related research. Such research focuses exclusively on the analysis of logging statements within source code in addition to their text-based output for a variety of goals.

Additionally, recent breakthroughs in AI research have seen the proliferation of Graph Neural Networks. These networks successfully combine the power of neural networks with the extensive data included in graph-based representation of real world elements.

This Chapter briefly summarizes the most recent and important contributions to both these sub-fields, before identifying the gap in research that NeuroLog tries to address by using a GNN for logging related research.

2.2 Previous Research

2.2.1 Log Analysis

With logging being relevant in both the industry and in academia, extensive research has been performed on the analysis of logging statements and their outputs for a variety of purposes. An example is the work conducted by Yuan et al. [37], which sees the creation of *SherLog*, a tool that is capable of automatically inferring what part of source code caused a failure, based on the output logs the failure generated. The approach of using the output of logging statements for failure analysis has been further explored, with the work of Xu et al. [33] and Fu et al. [8]. These research papers both systematically analyze the text messages contained within log files before combining them with source code analysis to train a machine learning model. The model can then be used with log files to detect anomalies and faults during execution. Finally, research on log analysis has also examined logs for the purposes of test analysis [17] as well as for natural language analysis [14].

An element that is common in most of the aforementioned research papers is that a significant amount of labor-intense work is required to create source code features that can be used with the machine learning approach the papers are utilizing. This work is very manual and convoluted, requiring extreme knowledge of state-of-the-art algorithms and their application before any substantial results can be achieved.

2.2.2 Logging Practices

The logging practices of developers have also been analyzed by several research teams in order to understand what forms a good logging statement and how developers choose severity levels or messages. Some examples of such research include the work done by Yuan et al. [35], which analyses the repositories of four large-scale, open-source projects. Specifically, the revision history of logging statements is examined to understand how developers log. As a proof of concept, the paper also implements a very simple tool that identifies problems with the assignment of severity levels. Other research has explored the logging research field by analyzing source code for anti-pattern detection [4]. This is achieved via the application of a custom tool, *LCAnalyzer*, to the source code of ten open-source projects. The tool detects anti-patterns that the researchers then analyze to gain insight and make recommendations on logging practices. Finally, the link between logging characteristics and code quality has been the work of Shang et al. [29]. A case study on four different open-source projects is performed to check the correlation between defect densities and the existence of logging statements. The researchers use this correlation to again make recommendations on how developers should log.

It is worth noting that almost all research papers in the sub-field of logging research collect a dataset of logging statements or their output. These papers then analyze their respective datasets to create logging-related recommendations for developers to follow. While this sub-chapter discusses research papers that explicitly intend to analyze the logging practices of developers, many valid discussions, insights and recommendations on practices can be found in most research papers that this thesis discusses or references.

2.2.3 Logging Recommendations

Logging recommendations can be split into two central components; *where to log* and *what to log*. Work in the *where to log* category focuses on the source code placement of logging statements and how to improve it. Work in the *what to log* category focuses on improving the content of

logging statements. This category encompasses the two central components of a logging statement: the severity level as well as the message. Furthermore, research in the *what to log* category can focus on the prediction of severity levels or messages of newly formed statements, in addition to the enhancement of already existing logging statements.

Where to Log: *Where to log* has seen some great exploration by researchers, with an example being the creation of LogAdvisor by Zhu et al. [40]. LogAdvisor was developed by first extracting structural, textual and syntactical features from the source code of two industrial projects and two open-source codes. These features are then further processed with feature selection techniques before training a classification model that can reliably predict where a logging statement should be placed in the source code. Similarly, Jia et al. [16] explores *where to log* by creating two custom feature extraction models, IDM and GIDM, and applying them on already existing logging statements. These models are designed to capture the semantics of log intention and the semantic equivalence between different log contexts, respectively. The models are used as the base of SmartLog, which is superior to its state-of-the-art counterparts in improving the location of logging statements in source code.

What to log: What to log has also been researched, starting with the work of Yuan et al. [36] which oversees the creation of *LogEnchancer*, a tool that improves already existing logging statements by adding relevant variables to the message component. *LogEnchancer* creates a custom algorithm that utilizes program analysis in combination with a constraint solver to detect variables that can enhance a logging message. Zhao et al. [38] present *Log20*, a tool that is capable of recommending *where* a logging statement should be placed in addition to *what* should be contained within, in the form of variable recommendation. Similarly to *LogEnchancer*, *Log20* operates via a custom dynamic programming algorithm that takes a program's runtime execution path and its frequencies as inputs. The algorithm then outputs a location for the logging message to be written at, in addition to a few variable recommendations for it.

There has been only one previous research paper that attempts to predict the severity of a logging statement: the work of Anu et al. [3], which created *VerbosityLevelDirector*. The tool was built by first creating a custom, feature extraction and selection algorithm. The algorithm uses code blocks to extract the required features before using a complex feature selection process. Data-clustering techniques are also used for noise reduction. The entire process is then applied to four open-source projects and the filtered features, alongside a dataset of detected logging statements, is used to train a classification model that can predict the severity levels of logging statements.

The observation that was made in Chapter 2.2.1 can also be made here. All of the work discussed in this sub-chapter achieves good results by creating and utilizing extremely complex and labor-intensive feature extraction algorithms. These algorithms are usually custom-built for the problem at hand and cannot be easily applied to other research work. Additionally, research has exclusively focused on the improvement of already existing logging messages, with no research conducted on the prediction of logging messages.

2.2.4 Graph Neural Networks

Graph Neural Networks, which stem from Convolutional Neural Networks, have had a large amount of research exploration recently due to the fact that many elements in the real world can be converted to graphs [39]. These graphs are especially useful when combined with a GNN, which is capable of extracting extremely complex features from them and using them for a variety of deep-learning purposes.

Recent SE research by Alamanis et al. and Fernandes et al. [1, 7] has explored the conversion of source code to a graph-based representation. This is achieved by encoding the source code contents into graph nodes and the relationships between the nodes as directed edges. The directed graphs can then be used with a GNN for various purposes, such as data type prediction [2] or the naming and selection of variables [1].

2.3 Gaps in Research

There are two gaps in research that NeuroLog tries to explore and address:

The application of GNNs to logging-related research. As discussed in this chapter, most logging research, even if it uses modern machine learning approaches, requires an immense amount of work to extract source code features. Even more work is needed for feature selection and other data modification techniques, such as noise reduction. This work is also usually optimized manually by researchers, further increasing development and research time costs. This has the unfortunate effect of increasing the barrier of entry for such research, as a researcher needs to have a heavy background in all of these areas before attempting logging-related research. NeuroLog instead uses a graph representation of source code in combination with a GNN to streamline the entire approach, greatly reducing the amount of work needed to train a classification model, while simultaneously creating a training approach that significantly reduces the research barrier of entry. It is important to note that feature extraction still occurs with NeuroLog, but it is a much more simplified process that is not custom built for severity or message prediction and can be effectively reused for other goals with little to no modification. A GNN can then "understand" the large sum of extracted data that is contained within a graph and learn how to best use it for the goal at hand.

Prediction of logging messages. To the best of our knowledge, no research has ever attempted to completely predict the messages of newly formed logging statements. NeuroLog is thus the first tool that attempts to achieve such a goal. While it is ultimately unsuccessful, much can be gained by analyzing the reasons for failure in order to make recommendations for future research.

3. LOG COLLECTION & OBSERVATIONS

To discuss the insight gained from analyzing the dataset of the logging statements, we must first explore how these logs were collected. As mentioned previously, the 20.066 statements were collected on two different instances. The first collection instance occurred when the supplied graph corpus was searched for logging statements, providing 8.954 unique logging statements. The second, which yielded 11.112 statements, occurred much later in the development of NeuroLog, when the training of a tokenization library required additional examples of logging messages. It is important to note that only the first 8.954 statements were used in the training of either the severity or message prediction ML models. The reasoning for this is covered in Chapter 4.3. This chapter will discuss how the logs were collected for both instances as well as the observations that can be made from the combined dataset.

```
Ex 1: logger.info("Creating _Listener that should also receive the thrown exception.");

Ex 2: logger.debug("About to check the payload" + "\n  Received: {}", payload);
```

Figure 1 Examples of Logging Statements

3.1 Conversion of Java to Cyclic Graphs

The provided corpus was generated using the features-javac, javac plugin [26], which is used to extract a feature graph from a Java source file that can be used with machine learning models. A program is represented as a directed graph with many edge types that model the relationships between the graph nodes, i.e the source code. The graph's central component is the abstract syntax tree (AST) which contains syntax nodes and syntax tokens. These are created via the analysis of the source code in order to extract its content as nodes in the graph.

Additionally, the program's structure, flow of control and flow of data is captured via specific edge types. Some examples are CHILD, which connects the components of the AST, NextToken, which connects syntax and LastRead or LastWrite, which track when a variable was last read

Table 1 Breakdown of Severity Levels Per Project

Project	trace	debug	info	warn	error	fatal	Total
Hystrix	0	28	3	33	33	0	97
cassandra	334	332	456	256	251	0	1629
dubbo	21	41	154	255	150	0	621
elasticsearch	836	1222	3789	534	460	3	6844
geronimo	0	52	26	51	61	0	190
grails-core	5	113	6	22	28	0	174
guice	0	0	6	0	0	0	6
hadoop	2	21	20	11	15	0	69
hbase	3	1	0	5	582	0	591
hibernate-orm	533	745	21	44	44	0	1387
kafka	203	459	377	186	276	0	1501
netty	26	217	43	148	29	0	463
okhttp	0	0	16	6	0	0	22
qpid	27	710	355	222	191	0	1505
sprint	51	113	130	34	27	0	355
spring-framework	354	860	128	128	125	1	1596
tomcat	113	808	186	363	539	8	2017
wildfly	584	172	57	73	111	2	999
Total	3092 (15.4%)	5894 (29.37%)	5773 (28.77%)	2371 (11.82%)	2922 (14.56%)	14 (< 0.001%)	**20066**

from or written to. There are several other edge types, with each being responsible of capturing useful program information. For more information on the graph generation process, please see Alamanis et al. [1].

3.2 Log detection

3.2.1 Gathering of the First Log Set

The corpus of graph files shared by the University of Cambridge originally had 16 different Java projects within it. The first step in logging detection that was taken was to manually check each project's source code to see if any logging statements existed inside of it. Figure 1 displays two examples of a logging statement within the source code of qpid-jms. Most logging libraries operate in the displayed way: that is, an invocation to the logging library via a named variable, followed by a function call which indicates the logging severity and has the logging message passed to it as a string. Words indicating a severity level (warn, error, etc.) or common logging variable names, such as logger or LOGGER, can be easily used in simple text-search tools such as grep to indicate whether a project's source code contains logging statements. This is precisely how the 16 different Java projects were manually examined. From the 16, only 9 had logging statements within them. The rest of the projects were discarded, and a more in-depth, error-free logging detection procedure could begin.

The first set, containing 8954 logging statements, was detected via in-graph analysis. Detecting the statements in the program graphs is preferred over textual detection in source code for two central reasons: Firstly, in order to prepare the corpus for ML training after the log detection process is complete, a few graph modifications are required. These graph modifications need information such as the in-graph location of the logging statement; this information cannot be discovered by textual analysis. More information on the process can be found in Chapter 4. Secondly, the graph files are encoded using Google's protocol buffers, resulting in much faster processing speeds when compared to any textual analysis due to its storage in a binary format.

The in-graph log detection was performed by a script that traverses each graph's nodes and checks the node's contents. Specifically, and very similarly to manual detection, the content had to be a variable call having a standard name, such as logger or LOGGER or log. Following a True comparison, the script would advance two nodes and check whether the element following the dot node was one of the preset severity levels, trace, debug, info, warn error or fatal. If the severity comparison was also True, the script would transverse as many nodes as it had to in order to reach a semicolon, indicating the end of the statement. Finally, it would extract the logging message, contained within the two outermost parentheses, and then save all relevant information, such as the severity level, logging message, project name and others to a JSON file. The process would be repeated for each and every project's graph files, eventually leading to a JSON array which had 8954 different statements contained within.

3.2.2 Gathering of the Second Log Set

The second set of logging statements was collected in order to enhance the tokenization of logging messages for the training of the message prediction model (Chapter 4.3), and an attempt was made to reuse the in-graph detection script by converting the source files into graphs. Unfortunately, doing so requires the compilation of the project's source code, which is a difficult task when working with very large, relatively old Java projects. Many different compilation issues were encountered during the building process, and eventually the in-graph analysis was abandoned in favor of a more traditional textual analysis that did not require project compilation. The textual search was conducted in a similar fashion to the in-graph search. Keywords such as logger or LOGGER were used in combination with severity searches and once a logging statement was discovered, its severity, message and project information was stored, ready for the tokenization training process. The second set contains 11.052 unique logging statements. Finally, these two sets were combined to form the final dataset containing 20.006 statements, in the form of a tab-separated file available on OneDrive [24].

3.3 Severity Insights & Recommendations

3.3.1 Developers do not use the fatal level

Table 1 displays all logging statements detected, broken down by severity levels. It is immediately apparent that there is a significant lack of fatal-level logs in all the projects. Out of the 20.066 logging statements, only 14 of them are of the fatal severity level, less than 0.001% of the entire

dataset. This clearly indicates that developers are unwilling to utilize the fatal severity level. In fact, it seems that developers do not want to use the fatal level at all, only using the other severity levels instead.

There are no official guidelines when it comes to logging statements [3]. Each implementation of a logging library makes its own recommendations on what the logging levels mean and how they should be utilized. It would be incorrect to attribute this lack of fatal logs to guidance instructing developers to ignore the fatal level altogether, as there is no indication of such a rule anywhere. Instead, it seems that developers have, much like many other elements in programming, ignored the fatal level in a natural, unguided way as programming and logging practices evolved.

The key recommendation that can be made by this insight is that there is no need for a fatal severity level. Moving forwards, a new logging library could completely discard this severity level in order to reduce confusion and simplify the logging process.

3.3.2 The trace level is often unutilized

The severity levels can be split into two sub-categories. The first one contains the trace, debug and info levels and is used when something relatively mundane needs to be reported. The rest of the levels, warn, error and fatal are used when a potentially or outright dangerous event occurs. When these two categories are examined side by side, an interesting observation becomes apparent. For both categories, only two out of the three levels have similar overall usage percent values. Debug and info both encompass roughly an equal share of the total dataset, at approximately 29%. However, trace is at a much lower value of approximately 15%. The exact same occurs when examining warn, error and fatal. Warn and error encompass 12-15% of the dataset but fatal sits at a disproportionately lower <0.001%. This difference in percent values suggests that trace, similarly to fatal, is avoided by developers in favor of the other two levels that report mundane events, debug and info. Of course, unlike fatal, developers do sometimes use the trace level, which indicates that the info level is not shunned like the fatal level.

Much like 3.3.1 discusses, it would be unwise to suggest that this is a deliberate choice by the development world. Again, this lack of usage of the trace level occurs in a natural, unguided way. The recommendation that was made in 3.3.1 for the fatal level could be made for the trace level as well. Perhaps it would be best to remove both the fatal and the trace levels. This would greatly simplify and streamline the logging process and reduce the amount of overhead for a developer. It would also have the benefit of reducing the work required for post-mortem analysis.

However, the recommendation to remove the trace level cannot be made as easily as it was made for the fatal level. Developers do use the trace level and whether to completely remove it is not an easy question to answer. A case study which examines what developers want and how they view this recommendation would be a great idea for future research work, while also cementing the validity of the idea.

Table 2 Various Statistics on Log Messages

Statistic	Value
Average message length (characters)	56.82
Minimum message length (characters)	0
Maximum message length (characters)	1203
Messages with variables	13165 (66%)
Messages with method calls	5770 (29%)
Messages with pure text	1131 (6%)

3.3.3 There is a lack of official guidelines for levels

Logging as a practice is extremely important to the development world. No matter the size of a project, having the capability to store run-time information for analysis is of the essence. However, logging severities have no official guidelines that define them and their correct use [3], leading to confusion and miscommunication on a critical development practice. It is easy for a single research paper or thesis such as this one to make recommendations for a single group of developers to adopt. The best course of action, however, is for developers to standardize the logging levels by examining all research conducted on them and making an informed, calculated decision on which logging levels to keep and how to best define them.

3.4 Message Insights

Table 2 and Figure 2 display various statistics that were obtained by analyzing all the messages within the logging statement dataset. The analysis was performed with the assistance of regex and the search parameters can be freely viewed alongside NueroLog's source code, on Github [9]. Close examination of these results yields some interesting insights on what developers log and how they log it.

3.4.1 Logging of pure text is rarely done

The underlying goal of logging is to log as much relevant information as possible, in order to enhance the effectiveness of logging and allow for easier post-mortems and general analysis of issues. One assumption that can be made is that this goal would usually entail the logging of some kind of runtime information instead of pure text. The data gathered for NeuroLog does indeed confirm this assumption: developers will almost always log runtime data obtained either by direct access to variables or by returning strings from methods.

3.4.2 Logging of variables is preferred over method calls

It also seems that developers prefer to directly access and log variables over calling methods that return the log information needed. The reason for this is not immediately apparent. One possible explanation is that method calling could often require the importing of other source packages and files that contain the required method. Doing so would imply that non-local (i.e. within the script that is currently logging) information is required. However, developers would usually need to store local information only, hence the preference for direct variable accessing.

It may also be entirely possible that the called methods are instead mostly contained within the logging script, meaning that the usage of methods or variables is virtually identical and plays no difference in the logging patterns of developers. This could be a subject of future research that focuses on pattern discovery via a questionnaire and source code analysis. Studying the logging message patterns of developers could assist in the creation of logging libraries, in addition to strengthening the benefits of logging.

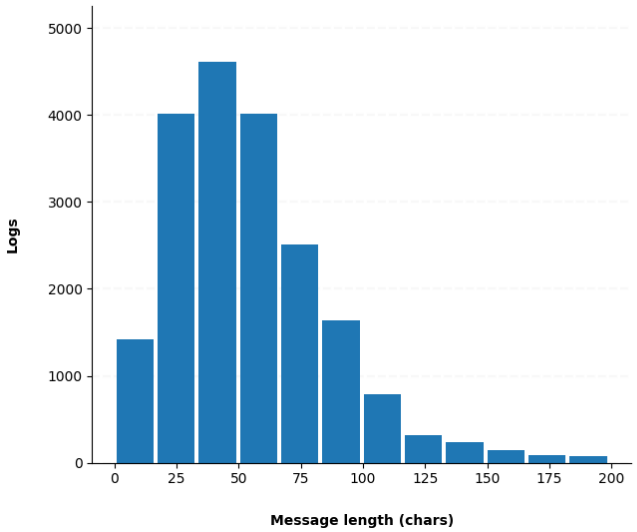

Figure 2 Message Length Histogram (<200 chars)

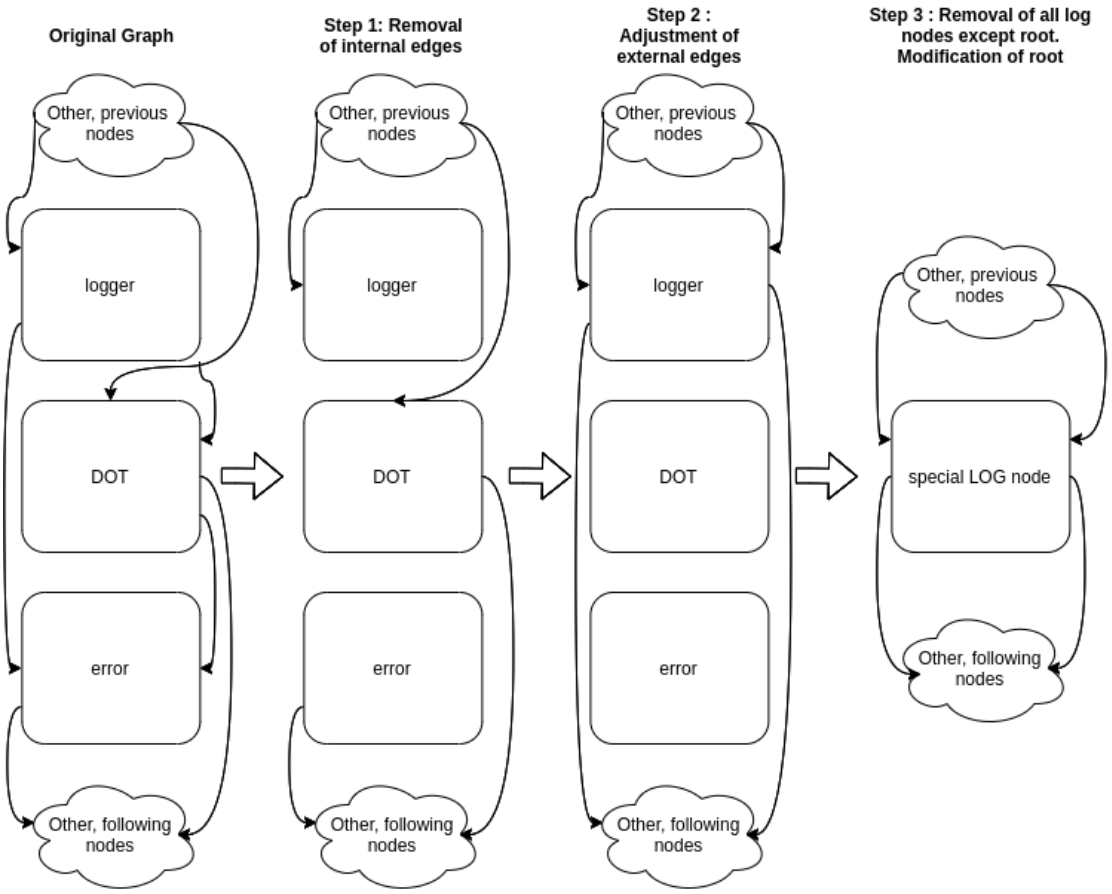

Figure 3 Graph Modification Process

3.4.3 Developers keep messages relatively short

Figure 2 presents a histogram of the message length of all messages that have less than 200 characters (>99% of data). The histogram suggests that developers prefer to keep the log message relatively short, with most messages having approximately 20-80 characters within them. At the same time, having extremely small messages of under 20 characters is avoided, most likely because such messages do not provide adequate logging information. Larger logging messages of over 90 characters are also avoided, possibly for a similar reason: having too much information to go through will result in delays and difficulties in understanding the message.

It is crucial to note that the message length in the source code does not equal the message length in a log output. As most messages contain calls to methods or direct variable access, or both, the logging output will usually be larger than the source code statement that creates it. Simultaneously however, the analysis of the source code message is of use, as it indicates what developers choose to prepend to the output message, which is usually the information that defines the logged variable or return data.

4. MACHINE LEARNING APPROACH

Following the detection and collection of the first set of approximately 9000 logging statements as well as relevant graph location data via in-graph analysis, the development of the machine learning component of NeuroLog began. This chapter analyses the various developmental steps that were taken in order to train the severity and message prediction models.

4.1 Corpus Generation via Graph Modification

In order to successfully train a model to predict either the severity or the message of a logging statement, a new, modified corpus of graph files is generated. This corpus consists of one graph file per logging statement and is generated by modifying each log's respective graph in a sequence of steps. Having one graph file per logging statement ensures that the model cannot be "confused" by having multiple targets per file and that it can identify where the unique prediction statement is in the graph file. The aim of the modification is to prepare the graph for machine learning by removing any data that might give away the prediction target, without damaging the graph's internal structure and flow of data logic. Figure 3 displays a visualized, step by step modification of a simplified set of nodes that all represent a single logging statement. For ease of understanding, this logging statement's message has been omitted. In detail, the steps are as follows:

Step 1: Removal of internal edges. Any edges that both originate and "point" to one of the nodes that belong to the logging statement are completely removed. These edges represent relationships between the log nodes and should not be in the graph following the deletion procedure in step 3.

Step 2: Adjustment of external edges. This step includes two different sub-steps. Firstly, any edge that originates from a node not belonging to the log nodes and targets one of the log nodes has its target modified to "point" to the root log node (i.e. the first node of the logging statement). Secondly, any edge that originates from a node belonging to the log nodes, except the root node, and targets an external node is modified to originate from the root node. As exhibited in figure 3, this means that all log nodes except the root node are orphaned of any edges, with the root node "taking" all those edges instead.

Step 3: Removal of all log nodes except root. Modification of root. All log nodes except the root are completely removed. Additionally, the root has its contents scrubbed and its type is replaced by a special LOG type. This special LOG node is unique per graph file, which allows it to be used by the machine learning model as the "target" (i.e. prediction) node.

During development of the graph modification script it was noticed that developers would often directly import the logging level that they wished to use instead of importing all log levels with a star import (i.e. import logger.error over import logger.*). This use of direct import created an issue in which the log level could be "leaked" to the machine learning model, effectively teaching it that the best way to predict severity is to examine the import statements, instead of actually attempting to predict via the data contained within the graph file. To rectify this, an additional check is performed during the graph modification procedure. Any set of nodes that contain an import statement that gives away the log severity have their contents completely removed. This ensures that severity cannot be leaked, while also preserving as much graph information as possible.

4.2 Training of Severity Prediction Model

With the new, modified corpus of graph files generated, the next step in NeuroLogs development can begin in order to train a classification model that can successfully predict the severity level of a logging statement. In order to achieve this goal, several technologies were utilized. A short summary on them and their contribution to the training procedure follows:

4.2.1 ptgnn vs Other Libraries

While graph neural networks are a relatively new field in computing researching, there are several libraries available that one can use to assist with model training. Some examples are ptgnn, DGL, tf2-gnn and gnn [42, 6, 43, 44]. All of these libraries are written in Python, the de-facto language of Machine Learning, and all have the same overarching goal: to handle the heavy, mathematical computations of graph neural networks and greatly reduce the workload research needs in order to achieve results. For NeuroLog, these libraries were all considered, but ultimately ptgnn was chosen for the following reasons:

Reason 1) One of NeuroLog's collaborators is the creator of ptgnn. Miltiadis Alamanis is the developer of ptgnn and thus has an extremely intimate knowledge of its use as well as its internal workings. All of the GNN libraries are complex pieces of software that are difficult to understand and even more difficult to bug-fix, should something go wrong. Alamanis's experience with ptgnn proved to be invaluable during NeuroLog's development, whenever an issue was encountered.

Reason 2) ptgnn synergizes with the tools and ultimate goal of NeuroLog. A component of ptgnn is Graph2Sequence, which is used in Alamanis et al.'s paper on Graph Representation of code [1]. In fact, javac-features produces a graph that can be used with Graph2Sequence without any major structural modifications. Thus, the modified corpus, following some modification and data conversion, can be directly inputted to ptgnn to train the severity classification model. Using ptgnn allows for a massive reduction in workload through the usage of Graph2Sequence, as it ensures that no modifications to ptgnn's underlying source code need to be made for the severity prediction training.

4.2.2 Conversion of Data for ptgnn

As mentioned in 4.2.1, the final, modified graph corpus had to be converted into ptgnn's input format. This is a relatively simple procedure and is handled by a small Python script. The graph files are all condensed into three separate gzipped jsonl files, the train set, the validation set and the test set. Each of these jsonl files contain one graph per line as a JSON object and additionally contain the correct prediction that the model should find. The conversion script supports Azure ML, which enables the usage of a CPU cluster to greatly increase conversion speed.

Additionally, an interesting technical challenge appeared during the development of the conversion script of NeuroLog. As mentioned previously, each graph file has a single logging statement associated with it. It is therefore entirely possible (and probable) that a single source file with many logging statements will eventually generate many similar, modified graph files, each with one logging statement. Creating the train/validation/test sets by simply splitting all the graphs into three sets will have a very adverse result: the test or validation set can contain graphs that the model has seen before. To clarify, the severity prediction value in combination with the graph can never be identical, but two graphs with different severity predictions can both originate from the same source file, meaning that they are almost identical. Having these almost identical graphs into separate sets would negatively impact the model's training capabilities, in addition to compromising the results. To prevent this, the conversion script ensures that no "cross-contamination" of the sets can occur, by automatically grouping all graphs that were generated by the same source file to the same set.

4.2.3 Training of the Severity Model

The three sets that contain 8954 statements are then be fed into ptgnn's Graph2Sequence. ptgnn parses these files and their contents and automatically creates and trains a model to predict the correct severity level. In order to test that Graph2Sequence works correctly with the generated gzipped jsonl files, a "smoke" test is conducted. This is done by using the same, single jsonl file as the train, validation and test set in order to induce intentional overfitting. Any correctly functioning neural network should simply learn the correct order of predictions (i.e. over fit) and achieve a tremendously high accuracy score.

This smoke test was originally unsuccessful and was extremely useful, as it pointed to few small bugs in the graph modification algorithm that led to incorrect graph generation. With the bugs fixed, the smoke test succeeded, leading to a 98.1% accuracy score. The smoke test output, as well as the smoke train logs can be found on OneDrive [24].

Following the smoke test, the training of the severity model could begin. In order to expedite the training, an AzureML GPU cluster with 1 NVIDIA Tesla K80 card was utilized. The training, much like most of the development of NeuroLog, was done is a few sequential steps. Firstly, a relatively small subset of approximately 20% of the whole data was used to test that the network was indeed learning to classify the data. With positive results, the subset was increased to 50% before eventually, the entire dataset was used. Several different data splits were attempted, with the standard 60/20/20 split showing the best results (i.e. 60% validation, 20% validation, 20% test). Additionally, several modifications to the model parameters were tested, with an interesting observation being made; due to the lack of fatal-level logs, the model scored better if the fatal level was completely disregarded and the model only tried to predict one of the other five levels. Trying to learn when to classify the fatal level only "confused" the model, leading to worse scores. As a result, the model will never output a fatal prediction and has not been trained on such data at all.

With the exception of some minor bugs, no serious problems occurred during the development of the severity prediction model, and eventually a 97.03% accuracy rate was achieved. The final model is available for download alongside its logs and training sets, on OneDrive [24]. Chapter 5 discusses and analyzes the results obtained in detail.

4.3 Training of Message Prediction Model

From the begging of the project, the goal of NeuroLog was to create two models that could successfully predict a statement's severity level and message. Having established positive results with the first goal, the development of NeuroLog could now pivot to the creation and training of a message prediction model.

4.3.1 Background

Before examining and analyzing the creation of the message prediction model, it is vital to discuss the following background concepts. Firstly, ptgnn's Graph2Sequence is, as the name suggests, a tool that is used to predict a sequence of elements or tokens (i.e. a one-dimensional array). Of course, it can also be used to predict a sequence of one element, which is how it was used to train the severity prediction model. A logging message can be converted into a sequence of tokens through the process of tokenization. This message can then be fed into Graph2Sequence in a similar fashion to the training of the severity model. Graph2Sequence should, in theory, then try to predict the entire message.

In summary, the goal for NeuroLog's message prediction component was to tokenize all messages and pass them into Graph2Sequence as jsonl files, in combination with their respective graphs in order to train a model that could successfully predict a statement's message. The rest of this chapter presents the implementation details and discusses the difficulties encountered during development.

4.3.2 Tokenization and Modifications to ptgnn

As discussed above, the first step of development was to tokenize all the messages in the corpus of almost 9000 logs, so that they could be fed into Graph2Sequence. One of the most common algorithms to use when dealing with tokenization for machine learning purposes is that of byte-pair-encoding (BPE). The algorithm has various implementations, but a popular one is Google's sentencepiece library [41]. This library was used to train a tokenization model over the -then- entire dataset of 8954 statements. It immediately became apparent that the tokenization of the messages was not as successful as desired. The tokenization procedure would often break logged variables in two or fail to separate words in the intended way. Regardless, it was deemed best to leave the tokenization issue unfixed temporarily and continue development in order to test if ptgnn would crash. Some modification of the corpus conversion code for ptgnn input followed, and a set of new gzipped jsnol files were created.

However, while ptgnn supports the prediction of multiple output tokens, it does not have a way to calculate any natural language processing success metric. Therefore, before any testing, the testing code of ptgnn was modified to calculate the Bilingual Evaluation Understudy Score (BLEU) of a prediction. When ptgnn is used with the intention of predicting a statement's message, the test code automatically calculates the BLEU score, in order to quantify ptgnn's success.

4.3.3 First Test and Issue Encountered

Similarity to the severity model, several smoke tests were conducted in which Graph2Sequence attempted to train a message prediction model using the same training, validation and testing sets. The results were extremely problematic. Firstly, they completely fail to return a satisfactory BLEU score of around 0.8, with the average score of most tests ranging from 0.1 to 0.2. Additionally, the confidence value for most predictions was extremely low, less than 0.01%. Manual examination of the predictions revealed that the trained model would often either return the special unknown tokens that signify complete uncertainty or it would return completely incorrect predictions.

The first assumption made was that the tokenization issue was somehow responsible for this problem. To rectify this, a second set of logging statements was collected, bringing the tokenization training data up from 8954 to 20066 log messages. This second set would eventually be combined with the first set to obtain the insights and recommendations discussed in Chapter 3. The doubling of the train set did have the intended effect of improving the tokenization procedure, without, however, completely eliminating the issues described previously. Unfortunately, the improvement in tokenization did not translate to any improvement in the smoke test results.

4.3.4 More Attempts at Solving the Issue, Time limitations

Internal discussions on the issue resulted in the following observation: the tokenization issue should not be able to result in the complete failure of the smoke test. Even if the input tokens are far from optimal, the neural network should still be able to "learn them by heart", achieving a high BLEU score. As a result of this observation, the next attempt to fix this issue involved the modification of the model's parameters. Several modifications were made, such as the reduction of the vocabulary threshold or the modification of dropout rates. However, the test results did not show any improvement.

It is an unfortunate fact of research that there is only a finite amount of time that can be spent on a research project. This is especially true for a relatively small-scale, master's dissertation. Having spent weeks trying to solve the issue of message prediction and with the project deadline fast approaching, it was decided to halt research and begin writing this thesis. In retrospect, there are only two different possible causes of the issue:

1) There is an issue with the graph modification or the input data. This is unlikely, but not impossible. The only real difference between the severity prediction training and the message prediction training is that of the input prediction data. Instead of an array with a single element, which is the severity level, the array now has multiple elements that are the statement message that the model is trying to predict. This, in theory, should not have an impact on Graph2Sequence's performance. Simultaneously however, it is possible that some element of the graph structure does not cooperate with the message prediction training, while having no issues with the severity prediction training.

2) There is an internal bug with ptgnn. It is also possible that ptgnn has a bug somewhere within its source code that prevents it from working as intended. Unfortunately, even with the help of ptgnn's author, there is no easy way to even speculate on precisely what the bug is, without first spending an extreme amount of time analyzing the library's internal logs.

Regardless of the difficulties encountered and the fact that ultimately, NeuroLog's message prediction model was not successful, the work completed here represents a good first step in logging message prediction. Chapter 7 will cover future avenues of exploration in more detail.

5. EVALUATION

In summary, the main results of NeuroLog are: An achievement of 97.03% accuracy rate with the severity prediction model and a concrete, novel, attempt to implement and train a model that can successfully predict a logging statement's message. This chapter discusses and compares these results with past research in addition to analyzing their possible implications in academia as well as the development world.

5.1 Severity Results & Analysis

5.1.1 Comparison with Most-Common Model

As little research has been done on severity recommendation, a good evaluation strategy is to perform a comparison between NeuroLog and a model that will always predict the most common severity level. This most-common model would predict the debug level, as the level has the highest percentage of the data, at 29.37%. This would also mean that the model would have a 29.37% accuracy score. In contrast, NeuroLog's severity model is much more successful with an accuracy score of 97.03%. This comparison confirms that NeuroLog's model has learned to correctly classify the severity level of a logging statement, based on its graph structure and intricate relationships between the graph's nodes and edges.

5.1.2 Comparison with VerbosityLevelDirector

The only research paper that uses a classification model to make severity level recommendations is *An Approach to Recommendation of Verbosity Log Levels Based on Logging* by Anu et al. [3]. While Chapter 2 has covered the paper and its contribution, this sub-chapter aims to serve as a comparison between NeuroLog and VerbosityLevelDirector. It is not possible to directly compare these tools due to the fact that NeuroLog uses the accuracy metric and VerbosityLevelDirector uses the area under curve (AUC) metric. Despite this, several observations can be made when comparing the tools:

NeuroLog is more generalized. VersbosityLevelDirector works by analyzing logging statements that belong to two different focus blocks, exception handling blocks and condition check blocks. The tool uses these blocks in combination with the code within them to extract the source code features before training a random-forest model. NeuroLog does not require the logging statement to be inside any kind of block, analyzing all logging statements regardless of their surrounding code. Additionally, VerbosiyLevelDirector only analyzes four projects, Hadoop, Tomcat, Qpid and ApacheDS. These do not contain many logs, further reducing the possible generalization and applicability of VerbosityLevelDirector.

Both of the tools perform well. While direct comparison is impossible, it is evident that the approach that both tools use is successful in predicting logging severity. Both tools use the abstract syntax tree (AST) to perform feature extraction in order to assist with the model's training. The mutual success of these tools confirms that logging-related research on the prediction of *what to log* can yield substantial results, paving the way for more research on the matter.

VerbosityLevelDirector is better at log detection. Closer examination of VerbosityLevelDirector reveals that it is able to discover more logs in each of the four projects that it analyzes, compared to NeurLog's log discovery metrics. VerbosityLevelDirector does indeed use a more refined log discovery approach, which could be combined with NeuroLog's approach in order to reduce the number of missed logs.

NeuroLog is simple and requires little pre-processing of data. In order to develop VerbosityLevelDirector, its authors went through a complicated data modification procedure, which included but was not limited to noise reduction and calculation of special metrics, such as the Pointwise Mutual Information metric. On the contrary, NeuroLog only requires some modification of the graph data and a parameter fine-tuning process. Overall, this results in a significant reduction of research and development time for NeuroLog. Additionally, future modifications to NeuroLog will not be a time-consuming, difficult task.

Through this comparison it is evident that both tools are characterized by their own strengths and limitations. Therefore, what will inform future decisions on which tool or approach to use entirely lies with the researcher's preference for a generalized or specific analysis.

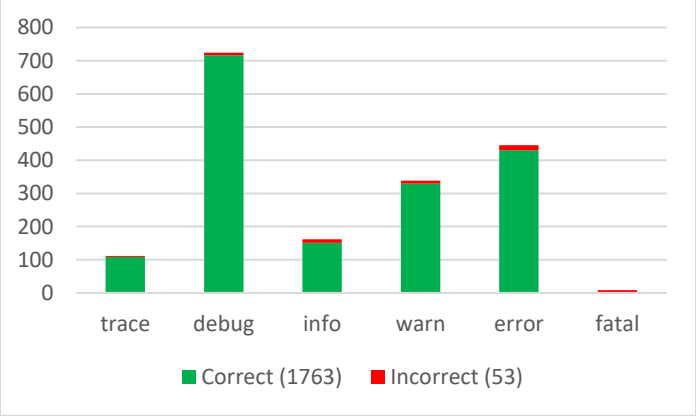

Figure 4 Severity Prediction Breakdown

5.1.3 An in-depth Discussion of the Test Results

Examination of the output of the test script, which is responsible for calculating the accuracy score of a trained severity prediction model and is available on GitHub [9], yields some interesting observations.

The model has very high confidence levels. For the majority of predictions, the model is extremely confident that the level it has predicted is the correct level. For correct predictions, the confidence value is usually 1.0 or extremely close to it. For incorrect predictions, the confidence value is still extremely high, mostly ranging in the 0.9-0.99 range. This is surprising and indicates that, for all severity levels (except fatal), the model seems to have identified a unique feature or group of features in the graphs that identifies ("maps" to) a specific severity level. If such a feature exists in a graph during testing, then the model knows with almost absolute certainty that it maps to a specific severity level, and that there is little chance it maps to a different level.

Of course, there is no way to test this hypothesis, as the internal workings of any machine learning model are a complete mystery to everyone who tries to examine them. Regardless, these results are very positive and could indicate that NeuroLog's approach can be applied to another programming language that is similar to Java. Such a language will have relatively similar formatting and flow of data, and NeuroLog might be able to learn to identify severity levels with the same confidence and success values as it has done in this case. Chapter 7 covers future avenues in more detail.

The model can predict all severities with equal success rates. Figure 3 depicts the predictions that the model has made on the unseen test set. Note that fatal results are all incorrect due to the model's inability to predict the level at all (Chapter 4.2.3). The test set was created randomly, hence the variation in the number of logs per severity. Regardless, relatively speaking, all severity levels have the same correct to incorrect prediction ratio. A few repeated executions of the training and testing sessions yield similar results. This indicates that the model does not have difficulties predicting any specific severity level. The equal success rates also strengthen the theory that the model can clearly differentiate between severity levels, without any one level having drastically similar features to another, which would lead to reduced success rates for those levels.

5.1.4 Theoretical Implications

The primary theoretical implication of NeuroLog's severity prediction model is that it confirms how valuable Graph Neural Networks are for the exploration of very complex and otherwise very difficult to understand data in logging related research. Additionally, these networks can in theory be used to handle any kind of classification task within source code and exploration of their potential as well as their limitations will surely result in some extremely interesting observations and findings.

Furthermore, NeuroLog's results in combination with the usage of javac-features act as an excellent verification that extracting features in the form of graphs from source code can create very expressive data that is capable of yielding great results. This is especially true when the data is combined with a methodology that can understand the intricacies and relationships between them; a Graph Neural Network. Chapter 7 covers suggestions for future research in more detail.

5.1.5 Practical Applications

The severity prediction model of NeuroLog has several possible practical applications. The first and probably best possible use of the model is to integrate it into a Java IDE to make recommendations on what severity level to use. It would be a relatively straight-forward task to construct a plugin that allows the user to select a line of source code. Following the user's input, the program would automatically convert the source file into a graph file, make the necessary modifications to its structure by adding the special LOG node at the right location (i.e. the user's line selection) and then feed it to the model. Finally, the IDE could display the prediction back to the user, alongside the confidence value. It would also be possible to cache the graph files during compilation, speeding up the process significantly. The existence of such a tool would greatly aid with a developer's decision on which level best fits their needs.

Furthermore, it is also possible to create software that checks logging statements that have already been written in order to verify if these statements have an appropriate severity level. Such a software could work by taking an entire Java project as its input and automatically compile it to convert all Java source files to graph files. Then, much like NeuroLog's training, these graph files would have all logging statements detected and stored. The graph files would then be modified and fed into NeuroLog's model and any output predictions that did not match the log level the developer used could be flagged for examination. Such a tool would allow for developers to check and fix any severity misclassifications for even the largest of projects, saving them from missing important logs due to incorrect severity levels. It may even be possible to extend the use of this tool to work with git. Any commits to the master branch would trigger a "log severity" check which would automatically let developers know if some major logging level misclassifications occur. This would ensure that any future logs will be "tagged" with the appropriate severity level, reducing confusion and enhancing any post-mortem analysis.

5.2 Message Prediction Results & Analysis

5.2.1 Novelty of Idea

Due to the variety of issues and time limitations discussed in Chapter 4, no substantial results are available for the message prediction model. What is available for discussion, however, is the fact NeuroLog's goal to predict the messages of logging statements is novel, and so is its approach. While the goal was not achieved, NeuroLog shows that such a goal is within the realm of scientific possibilities. Realistically, by reusing some components of NeuroLog's message prediction approach and combining them with their innovations, a researcher or team of researchers should be able to achieve positive message prediction results without an excessive requirement in time or effort.

5.2.2 Retrospective Changes to Methodology

In regard to NeuroLog's development, there are several suggestions to be made. These suggestions represent what could have been done differently to increase the probability that message prediction would have been successful.

Adopt a Natural Language Processing mindset. The two problems that NeuroLog attempted to solve are very different in nature. Severity prediction is a relatively straight forward, multi-classification issue that is very common in the machine learning world. On the other hand, message prediction is a complicated natural language processing issue, which also has the added usage of graph neural networks and source feature extraction. Due to the fact that the severity model development preceded the message model development, the developer of NeuroLog did not give enough weight to the natural language processing component, instead focusing mainly on the usage of GNN's with feature extraction. Had the issue been approached from a natural language processing point of view, several additional avenues of development would have been explored that may have resulted in better success with the issue at hand. For example, the tokenization issue that took up a significant amount of development time could have been prevented, had more consideration been given to the tokenization of messages from the start, an effect that a natural language processing mindset would have almost certainly led to.

Explore Additional Tools. ptgnn was chosen for several reasons, which have been analyzed previously, in Chapter 4.2.1. However, while these reasons are valid and ptgnn was extremely useful for severity prediction, the usage of ptgnn for message prediction might not have been the best possible option. This is because ptgnn is not optimized for natural language processing. It is entirely capable of doing such a task due to its nature, but it has not been built as a natural language prediction tool. The usage of a dedicated GNN, natural processing tool, such as GNNs-for-NLP [30] might have had significantly better chances of producing positive results. This approach would have taken more time, as it would have required the conversion of the graphs to the tool's required input, but such a time investment would have been similar to the amount of time that was spent trying to solve the issues encountered during message prediction with ptgnn.

5.2.3 Possible Theoretical Implications

There are a few possible, theoretical implications of a model that can successfully predict the logging message of a statement. Graph-based, natural language processing algorithms have existed for years, but the usage of graph neural networks to achieve success in a natural language processing task has only recently been explored by the research community. A success in NeuroLog's message prediction would be a welcome research addition in the exploration of GNN's for natural language processing tasks.

Additionally, a success in message prediction would illuminate new research avenues in the usage of graphs that are extracted from source code for the purposes of natural language prediction. This is due to the fact that in source code, there are many different possible locations that can contain natural language processing appropriate text. Such locations are in error messages and documentation text, amongst others, and their prediction and verification could be the work of future research.

5.2.4 Possible Practical Applications

The possible applications of a successful message prediction model are very similar to the applications of the severity prediction model. It would be entirely possible to integrate such a model with an IDE to make suggestions on *what* a developer should log. The severity prediction model could even be used in combination with the message prediction model to create a "complete" logging statement assistance IDE tool. The severity model would first be used to recommend a severity level and once a developer chose the level they wanted, the tool would produce an example message to be used with the severity level. The existence of a "complete" logging assistance tool would allow developers to spend less time on logging related coding and instead focus on other, more pressing development issues.

Finally, similarly to the severity prediction model, a tool that checks written log messages could be developed in order to make recommendations on ways to improve the messages. This code checker tool could check both the severity and the message and display warnings if any major discrepancies are found in between the written statement and the predicted statement.

6. THREATS TO VALIDITY

The authors acknowledge that there are a few possible threats to the validity and applicability of NeuroLog. These threats are as follows:

6.1.1 Lack of Developer Questionnaire

Perhaps the biggest threat to the validity of NeuroLog is that it is not accompanied by a real-world case study which quantifies its success with developers. NeuroLog achieves very good results in its severity prediction model, having accuracy rates of 97.03% on previously unseen data. However, it is not known if this accuracy rate "translates" well to a non-research setting in the development world. Developers may not agree with some of the predictions that the severity model makes, even if they fall within the 97.03% "correct" percent group. Additionally, it cannot be assumed that developers will want to adopt and use a commercial version of NeuroLog, such as an IDE log recommendation tool.

6.1.2 Analysis of Only One Programming Language

NeuroLog, unfortunately, only supports one programming language for training and prediction: Java. The feature extraction tool, javac-features, works exclusively with Java source code. Additionally, NeuroLog's prediction models have also only been trained with Java based graph files and, as a result, the models may not be usable with any other programming language, even if the source code of the language is correctly translated to a graph file using its respective AST.

Furthermore, the entire corpus that is the basis of the discussion in Chapter 3 contains Java logging statements only. This usage of a single language has the effect of severely limiting the applicability of these recommendations for other programming languages. For example, the developers of a different language such as C might use the fatal level for their logging statements often, which would nullify this thesis's recommendation of removing the level. Such a possibility exists for all programming languages other than Java.

7. FUTURE WORK

7.1.1 Verification of Severity Model Success

To rectify the two possible problems discussed in 6.1.1, a questionnaire can be designed that presents several examples of NeuroLog's severity predictions. Developers could then indicate whether they agree with NeuroLog's output. Doing so would generate an additional success metric for NeuroLog, one that would indicate its real-world success rate, versus the more clinical, research-based accuracy rate that this thesis presents. Additionally, this questionnaire could have questions that ask if developers would adopt a non-research, commercial version of

NeuroLog's prediction model, in order to gauge the applicability of NeuroLog in addition to the benefits of any potential future research.

7.1.2 Enhancement of NeuroLog's Applicability

Chapter 6.1.2 discusses several issues stemming from the usage of a single programming language for training and prediction. These problems can all be solved by expanding NeuroLog to be capable of extracting and training on more programming languages such as C, C++ or even Python. This could be done by collecting a separate logging statement corpus for each programming language using the language's AST. Prediction models could then be trained with each corpus. Finally, these models could be used with their respective languages to make logging predictions. This expansion of NeuroLog will have two central benefits:

NeuroLog will become much more applicable. This benefit is relatively self-explanatory. Having the ability to predict severity levels for multiple languages will result in a more general, applicable tool that will have more value for both developers and researchers.

More log related recommendations can be made. The further analysis of the new corpuses will provide insight to the logging practices of many different languages. These insights can be used to not only make recommendations for each language specifically, but also to make recommendations on logging practices in general.

7.1.3 On the Pursuit of Logging Message Prediction

NeuroLog attempted to predict logging messages and ultimately was unsuccessful, but despite this fact, several recommendations can be made for future work. NeuroLog's attempt to perform both severity prediction and message prediction is partially the reason that message prediction failed. Thus, the goal of message prediction should ideally be the sole work of a single, well-focused research paper. Additionally, following the retrospective analysis in Chapter 5.2.2, it is the opinion of the author that message prediction should be addressed as a natural language processing problem that also uses a GNN, instead of the opposite. Any future research work should use specific natural language processing libraries and methodologies to address the possible issues that may occur during development.

7.1.4 Graph-based Source Code Research Avenues

Converting source code into its graph representation before using it with a GNN for multi-classification issues is an extremely powerful, potent, research tool and avenue, as shown by NeuroLog's success with the severity prediction model. Future work can apply NeuroLog's approach in order to answer a huge amount of research questions. Realistically, almost any programming concept that has a few different programming "choices" can be considered a classification problem. Some examples are: different data-types (both primitive and non-primitive), different array types (set, map, 1 dimensional, 2 dimensional etc.), conditional statements and others. A GNN that resembles NeuroLog's severity model could easily learn to classify any of these groups in order to assist developers and advance AI and programming research.

8. CONCLUSION

We collect a dataset of 20.066 logging statements. We analyze and discuss the dataset in order to gain insight into how and why developers log, presenting recommendations for future libraries to adopt. From the dataset, 8.954 logging statements are collected from graph-based representation of code. These graphs are used in combination with a GNN to train a machine learning model that is capable of predicting the severity of Java logging statements with a 97.03% accuracy. Additionally, we attempt to create a model that can predict the message component of logging statements. This attempt is unsuccessful, but recommendations are made for future work.

9. REFERENCES

[1] Allamanis, M. et al. 2018. Learning to Represent Programs with Graphs. arXiv:1711.00740 [cs]. (May 2018).

[2] Allamanis, M. et al. 2020. Typilus: Neural Type Hints. Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. (Jun. 2020), 91–105. DOI:https://doi.org/10.1145/3385412.3385997.

[3] Anu, H. et al. 2019. An Approach to Recommendation of Verbosity Log Levels Based on Logging Intention. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME) (Sep. 2019), 125–134.

[4] Chen, B. and Jiang, Z.M. 2017. Characterizing and Detecting Anti-Patterns in the Logging Code. 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE) (May 2017), 71–81.

[5] Dash, S.K. et al. 2018. RefiNym: using names to refine types. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (Lake Buena Vista, FL, USA, Oct. 2018), 107–117.

[6] Deep Graph Library: https://www.dgl.ai/. Accessed: 2020-08-15.

[7] Fernandes, P. et al. 2020. Structured Neural Summarization. arXiv:1811.01824 [cs, stat]. (May 2020).

[8] Fu, Q. et al. 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. (Dec. 2009), 149–158.

[9] georgiosN/NeuroLog: https://github.com/georgiosN/NeuroLog. Accessed: 2020-09-04.

[10] Gilmer, J. et al. 2017. Neural Message Passing for Quantum Chemistry. arXiv:1704.01212 [cs]. (Jun. 2017).

[11] Gori, M. et al. 2005. A new model for learning in graph domains. Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005. (Jul. 2005), 729–734 vol. 2.

[12] Guo, P. et al. 2006. Dynamic inference of abstract types. (Jan. 2006), 255–265.

[13] Hassani, M. et al. 2018. Studying and detecting log-related issues. Empirical Software Engineering. (Mar. 2018). DOI:https://doi.org/10.1007/s10664-018-9603-z.

[14] He, P. et al. 2018. Characterizing the Natural Language Descriptions in Software Logging Statements. 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE) (Sep. 2018), 178–189.

[15] Hellendoorn, V.J. et al. 2018. Deep learning type inference. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018 (Lake Buena Vista, FL, USA, 2018), 152–162.

[16] Jia, Z. et al. 2018. SMARTLOG: Place error log statement by deep understanding of log intention. 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER) (Mar. 2018), 61–71.

[17] Jiang, H. et al. 2017. What Causes My Test Alarm? Automatic Cause Analysis for Test Alarms in System and Integration Testing. arXiv:1703.00768 [cs]. (Mar. 2017).

[18] Li, Y. et al. 2017. Gated Graph Sequence Neural Networks. arXiv:1511.05493 [cs, stat]. (Sep. 2017).

[19] Lin, H. and Bilmes, J. 2011. A Class of Submodular Functions for Document Summarization. Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies (Portland, Oregon, USA, Jun. 2011), 510–520.

[20] Lin, H. and Bilmes, J. 2010. Multi-document Summarization via Budgeted Maximization of Submodular Functions. Human Language Technologies: The 2010 Annual Conference of the

North American Chapter of the Association for Computational Linguistics (Los Angeles, California, Jun. 2010), 912–920.

[21] Log4j – Apache Log4j 2: https://logging.apache.org/log4j/2.x/. Accessed: 2020-07-04.

[22] Logback Home: http://logback.qos.ch/index.html. Accessed: 2020-07-04.

[23] Montanari, M. et al. 2012. Evidence of log integrity in policy-based security monitoring. IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012) (Jun. 2012), 1–6.

[24] NeuroLog Files: https://liveuclac-my.sharepoint.com/personal/ucabgn0_ucl_ac_uk/_layouts/15/onedrive.aspx?id=%2Fpersonal%2Fucabgn0%5Fucl%5Fac%5Fuk%2FDocuments%2FNeuroLog%2FNeuroLogFiles%2E7z&parent=%2Fpersonal%2Fucabgn0%5Fucl%5Fac%5Fuk%2FDocuments%2FNeuroLog&originalPath=aHR0cHM6Ly9saXZldWNsYWMtbXkuc2hhcmVwb2ludC5jb20vOnU6L2cvcGVyc29uYWwvdWNhYmduMF91Y2xfYWNfdWsvRVkzVVZ5eDNOcDVBdmFCcERmQmstSDhCalA1YllxNzcwUGJrbXNUUFlRWlk1QT9ydGltZT1JaU1mTVV4UTJFZw. Accessed: 2020-09-03.

[25] Raychev, V. et al. 2015. Predicting Program Properties from "Big Code." ACM SIGPLAN Notices. 50, 1 (Jan. 2015), 111–124. DOI:https://doi.org/10.1145/2775051.2677009.

[26] Rice, A. 2020. acr31/features-javac.

[27] Scarselli, F. et al. 2009. The Graph Neural Network Model. IEEE Transactions on Neural Networks. 20, 1 (Jan. 2009), 61–80. DOI:https://doi.org/10.1109/TNN.2008.2005605.

[28] Shang, W. et al. 2013. Assisting developers of Big Data Analytics Applications when deploying on Hadoop clouds. 2013 35th International Conference on Software Engineering (ICSE) (May 2013), 402–411.

[29] Shang, W. et al. 2015. Studying the relationship between logging characteristics and the code quality of platform software. Empirical Software Engineering. 20, 1 (Feb. 2015), 1–27. DOI:https://doi.org/10.1007/s10664-013-9274-8.

[30] Vashishth, S. et al. 2019. Graph-based Deep Learning in Natural Language Processing. (Nov. 2019).

[31] Wang, Z. et al. 2014. An Analysis of Research in Software Engineering: Assessment and Trends. arXiv:1407.4903 [cs]. (Jul. 2014).

[32] What is Azure Machine Learning: https://docs.microsoft.com/en-us/azure/machine-learning/overview-what-is-azure-ml. Accessed: 2020-07-26.

[33] Xu, W. et al. 2009. Detecting large-scale system problems by mining console logs. Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (Big Sky, Montana, USA, Oct. 2009), 117–132.

[34] Yuan, D. et al. 2012. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. (2012), 293–306.

[35] Yuan, D. et al. 2012. Characterizing logging practices in open-source software. 2012 34th International Conference on Software Engineering (ICSE) (Jun. 2012), 102–112.

[36] Yuan, D. et al. 2011. Improving software diagnosability via log enhancement. ACM SIGPLAN Notices. 46, 3 (Mar. 2011), 3–14. DOI:https://doi.org/10.1145/1961296.1950369.

[37] Yuan, D. et al. 2010. SherLog: error diagnosis by connecting clues from run-time logs. ACM SIGPLAN Notices. 45, (Mar. 2010), 143. DOI:https://doi.org/10.1145/1735971.1736038.

[38] Zhao, X. et al. 2017. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. Proceedings of the 26th Symposium on Operating Systems Principles (New York, NY, USA, Oct. 2017), 565–581.

[39] Zhou, J. et al. 2019. Graph Neural Networks: A Review of Methods and Applications. arXiv:1812.08434 [cs, stat]. (Jul. 2019).

[40] Zhu, J. et al. 2015. Learning to Log: Helping Developers Make Informed Logging Decisions. (May 2015), 415–425.

[41] 2020. google/sentencepiece. Google.

[42] 2020. microsoft/ptgnn. Microsoft.

[43] 2020. microsoft/tf2-gnn. Microsoft.

[44] 2020. sailab-code/gnn. SAILab - Siena Artificial Intelligence Laboratory.

10. Declarations

This report is submitted as part of requirement for the MSc Software Systems Engineering degree at UCL. It is substantially the result of my own work except where explicitly indicated in the text.