Constructing Caveat Contracts from API Documentation and its Applications

Thien Bui-Nguyen

A thesis submitted for the degree of
Bachelor of Advanced Computing (Honours) at
The Australian National University

October 2019

Except where otherwise indicated, this thesis is my own original work.

Thien Bui-Nguyen
23 October 2019

to my parents Tuyet and Linh, and my sister Quyen

Acknowledgments

First, I would like to thank Dr. Zhenchang Xing for being an excellent supervisor. Thankyou for being patient, encouraging and supportive at all times. Without your guidance, this thesis would not have been completed.
To Jiamou Sun and Xiaxue Ren, thank you for yours ideas and support. You helped me learn, contribute and extend your work.
Finally, I would like to thank my parents, Tuyet and Linh, along with my sister, Quyen, for their words of wisdom and positivism throughout my time at ANU.

Abstract

This thesis investigates the methods of linking Application Programming Interface (API) caveats to code. API caveats deal with the accessibility of API usage knowledge in API documentation and are defined as the constraints that specify what developers are allowed or not allowed to do. Previous research has suggested that API caveats comprise a significant amount of issues developers face when using an API. Linking API caveats to code can, therefore, help developers learn about correct API usage and avoid potential misuses.

A previously proposed method applied sentence embedding and other Natural Language Processing techniques to API caveat sentences and the text from Stack Overflow answers. Cosine similarity was used to determine the similarity of the API caveat and answer text vectors, allowing sample code from Stack Overflow answers to be indirectly linked to API caveats. This thesis applies the previous approach to GitHub data to determine its effectiveness when applied to a different type of software text data. I extract 73,831 API caveat sentences from the Java 12 API documentation, and 629,933 sentences from GitHub issue sentences related to Java repositories throughout 2018. I then create 4 information retrieval systems based on TF-IDF, word2vec, BM25 and word2vec + BM25 to link the caveat sentences with GitHub sentences. However, a significant lexical gap was discovered and only 1% of query results were relevant for all models tested. In comparison to previous work, the main issue was concluded to be the different purposes of GitHub text data: bug reporting and suggestions instead of general Q&A. Furthermore, GitHub data is considerably noisier and can be relevant to a multitude of APIs while Stack Overflow provides a clear distinction between different question/post categories.

The negative results prompted an investigation into a more direct method to link API caveats to code. The concept of *caveat contracts* is introduced, which are similar to code contracts that can detect bugs while coding, but are derived from API caveat sentences. I use and combine proposed methods of sentence normalisation and heuristics to parse the Java 12 API caveat sentences and extract *not-null* and *range limitation* constraints. This is used to construct 4,694 unique contracts for the Java 12 API. An evaluation of this approach shows an accuracy, precision, recall and F1 score of 0.77, 0.96, 0.73, and 0.83 respectively. I then develop an IntelliJ *checker* plugin that utilises these caveat contracts and static code analysis to automatically check for API misuse in real-time.

Overall, this thesis presents the complete process of linking API caveats to code via the use of caveat contracts. The key contributions include a simplistic parsing approach to extract constraints from a subset of exception related API caveats and the implementation of a checker program that serves as a proof-of-concept for real-time bug detection.

Software

All software developed for this thesis is made publicly available.

Caveat extraction, GitHub data extraction, information retrieval systems, caveat contracts construction.
`https://github.com/thienbuinguyen/github-api-caveats`

IntelliJ Plugin (checker program for API misuse detection).
`https://github.com/thienbuinguyen/ApiCaveatRules`

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An Application Programming Interface (API) provides a set of functions to interact with some software. However, the correct usage of APIs often requires knowledge and understanding specific to that API. Incorrect usage, which is referred to as API misuse in this thesis, can lead to severe bugs for developers. Fortunately, API reference documentation typically provides information to guide developers to correct usage by describing the constraints associated with different components of the API. A recent study by Li et al. [2018] researched a particular subset of these constraints that describe what developers can and cannot do. This class of constraints is termed as *API caveats*. An approach for extracting these API caveats was also proposed based on using their syntactical patterns. Improving the accessibility of API documentation by helping developers understand API caveats was then investigated by Sun [2018]. The approach used NLP techniques with sentence embedding of Android API caveat sentences and community text from the Q/A forum Stack Overflow, which was used to indirectly collect code examples from the programming community and augment API documentation. This thesis seeks to extend upon the approach in Sun [2018] by testing it in a different domain abundant with open source code: GitHub. Besides this, alternative methods of linking API caveats to code are investigated to improve comprehension of API caveats in general. In particular, I focus on constructing *caveat contracts* from the natural language of API caveats. This concept is similar to code contracts and can be used in static code analysis tools to check for API misuses in real-time.

1.1 Motivation

Suppose a developer was starting their journey on learning Java. A topic that may pique their interest is file handling: how to read text files into their programs. A common first strategy involves using Google to search for "java how to read files" (for example). The first result of this is a tutorial webpage by GeeksforGeeks[1] that provides multiple methods and examples for reading a file in Java. The first example is shown in Listing 1.1.

[1]https://www.geeksforgeeks.org/different-ways-reading-text-file-java/

Listing 1.1: File reading java code example from GeeksforGeeks

```
1  // Java Program to illustrate reading from FileReader
2  // using BufferedReader
3  import java.io.*;
4  public class ReadFromFile2
5  {
6      public static void main(String[] args)throws Exception
7      {
8          // We need to provide file path as the parameter:
9          // double backquote is to avoid compiler interpret words
10         // like \test as \t (ie. as a escape sequence)
11         File file = new File("C:\\Users\\pankaj\\Desktop\\test.txt");
12
13         BufferedReader br = new BufferedReader(new FileReader(file));
14
15         String st;
16         while ((st = br.readLine()) != null)
17         System.out.println(st);
18     }
19 }
```

At this point, the next step a developer might take is to copy the code shown in Listing 1.1 directly into their program. Pleased with their clever thinking, the developer might try to execute their program only to find a `FileNotFoundException` is thrown. This is because the file path in the `File` constructor call has not been changed to the appropriate file path relative to their system. In an Integrated Development Environment (IDE) such as IntelliJ, information about the exception would also be exposed. This includes the associated line number in which the exception was thrown. However, a confounding result would be discovered: the exception is not associated with the `File` constructor in line 11 (where the file path is set), but with the `FileReader` constructor from line 13. In other words, the file path appears to be accepted by `File`.

A logical next step would involve searching for the reference documentation of these Java classes[2]. By reading the documentation of `FileReader`'s constructor, the developer will discover a line that says "Throws: FileNotFoundException - if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading". Also checking the documentation of `File`'s constructor, they will find that no such warning is provided. Rather, they may notice that `File` contains a `exists()` method used to verify whether a file exists *after* instantiating the `File` object. Only after this arduous process does the

[2]https://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html
https://docs.oracle.com/javase/7/docs/api/java/io/File.html

developer understand the source of the problem (the file path specified in this scenario) alongside correct usage of `File` and `FileReader` for reading files. Although the above example presents a simplistic view of how a new developer might approach learning an API, we observe the constraints associated with an API introduce a significant problem for all programmers. This problem lies in the fact that "you don't know what you don't know", meaning the usage of an API requires a considerable understanding of the different components and their interactions before they can be used. This is a time-consuming endeavour given the number and complexity of different APIs.

For reference, several examples of caveats from the Java 12 API are shown in Tables 1.1, 1.2 and 1.3. Note that the community text provided is from answers or comments addressing an issue raised by another developer involving the API caveat. These questions/posts generally include superfluous information and are excluded for brevity. We observe from 1.1 and 1.2 that community solutions for API caveat related problems essentially attempt to describe the API caveats in a simplified manner. Table 1.3 showcases an example where the API documentation does not mention the caveat, but community answers can provide additional information to fix programming bugs. Hence, it can be seen that API caveats provide essential information for correct API usage, but its portrayal/accessibility may be insufficient due to issues such as complex descriptions or lack of examples. A previously proposed solution was to augment API documentation by using text and code examples from the programming community, which was mainly focused on Q&A platforms such as Stack Overflow. However, other community-centered platforms like GitHub exist that contain significantly more code given its use for hosting Free and Open-Source Software, but lack the community text structure of a Q&A website. This thesis, therefore, attempts to test whether previously applied approaches could obtain better results in the different domain of GitHub and whether other methods of linkage can be used to connect API caveats to code.

	API Caveat
API Element	*String.indexOf(int n)*
Method Summary	Returns the index of the first occurrence of the specified character in this string
Caveat Description	Indices are positive, but this method can return a -1 value
Relevant Stack Overflow Comment #1	"If <code>indexOf</code> cannot find the required string, it returns <code>-1</code>."[3]
Relevant Stack Overflow Comment #2	"If <code>.</code>isn't present, <code>indexOf</code> returns -1 as a sentinel"[4]

Table 1.1: Example of a caveat for `String.indexOf(int n)` alongside relevant Stack Overflow/GitHub comments.

	API Caveat
API Element	*ArrayList.subList(int fromIndex, int toIndex)*
Method Summary	Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive
Caveat Description	the starting index (fromIndex) is inclusive while the ending index (toIndex) is exclusive
Relevant GitHub Comment	"...subList toIndex is exclusive..."[5]
Relevant Stack Overflow Comment	"The first index is inclusive, the other is exclusive."[6]

Table 1.2: Example of a caveat for `ArrayList.subList(int fromIndex, int toIndex)` alongside relevant Stack Overflow/GitHub comments.

[3]https://stackoverflow.com/questions/40314998/java-substring-out-of-range-1

[4]https://stackoverflow.com/questions/10859633/java-string-index-out-of-range-1-exception-using-indexof

[5]https://github.com/viritin/viritin/issues/273

[6]https://stackoverflow.com/questions/37256605/why-does-arraylist-sublist0-n-return-a-list-of-size-n

	API Caveat
API Element	*HashMap.size()*
Method Summary	Returns the number of key-value mappings in this map
Caveat Description	Negative values can be returned in multi-threaded environments
Relevant GitHub Comment	"Since HashMap is not threadsafe it's possible to get HashMap size as -1 if you try removing items concurrently."[7]
Relevant Stack Overflow Comment	"HashMap is not thread-safe. Therefore, if there is any point where two threads could update a HashMap without proper synchronization, the map could get into an inconsistent state."[8]

Table 1.3: Example of a caveat for `HashMap.size()` alongside relevant Stack Overflow/GitHub comments.

1.2 Main Research Challenges

The primary goal of this thesis is the investigation of methods for linking API caveats to code. Ideally, this would allow the detection of API misuse and help developers understand the correct usage of a given API. Previous work has focused on an indirect approach of performing this linkage: caveats sentences were matched with community text surrounding code examples on Q&A websites such as Stack Overflow. This utilised NLP techniques involving sentence embedding to form information retrieval systems. The caveat sentences were then used as queries and the information retrieval systems used to locate similar community text with associated code examples from Stack Overflow. For this thesis, one objective was to determine whether the same approach could be applied to a different domain such as GitHub. This is a complex task that involves four components: (1) extraction of API caveats, (2) sentence embedding, (3) matching caveats sentences to similar sentences from GitHub, and (4) presenting code examples to developers. The scope of this project covers components 2 and 3. This is because the extraction method used by Sun [2018] was found to be sufficient for allowing sentences to be matched (component 1). Furthermore, a simple Graphical User Interface (GUI) was found to help users better understand API caveats (component 4). In terms of component 2, sentence embedding is a modern approach in NLP that transforms sentences into vectors. These vectors retain the semantic meaning of their original sentences Palangi et al. [2016]. However, this typically requires complex models (such as

[7]https://github.com/RuedigerMoeller/fast-serialization/issues/223
[8]https://stackoverflow.com/questions/5283294/can-a-java-hashmaps-size-be-out-of-sync-with-its-actual-entries-size

neural networks) to learn important features of sentences.

Matching API caveats to code examples is also non-trivial given the semantic/lexical gap that exists between natural language and programming code. An example can be seen with the API caveat sentence "UnsupportedEncodingException - If the named charset is not supported" from the Java 12 Documentation (method `getBytes` of class `String`). Developing a generalised representation of this for computers to understand is particularly difficult. It requires consideration of context, that "charset" refers to a parameter named "charsetName", alongside what exactly denotes a supported "charset". The fourth component refers to the problem of how to improve API documentation in terms of helping users better understand them. Overall, these components only represent a part of the challenges of this thesis. As negative results were found using the previous approach for GitHub, an alternative method of linking API caveats to code was investigated. This introduced additional research challenges involving sentence parsing and static code analysis.

Sentence parsing involves numerous difficulties in the process of interpreting semantic meaning. It requires identification of individual words, contextual information, punctuation and subtle features such as intonation Mitchell [1994]. Several fields of NLP exist to determine specific properties of a sentence, such as Named Entity Recognition for identifying which entity certain words refer to. An example of why sentence parsing is a complex topic is provided by Ratinov and Roth [2009] with the news headline "SOCCER - PER BLINKER BAN LIFTED". Without some background knowledge, it is difficult to recognize that "BLINKER" refers to a soccer player. Therefore, the scope of this thesis in terms of sentence parsing is limited to applying and extending existing parsing techniques to construct caveat contracts.

Other challenges for this thesis are those involved with static code analysis, which refers to the process of examining the source code of programs before executing them Baca et al. [2009]. This is typically used to "find bugs and reduce defects in a software application" Bardas et al. [2010]. However, implementing static code analysers, which we refer to as *checkers*, is a non-trivial task. This is observed with existing static analysis tools that can detect complex vulnerabilities but also commonly produce false-positive alarms Zitser et al. [2004]. Besides this, the performance of checker programs must also be considered as increasing the complexity of checks would also require more difficult computations and more time. The scope of this thesis is for static code analysis is limited to implementing a checker program that can utilise caveat contracts.

1.3 Thesis Outline

An overview of API caveats and the background work of this thesis is described in Chapter 2. Chapter 3 extends upon the work of Sun [2018] and Ren et al. [2018] for linking API caveat sentences to code examples in a different domain: GitHub. A more direct approach for linkage is described in Chapter 4 with the construction of caveat contracts and static code analysis. In Chapter 5, the findings of this thesis are discussed with remarks for future work.

1.4 Main Contributions

The main contributions of Chapter 3 is the discovery of a significant lexical gap between community text from GitHub and (Java 12) API caveat sentences, which impedes previous approaches used for linking API caveats to code examples.
The main contributions of Chapter 4 are:

- Parsing of a subset of API caveats from the Java 12 API to formulate caveat contracts.
- Implementation of a checker program (IntelliJ plugin) that uses static code analysis to automatically check caveat contract compliance in real time.

Chapter 2

Related Work

This chapter provides an overview of the background for API caveats, linking API caveats to code examples with NLP techniques, and the use of static code analysis to detect API errors or misuses. The term *API reference documentation* or *API documentation* is adopted throughout this thesis to refer to the set of *documents* that are indexed by an *API element* such as a class or method (Maalej and Robillard [2013]). For example, the Java 12 reference documentation consists of numerous documents as web-pages with each describing a specific Java class (`String`, `ArrayList` etc.).

Section 2.1 references the papers that termed *API caveats* and extended upon this concept for linkage of API caveats to code.

Section 2.2 mentions related works that have applied NLP techniques to the domain of API documentation.

Section 2.3 references related works that applied static code analysis to the domain of API documentation.

2.1 API Caveats

API reference documentation consists of a taxonomy of knowledge types that are defined by Maalej and Robillard [2013]. A particular knowledge type identified as *directives* "specifies what users are allowed/not allowed to do with the API element". This is identified as a notable component of API reference documentation by Li et al. [2018] as it includes all forms of constraints. It is therefore referred to as *API caveats*. These constraints are of particular interest as they can be used to determine the accessibility of an API document. Specifically, accessibility of documentation is an essential part of any framework because it is used to describe functionality and usage of the associated API. A formative study was conducted by Li et al. on Stack Overflow to highlight the prevalence of issues involving API caveats for programmers. Also, a set of syntactic patterns were identified that could be used to recognise API caveats (see Table 3.1). Moreover, API caveats have many practical applications such as the examination of the quality/validity of Stack Overflow answers Ren et al. [2018], construction of a knowledge graph for information retrieval purposes or entity-centric searches of API caveats Li [2018], and augmentation of caveats with code examples Sun [2018]. In particular, the work by Sun [2018] is the foundations for Chapter 3. It was shown that one possible solution for linking caveats to code is by indirectly using community text from Q&A websites. This approach used word2vec sentence embedding to Android API caveat sentences and answer posts on Stack Overflow. The cosine similarity of these vector outputs could then be used to determine the relatedness of sentences. The code examples from the answer and question posts could then be inferred as "good" and "bad" code examples respectively for the given API caveat.

2.2 Natural Language Processing for API Documents

Other NLP-centered works for linking API documentation to code have also been proposed. For example, CROKAGE (Crowd Knowledge Answer Generator) is a tool that takes a query input and returns programming solutions consisting of both code and explanations. This is achieved with several NLP models and word embedding approaches including Lucene Index, FastText, IDF and an API inverted index. For relevance scoring, CROKAGE also utilises the BM25 function and TF-IDF. Note that BM25 and TF-IDF are explained in further detail in Chapter 3 (Section 3.2). An explanation of the other methods will not be presented here as they are not used. Besides this, another solution proposed is called BIKER (Bi-Information source-based KnowledgE Recommendation), which focuses on helping developers find APIs appropriate for their programming tasks Huang et al. [2018]. BIKER uses a combination of language models involving IDF and word2vec, and similar to the other works mentioned retrieves relevant posts from Stack Overflow via similarity functions for a given query. The advantage of deep learning with word2vec has also been investigated by Van Nguyen et al. [2017]. This involved representing source code in a higher-dimensional space (i.e. vectors).

In terms of parsing semi-structured natural language, Pandita et al. [2012] proposes an approach using part-of-speech (POS) tagging, and phrase and clause parsing to identify sentences with code contracts. They then describe these contracts via first-order logic (FOL) expressions with precision and recall of 91.8% and 93% respectively. POS tagging refers to the identification of part-of-speech categories for words within some text (such as noun, verbs and adjectives), while clause and phrase parsing refers to analysis of the syntactic features and grammar used for those forms of text with computers. Similarly, Jdoctor is proposed by Blasi et al. [2018], which uses sentence normalisation techniques to translate Javadoc comments to executable Java expressions. This tool generates procedure specifications that are analogous to caveat contracts but differ in representation and application. The caveat contracts in Chapter 4 are kept in an abstract form that is only resolved to Java expressions by a checker program (IntelliJ plugin). Overall, Jdoctor boasts an overall precision and recall of 92% and 83% respectively, which we use as a baseline objective for this thesis.

2.3 Code Analysis & its Applications with API Documents

A code-analysis centered solution for linkage of API documents and code was proposed by Subramanian et al. [2014]. This uses deductive linking, a process where the abstract syntax tree (AST) of a code snippet from online websites is analysed. ASTs are tree representations of the syntactic structure of source code and are explained in further detail in Chapter 4. The relevant API elements are then detected and linked to associated API documents. Another approach combines NLP and code analysis to detect errors such as obsolete code samples Zhong and Su [2013]. For this, the names of API elements from code samples are compared to the set of all existing API elements. Mismatches in API element names are then reported as documentation errors. Another interesting approach for detecting API misuse ignores API documentation and data-mines correct code usage using a large set of code samples on GitHub Zhang et al. [2018]. This is achieved by traversing ASTs and generating API call sequences, representations of API calls and their surrounding context. Correct usage patterns are then applied to Stack Overflow to determine how reliable code examples are on Q&A websites. Besides this, Zhou et al. [2017] provides a method of detecting defects between API documents and their code implementations by extracting the constraints from directives and code as FOL expressions. These expressions are then resolved with a Satisfiability Modulo Theories (SMT) solver. In particular, several categories of constraints for directives identified in that work represent the largest portion of API documentation (43.7%) Monperrus et al. [2012a]. To reduce the scope of mapping API caveats to code, the heuristic patterns and regular expressions from that work are used as the building blocks for the construction of caveat contracts in Chapter 4.

The concept of mutation analysis has also been suggested for exposing API misuse by Wen et al. [2019]. Mutation analysis consists of many, small modifications to a program that are then executed with a given test suite. Execution data from the stack trace of these programs can then be used to determine the patterns that constitute an API misuse. For an example of the applications of static code analysis, Bae et al. [2014] proposes an error detection method for JavaScript web applications using the fact that Web APIs are typically specified in a certain format known as Interface Definition Language, which exposes function semantics. From this, correct API usage can be inferred.

2.4 Summary

This chapter provides an overview of the related works for API caveats, NLP applied to API documentation and static code analysis for API misuse detection. In particular, key background information for Chapters 3 and 4 is provided, and some terminology used is defined to avoid ambiguity for readers. In Chapter 3, the techniques in Sun [2018] are applied to GitHub data to investigate linking API caveats to a different community platform domain (GitHub).

Chapter 3

Locating API Misuse Examples in GitHub Data

This chapter provides an overview of the process used for extracting caveats from the Java 12 API documentation, extracting GitHub text and sample code data, and the use of NLP techniques (information retrieval systems) to perform sentence matching. In particular, it was found that a significant lexical gap existed with API caveats and GitHub data, resulting in this approach to be unsuccessful. The purpose of this chapter is to, therefore, be a reference and act as a bridge for an alternative approach used in Chapter 4.

Section 3.1 provides an overview of previously proposed methods for linking API caveats to code examples and how the same approach can be applied to GitHub.

Section 3.2 describes TF-IDF, word2vec and BM25 as information retrieval systems alongside the architecture design of the approach for this chapter.

Section 3.3 discuses the implementation process, which consists of the the extraction of GitHub data and API caveats from the Java 12 API documentation, and the process used for sentence matching via the creation of information retrieval systems.

Section 3.4 discusses the findings of this chapter.

3.1 Introduction

One approach for linking API caveats to code examples is to utilise code examples provided by the programming community from Q&A websites. These websites allow users to post questions that contain some "buggy" code that requires fixing and have more experienced programmers provide suggestions or fixes. A core idea for linkage then involves locating questions that are related to API caveats and using their code examples as examples of API misuse. Note that conversely, the code examples from answer posts can be considered correct API usage samples. This can be achieved by measuring the lexical/semantic similarity between the natural language surrounding these code examples with the descriptions of API caveats. In particular, the method proposed by Sun [2018] and Ren et al. [2018] uses sentence embedding to show that sentences in answer posts typically have high similarity to sentences of API caveats. Sentence embedding involves transforming sentences into some high dimensional vector space such that computers can "understand" them. For example, a sentence such as "I like apples" can be mapped to some sequence of numbers (a vector) that retains the original meaning of the sentence. The cosine similarity function can then be used to compute the cosine of the angle between two vectors. With sentence embeddings, the similarity function therefore outputs a score that represents the similarity of two sentences. This chapter aims to extend upon the previous approach by applying it to another domain: GitHub.

GitHub is a website that is abundant with community text and code examples, but its central focus is for providing an interface to a version control system (which is used to help developers maintain their projects over time). As a result, GitHub is commonly used for hosting free and open-source software and contains code examples for many different APIs. According to GitHub's internal statistics[1], the website includes over 31 million developers and 96 million project repositories in 2018. A key component of GitHub is its *issues* features, which lets developers mainly report bugs for a repository (though it is also used to ask questions or make suggestions). Similar to a Q&A website, these issues can contain multiple *comments* from different users. Comments are used to answer the questions of the issue. Therefore, it can be seen that GitHub shares some similarities to Q&A websites with the addition of a significantly larger collection of example code for analysis. The same approach from Sun [2018] and Ren et al. [2018] can then be applied to GitHub to investigate if API caveat linkage to code examples can be improved with a different domain.

I extract 1,855,870 GitHub comments related to Java throughout 2018 via the GitHub Archive project. I then perform data cleaning and filtering to find comments associated with GitHub issues containing code examples, resulting in 629,933 comment sentences. Next, I data crawl the Java 12 API documentation to extract 73,831 API caveat sentences. I then perform similar text preprocessing and tokenisation to Sun [2018] to create 4 information retrieval systems consisting of TF-IDF, word2vec, BM25 and word2vec + BM25. This is used to match the API caveat sentences with GitHub comment sentences. I evaluate the information retrieval systems with a statistical random sampling method to compute the mean precision@3. However, all models performed considerably worse in comparison to the studies on Stack Overflow, with a mean precision@3 close to 0. A significant lexical gap was identified as the cause of negative results. This was a consequence of the generic caveat sentences in the Java 12 documentation in combination with the fact that GitHub data relates to many different APIs, and fundamental differences in the community platforms.

[1]https://octoverse.github.com/

3.2 Design

Section 3.2.1 provides an overview of the TF-IDF statistic followed by a description of the word2vec model for sentence embedding (Section 3.2.2) and Okapi BM25 function (Section 3.2.3). Note that for these sections, *documents* is used interchangeably with *sentences* and *terms* is used interchangeably with *words*. An architecture design overview of this chapter is shown in Figure 3.1. This resembles the architecture overview from Sun [2018], with key differences in the *Candidate Matching* step of the *Matching Phase*. The differences include the use of several, different information retrieval systems in addition to an extension of the pure word2vec sentence embedding of the previous approach. A GUI was intended but also not completed due to the negative results of matching.

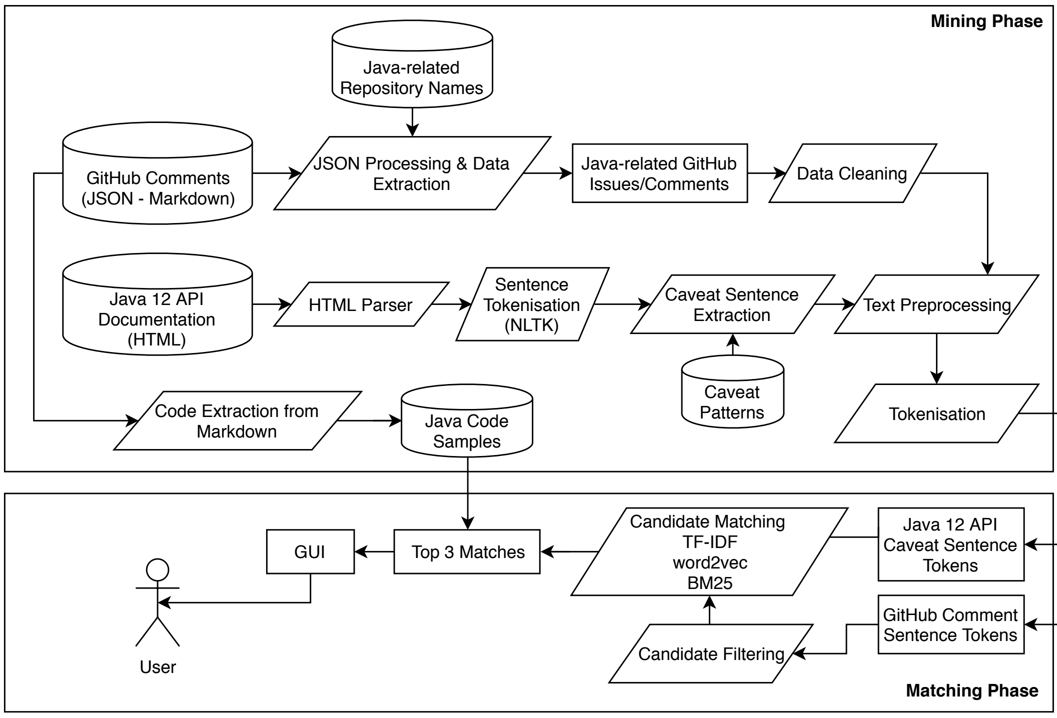

Mining Phase
Java-related Repository Names
GitHub Comments (JSON - Markdown)
JSON Processing & Data Extraction
Java-related GitHub Issues/Comments
Data Cleaning
Java 12 API Documentation (HTML)
HTML Parser
Sentence Tokenisation (NLTK)
Caveat Sentence Extraction
Text Preprocessing
Caveat Patterns
Tokenisation
Code Extraction from Markdown
Java Code Samples
GUI
Top 3 Matches
Candidate Matching TF-IDF word2vec BM25
Java 12 API Caveat Sentence Tokens
GitHub Comment Sentence Tokens
Candidate Filtering
User
Matching Phase

Figure 3.1: Architecture design of the work for this chapter.

3.2.1 TF-IDF Sentence Matching

TF-IDF is short for *term frequency-inverse document frequency*. TF is based on the frequency of a term within a document while IDF refers to the inverse of the number of documents that contain the given term Robertson [2004]. TF-IDF is computed as $TF \cdot IDF$, which is the multiplication of two measures. The formal definition of IDF is shown in Equation 3.1, where t is a token and D is the documents in the corpus (a collection of documents). The overall motivation for TF

is based on the assumption that meaningful words tend to appear more often within a document. However, this assumption means common terms such as "the" and "a" would be incorrectly emphasised. The IDF is therefore used to minimise the weights of words that appear frequently across the entire corpus.
The TF-IDF heuristic is a widely used technique for information retrieval systems that have been successful in several cases Havrlant and Kreinovich [2017]. For sentences, TF-IDF typically follows a bag-of-words approach in which vectors have a length equal to the vocabulary size and each component of the vector corresponds to the TF-IDF score of a single term.

$$idf(t, D) = log \frac{\mid D \mid}{1 + \mid \{d \in D : t \in d\} \mid} \tag{3.1}$$

3.2.2 Word2Vec Embedding

Word2vec involves a computationally efficient neural network that is trained to learn and produce word embeddings. The original motivation for this was to capture contextual information such that words could be predicted in a sequence of words Mikolov et al. [2013a]. The two models that can be used with word2vec is Continuous Bag-of-Words (CBOW) and Skip-Gram, though only an overview of Skip-Gram will be provided as it is the model used for sentence embedding in this chapter. The Skip-Gram model effectively predicts surrounding words given the current word and has the property that "simple vector addition can produce meaningful results" Mikolov et al. [2013b]. An example of this provided in Mikolov et al. [2013b]. Suppose *vec(x)* represents the vector form of word x, then vec("Germany") + vec("capital") would result in a vector close to vec("Berlin"). A simple method for applying word embeddings to sentence embeddings is to use the sum of the word vectors of a sentence. This is the method used for sentence embeddings in 3.3.6.

3.2.3 BM25 Ranking

BM25 is a ranking function that outputs the relevance of a document relative to other documents for a some query. It is based on a probabilistic model and is computed from Equation 3.2, where $f(q_i, D)$ is the term frequency of term q_i (the i^{th} term in document D), $|D|$ is the document length, k_1 and b are tuning variables, and IDF' is an alternative version of IDF that is defined by Equation 3.3 Manning et al. [2008]. In particular, $b = 0.75$ and $k_1 \in [1.2, 2]$ are reasonable values reported by Manning et al.. The values $b = 0.75$ and k_1 were therefore used in 3.3.6. BM25 was considered a state-of-the-art model for information retrieval in previous years Pérez-Iglesias et al. [2009]. It is noted that since the BM25 scores are only comparable for a specific query, they can be normalised to the range 0-1 with 0 representing the least amount of relevance and 1 representing the highest. This allows its ranking score to be used in conjuction with cosine similarity for sentence matching.

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}'(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot (1 - b + b \cdot \frac{|D|}{\text{avgdl}})} \tag{3.2}$$

$$\text{IDF}'(q_i) = \log \frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} \tag{3.3}$$

3.3 Implementation

Implementation of the data retrieval and extraction of GitHub comments alongside Java 12 API caveats was completed in Python 3.6. In particular, the Gensim library was used to apply word2vec sentence embedding to GitHub comment sentences and Java API caveat sentences. The Sci-kit library was used to obtain sentence vectors from TF-IDF sentence embedding. Rankings for BM25 were computed manually using its associated function. Section 3.3.1 describes the collection of GitHub data and its extraction for sentence embedding and matching. Section 3.3.2 provides an overview of the Java 12 API documentation and extraction of its API caveat sentences. Section 3.3.3 explains the sentence preprocessing steps performed, followed by filtering steps to ease the scope of sentence matching required for the large sets of sentences (Section 3.3.4). Section 3.3.5 illustrates the tools used for sentence embedding, and finally, Section 3.3.6 describes how the Java 12 API caveat sentences were matched with sentences from GitHub comments.

3.3.1 GitHub Data Extraction

There are three notable sources for GitHub data: (1) the GitHub Representational State Transfer (REST) API, (2) the GitHub Archive project, and (3) GitHub itself by cloning a repository. For this project, the GitHub REST API and GitHub Archive project was utilised due to the memory and processing requirements of cloning each repository of interest with option 3. However, the GitHub REST API contains several limitations that inhibit its usefulness for a large corpus of community text. This includes limitations on the request rate (30 requests per minute using the "search" API) and query options (e.g. it does not allow complex searches such as "all GitHub issues associated with Java projects in January", or more than 100 results per query). For option 2, GitHub Archive is a project that creates hourly archives of all GitHub data starting from 2/12/2011 as JSON objects. The archives include over 20 event types that are provided by GitHub such as the creation of an issue or comment. For issues and comments, the raw text of the posts is included in Markdown format, which represents the community text of interest. However, these objects do not provide metadata for which programming languages or APIs are relevant. This makes it difficult to filter out irrelevant posts for information retrieval systems in later steps. Thus, we combine the queries of the GitHub REST API to identify the repositories that are Java related, then link them with the community text captured by GitHub Archive.

In more detail, GitHub provides a REST API to allow developers to query for data on GitHub such as metadata about a particular repository or the number of repositories use some programming language. The REST API imposes a rate limit on user requests to prevent flooding of the GitHub servers. For this API, its "search" functionality is the main solution for searching repositories or issues given specific conditions such as the time in which it was created. The API does not provide functionality to search for the contents of GitHub issues and comments across multiple repositories. Furthermore, the API is limited to a maximum of 100 results for a single query. The API is therefore only useful for finding which repositories contain Java code. The rate limit imposed is circumvented by performing numerous HTTP GET requests in sequence. This involves sending queries with a timer between each request and modified query parameters for (daily) creation times. A time window of 2009 to 2019 was chosen alongside a restriction of at least 2 "stars" to reduce the scope of projects collected to those that contained Java code and likely had more than 1 developer involved (as the number of stars a repository contains can be used to gauge its popularity). The PyGitHub Python library, in particular, is used to implement these queries to the GitHub REST API. Overall, collecting repository names of Java-related projects from 2009-2019 projects that contain at least 2 stars yields 291,152 results.

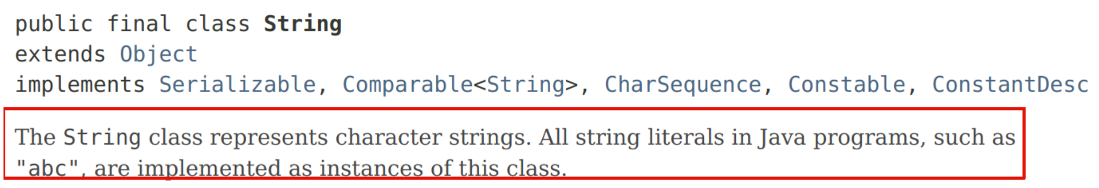
GitHub Archive captures an event known as "IssueCommentEvent", which contains various information about a comment for a particular issue. The archives are saved hourly. Therefore, a script is used to automate the process of downloading all archived data for 2018. Note that only data from 2018 is collected to reduce the amount of time required for querying the entire dataset and as an initial study. Multiprocessing is used in particular with a Python script to perform concurrent downloads and reduce the time required for data collection. After this, the extraction of relevant "IssueCommentEvent" objects is performed by checking whether the associated repository of a comment is Java related based on the list attained from the GitHub REST API. Notable information of these objects such as the text body of an issue comment and its title is extracted for natural language processing in later steps. Overall, this extraction process results in 627,450 GitHub issues and 1,855,870 issue comments to be collected.

3.3.2 Java 12 Documentation Caveat Extraction

To begin extracting API caveats, the API documentation must first be collected. At the time of writing, the Java Standard Edition 12 API was chosen for analysis, which was the latest Java version and documentation available. This API was chosen as it comprises the standard library for Java and is, therefore, expected to be the most generalised API used by most Java projects. Its documentation consists of Hypertext Markdown Language (HTML) pages for each class of the Java Development Kit (JDK) 12. In particular, the API documentation contains a webpage in HTML that lists the complete class hierarchy tree of the Java standard library.[2] This information allows us to data crawl the entire Java API documentation. First, the Uniform Resource Locator (URL) of all classes are mined from the class hierarchy page by collecting all hyperlink references on the page found within the appropriate HTML `section` element. The Beautiful Soup[3] library is used to parse the HTML content. The relative URLs of each class are found by locating list item elements (`li`) then anchor elements (`a`) residing within. From this, absolute URLs are constructed for 4,865 classes. Next, the HTML pages for each class is downloaded by recursively sending HyperText Transfer Protocol (HTTP) GET requests for each of the URLs generated. A total of 4,712 classes are found alongside 4,172 constructors and 33,827 methods. At this point, an important note to make for the Java SE 12 documentation is that it is automatically generated from comments in the source code implementation with the JavaDoc tool and is well-structured. For example, the parameters and possible exceptions for a method are consistently placed within certain HTML elements across all HTML pages. Hence, it is relatively simple to identify and extract sentences for each API element alongside additional information such as whether a method is deprecated from the existence of a `div` element with the "deprecationBlock" Cascading Style Sheets (CSS) class for example.

The extraction of caveat sentences is performed by creating a set of regular expressions based on keywords and patterns identified in Li et al. [2018]. These regular expressions allow us to check whether arbitrary strings contain some predefined patterns. API caveat sentences are then identified by checking if one of the regular expressions finds a match within a sentence from the API. This is executed recursively for all of the HTML pages, in which 107,601 caveat sentences are found. Of these sentences, 9,964 are regarded as *class level sentences*, which are sentences that describe the overall class and are located in the class description section of API documents (typically at the top of each API document). A snippet of this is shown in Figure 3.2. The Java 12 API is also structured such that each method/field/constructor for a given class contains a self-contained section that describes details specific to that element. An example of this is shown in Figure 3.3. We refer to the sentences that appear in the description of these sections as element level sentences. A total of 37,578 element level sentences are identified as caveat sentences. Other notable locations for sentences are sections that describe the parameters, exceptions or return value (if it is a method) of a particular API element. We refer to sentences found in those areas as *parameter sentences*, exception sentences and *return sentences* respectively. The number of caveat sentences found at these levels is 12,614, 32,623 and 11,765. Besides this, it is discovered that 1,522 API elements of the Java 12 API are deprecated.

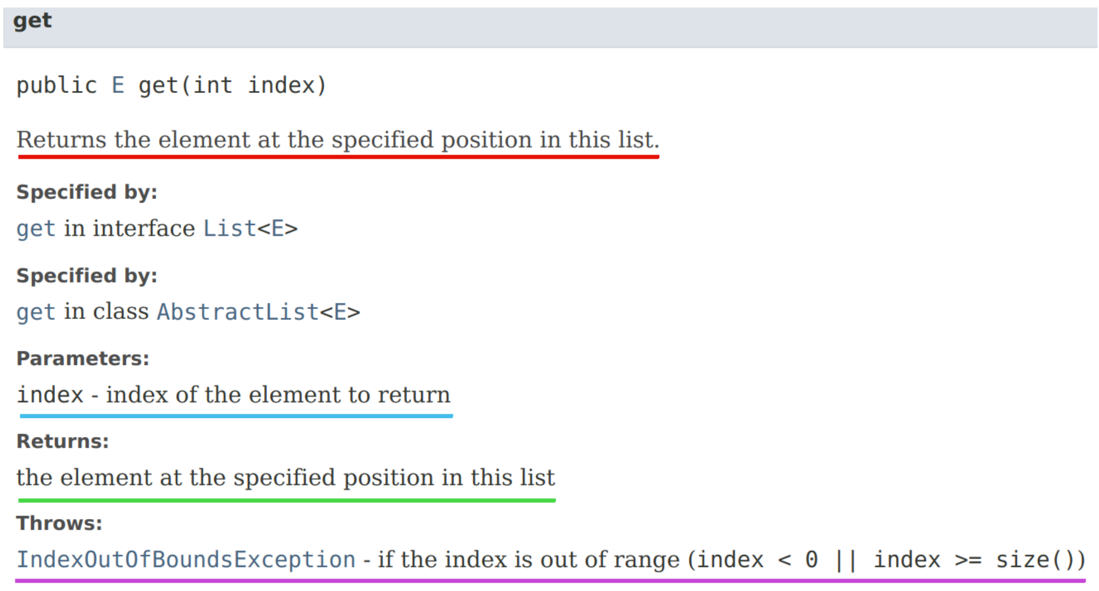
[2]https://docs.oracle.com/en/java/javase/12/docs/api/overview-tree.html
[3]https://pypi.org/project/beautifulsoup4/



```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

Figure 3.2: Example of the description section for the Java `String` API documentation. Sentences located in this area (highlighted in red) are referred to as *class level sentences*.



get

```
public E get(int index)
```

Returns the element at the specified position in this list.

Specified by:

`get` in interface `List<E>`

Specified by:

`get` in class `AbstractList<E>`

Parameters:

`index` - index of the element to return

Returns:

the element at the specified position in this list

Throws:

`IndexOutOfBoundsException` - if the index is out of range (`index < 0 || index >= size()`)

Figure 3.3: Example of the API documentation for `ArrayList.get(int index)`. Sentences located in the red underlined section, blue underlined section, green underlined section, and purple underlined section are referred to as *element sentences*, *parameter sentences*, *return sentences* and *exception sentences* respectively.

Category	Subcategory	Syntactic Pattern Examples
Explicit	Error/Exception	“insecure”, “susceptible”, “error”, “null”, “exception”, “susceptible”, “unavailable”, “not thread safe”, “illegal”, “inappropriate”,
	Recommendation	“deprecate”, “better/best to”, “recommended”, “less desirable” “discourage”
	Alternative	“instead of”,“rather than”,“otherwise”
	Imperative	“do not”
	Note	“note that”, “notably”, “caution”
Restricted	Conditional	“under the condition”, “whether ...”, “if ...”, “when ...”, “assume that ...”
	Temporal	“before”, “after”
Generic	Affirmative	“must”, “should”, “have to”, “need to”
	Negative	“do/be not ...”, “never”
	Emphasis	“none”, “only”, “always”

Table 3.1: API caveat categories and syntactic patterns from Li et al. [2018].

3.3.3 Data Preprocessing

Data cleaning and text preprocessing are required to allow NLP techniques/models to perform correctly. Numerous preprocessing steps are therefore applied to prepare the sentences for word and sentence embedding. In particular, the comments are in Markdown format, allowing simple removal of unwanted elements such as code blocks (which are wrapped between a pair of `'''`). Furthermore, all URLs, supplementary white space characters, apostrophes and punctuation (except for full stops appearing after a word) are removed. In particular, full stops are retained in consideration of inline code such as “array.length”. The removed parts of the text are substituted with a white-space character so that individual components of references to API elements (e.g. “array.length”) can be separated correctly during word tokenisation. For sentence tokenisation, which separates a paragraph into its sentences, full stops that were followed by a space were used to identify the split points. We note that a custom sentence tokeniser is required to prevent cases where full stops within inline code would result in a sentence split. Finally, word tokenisation is performed for each preprocessed sentence. This simply involves splitting the sentences based on white space characters such that each sentence is represented as a list of tokens. This representation is essential for further NLP tasks as it simplifies parsing/processing for computers.

Next is the removal of a customised set of English stop words provided by the NLTK library[4]. Stop words are common words that add little value/meaning for NLP such as "a" and "is". For the GitHub data, stop words that were of length 1 were retained such that single character variable names like "i" (commonly used by Java programmers in `for` loops) are retained. Overall, preprocessing and tokenisation of all GitHub comments results in 1,855,870 tokenised sentences. It is noted here that in Sun [2018], two additional steps are performed for the sentences: (1) coreference resolution, where pronouns such as "it" and "this" is substituted with the closest API names, and (2) lexicon construction in which an API lexicon (i.e. a vocabulary set of API element names) is constructed from the code of question and answer posts of Stack Overflow. Both of these steps are intended to reduce the complexity and noise of sentence embedding and matching. However, an initial search of the GitHub data via Google found that finding examples of API misuse from GitHub data is significantly more difficult than with Stack Overflow. For simplicity, step 1 is therefore ignored. Step 2 is reduced to keyword matching of both the API element name and its associated class name in Section 3.3.4 when determining the relevant API elements of each GitHub comment, and hence, the list of potentially relevant API caveats for those comments.

3.3.4 Candidate Filtering

The caveat sentences that were associated with deprecated API elements or were not an element sentence, parameter sentence or exception sentence were filtered. In particular, the sentences for deprecated elements are ignored because those caveats are typically obvious to developers as Integrated Development Environments (IDEs), software that support programmers with development, highlight cases where an API element is deprecated. This is accomplished using the `@Deprecated` annotation for Java that is tagged onto deprecated methods by API developers. I only focus on element, parameter and exception sentences to reduce the scope of sentences during sentence matching with GitHub comments and because these sentences were found to describe the most explicit caveats (such as the "index < 0 || index >= size()" constraint in the exception sentence of Figure 3.3). It is hypothesised that these explicit types of caveats are the most likely caveats to be matched to community sentences as they would require some form of paraphrasing to explain and would result in obvious errors and exceptions to be thrown.

For reference, it is observed that several caveat sentences extracted are invalid: they contain snippets of code or do not necessarily specify a constraint for users. Example of caveat sentences containing code snippets is shown in Listing 3.1, 3.2 and 3.3. An example of an API caveat that does not specify a constraint is the class level sentence "Java implementations must use all the algorithms shown here for the class Random, for the sake of absolute portability of Java code." from the `Random`

[4]https://www.nltk.org/index.html

class[5]. To counteract sentences containing code snippets, those that exceeded an arbitrarily (generous) limit of 400 characters were excluded from further analysis and sentence matching. The other type of invalid caveat sentences were ignored as they appeared to represent a small portion of the extracted caveats. In addition to this, it was noted that matching all of the GitHub issue comment sentences against every API caveat sentence would require significant processing time/power. The potentially relevant API caveats for each GitHub comment were therefore identified first to filter out non-relevant API caveats. This was achieved by concatenating the text of each comment and their associated GitHub issue, transforming the string to lowercase characters only, and performing a sub-string search for the (lowercase) class name and API element name of the API caveats. Caveats that were potentially relevant to more than 1000 GitHub comments were restricted to those 1000 GitHub comments to further reduce computation. This restriction involved 213 of the 21,932 API elements from the Java 12 documentation. We note that this limits the search area of sentence matching for those API caveats considerably, but is acceptable for an initial attempt of sentence matching. The total number of caveat sentences after preprocessing and all filtering functions totalled 73,831.

For GitHub comments, those that were not attached to an issue containing a code block were filtered. This is because the overall purpose of sentence matching is to indirectly link API caveats to code examples. From the dataset, 85,318 issues contained a code block with 290,019 associated comments. Those comments contained 629,933 sentences in total. The next step for sentence matching was to perform sentence embedding.

[5]https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/Random.html

Listing 3.1: An example of a caveat sentence extracted from the `javax.swing.Spring` documentation containing some snippets of code or mathematical expressions

```
If we denote Springs as [a, b, c], where a <= b <= c,
we can define the same arithmetic operators on Springs:

[a1, b1, c1] + [a2, b2, c2] = [a1 + a2, b1 + b2, c1 + c2]
-[a, b, c] = [-c, -b, -a]
max([a1, b1, c1], [a2, b2, c2]) = [max(a1, a2), max(b1, b2), max(c1, c2)]
```

Listing 3.2: An example of a caveat sentence extracted from the `java.security.cert.X509CRL` documentation explaining the structure of a `TBSCertList` object.

```
The ASN.1 definition of tbsCertList is:

TBSCertList ::= SEQUENCE {
        version         Version OPTIONAL,
        -- if present, must be v2
        signature        AlgorithmIdentifier,
        issuer          Name,
        thisUpdate       ChoiceOfTime,
        nextUpdate       ChoiceOfTime OPTIONAL,
        revokedCertificates   SEQUENCE OF SEQUENCE {
        userCertificate     CertificateSerialNumber,
        revocationDate     ChoiceOfTime,
        crlEntryExtensions   Extensions OPTIONAL
        -- if present, must be v2
        } OPTIONAL,
        crlExtensions      [0] EXPLICIT Extensions OPTIONAL
        -- if present, must be v2
}
```

Listing 3.3: An example of a caveat sentence extracted from the `java.text.BreakIterator` documentation that contains some sample code.

```
Creating and using text boundaries:

public static void main(String args[]) {
        if (args.length == 1) {
                String stringToExamine = args[0];
                //print each word in order
                BreakIterator boundary = BreakIterator.getWordInstance();
                boundary.setText(stringToExamine);
                printEachForward(boundary, stringToExamine);
                //print each sentence in reverse order
                boundary = BreakIterator.getSentenceInstance(Locale.US);
                boundary.setText(stringToExamine);
                printEachBackward(boundary, stringToExamine);
                printFirst(boundary, stringToExamine);
                printLast(boundary, stringToExamine);
        }
}
```

3.3.5 Sentence Embedding

Sentence embedding provides a method of comparing the similarity of two sentences by mapping them to vectors and computing the angle between the two vectors. The process of representing variable-length sentences as fixed-length vectors has been a major research topic related to deep learning approaches for NLP Adi et al. [2016]. TF-IDF based vector representations were created for the sentences using the Scikit-learn library[6]. This involved the class `TfidfVectorizer` and calling its `fit_transform` function to create vectors for all API caveat sentences. For word2vec sentence embedding, the `word2vec` class from the `gensim` library is imported and used. Note that the input for sentence embedding is the preprocessed, tokenised and filtered forms of caveat sentences and Github comment sentences. Hence, TF-IDF and word2vec vectors were generated for the 629,933 Github comment sentences and 73,831 caveat sentences.

3.3.6 Candidate Matching

Sentence matching is accomplished using cosine similarity for the sentence embedding vectors. For BM25, its function already outputs a ranking score for a particular match pair and does not require further computations. To evaluate the cosine similarity of the vectors, each of the 73,831 caveat sentence vectors is compared to a unique subset of the 629,933 sentence vectors from GitHub

[6]https://scikit-learn.org/

comments that are deemed potentially relevant (this was computed in Section 3.3.4). The equation in 3.4 is then used to compute the similarity scores. A similarity score of 1 represents high sentence similarity between two sentence vectors, while 0 represents dissimilarity. Given that BM25 ranks the sentence vectors relevant to each other, the BM25 scores were then normalised to the range 0 and 1, with 1 being representing the highest similarity. Thus, scores ranging from 0-1 were computed for each caveat sentence against their subset of potentially relevant GitHub comment sentences for TF-IDF, word2vec, and BM25. A weighted combination of BM25 and word2vec was then considered (with equal weightings for both scores) since using both approaches could be expected to balance the advantages/disadvantages of sentence embeddings and a probabilistic model.

$$\cos(\mathbf{A},\mathbf{B}) = \frac{\mathbf{AB}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n}\mathbf{A}_i\mathbf{B}_i}{\sqrt{\sum_{i=1}^{n}(\mathbf{A}_i)^2}\sqrt{\sum_{i=1}^{n}(\mathbf{B}_i)^2}} \tag{3.4}$$

To evaluate the performance of each information retrieval system (and the approach in general), a statistical sampling size from a statistical sampling method from Singh and Mangat [2013] is used. Specifically, the minimum number of samples required to ensure the estimated population mean is within a given confidence level and error margin can be calculated by the formula:

$$min = \frac{\frac{z^2 \times 0.25}{e^2}}{1 + \frac{(\frac{z^2 \times 0.25}{e^2} - 1)}{p}} \tag{3.5}$$

Where z is the z-score associated with the desired confidence level, e is the error margin and p is the population size. From this, an estimated sampling size of 383 is required given a 95% confidence interval, 5% error margin and population size of 73,831 (i.e. the number of preprocessed and filtered caveat sentences). However, a population size of 384 was used for consistency with Ren et al. [2018] in which notably more API caveat sentences from the Android documentation were analysed (160,112). The precision at K metric (precision@k) is chosen for a simplistic, initial evaluation of a query for the information retrieval systems, where K is set to 3 to reduce the amount of manual labelling required for all approaches (TF-IDF, word2vec, BM25 and word2vec + BM25). This is then combined with the mean average precision metric to quantify the overall performance of the systems across multiple queries. Hence, I generate a different random sample of caveat sentences for each information retrieval system and collect the top 3 results of those queries. Note that separate sampling is performed for each model since we also aim to observe and evaluate the variety of caveat sentences and their relevant GitHub comments for an initial study. The random sampling resulted in 959, 936, 1129 and 1051 results for the TF-IDF, word2vec, BM25, and word2vec + BM25 information retrieval systems for manual labelling. The sampling sizes are inconsistent because some caveats sentences can have less than 3 GitHub comments that are deemed potentially relevant in the given dataset.

3.4 Results

Labelling of the sentence matching results was conducted by manually checking whether the given API caveat appeared to be relevant to the GitHub comment containing the matched sentence. In particular, the caveat is compared against the entire GitHub comment such that contextual information from the GitHub comments can also be considered. Each query and result pair were annotated with either *relevant* or *non-relevant* based on whether the GitHub comment referred to the API caveat. A significantly small percentage of the results were found to be relevant to their API caveat query, with only 9, 16, 5 and 11 results identified as relevant for the TF-IDF, word2vec, BM25, and word2vec + BM25 information retrieval systems. In other words, only about 1% of the query results for the random samples are relevant to their API caveats. The mean average precision@3 for all of the information retrieval systems is approximately 0. It is noted that using a similar version of the code on another work[7] based on Stack Overflow and Android API caveats showed promising results. I, therefore, perform an analysis of these findings to understand why the systems performed poorly for GitHub data.

[7] `https://xin-xia.github.io/publication/ase192.pdf`

Listing 3.4: Example of a GitHub comment containing an error log from `https://github.com/ChrisRM/material-theme-jetbrains/issues/863`

```
similar problem just now, Linux (CentOS), upgrading from 2018.1 to 2018.2.
(I use the Darcula theme, and this stack trace is
    suggestive...)\r\n\r\njava.lang.ClassCastException: java.lang.Boolean cannot
    be cast to com.intellij.openapi.actionSystem.ex.ComboBoxAction\r\n\tat
    com.intellij.ide.ui.laf.darcula.ui.DarculaButtonUI.
getComboAction(DarculaButtonUI.java:75)\r\n\tat
    com.intellij.ide.ui.laf.darcula.ui.DarculaButtonUI.
getDarculaButtonSize(DarculaButtonUI.java:236)\r\n\tat
    com.intellij.ide.ui.laf.darcula.ui.DarculaButtonUI.
...
```

To verify the results, the API caveat queries of the random samples and the results returned by the information systems are examined. It was found that numerous GitHub comments contained code or error logs as text. Manually checking the preprocessing/filtering steps on these comments, I found that several sentences were not preprocessed/filtered correctly because of missing code tags in the Markdown text. In other words, users occasionally include code or error logs besides natural language text. An example of this is shown in 3.4, where the error logs are inline with actual sentences. Note that the example has been truncated and long lines were split for better viewing. This would result in the text from those code and error logs to be processed as natural language words during information retrievals. This is a particularly hard problem to fix given that code and error logs do use a single, specific template and how they are presented is dependent on the user. Besides this, it was noted that the repositories associated with GitHub comments could utilise many different APIs. This means that the text from GitHub comments could be related to numerous APIs and their caveats, which is not typically distinguished.

In comparison to Stack Overflow, users on Q&A platforms can usually "tag" their questions under a certain topic (such as Android or a specific API). However, GitHub does not provide that functionality as its community text options are intended for bug reports or suggestions. This means it is much harder to differentiate the GitHub comments by API relevance. The consequence of this was seen in cases where the correct API class name and element name of a given API caveat was mentioned in the GitHub comment, but were generic names (e.g. `truncate`, `compareTo`, `getOffset`), that were associated with a different API. It is noted that perhaps linking caveats for a more specific API (such as specific Java libraries like ReactiveX[8] or OkHttp[9]) could yield better results for GitHub data since those projects have their own GitHub repositories. Issues and comments for those

[8]http://reactivex.io/

[9]https://square.github.io/okhttp/

repositories would therefore likely be related to their respective APIs. Another possible reason for the negative results is the caveat sentences from the Java API. It was observed that some of the caveats could be considered too generic. For example, the `set` and `get` methods of the `java.util.ArrayList<E>` class both share the same sentence: "IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())". Even with the process of coreference resolution used in Sun [2018] to deal with similar caveat sentences, it is difficult to distinguish which API element the caveat relates to. An example where unrelated classes and methods of the Java API contain similar caveat sentences is from the constructor of `java.util.HashMap<K,V>` that says "NullPointerException - if the specified map is null" and from the `addAll` method of `java.util.ArrayList<E>` that says "NullPointerException - if the specified collection is null".

Another possible reason that GitHub is considerably worse for linking API caveat sentences is that its community text serves a different purpose compared to Q&A websites such as Stack Overflow. In particular, community text regarding API caveats would likely be in the format of a bug report, and answers to specific caveats can already be found on platforms catered to answering those caveats (i.e. Stack Overflow). Other forms of natural language that could be analysed for API caveat linkage is *commit* messages. Commits are typically individual file changes, where developers can attach a short message provide an overview of what changes were made. Another option is with pull requests, which contains a set of changes and longer discussions and reviews of these changes. However, both commits and pull requests are non-viable approaches for linkage as it is difficult to identify which code examples are associated with their natural language text. This is because both options can involve changes made to different lines of code files, and their descriptions for the changes made must be similar to the descriptions of API caveats. Code that has been uploaded to GitHub has also (likely) been executed, meaning explicit types of API caveats that result in exceptions/errors would be fixed beforehand.

Ultimately, it can be seen that due to the usage of same/similar sentences for different API elements, the inclusion of many different APIs on GitHub, and different purpose of GitHub results in a significant lexical gap for the Java 12 API caveat sentences and community text from GitHub. Though, a closer look at the caveat sentences and information retrieval results, in general, allowed an important observation to be made: caveats from exception sentences tend to describe explicit constraints in the Java 12 API documentation. These types of caveats would, therefore, be more applicable in a real-time setting where code could be analysed before it was released to GitHub. It is noted that these exception level caveats represent a large proportion of the API caveats (32,623 of 107,601 sentences). Finding a method to apply these constraints directly to code would be more beneficial in terms of both preventing issues with API caveats and helping programmers understand them. Overall, the poor results from this approach lead to the alternative idea of representing exception level caveats in a way where code analysis could be performed in Chapter 4.

3.5 Summary

This chapter described the background work that is used for linking API caveats to code examples and focuses on extending the solutions to a different domain: GitHub. The data of GitHub is of interest for API caveat linkage because it is another community platform that contains an abundance of code. However, GitHub is inherently different from the focus of Sun [2018] and Ren et al. [2018], which was strictly Q&A community websites. The Java 12 API documentation was used instead of the Android API documentation for API caveats as it could be generalised to more projects on GitHub. After extraction, 107,601 caveat sentences are found for the Java 12 API. For GitHub data, the GitHub Archive project is used to collect all GitHub events throughout 2018. The complete list of Java-related projects that had at least 2 stars is collected using the GitHub REST API, finding 291,152 projects. A total of 1,855,870 GitHub comments related to Java in 2018 are then identified from the GitHub Archive dataset. The sentences underwent data preprocessing, filtering, and sentence embedding (based on TF-IDF and word2vec) to perform sentence similarity computations between the two GitHub comment sentences and Java 12 API caveat sentences. After filtering, this involved matching 73,831 caveat sentences against 629,933 sentences from GitHub comments.

Information retrieval systems were created based on TF-IDF, word2vec, BM25, and word2vec + BM25, with caveat sentences used as queries and query results being GitHub comment sentences (ranked according to their relevance). Manual labelling was performed to measure the performance of the information retrieval systems. A statistical random sampling method was used with a 95% confidence interval and 5% error margin. It was found that 1% or less of the query results were found relevant for all of the information systems used. The cause of this was discussed and suspected as a result of the differences in GitHub as a community platform in comparison to Q&A websites. GitHub data is also polluted with many different APIs that makes it harder to discern which API arbitrary text is associated to.
In Chapter 4, I take a more direct approach to linking API caveats to code by transforming API caveat sentences into *caveat contracts* and conducting static code analysis.

Chapter 4

API Contracts Construction with Static Code Analysis

This chapter explains the concept of constructing caveat contracts from API caveats. In particular, it provides an overview of how caveat sentences from the Java 12 API documentation can be used to construct coding rules similar to code contracts. Applications of static code analysis for these caveat contracts are then explored, with one sample implementation for detecting bugs in real-time described.

Section 4.1 provides background information for code contracts and an overview of the work conducted for this Chapter: constructing code contracts from API caveat sentences and using static code analysis to apply the contracts to code in real-time.

Section 4.2 describes the design framework for generating and using code contracts. This includes a statistical analysis on the Java 12 API caveats, a description of the idea used for extracting information from caveat sentences for code contracts construction, and the concept of checker programs that can utilise the code contracts.

Section 4.3 explains the implementation of API contracts construction and static code analysis in an IntelliJ proof-of-concept plugin.

Section 4.4 showcases the IntelliJ plugin developed that uses caveat contracts to detect API misuse.

4.1 Introduction

Code contracts are a concept derived from object-oriented principles in which preconditions, postconditions, and invariants are defined for different software components. Specifically, the principle of "design by contract" suggests the use of specifications for code referred to as contracts. This improves code correctness and robustness as software components can only interact via obligations to code contracts. Using this concept, we can reduce the problem of linking API caveats to source code to the problem of mapping API caveats to code contracts, which we refer to as *API caveat contracts* or simply *caveat contracts* in this thesis. This can be used in real-time to provide feedback during the programming/development process in regards to API caveats. As a continuation of the previous chapter, I perform a statistical analysis of several caveat types for the Java 12 API documentation based on the work by Zhou et al. [2017]. I then propose a parsing technique to construct contracts from API caveats exception sentences, which represent a large proportion of API caveats for the Java 12 API and typically result in exceptions to be thrown. This extends upon the parsing techniques used by both Zhou et al. [2017] and Blasi et al. [2018] to collect a subset of API caveats related to explicit constraints including range limitations or not-null constraints. From this, I construct a total of 4,694 unique caveat contracts. Finally, I develop a checker plugin for IntelliJ that can use the caveat contracts to highlight violations of these API contracts in real-time.

4.2 Design

The first step to generating contracts for API caveats is the extraction of API caveat sentences. This process is described in the previous chapter (Section 3.3.2), and all caveat sentences extracted are re-used for this chapter. We recall that caveat extraction for the Java 12 reference documentation yields 107,601 caveat sentences, where a significant proportion of the sentences are exception sentences or parameter sentences (approximately 30% and 11% respectively). These are analogous to the subset of constraints identified and focused in Zhou et al. [2017] and found to represent the largest portion (43.7%) of API documentation in Monperrus et al. [2012b]. From Zhou et al. [2017], a set of heuristic rules was created from manual inspection that classifies the types of these constraints as (1) nullness not allowed, (2) nullness allowed, (3) type restriction and (4) range limitation. In particular, sentence normalisation techniques were used in both Zhou et al. [2017] and Blasi et al. [2018] in preparation for parsing techniques. The NLP techniques of POS tagging (identification of part-of-speech categories like nouns for words), dependency parsing (identification of the syntactic structure of sentences) and open information extraction (extraction of relation tuples from plain-text) are then applied to construct FOL expressions representing the constraints. An SMT solver can then be used to compare whether two FOL expressions were equivalent. This was used in Zhou et al. [2017] by comparing API documentation constraints with the associated code implementations to check whether API documentation was defective. The FOL expressions constructed from code involved traversing the ASTs of programs to extract exceptions thrown and the conditions for those exceptions.

For this thesis, I mainly focus on the not-null and *range limitation* constraints identified by Zhou et al. [2017] to reduce the scope of caveat mappings to contracts (given the diversity of API caveats). However, it should be noted that other categories of API caveats could also be transformed with other NLP techniques.

Besides this, I base my parsing approach on the sentence normalisation technique used in the previously mentioned work. For static code analysis, I focus on the process used by Zhou et al. [2017] where relevant code elements are first identified, followed by an appropriate analysis of those elements. This can be used by a plugin for some IDE to highlight contract violations (API misuses) while coding. However, I note that another method used in Zhang et al. [2018], which involves creating an alternative representation of code from ASTs containing only important features of an API method call, could also be used. Overall, the design architecture of the approach for this chapter is shown in Figure 4.1.

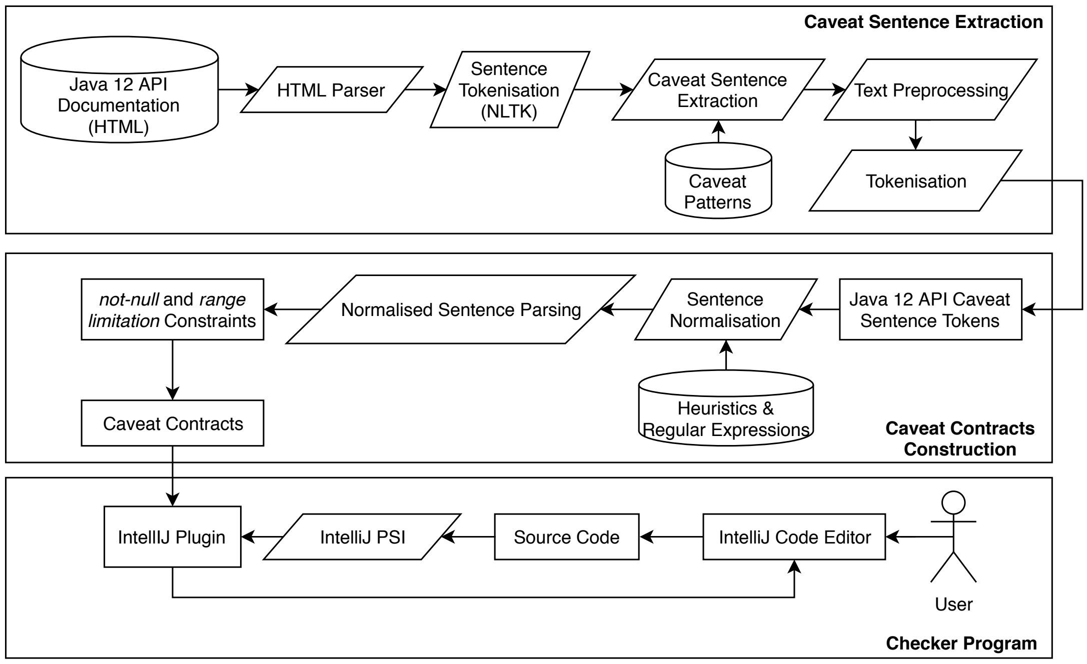
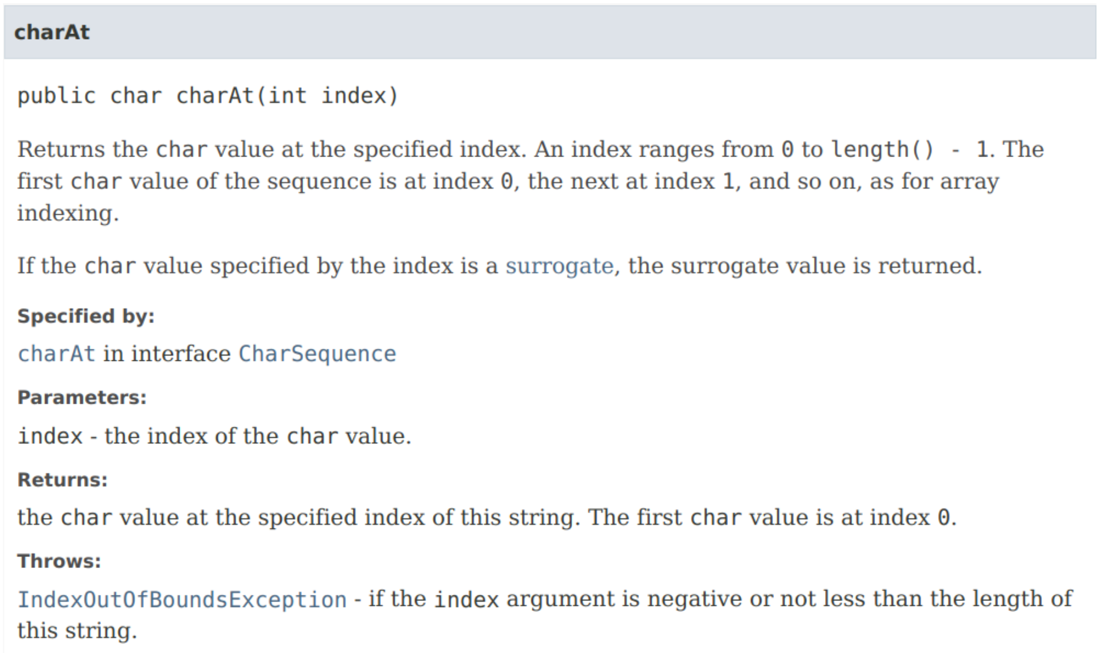

Caveat Sentence Extraction
Java 12 API Documentation (HTML)
HTML Parser
Sentence Tokenisation (NLTK)
Caveat Sentence Extraction
Text Preprocessing
Caveat Patterns
Tokenisation
not-null and range limitation Constraints
Normalised Sentence Parsing
Sentence Normalisation
Java 12 API Caveat Sentence Tokens
Caveat Contracts
Heuristics & Regular Expressions
Caveat Contracts Construction
IntelliJ Plugin
IntelliJ PSI
Source Code
IntelliJ Code Editor
User
Checker Program

Figure 4.1: Architecture design of the work for this chapter.

Section 4.2.1 provides an overview of the statistical analysis performed for the Java 12 API caveats. This is followed by Section 4.2.2 that describes the design idea for parsing and extracting constraints from API caveats. Finally, Section 4.2.3 gives an overview of ASTs and how checker programs can utilise caveat contracts.

4.2.1 Java 12 Caveat Statistics Analysis

An observation on the exception sentences of the Java 12 documentation is their consistent structure. These sentences follow a template of "`exception - description`", where `exception` is the exception class thrown and `description` describes conditions required for the exception to be thrown. An example of this is shown in Figure 4.2. It is noted that similar structures are used for other sentences such as the parameter sentences, which follow a template of "`param - description`", where `param` is the name of the parameter for a given method/constructor and `description` is the actual sentence describing some information about `param`. Overall, this information can be used to trivially separate key parts of caveat sentences found within these sections (i.e. the subject and associated description). For parsing constraints, we are only interested in the description parts of exception sentences, as this is where constraints imposed are described. Therefore, only the descriptions of these sentences are used for analysis. Next, I filter the corpus of exception sentences to obtain a unique set. This is because identical descriptions could be mapped to the same caveat contract, but with different information in regards to what exception is thrown. Hence, I generate a random sample of exception, caveat sentences based on Equation 3.5 for a 95% confidence interval, 5% error margin and population size of 4915 (unique exception sentences), which gives an estimated sample size of 356. I also collect a random sample for parameter sentences to compare to the results of Zhou et al. [2017], which uses the same confidence interval and error margin, but has a population size of 2704, resulting in an estimated sample size of 336.


charAt

`public char charAt(int index)`

Returns the `char` value at the specified index. An index ranges from `0` to `length() - 1`. The first `char` value of the sequence is at index `0`, the next at index `1`, and so on, as for array indexing.

If the `char` value specified by the index is a surrogate, the surrogate value is returned.

Specified by:

`charAt` in interface `CharSequence`

Parameters:

`index` - the index of the `char` value.

Returns:

the `char` value at the specified index of this string. The first `char` value is at index `0`.

Throws:

`IndexOutOfBoundsException` - if the `index` argument is negative or not less than the length of this string.

Figure 4.2: API documentation for the `charAt` method of the `java.lang.String` class

Manual labelling is then required for the samples to identify the prevalence of different caveat types for the sentences. In particular, the categories identified in Zhou et al. [2017] of *not-null*, *range limitation* and *type restriction* are used as labels with the addition of *ambiguous* to account for sentences that do not match any of the former classes. The *not-null* category involves sentences that specify some parameter cannot be the `null` value. The *range limitation* category specifies some numerical limitation on a parameter such as a non-negative requirement. Finally, the *type restriction* category indicates that a parameter must a particular class type or one of several types. The *nullness allowed* category that was also identified is not considered because it only describes an acceptable condition for contracts, whereas the other categories describe the conditions that constitute an API misuse (or caveat). Overall, the results of this for the parameter sentences sample is shown in Table 4.1, while the results for exception level sentences is shown in Table 4.2. We note that for 4.2, the counts contribute add up to more than 356 because 6 of the labelled caveat samples fit both the *not-null* category and the *range limitation* category.

	Labels			
	ambiguous	**not-null**	**range limitation**	**type restriction**
Count	291	26	19	0

Table 4.1: Manually labelled results for classes of 336 randomly sampled parameter level caveat sentences.

	Labels			
	ambiguous	**not-null**	**range limitation**	**type restriction**
Count	242	73	46	1

Table 4.2: Manually labelled results for classes of 356 randomly sampled exception level caveat sentences.

From Table 4.1, it can be seen that approximately 8% of unique parameter caveat sentences impose a *not-null* constraint and approximately 6% of the unique parameter caveat sentences impose a *range limitation* constraint. Despite the small percentage of sentences in parameter sentences sample that fit these categories, it is important to note that they represent an important type of API caveats that can cause software failures from exceptions. Furthermore, these caveat types generally contain explicit constraint descriptions that have little dependencies on other API elements, making them simpler to parse and an adequate baseline for constructing caveat contracts. In contrast, the results from Table 4.2 show that a considerably larger subset of 20% of unique exception caveat sentences specify a *non-null* constraint. This is also observed for the *range limitation* category with approximately 13% of sentences labelled.

An analysis of other categories is also conducted to determine what other caveat contracts can be constructed. In particular, the categories identified in Zhang et al. [2018] are derived from API misuse patterns data-mined from code snippets on Stack Overflow, but can also be mapped into contracts. For example, *missing control constructs* can be represented by a caveat contract that defines the control structure around some API call as a requirement. The same concept can also be applied to *missing or incorrect order of API calls* and *incorrect guard conditions*. For the subcategories of *missing control constructs*, which include *missing exception handling*, *missing if checks* and *missing finally*, we observe that they would all require explicit explanations for usage of these control structures. This is because usage of a control structure such as `if` or `finally` cannot be inferred without those stating those keywords. Furthermore, *incorrect guard conditions* could be considered a superset of the constraints analysed previously in Zhou et al. [2017] for the Java API documentation, but sentences that do not fit a category identified by Zhou et al. [2017] could be expected to be rare occurrences.

I focus on *missing control structures* and *missing or incorrect order of API calls* as categories to analyse. Using a similar approach to before, the estimated sample size required is calculated with Equation 3.5 for sentences from the method/constructor description, parameter section or return value section, which are 321, 377, 336 and 353 respectively for unique sentence population sizes of 8,829, 22,465, 2704 and 4,426. The exception section sentences are not considered as they would all be categorised as descriptions of *missing exception handling* or *incorrect guard conditions*. This emphasises the fact that the exception sentences are an important subset of API caveats. The labelled results are shown in Table 4.3. Note that *Control* refers to *missing control structures*, *Temporal* refers to *missing or incorrect order of API calls* and *guard* refers to *incorrect guard conditions*.

		Labels			
		Ambiguous	**Control**	**Temporal**	**Guard**
Sentence Location	**Constructor**	264	21	6	32
	Method	360	9	4	5
	Parameter	257	2	2	75
	Return	351	0	1	0

Table 4.3: Manually labelled categories for caveat sentences found in different locations of the Java 12 API documentation.

The results in Table 4.3 show the size of *Guard* caveat sentences is significantly larger in parameter sentences and the highest for constructor sentences. Meanwhile, method sentences have roughly equal caveat sentences belonging to the *Control*, *Temporal* or *Guard* categories. Overall, this indicates that perhaps API documents rarely contain API methods that require specific call orders or additional control structures, and caveats related to some guard conditions are the most prevalent. This is an interesting result given that the most prevalent type of API misuse found by Zhang et al. [2018] for 66,897 Stack Overflow posts was related to *missing control constructs*, followed by *incorrect guard conditions* then *missing or incorrect order of API calls*, which suggests that the Java 12 API documentation does not contain sufficient information for developers to handle API caveats involving control structures.

4.2.2 API Contracts Construction

Given the results found from statistical analysis in 4.2.1, the next step is to collect a set of API caveat sentences and attempt to transform them into caveat contracts. As a baseline study, the API caveats contained within the *not-null* and *range limitation* categories were found to represent a sufficient proportion of exception caveat sentences (see Table 4.2) and are therefore chosen. We note that this allows us to base our solution on the 64 heuristic rules and 29 regular expressions from Zhou et al. [2017] designed for parsing mainly *not-null* and *range limitation* constraints. For reference, the approach in Zhou et al. [2017] required intensive manual analysis to formulate the heuristics. Hence, extend upon the methods used with a key observation of English sentences and sentence normalisation techniques used by Blasi et al. [2018] to propose a much simpler approach of extracting constraints and constructing caveat contracts in general.

To design a simpler method for parsing API caveats with a *not-null* constraint, we observe that sentences must mention the term "null" to either specify if a `null` value is allowed or not allowed in code. Furthermore, given the structural information of the Java 12 API documentation for parameter sentences and exception sentences, extracting additional information such as the subjects of the sentences is simple (as described in Section 4.2.1). Therefore, dependency parsing POS tagging is not required. Another observation made based on the heuristics from Zhou et al. [2017] is the prevalence of the subject-verb-object (SVO) ordering for a given sentence Dryer [2005]. Specifically, English is known to follow SVO ordering despite other possible orderings such as subject-object-verb used in Japanese or subject-verb-object in Mandarin. This structure can be seen from rules 1 and 17 of Table 4.4 and all shown in Table 4.5. For rule 20, we observe the use of "non-null' as a predicative adjective to the subject, which is the only *non-null* heuristic formulated that has "null" appearing before the subject.

The final observation for *not-null* caveats is that whether some API method/constructor's parameter can be null is a boolean condition. In other words, this category of caveats represents the simplest form of a caveat contract as it only needs to specify whether a null value is accepted or not accepted. Given this information, a general approach to identify whether an API caveat is of the *non-null* category is to filter out caveat sentences that do not contain the sub-string "null". Next, we assume that API documentation aims to be simplistic and mentions of nullness within certain sentences (i.e. parameter or exception sentences) can be regarded as a *non-null* constraint. This is particularly true for the Java 12 API documentation as exception sentences (for example) describe the conditions required for the relevant exception to be thrown. Hence, any mention of "null" could be assumed to indicate a null value will result in an exception. We do note however that this assumption does not necessarily hold for other API documentation.
An alternative approach for parsing the *non-null* and *range limitation* API caveats is to utilise sentence normalisation techniques used in Zhou et al. [2017] and Blasi et al. [2018]. In particular, several regular expressions are identified by Zhou et al. to perform substitutions within a sentence before dependency parsing. These expressions are used to detect cases such as the names of variables, classes, and

Rule Number	Heuristic
1	[something] be/equals null
17	Value of [something] be/equals null
20	Non-null [something]

Table 4.4: Example of 3 of 20 heuristic rules for nullness not allowed from Zhou et al. [2017]. Note that the complete list can be found in 4.4 (Appendix).

Rule Number	Heuristic
1	[something] >/</= [value]
8	[something] be not negative/positive/false/true
20	[something1] equals [something2]

Table 4.5: Example of 3 of 23 heuristic rules for range limitations from Zhou et al. [2017]. Note that the complete list can be found in 4.5 (Appendix).

mathematical expressions, which are then replaced with a predefined token to facilitate dependency parsing. This process is a form of *sentence normalisation*. However, rather than using a dependency parser, we can simply use build heuristic rules based on the structure of SVO in English sentences. For example, a sentence in the exception section of the `ArrayBlockingQueue` constructor for class `java.util.concurrent.ArrayBlockingQueue<E>` says:

```
IllegalArgumentException - if capacity is less than c.size(), or less than 1
```

If we ignore the exception, and focus on the description text after the hyphen as described in Section 3.3.2, we obtain:

```
if capacity is less than c.size(), or less than 1
```

From this, we can then apply sentence normalisation techniques previously used. The rules formulated for this project are shown in 4.6 and are based on those used by Blasi et al. [2018]. These rules are used first in the normalisation process. For the given example, the first phase of normalisation results in:

```
if capacity is < c.size(), or < 1
```

The second phase of sentence normalisation involves the regular expressions shown in Table 4.7 from Zhou et al. [2017]. This allows us to utilise the SVO structure of English to perform a greedy parsing approach. The result of this is:

```
if @PARAM1 is @EXPR1, or @EXPR2
```

Here, “@PARAM1” represents “capacity”, which is the first parameter of the method, and “@EXPR1” and “@EXPR2” represent the mathematical expressions within the sentence. Finally, we extract the constraints from the sentence by using

Regular Expression	**Normalisation Form**
(not? (less | shorter) than) | ((greater | larger) than or equal to)	>=
(not? (greater | larger | longer) than) | ((less | shorter) than or equal to)	<=
(less | shorter) than	<
(greater | larger | longer) than	>
((is | are)? not negative) | ((be)? non-negative)	>=0
((is | are)? not positive) | ((be)? non-positive)	<=0
(is | are)? negative	<0
(is | are)? positive	>0
not equal(to)?	!=
equal to	==

Table 4.6: Regular expressions used to normalise mathematical phrases.

exploiting the linear nature of the SVO structure that allows words of a sentence to be considered in sequence from left to right. We first we tokenise the sentence such that we have a list of tokens, then iterate over the list in a single pass. States are saved throughout the iteration process based on the token observed at each step of the iteration. These states contain information such as whether a negating word (e.g. "not", "false") or expression was encountered. A simplistic rule for the parsing is to then construct a caveat contract whenever an expression token is encountered. The last parameter observed so far is then used to complete this expression. For the given example, both "@EXPR1" and "@EXPR2" are therefore completed with "@PARAM1" to form the constraints $capacity < c.size()$ and $capacity < 1$. We note that for the scope of this project, the rules used to construct these constraints are simplistic and only consider these mathematical constraints alongside logical "or" conditions as shown in the example above. However, this is sufficient to map API caveat sentences of the *not null* and *range limitation* categories. Other constraint types such as temporal rules for API calls would require much more complex parsing techniques.

Type	**Description**	**Regular Expression**
Specific Values	0.0, 0.1f, etc.	\W(-)?[0-9]+(,[0-9]+)*((\.[0-9]+)?[a-z]*)\W
	Member value of objects e.g. Location.x	\W(^(java\.|javax\.|org\.))? ([A-Za-z_]+\w+\.)+[a-z_]+[a-z0-9_]* [^\.A-Za-z0-9_]
Class methods and static members	Class methods, e.g., ClassA.func(Param1)	\W[A-Za-z_]+[A-Za-z_0-9]* (\.[A-Za-z_]+[A-Za-z_0-9])* (#[A-Za-z_]+[A-Za-z_0-9]*)?([^()]*)\W
	Static member, e.g. Desktop.Action#OPEN	\W([A-Za-z_]+[A-Za-z_0-9]*(\.[A-Za-z_]+ [A-Za-z_0-9])*)?(#[A-Za-z_]+ [A-Za-z_0-9]*)[^A-Za-z0-9_()]
	All upper case	\W(\w+\.)*([A-Z]+_)*[A-Z]+\W
	Class name	\W([A-Za-z_]+\w+\.)*[A-Za-z_]*[A-Z]+ \w+[^\.A-Za-z0-9_]
Expressions	A - B	\W\w+((\s+-)|(-\s+)|(\s+-\s+))\w+\W
	A + B	\W\w+\s*\+\s*\w+\W
	A * B	\W\w+\s**\s*\w+\W
	A..B	\W(?\s*\w+\s*)?\s*\.\s*\.\s*(?\s*\w+\s*)?\W
	[A, B]	\W[\s*\w+\s*,\s*\w+\s*]\W
	[A..B]	\W[\s*\w+\s*(\.\s*\.\s*)\s*\w+\s*]\W
	A <\<= B <\<= C	\W\w+\s*&lt;=?\s*\w+\s*&lt;=?\s*\w+\W
	A >\>= B >\>= C	\W\w+\s*&gt;=?\s*\w+\s*&gt;=?\s*\w+\W
	From A to B	\W(from\s+)?\w+\s+to\s+\w+\W
	A != B	\W\w+\s*!=\s*\w+\W
	Enumeration expression	\W (\s*\w+\s*)(,\s*\w+\s*)+,?\s*or\s*\w+\W

Table 4.7: The regular expressions identified in Zhou et al. [2017] for sub-sentence substitutions and dependency parsing.

4.2.3 Checker Programs

The previous section described a solution for constructing caveat constructs and assumed that program analysis tools exist which can utilise these contracts. Fortunately, such tools exist and are contained within a field known as static code analysis Louridas [2006]. Static code analysis uses checker tools that can perform checks on other programs without executing them. A common data structure used by these tools is Abstract Syntax Trees (ASTs), which models the structure of source code as a tree. Syntactic details of the code are not represented which enable an analysis of code to be significantly simpler. Consider the following code for the Euclidean algorithm:

```
while b != 0
    if a > b
        a := a - b
    else
        b := b - a
return a
```

The AST of the program is shown in 4.3. In particular, ASTs can be used to extract important information such as an API method call, what control structures (such as `for` or `if` blocks) they reside in, or what parameters are passed to the method.
Given a caveat contract, we could traverse the AST of a program and identify any code expressions and statements that correspond to the relevant API element.
Checking if a caveat contract is satisfied then requires one or more of the following processes: observing the surrounding API calls, identifying relevant expressions and statements, inspecting the various control flows of the program, evaluating expressions, and inspecting the API call parameters.
The IntelliJ platform was chosen for the development of a proof-of-concept plugin given that it was one of the three most popular Java IDEs (next to Netbeans and Eclipse) and provides a simple interface for accessing the AST of Java code. IntelliJ provides an interface known as the Program Structure Interface (PSI) that allows developers to interact with the AST of a program approaches to navigation of an AST, making the development of a plugin relatively simple.

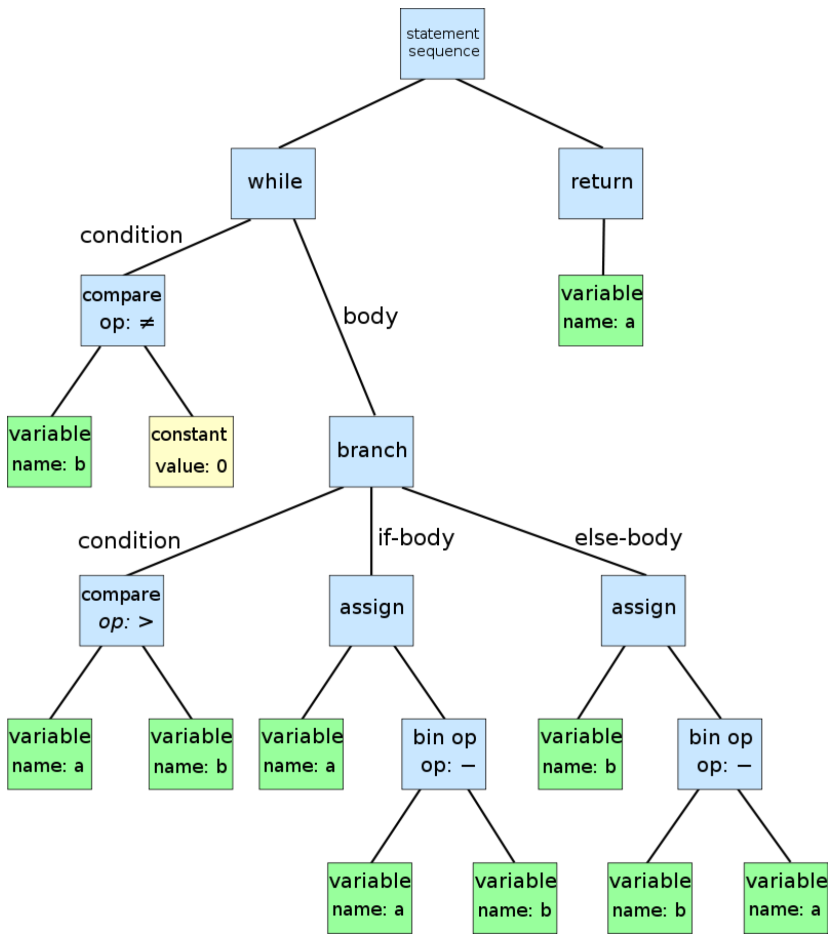


Figure 4.3: Example of an AST for the Euclidean algorithm. Copied from `https://en.wikipedia.org/wiki/Abstract_syntax_tree`

4.3 Implementation

Implementation of the caveat contract construction for the Java 12 API is completed using Python 3.6 and its standard libraries. The contracts generated are exported as a JSON array to be used in other applications (i.e. the plugin) and can be found in the project repository of the IntelliJ plugin (`./exception_range_rules_filtered.json`).

4.3.1 Java API Caveat Contracts

The caveat sentences extracted from the previous chapter (Section 3.3.2) was reused and loaded as a Pandas library[1] `DataFrame` object. The object consists of rows of individual caveats while the columns represent various information about a caveat such as the corresponding caveat sentence and which Java class it belonged to. This

[1]https://pandas.pydata.org/

allowed filtering functions to be easily applied such that a subset of the caveats could be selected based on some condition. Constructing the caveat contracts involves following the approach described in 4.2.2. In particular, this involves defining the regular expressions from 4.7 and 4.6 in Python's `re` library syntax. A preprocessing step is then performed to reduce the set of caveat sentences that are clearly not *not null* or *range limitation* caveats. I create generalised regular expressions used to identify the caveat sentences that belong to these categories from the fact that sentences of the *range limitation* category, for example, would likely contain soon mathematical logical operators such as "<" for less than/greater than. The regular expressions used can be found in Tables 4.8, 4.9, 4.10 and 4.11. Applying these filters reduces the set of invalid caveat sentences and the amount of work required for manually labelling during evaluation.

Regular Expression
null

Table 4.8: Regular expressions for identifying exception sentences of the *not-null* category.

Regular Expression
not(be)? null
non-null

Table 4.9: Regular expressions for identifying parameter sentences of the *not-null* category.

Regular Expression
< | > | =
equal | equal to | equivalent to | illegal value | is (nan | infinite | empty)
\b(less | smaller | greater | larger)\b
\b(range | negative | positive | non-negative | non-positive)\b

Table 4.10: Regular expressions for identifying exception sentences of the *range limitation* category

Regular Expression
< | > | =
(less | smaller | greater | larger) than
negative | positive | non-negative | non-positive

Table 4.11: Regular expressions for identifying parameter sentences of the *range limitation* category.

Before the implementation of sentence parsing and contract construction, I random sampling to the sentences for evaluation purposes in later steps. The unique sentences from the *not-null* exception and parameter sentences are identified, totalling 835 and 193 respectively. For the *range limitation* category, the number of unique exception and parameter sentences is 193 and 149. I then randomly generate a sample of size 100 for each of these sets, which is an appropriate sample size given the small sets of unique sentences and overall simplicity. For the *not-null* categories of exception and parameter sentences, the relevant parameter for the constraint (if any) was identified. As an example, for the sentence "prefix - the prefix of the tag, may not be null", *prefix* would be marked as the constraint subject. Since the parameter sentences follow a template as described in 4.2.1, we note that an algorithm to extract the relevant parameter is trivial whereas for an exception sentence like "if the specified sorted set is null"[2], parsing is harder given that the name of the relevant parameter ("s" in this case) is not used.

Labelling of the *range limitation* sentences was completed by expressing constraints as mathematical expressions. An example of this is with the sentence "IllegalArgumentException - if iv is null or (iv.length - offset < 2 * (wordSize / 8))"[3], the constraint of the sentence would be identified as $iv.length - offset < 2 * (wordSize/8)$. After manual labelling, 90% of the 100 random parameter sentences filtered for *not-null* were found to actually describe a *not-null* constraint. For the exception level sentences, 87% were found to describe a *not-null* constraint. For the *range-limitation* samples, 49% and 72% of the parameter and exception sentences were found to express a *range limitation* constraint, respectively. These results reveal that using basic sub-string searches is viable for identifying *not-null* constraints, but more complex approaches are required for identifying (and extracting) *range limitation* caveats.

For implementing the sentence parsing, we only focus on the exception sentences as a baseline. The sentence normalisation approach is implemented as described in 4.2.2, with regular expression from Tables 4.7 and 4.6 used to perform substitutions with labelled words. The exception sentences are then tokenised by white-space characters and a simple `for` loop is used to iterate through the tokens of each sentence. Constraints are then extracted throughout the iteration process when

[2]Taken from the constructor of `java.util.TreeSet<E>`

[3]Taken from the `RC5ParameterSpec` constructor of the `javax.crypto.spec.RC5ParameterSpec` class

incomplete expression tokens are observed as described previously. Caveat contracts are then constructed and represented as individual objects that contain the associated class, and API element name, alongside the parameter name, constraint operator (such as the "<" operator to represent less than) and comparison value. This process results in 4,694 unique caveat contracts. We note that other representations might be required for other caveat categories, but this is sufficient for *not null* and *range limitation* caveats and for use by a checker program.

Manually comparing the caveat contracts constructed and the mathematical expressions from earlier, 41 of the 72 API caveat sentences that contain a range limitation constraint were correctly extracted. Furthermore, the approach was able to extract partially correct constraints for 13 other caveat sentences (where we define partial as some constraints that would also cause an exception was identified). Hence, 54 of the caveat sentences had caveat contracts that could be applied to static code analysis. We note that the approach proposed also collects caveats of the *not null* category by simply using an equality operator ("=" or "!="). Due to this fact, I only focus on caveat constructs using this method and the alternative method proposed using basic keyword searching is not included in Section 4.3.2.

	Predicted Non-constraint	**Predicted Constraint**
Actual Non-constraint	25	2
Actual Constraint	19	54

Table 4.12: Confusion matrix for the sentence normalisation approach proposed.

Overall, the results found are represented in terms of a confusion matrix in 4.12 to allow computation of accuracy, precision, recall, and F-measure scores. It can be seen that the true positive (TP) is 54, true negative (TN) is 25, false positive (FN) is 2, and the false negative is 19. Using the equations 4.1, 4.2, 4.3 and 4.4, the accuracy is 0.77, recall is 0.73, precision is 0.96 and F1 score is 0.83. These results suggest that applying a simplistic sentence normalisation is a viable approach for extracting constraints from sentences that describe exceptions thrown under certain conditions.

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.1}$$

$$\text{recall} = \frac{TP}{TP + FN} \tag{4.2}$$

$$\text{precision} = \frac{TP}{TP + FP} \tag{4.3}$$

$$\text{F1} = \frac{2 \cdot recall \cdot precision}{recall + precision} \tag{4.4}$$

4.3.2 IntelliJ Plugin with Static Code Analysis

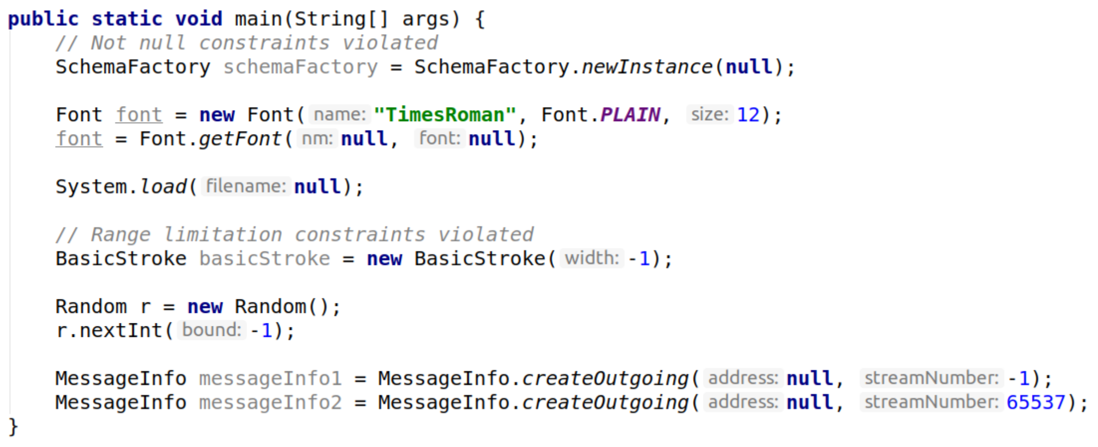
IntelliJ's PSI provides the `AbstractBaseJavaLocalInspectionTool` class that can be extended for creating plugins involving static code analysis. This is used to define a *visitor* that traverses the AST of a program periodically. In addition to this, I defined several classes to represent the concepts of an API method, API class, caveat, and for storing a collection of all caveat contracts loaded from 4.3.1. Java's `HashMap` is used as the underlying data structure for storing the caveat contracts. This allows searching for the contracts of an arbitrary method to consist of two simple, efficient steps: obtaining the methods attached to a certain class (via the full class name as a hash) and finding the correct method by searching through the associated list of methods. It is noted that further optimisations could be used in future to quicken the retrieval of contracts for a given API method. An example of this is using the hash of the method signature to map directly to its caveat contracts. However, a simple design was chosen given that the plugin was a proof-of-concept application.

The code analysis process involves implementing the visit function for the visitor such that each expression call in the program is identified and analysed. Specifically, the full class name, method name and argument types are identified and compared to the set of caveat contracts associated with that API element. Each caveat contract is then invoked and checked against values provided by the AST, such as the argument values provided to the API call. For the *not-null* constraints, this check simply requires comparing the argument values to a `null` value. For the *range limitation* constraints, the logical operators ($<$, $\leq$, etc.) are identified from the caveat contracts and mapped to a Java boolean expression involving the included alongside the comparison value from the contracts. This meant that an SMT solver was not required and we could simply apply the range constraints within Java code, though it is noted that an SMT solver could be used in for more complex constraints. As a baseline checker program, only argument values passed directly to an API method call are analysed. IntelliJ's PSI then provides the functionality to register a problem that will be displayed within the code editor of the IDE. Each caveat contract that was found to be violated would then result in a problem to be registered for the associated API call.

4.4 Results

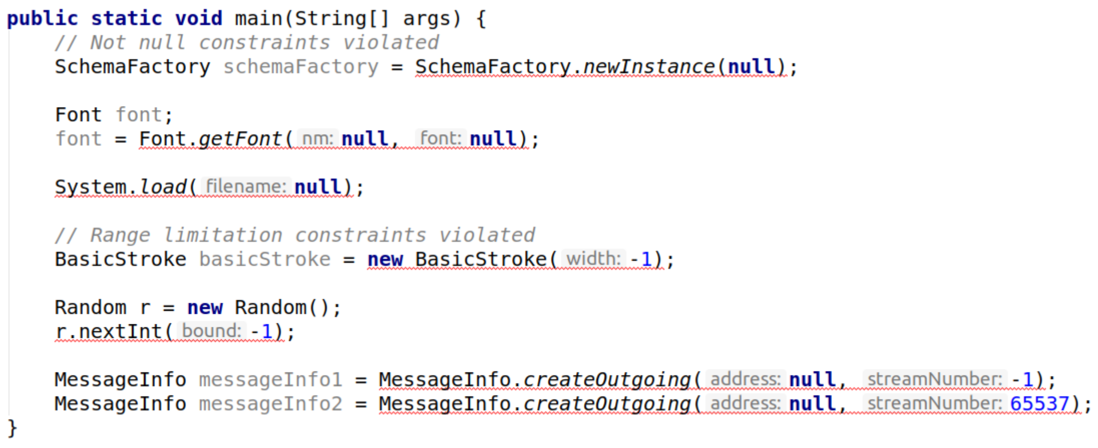
The complete process of constructing caveat contracts from some API documentation and applying it to a checker program was described. To showcase the checker plugin and its functionalities several examples are of constraint violating code is shown in Figure 4.4. Using the plugin, the result is each constraint-violation is highlighted in IntelliJ with red squiggly lines as shown in Figure 4.5. Hovering the cursor over any of these highlighted problems will reveal a pop-up that provides more detailed information about the caveat contract violation. An example of this is shown in 4.6. These examples are shown with IntelliJ version

2018.2.4 (community edition).



```
public static void main(String[] args) {
    // Not null constraints violated
    SchemaFactory schemaFactory = SchemaFactory.newInstance(null);

    Font font = new Font( name: "TimesRoman", Font.PLAIN,  size: 12);
    font = Font.getFont( nm: null,  font: null);

    System.load( filename: null);

    // Range limitation constraints violated
    BasicStroke basicStroke = new BasicStroke( width: -1);

    Random r = new Random();
    r.nextInt( bound: -1);

    MessageInfo messageInfo1 = MessageInfo.createOutgoing( address: null,  streamNumber: -1);
    MessageInfo messageInfo2 = MessageInfo.createOutgoing( address: null,  streamNumber: 65537);
}
```

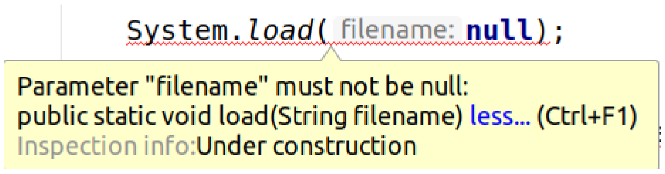
Figure 4.4: Example of IntelliJ's (lack of) problem highlights for obvious constraint violations.



```
public static void main(String[] args) {
    // Not null constraints violated
    SchemaFactory schemaFactory = SchemaFactory.newInstance(null);

    Font font;
    font = Font.getFont( nm: null,  font: null);

    System.load( filename: null);

    // Range limitation constraints violated
    BasicStroke basicStroke = new BasicStroke( width: -1);

    Random r = new Random();
    r.nextInt( bound: -1);

    MessageInfo messageInfo1 = MessageInfo.createOutgoing( address: null,  streamNumber: -1);
    MessageInfo messageInfo2 = MessageInfo.createOutgoing( address: null,  streamNumber: 65537);
}
```

Figure 4.5: Example of how the developed plugin handles API caveat contract violations.

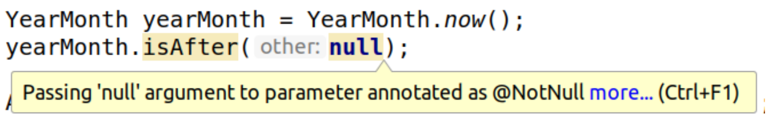
For interest, it is noted that IntelliJ does implement its version of contracts that require code comment annotations. However, these contracts require special syntax specific to IntelliJ and must be manually implemented for each method and are inconsistent. An example of this inconsistency is shown in Figures 4.7 and 4.8. In 4.7, a *not-null* constraint is violated for the `isAfter` API call and correctly highlighted by IntelliJ's contracts, but in 4.8 another *not-null* constraint is violated for the `add` method of a `PriorityQueue` object, which throws a `NullPointerException` if executed. Hence, it can be seen that the Intelli's code contracts contain two major drawbacks: it requires the manual implementation of IntelliJ contracts for each method and is prone to errors or inconsistency. The plugin implemented can solve these problems by automating the process of mapping sentences from the Java 12 API documentation to custom caveat contracts. Furthermore, it allows the second problem with some manual intervention as each caveat constraint specifies a single constraint. It can, therefore, be seen that these caveat contracts offer some improvement and can also be extended to other IDEs.



```
System.load( filename: null);
```

Parameter "filename" must not be null:
public static void load(String filename) less... (Ctrl+F1)
Inspection info:Under construction

Figure 4.6: Example of the displayed problem message for an API caveat contract violation.

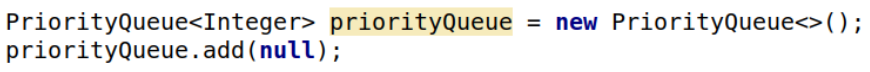


```
YearMonth yearMonth = YearMonth.now();
yearMonth.isAfter( other: null);
```

Passing 'null' argument to parameter annotated as @NotNull more... (Ctrl+F1)

Figure 4.7: Example of IntelliJ's contracts.

Overall, the plugin in its current iteration is able to highlight the explicit contract violations from the Java 12 API documentation related to *not-null* or *range limitation* constraints. It is clear that with the addition of other categories of API caveats, the plugin could be used to highlight more complex errors and problems for developers in real-time, minimising API misuse and helping developers learn the correct usage of an API. In terms of constructing caveat contracts, further research could yield methods that both improve the precision and recall of constraints extraction. It is noted that only one API documentation was analysed for the scope of this thesis, but the methods proposed apply to other languages and API documentation. However, one drawback of the plugin is that it can only display warnings after an API misuse, meaning an alternative method is required to describe an API caveat to users before it has occurred. This could involve listing potential API caveats from those currently used in a program, though is outside of the scope of this thesis. In conclusion, the use of static code analysis shown reveals that that natural language can be mapped to source code to detect API misuse, improve understanding of an API, and potentially increase the efficiency of development by preventing bugs/errors.



```
PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();
priorityQueue.add(null);
```

Figure 4.8: Example of IntelliJ's contract inconsistency.

4.5 Summary

This chapter focused on the concept of code contracts and its how it could be associated with API caveats. The idea of transforming API caveats to contracts was presented and implemented for a subset of API caveats related to *not-null* and *range limitation* constraints. Statistical analysis was also performed to determine the prevalence of these categories of caveats in the Java 12 API documentation. It was found that 20% of unique API caveat sentences appearing in the exception sections of the documentation imposed a *not-null* constraint. For the *range limitation* category, 13% of unique API caveat sentences were found to describe a constraint involving range. These categories were chosen for investigation and implementation of code contracts as they provided explicit constraints that result in exceptions. They can also be regarded as the simplest caveats and considered as a baseline for mapping natural language to code contracts. Furthermore, an approach for parsing of API caveat sentences to construct caveat contracts was proposed. This was used to extract the not-null and range constraints from sentences, resulting in 4,694 unique caveat contracts. The design and implementation of a proof-of-concept IntelliJ plugin that acted as a checker was also presented. The potential of this approach for linking natural language to source code was also discussed. In particular, the plugin showcases that API misuses can be detected in real-time, improve understanding of APIs and direct developers to correct usage of an API.
In Chapter 5, the findings of this thesis are summarised and ideas for future work are presented.

Chapter 5

Conclusion

In this chapter, I summarise the topic of investigation, the main challenges identified in Chapter 1 and how they relate to the main contributions of this thesis and make suggestions for future work.

Section 5.1 provides an overview of this thesis in terms of topic, challenges, contributions and findings.

Section 5.2 describes some future work ideas relevant to API caveats and constructing API caveat contracts.

5.1 Summary

This thesis presents the approach of NLP techniques to link Java 12 API caveats to community text from GitHub. Besides this, methods to construct caveat contracts from API documentation and its applications are also explained. Chapter 1 introduces the concept of API caveats. A realistic scenario is also provided to describe the motivation for linking API caveats to code, and the main challenges of this thesis are identified. Chapter 2 presents related work that defines the key concepts of this thesis: API caveats, using NLP techniques to augment API caveats with code examples, and the applications of static code analysis with API documentation. In Chapter 3, the process of extracting the documentation and caveats from the Java 12 API is described in detail alongside the extraction process of GitHub community text data and the construction of multiple information retrieval systems based on TF-IDF, word2vec, BM25. The negative results of this approach are discussed and used to introduce an alternative method applied in Chapter 4. Chapter 4 presents the idea of caveat contracts and the transformation of API caveats to form caveat contracts. The design of a checker program that can utilise the caveat contracts is given.

The main challenges of this thesis (Chapter 1) revolve around how API caveats can be linked to code. The challenges consist of effective sentence embedding and matching, sentence parsing techniques to extract constraints and the use of static code analysis with caveat contracts. In Chapter 3, I tackled the problem of sentence embedding and matching by extending the work of Sun [2018] with additional information retrieval system models. Specifically, information retrieval systems built upon TF-IDF, BM25 and word2vec + BM25 were used in addition to just word2vec to investigate the differences of embedding models and how this approach could be applied to GitHub. I attempt to match 73,831 API caveat sentences against 629,933 GitHub comment sentences using the models mentioned and perform statistical sampling with manual labelling to determine the performance of these information retrieval systems. However, a significant lexical gap was observed given that only 1% of query results were found relevant to their API caveat queries. The negative result of this reveals that linkage via sentence embeddings cannot be performed in the presence of large lexical gaps.

In Chapter 4, I tackle the challenges of sentence parsing for constraints extraction by utilising and combining ideas from Zhou et al. [2017] and Blasi et al. [2018] to propose a sentence normalisation approach. I conduct a statistical analysis of the Java 12 API documentation for constraints recognised in Zhou et al. [2017] as *not-null* and *range limitation* constraints to showcase their prevalence in the Java API documentation. I find that 20% of unique caveat sentences from exception sections of the documentation specify a *non-null* constraint, while approximately 13% of these sentences specify a *range limitation* constraint. I tackle the challenge of static code analysis by developing a proof-of-concept checker and showing how it can detect contract violations. Overall, I construct 4,694 unique caveat contracts and develop an IntelliJ plugin to automatically detect caveat contract violations in real-time.

5.2 Future Work

Extending caveat contracts to include other caveat categories. Other interesting caveat types mentioned in Chapter 1 should also be considered for caveat contracts. These require much more complex parsing techniques. In addition, other methods to resolve dependencies of API caveats should be investigated. For example, the "add" method of "ArrayList" in Java specifies "IndexOutOfBoundsException - if the index is out of range (index < 0 || index > size())". Identifying that "size()" refers to the `size` method of `ArrayList` is a complex problem given that cross-class dependencies exist.

Detecting bugs in existing software and more complex static code analysis. The automatic detection of bugs remains a challenging problem. Two notable works that the methods proposed in this thesis can be combined with is Zhang et al. [2018], which uses API call sequences to detect bugs, and Wen et al. [2019], which uses mutation analysis to detect bugs. In Zhang et al. [2018], the Boa programming language is used as an interface to a large number GitHub repositories associated ASTs to data-mine correct usage patterns of APIs. Caveat contracts could be applied to this to detect API misuse in free, open-source software. Caveat contracts could also be integrated into development pipelines as checks that are performed before code is uploaded to GitHub. Besides this, methods to effectively present these errors and suggest fixes to developers still require further research as this thesis only identifies misuse after it has occurred. More complex static analysis for special API caveats, such as those that deal with the temporal order of API calls, is another topic for investigation. For Wen et al. [2019], I suggest the topic of mutating caveat contracts to create correct code usage patterns. Specifically, this could be used to generate test suites that add an extra layer of code validation and overall development efficiency.

Testing other APIs, programming languages and natural languages. Evaluating the practicality of caveat contracts and their construction for other APIs is a useful topic that should be investigated. This is because differing results can be expected for even minor differences such as dynamically typed instead of statically typed programming language. Furthermore, the different syntactic styles used by developers for documentation presents a challenge for sentence parsing. Methods to construct caveat contracts for other languages (like Mandarin) would also be useful.

Chapter 6

Appendix

6.1 Heuristic Rules from Zhou et al. [2017]

6.1.1 Not-null Heuristics

Rule	Nullness Not Allowed Heuristics (@exception or @throws)
1	[something] be/equals null
2	[something] be equal/equivalent to null
3	[something1] or [something2] be/equals null
4	[something1] or [something2] be equal/equivalent to null
5	[something] parameter be null
6	The specified [something] be null
7	Any/none of [something] be null
8	Either/neither/any/all/both/none parameter(s) be/equals null
9	Either/neither/any/all/both/none parameter(s) be equal/equivalent null
10	Either/neither [something1] or/nor [something2] be/equals null
11	Both [something1] and [something2] be null
12	The [type] be null
13	[parameter phrase] be/equals null
14	[parameter phrase] be equal/equivalent null
15	[something]âĂŹs value be/equals null
16	[something]âĂŹs value be equal/equivalent null
17	Value of [something] be/equals null
18	Value of [something] be equal/equivalent null

Table 6.1: Complete list of heuristic rules for exception nullness not allowed.

Rule	Nullness Not Allowed Heuristics (@param)
19	[something] can not be null
20	Non-null [something]

Table 6.2: Complete list of heuristic rules for parameter nullness not allowed.

6.1.2 Range Limitation Heuristics

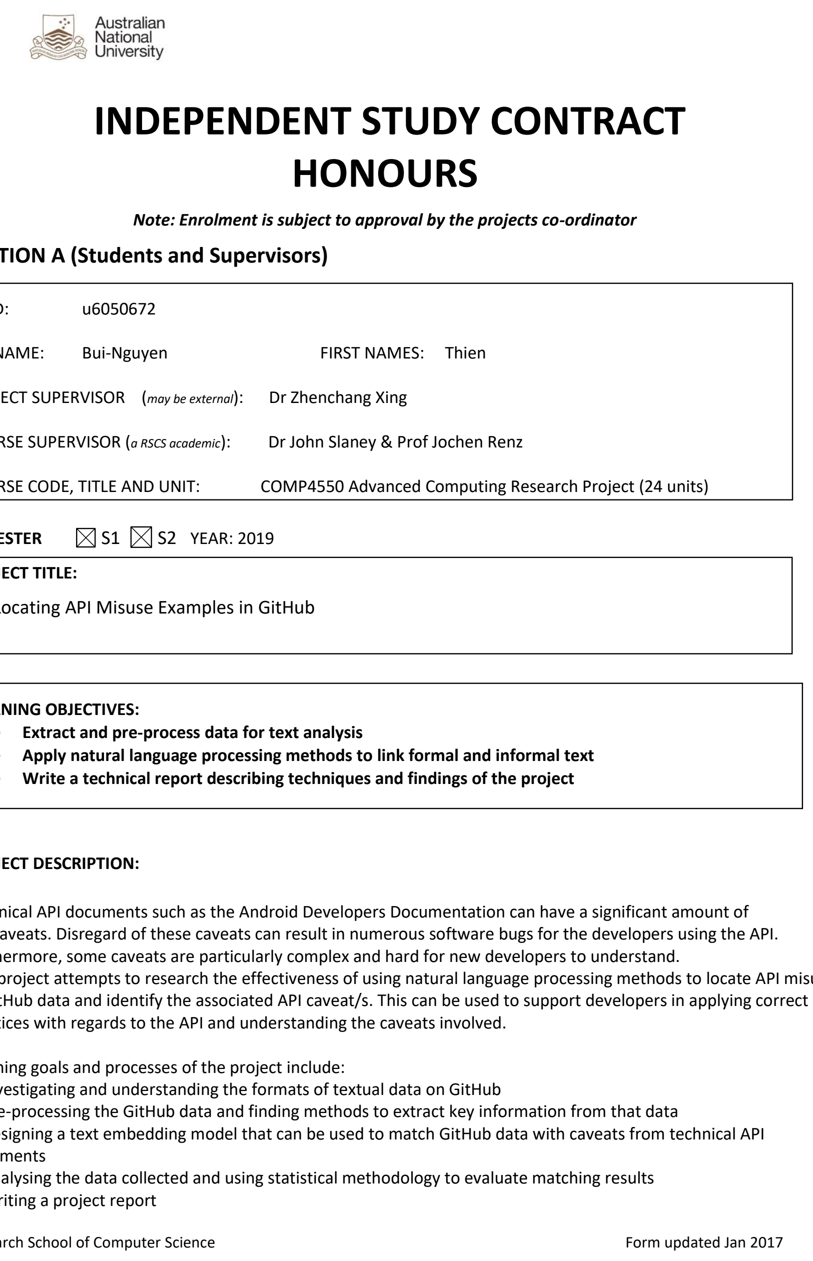
Rule	Range Limitation Heuristic (@exception or @throws)
1	[something] >/</= [value]
2	[something] be {not} less/greater/larger/equal/equivalent than/to [value]
3	[something] equals [value]
4	[something1] or/and [something2] be {not} less/greater/larger/equal/equivalent than/to [value]
5	Computing [expression] be not less/greater/larger/equal/equivalent than/to [value]
6	Computing either [expression1] or [expression2] be {not} less/greater/larger/equal/equivalent than [value]
7	Product/sum of [something1] and [something2] be not less/greater/larger/equal/equivalent than/to [value]
8	[something] be not negative/positive/false/true
9	[something1] or/and [something2] be not negative/positive/false/true
10	[something1] and [something2] be not the same
11	[something1] equals [something2]
12	[something] be not in/out of/outside of range [range value]
13	[something] be not in/out of bounds
14	[something] be not [value]
15	[range expression] (only the expression,like)
16	[something] be not between [value1] and [value2]
17	[something] be not [value set]
18	[something] be not one of [value set]
19	[something] be not one of following: [value set]
20	[something] be not one of supported data, which are [value set]

Table 6.3: Complete list of heuristic rules for exception range limitations.

Rule	**Range Limitation Heuristic (@param)**
21	[something] can/must not be negative/positive/non-negative/non-positive
22	[something] must be greater/less/larger than [value]
23	[something] be greater/less/larger than [value]

Table 6.4: Complete list of heuristic rules for parameter range limitations.

6.2 Independent Study Contract



Australian National University

INDEPENDENT STUDY CONTRACT
HONOURS

Note: Enrolment is subject to approval by the projects co-ordinator

SECTION A (Students and Supervisors)

UniID:	u6050672		
SURNAME:	Bui-Nguyen	FIRST NAMES:	Thien
PROJECT SUPERVISOR (*may be external*):	Dr Zhenchang Xing		
COURSE SUPERVISOR (*a RSCS academic*):	Dr John Slaney & Prof Jochen Renz		
COURSE CODE, TITLE AND UNIT:	COMP4550 Advanced Computing Research Project (24 units)		

SEMESTER [✓ S1] [✓ S2] YEAR: 2019

PROJECT TITLE:

Locating API Misuse Examples in GitHub

LEARNING OBJECTIVES:

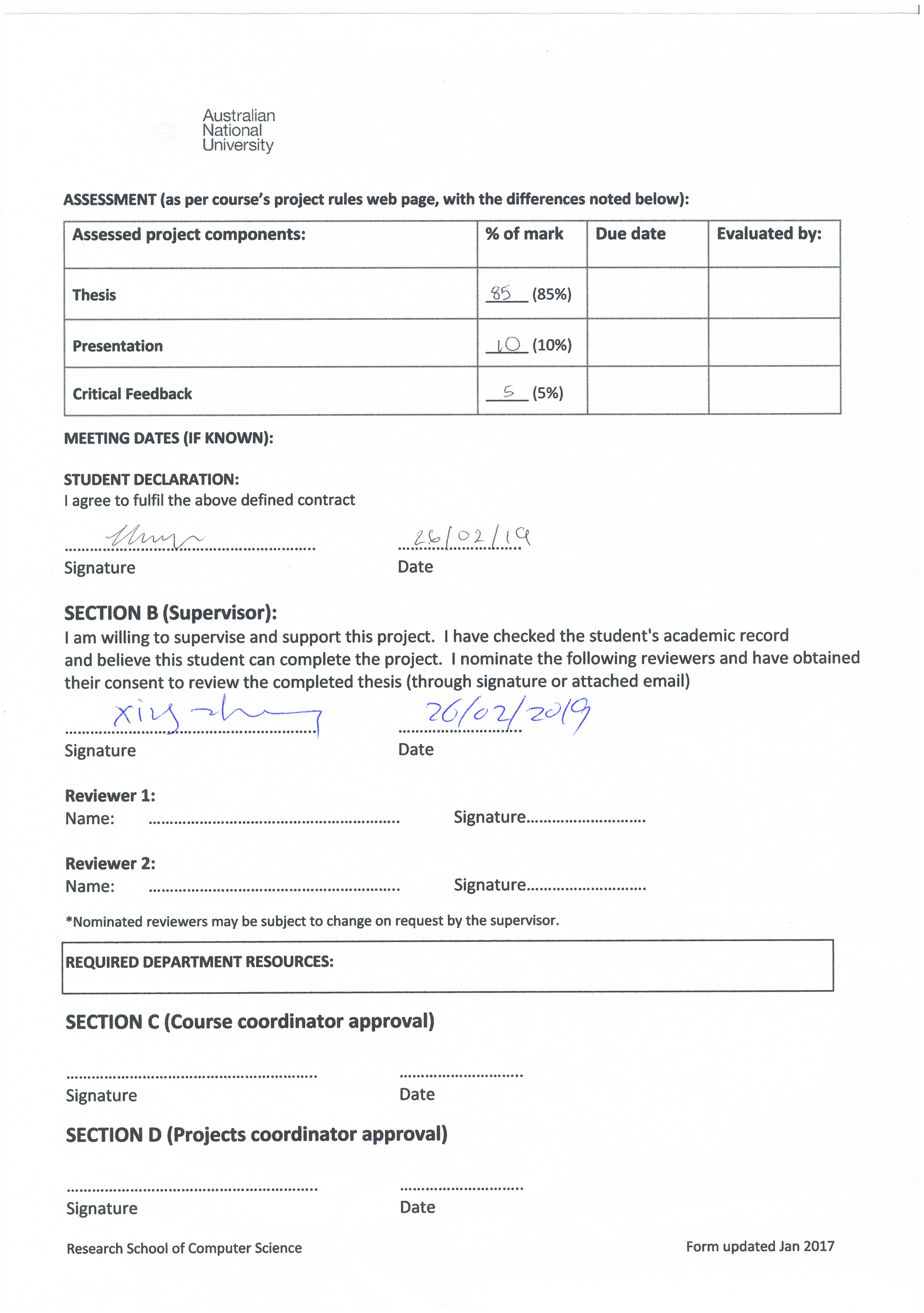
- **Extract and pre-process data for text analysis**
- **Apply natural language processing methods to link formal and informal text**
- **Write a technical report describing techniques and findings of the project**

PROJECT DESCRIPTION:

Technical API documents such as the Android Developers Documentation can have a significant amount of API caveats. Disregard of these caveats can result in numerous software bugs for the developers using the API. Furthermore, some caveats are particularly complex and hard for new developers to understand.
This project attempts to research the effectiveness of using natural language processing methods to locate API misuse in GitHub data and identify the associated API caveat/s. This can be used to support developers in applying correct practices with regards to the API and understanding the caveats involved.

Learning goals and processes of the project include:
1. Investigating and understanding the formats of textual data on GitHub
2. Pre-processing the GitHub data and finding methods to extract key information from that data
3. Designing a text embedding model that can be used to match GitHub data with caveats from technical API documents
4. Analysing the data collected and using statistical methodology to evaluate matching results
5. Writing a project report

Research School of Computer Science Form updated Jan 2017



Australian
National
University

ASSESSMENT (as per course's project rules web page, with the differences noted below):

Assessed project components:	**% of mark**	**Due date**	**Evaluated by:**
Thesis	85 (85%)		
Presentation	10 (10%)		
Critical Feedback	5 (5%)		

MEETING DATES (IF KNOWN):

STUDENT DECLARATION:
I agree to fulfil the above defined contract

[illegible] 26/02/19
Signature Date

SECTION B (Supervisor):
I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project. I nominate the following reviewers and have obtained their consent to review the completed thesis (through signature or attached email)

[illegible] 26/02/2019
Signature Date

Reviewer 1:
Name: .... Signature............

Reviewer 2:
Name: .... Signature............

*Nominated reviewers may be subject to change on request by the supervisor.

REQUIRED DEPARTMENT RESOURCES:

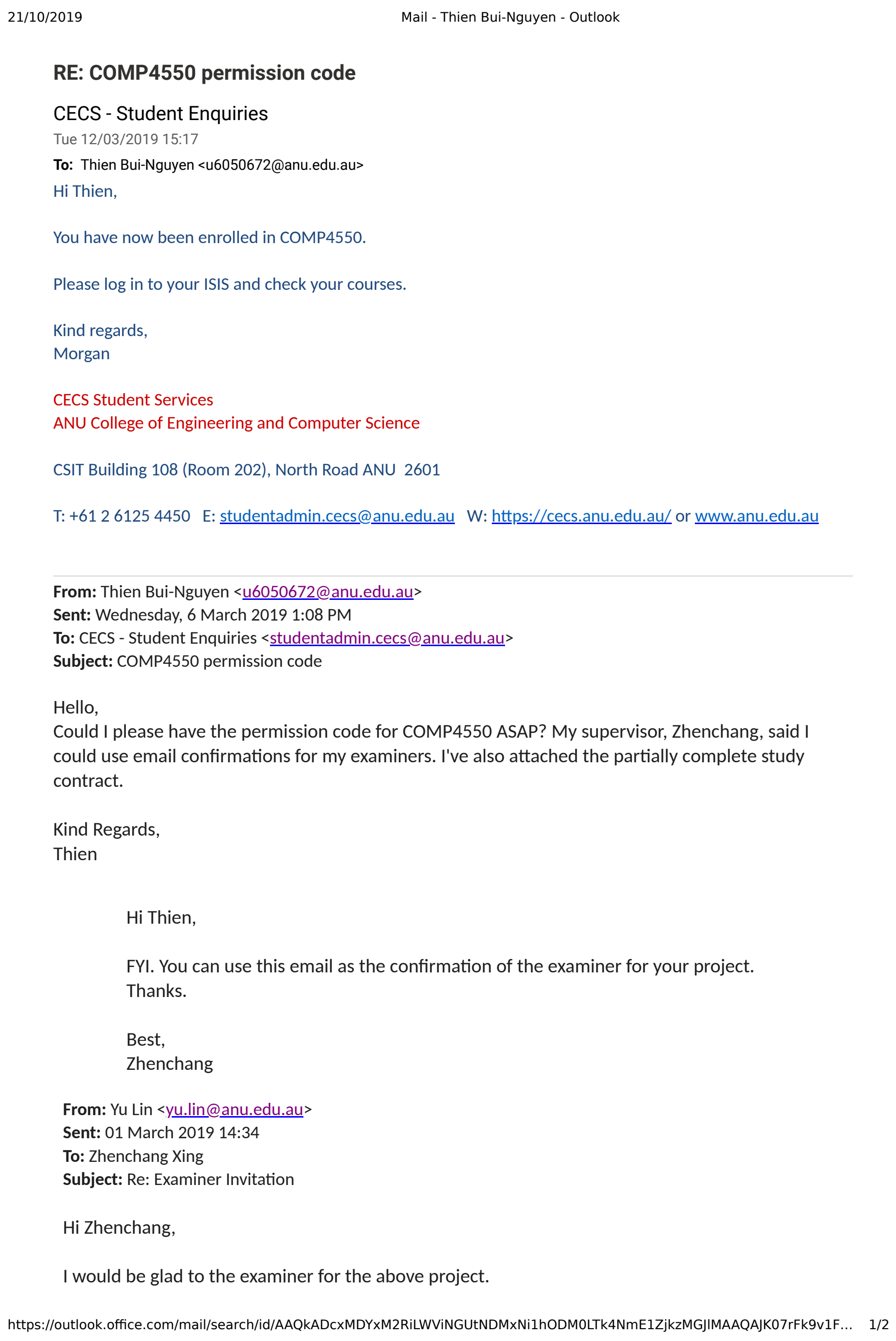
SECTION C (Course coordinator approval)

....
Signature Date

SECTION D (Projects coordinator approval)

....
Signature Date

Research School of Computer Science Form updated Jan 2017



RE: COMP4550 permission code

CECS - Student Enquiries
Tue 12/03/2019 15:17
To: Thien Bui-Nguyen <u6050672@anu.edu.au>

Hi Thien,

You have now been enrolled in COMP4550.

Please log in to your ISIS and check your courses.

Kind regards,
Morgan

CECS Student Services
ANU College of Engineering and Computer Science

CSIT Building 108 (Room 202), North Road ANU 2601

T: +61 2 6125 4450 E: studentadmin.cecs@anu.edu.au W: https://cecs.anu.edu.au/ or www.anu.edu.au

From: Thien Bui-Nguyen <u6050672@anu.edu.au>
Sent: Wednesday, 6 March 2019 1:08 PM
To: CECS - Student Enquiries <studentadmin.cecs@anu.edu.au>
Subject: COMP4550 permission code

Hello,
Could I please have the permission code for COMP4550 ASAP? My supervisor, Zhenchang, said I could use email confirmations for my examiners. I've also attached the partially complete study contract.

Kind Regards,
Thien

Hi Thien,

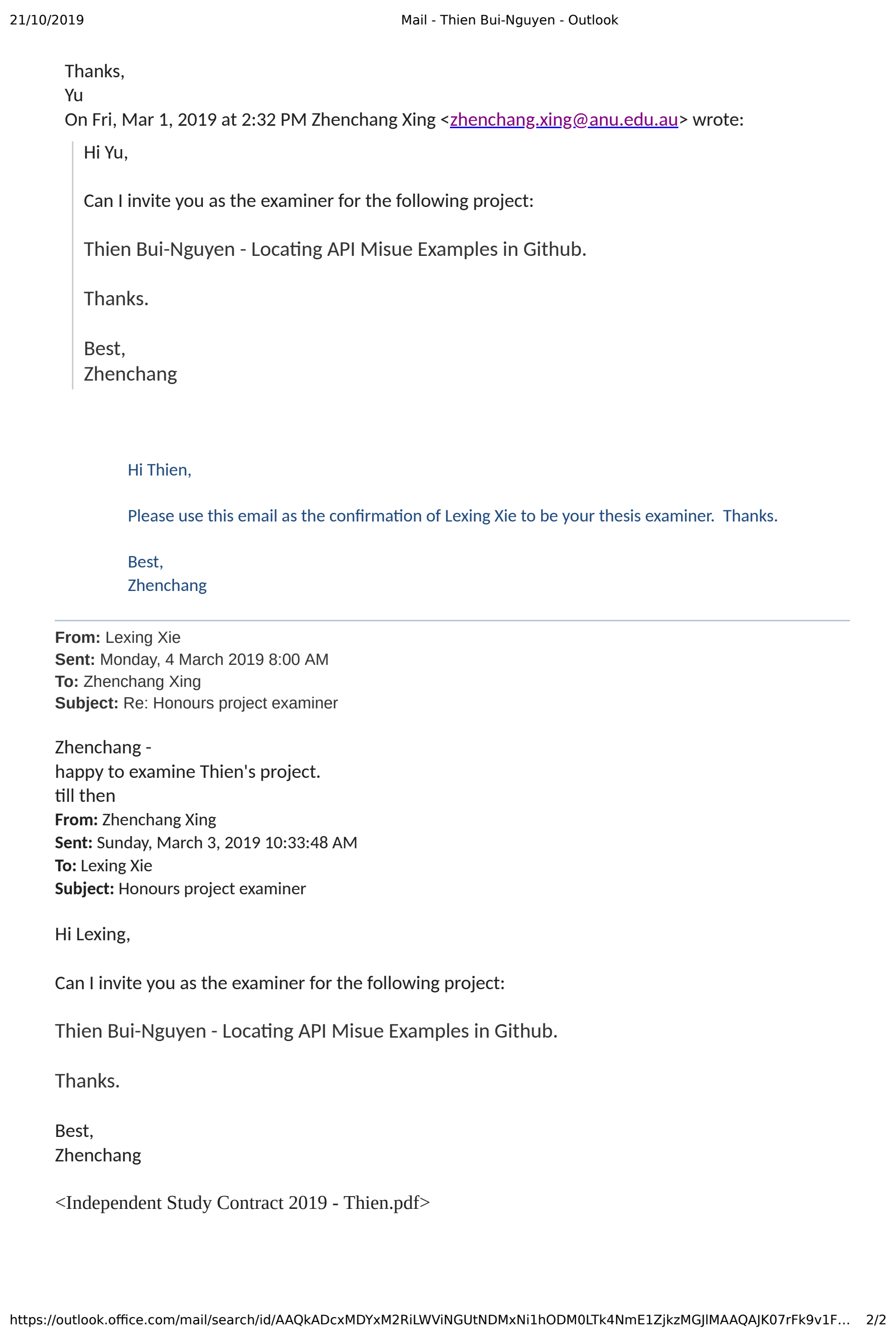
FYI. You can use this email as the confirmation of the examiner for your project.
Thanks.

Best,
Zhenchang

From: Yu Lin <yu.lin@anu.edu.au>
Sent: 01 March 2019 14:34
To: Zhenchang Xing
Subject: Re: Examiner Invitation

Hi Zhenchang,

I would be glad to the examiner for the above project.



21/10/2019 Mail - Thien Bui-Nguyen - Outlook

Thanks,
Yu
On Fri, Mar 1, 2019 at 2:32 PM Zhenchang Xing <zhenchang.xing@anu.edu.au> wrote:

> Hi Yu,
>
> Can I invite you as the examiner for the following project:
>
> Thien Bui-Nguyen - Locating API Misue Examples in Github.
>
> Thanks.
>
> Best,
> Zhenchang

Hi Thien,

Please use this email as the confirmation of Lexing Xie to be your thesis examiner. Thanks.

Best,
Zhenchang

From: Lexing Xie
Sent: Monday, 4 March 2019 8:00 AM
To: Zhenchang Xing
Subject: Re: Honours project examiner

Zhenchang -
happy to examine Thien's project.
till then
From: Zhenchang Xing
Sent: Sunday, March 3, 2019 10:33:48 AM
To: Lexing Xie
Subject: Honours project examiner

Hi Lexing,

Can I invite you as the examiner for the following project:

Thien Bui-Nguyen - Locating API Misue Examples in Github.

Thanks.

Best,
Zhenchang

<Independent Study Contract 2019 - Thien.pdf>

https://outlook.office.com/mail/search/id/AAQkADcxMDYxM2RiLWViNGUtNDMxNi1hODM0LTk4NmE1ZjkzMGJlMAAQAJK07rFk9v1F… 2/2

Bibliography

Adi, Y.; Kermany, E.; Belinkov, Y.; Lavi, O.; and Goldberg, Y., 2016. Fine-grained analysis of sentence embeddings using auxiliary prediction tasks. *arXiv preprint arXiv:1608.04207*, (2016). (cited on page 24)

Baca, D.; Petersen, K.; Carlsson, B.; and Lundberg, L., 2009. Static code analysis to detect software security vulnerabilities-does experience matter? In *2009 International Conference on Availability, Reliability and Security*, 804–810. IEEE. (cited on page 6)

Bae, S.; Cho, H.; Lim, I.; and Ryu, S., 2014. Safewapi: web api misuse detector for web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 507–517. ACM. (cited on page 10)

Bardas, A. G. et al., 2010. Static code analysis. *Journal of Information Systems & Operations Management*, 4, 2 (2010), 99–107. (cited on page 6)

Blasi, A.; Goffi, A.; Kuznetsov, K.; Gorla, A.; Ernst, M. D.; Pezzè, M.; and Castellanos, S. D., 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 242–253. ACM. (cited on pages 9, 31, 36, 37, and 50)

Dryer, M. S., 2005. 81 order of subject, object, and verb. *The world atlas of language structures, ed. by Martin Haspelmath et al*, (2005), 330–333. (cited on page 36)

Havrlant, L. and Kreinovich, V., 2017. A simple probabilistic explanation of term frequency-inverse document frequency (tf-idf) heuristic (and variations motivated by this explanation). *International Journal of General Systems*, 46 (01 2017), 27–36. `doi:10.1080/03081079.2017.1291635`. (cited on page 15)

Huang, Q.; Xia, X.; Xing, Z.; Lo, D.; and Wang, X., 2018. Api method recommendation without worrying about the task-api knowledge gap. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 293–304. ACM. (cited on page 9)

Li, H.; Li, S.; Sun, J.; Xing, Z.; Peng, X.; Liu, M.; and Zhao, X., 2018. Improving api caveats accessibility by mining api caveats knowledge graph. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 183–193. `doi:10.1109/ICSME.2018.00028`. (cited on pages xv, 1, 8, 18, and 20)

Li, S., 2018. Constructing software knowledge graph from software text. (2018). (cited on page 9)

Louridas, P., 2006. Static code analysis. *IEEE Software*, 23, 4 (July 2006), 58–61. `doi:10.1109/MS.2006.114`. (cited on page 39)

Maalej, W. and Robillard, M. P., 2013. Patterns of knowledge in api reference documentation. *IEEE Transactions on Software Engineering*, 39, 9 (2013), 1264–1282. (cited on page 8)

Manning, C. D.; Raghavan, P.; and Schütze, H., 2008. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA. ISBN 0521865719, 9780521865715. (cited on page 15)

Mikolov, T.; Chen, K.; Corrado, G.; and Dean, J., 2013a. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, (2013). (cited on page 15)

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G. S.; and Dean, J., 2013b. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, 3111–3119. (cited on page 15)

Mitchell, D. C., 1994. Sentence parsing. *Handbook of psycholinguistics*, (1994), 375–409. (cited on page 6)

Monperrus, M.; Eichberg, M.; Tekes, E.; and Mezini, M., 2012a. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17, 6 (2012), 703–737. (cited on page 10)

Monperrus, M.; Eichberg, M.; Tekes, E.; and Mezini, M., 2012b. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17 (05 2012). `doi:10.1007/s10664-011-9186-4`. (cited on page 31)

Palangi, H.; Deng, L.; Shen, Y.; Gao, J.; He, X.; Chen, J.; Song, X.; and Ward, R., 2016. Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, 24, 4 (2016), 694–707. (cited on page 5)

Pandita, R.; Xiao, X.; Zhong, H.; Xie, T.; Oney, S.; and Paradkar, A., 2012. Inferring method specifications from natural language api descriptions. In *Proceedings of the 34th International Conference on Software Engineering*, 815–825. IEEE Press. (cited on page 9)

Pérez-Iglesias, J.; Pérez-Agüera, J. R.; Fresno, V.; and Feinstein, Y. Z., 2009. Integrating the probabilistic models bm25/bm25f into lucene. *arXiv preprint arXiv:0911.5046*, (2009). (cited on page 15)

Ratinov, L. and Roth, D., 2009. Design challenges and misconceptions in named entity recognition. In *Proceedings of the Thirteenth Conference on Computational Natural Language Learning (CoNLL-2009)*, 147–155. Association for Computational Linguistics, Boulder, Colorado. `https://www.aclweb.org/anthology/W09-1119`. (cited on page 6)

Ren, X.; Xing, Z.; X; Xia; Lo, D.; Grundy, J.; and Sun, J., 2018. Trustdoc: Documents speak louder than votes. (2018). (cited on pages 6, 9, 13, 25, and 29)

Robertson, S., 2004. Understanding inverse document frequency: on theoretical arguments for idf. *Journal of documentation*, 60, 5 (2004), 503–520. (cited on page 14)

Singh, R. and Mangat, N. S., 2013. *Elements of survey sampling*, vol. 15. Springer Science & Business Media. (cited on page 25)

Subramanian, S.; Inozemtseva, L.; and Holmes, R., 2014. Live api documentation. In *Proceedings of the 36th International Conference on Software Engineering*, 643–652. ACM. (cited on page 10)

Sun, J., 2018. Augment api caveats with erroneous code examples. (2018). (cited on pages 1, 5, 6, 9, 11, 13, 14, 21, 28, 29, and 50)

Van Nguyen, T.; Nguyen, A. T.; Phan, H. D.; Nguyen, T. D.; and Nguyen, T. N., 2017. Combining word2vec with revised vector space model for better code retrieval. In *Proceedings of the 39th International Conference on Software Engineering Companion*, 183–185. IEEE Press. (cited on page 9)

Wen, M.; Liu, Y.; Wu, R.; Xie, X.; Cheung, S.-C.; and Su, Z., 2019. Exposing library api misuses via mutation analysis. In *Proceedings of the 41st International Conference on Software Engineering*, 866–877. IEEE Press. (cited on pages 10, 50, and 51)

Zhang, T.; Upadhyaya, G.; Reinhardt, A.; Rajan, H.; and Kim, M., 2018. Are code examples on an online q&a forum reliable?: a study of api misuse on stack overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 886–896. IEEE. (cited on pages 10, 32, 34, 35, and 50)

Zhong, H. and Su, Z., 2013. Detecting api documentation errors. In *ACM SIGPLAN Notices*, vol. 48, 803–816. ACM. (cited on page 10)

Zhou, Y.; Gu, R.; Chen, T.; Huang, Z.; Panichella, S.; and Gall, H., 2017. Analyzing apis documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17 (Buenos Aires, Argentina, 2017), 27–37. IEEE Press, Piscataway, NJ, USA. `doi:10.1109/ICSE.2017.11`. `https://doi.org/10.1109/ICSE.2017.11`. (cited on pages xiii, xv, 10, 31, 32, 33, 35, 36, 37, 39, 50, 52, and 53)

Zitser, M.; Lippmann, R.; and Leek, T., 2004. Testing static analysis tools using exploitable buffer overflows from open source code. In *ACM SIGSOFT Software Engineering Notes*, vol. 29, 97–106. ACM. (cited on page 6)