6CCS3PRJ

Git Commit Message Analysis using Natural Language Processing

Final Project Report

Author: Mateusz Przewlocki

Supervisor: Dr Jeroen Keppens

Programme Title: BSc Computer Science

Student Number: 1609577

Originality Avowal

Abstract

Git commit messages form a log of changes made within a project that uses Git for version control. Often, these messages do not follow guidelines, rendering logs of changes unreadable. Guidelines that have been written are not usually enforced and enforcing them automatically is not necessarily a trivial task. Through the use of natural language processing, this project aims to produce a way for software developers to check that their commit messages follow established convention, and to produce suggestions that could be used for messages that do not.

Acknowledgements

I would like to thank my supervisor, Dr Jeroen Keppens, for offering his time, support and expertise while I developed this project. In addition, I would like to thank my friends and family for supporting me through the course of this project.

Contents

Chapter 1 – Introduction

1.1 Motivation

Git is a system used for version control in software engineering projects. A Stack Overflow survey from 2015 [18] revealed that 69.3% of the 16,694 respondents used Git for version control, showing that Git is one of the most popular version control systems used today and that Git is an industry standard. One of the fundamentals of Git are commits, which are accompanied by messages that describe what has been changed in that commit. Guidelines have been written for these Git commit messages [5] although these are not always enforced, leading to inconsistent commit histories. A significant problem with this, as Aviv Ben-Yosef writes [1], is that when developers are added to a project, they will not be able to easily understand commits from the past, which will hinder development work. Another problem Ben-Yosef describes is that even those developers who were involved with the repository from the start might forget what a commit changed, and poor commit messages will not help in this. In contrast, good Git commit messages are useful to find which specific commit may have led to a feature being broken, when making changes to code and when using Git commands such as bisect.

Natural language processing deals with the language humans use every day, which is developed naturally, as opposed to formal languages that computers can deal with more easily. Although natural language is ambiguous and the rules are flexible, several techniques have been developed for natural language processing, most of which lend themselves to machine learning due to their tendency to be very mathematical. For example, there are distributional, frame-based and model-theoretical approaches to natural language processing, each dealing with natural language differently.

The motivation of this project is to, therefore, create a way to enforce appropriate guidelines for Git commit messages, which should allow developers to produce readable logs of changes that are easily understandable by anyone who should read them. Not only that, this project should also make developers consider how they write commit messages more seriously.

1.2 Aims and Objectives

The aim of this project is to learn about and utilise natural language processing methods, in conjunction with pre-programmed Git commit message guidelines, to attempt to fix poorly written Git commit messages such that they follow the guidelines. This should allow for a wide range of Git commit messages to be analysed and corrected in a relatively short period of time compared to manual analysis. Common problems with commit messages include that they are vague, that the wrong tense is used or even that the grammar is wrong. The system produced will attempt to solve these problems firstly by detecting them and showing the user what is wrong with the message, and then by attempting to automatically correct the mistake.

The objectives of this project were:

- To learn about the Git commit message guidelines,
- To learn about natural language processing and the different approaches to it,
- To identify and use a suitable natural language processing library,
- To produce an application which can identify and solve common problems with commit messages, using the Git commit message guidelines, and
- To evaluate this application against manual analysis of commit messages.

This has been achieved by creating an application which can obtain a log of commit messages from a Git repository, analysing them against established guidelines, and producing suggestions on how to change each message to make it follow the guidelines. The resulting application is then evaluated by comparing it to manual analysis of a data set of selected commit messages.

There is, however, the issue of changing the history of the repository to be considered while undertaking this project, as changing the history of the repository forces anyone who has cloned a repository to manually fix their local history, and as such, is strongly discouraged. [2]

Chapter 2 – Background

2.1 Git commit messages

2.1.1 Message guidelines

Git commit messages form a log of changes to a Git repository. As such, messages should follow guidelines for readability [3] as well as to make collaboration with others in the same repository easier [6]. Good commit messages should aim to tell the reader what has been done and why it has been done [4], as opposed to how it has been done. Code is self-explanatory and in-line comments are often enough to explain how something has been done.

The guidelines written by Tim Pope [5] are frequently cited by various sources on the matter of writing good commit messages. Most of these guidelines involve technical requirements that could be detected and fixed trivially. For example, Pope notes that the first (or subject) line of messages should not exceed 50 characters, and the more detailed text that follows it should be wrapped to about 72 characters per line. Further, the subject line of a commit message should not end with a full stop, and the first letter should be capitalised, as it is a title, not a sentence. [3]

A non-trivial guideline is that the present imperative be used in the subject line. The reason for this is that Git uses the present imperative by default, as with the merge and revert operations. [3] Subject lines should be written as if they were commands to be done in the present, not as indications of what has been done. Pronouns should also be avoided. Messages such as “bug fixes” or “I fixed a bug” are not acceptable, however messages such as “fix bug” are.

Another non-trivial guideline is that messages should aim to tell the reader what has been done and why it has been done, as previously mentioned. A third guideline is that a motivation for the change that the commit applies should be included, and the change should be contrasted to previous behaviour within the body of the commit [6]. In other words, commit messages should not be vague and be precise in informing users who read the commit history of what has been accomplished.

2.1.2 Commit message research

To better understand the problem of commit messages not following guidelines, a set of 201 anonymised Git commit messages from public repositories has been collated, and can be found in Appendix B. This set will be used for the evaluation of the system produced by this project.

The messages in this set come from a wide variety of Git repositories to obtain better representation. Most of these repositories were found on GitHub's "explore" page[1]. The first ten commit messages at the time of access of the repository were selected and placed in the data set. Messages generated automatically, such as "Update README.md" or "Merge branch", as well as duplicates of the same message, were ignored.

An observation that has been made while collating this data set is that most messages are limited to the subject line. As such, to better facilitate analysis of the data set, the data set is also limited to only the subject lines of commit messages. However, the tool produced in this project should be capable of analysing not only the subject line, but also the body of the commit message, which has different guidelines to the subject line. Another observation is that not all commit messages are in English, which might limit the utility of the application produced in this project somewhat.

One observation about the data set are that in most cases, the present imperative is not used in the subject line. Instead, most often some variant of the past tense is used. Another is that grammar mistakes are also somewhat common, and while a bit of a nit-pick, these can make a log of commits look less presentable and less readable. A third observation is that some commit messages are overly vague, sometimes being only one or two words. Some are limited to being as vague as "fix typo" or a version number, or even a string consisting only of the letter "e." This is not helpful in situations like those described in Chapter 1.1.

For numerical analysis of the data set, each message was manually evaluated against the guidelines described in 2.1.1. Guideline violations that messages were analysed against were:

[1] Explore · GitHub - https://github.com/explore

- Wrong tense – any other tense than the present imperative was used.
- No verb – there is no verb such as “add”, “fix” or “create” when mentioning a feature or bug.
- Vague – the message was too short to meaningfully describe what was achieved.
- Grammar mistakes – the subject line did not start with a capital letter, ended with a full stop or had any other grammatical mistake.
- Excessive length – the subject line was wrapped by GitHub with an ellipsis (…).

The findings of this analysis have been placed in Table 2.1.2.1.

Guideline violation	**No. of messages**	**% of messages in data set**
Wrong tense	44	22%
No verb	29	14.5%
Vague	55	27.5%
Grammar mistakes	80	40%
Excessive length	4	2%

Table 2.1.2.1 – a numerical analysis of the data set.

From this analysis, by far the most common mistakes were grammar mistakes – these were made 40% of the time in the messages in the data set. Vagueness and incorrect tense were also prominent, followed by lack of a verb. This is concerning as it goes to show how many commit messages do not adequately describe what has been changed, and how many superfluously add suffixes to verbs when it is not necessary. Only four messages in the whole data set were of excessive length, meaning that this is not the greatest concern, even if it is a trivial fix.

While this analysis may be flawed as it was manual, it should give adequate insight into which guideline violations are most common, and as such, which are of greater priority for the solution to deal with. From this analysis, and the collation of the data set, not all messages will be able to be automatically fixed, however there are many that could easily be fixed with only one or two changes, easily improving the readability of a Git commit history.

2.2 Features of natural language

Natural language is defined as "*language that has developed in the usual way as a method of communicating between people, rather than language that has been created, for example for computers.*" [19]. As opposed to formal languages, such as programming languages, natural languages evolved naturally. Examples of natural languages include English, Spanish and French. [20]

Natural and formal languages have many features in common, such as tokens – basic elements of the language, structure – the way the tokens are arranged, parsing – the process of figuring out the structure of the sentence and semantics – the meaning of the sentence. [20] The table below describes these in relation to commit messages.

Tokens	e.g. add, remove, create, modify, fix, bug, comment, feature
Structure	Verb, imperative, followed by a noun and an optional sequence of nouns and verbs e.g. "Add feature to allow comments"
Semantics	In the example above, a feature was added to a program that allows users to create, post and presumably delete comments.

Table 2.2.1 – features of a natural language in relation to commit messages

However, in contrast to formal languages, natural languages are full of ambiguity and redundancy. [20] In a natural language, there are multiple ways of representing the same information which is a result of the ambiguity of natural languages. For example, a commit message could say "create feature" which would mean the same thing as "add feature." This is one of the reasons why processing natural languages programmatically is difficult – meanings of words may not necessarily be "hardcoded" into the program. The context of the sentence is important.

Another reason why natural language processing is difficult is because natural languages are full of idiom and metaphor [20], as opposed to formal languages. For example, sentences such as "this program is on fire" may pose trouble for a conventional program built to process natural language as it may infer that the program is literally on fire, when the meaning of the sentence is different. In the context of Git, as discussed earlier, commit messages tend to only literally describe what the commit accomplished, so this is not necessarily a problem, although there may be messages that are exceptions to this.

2.3 Approaches to natural language processing

Natural language processing is defined as "*the application of computational techniques to the analysis and synthesis of natural language and speech.*" [7]

The approaches to natural language processing can be broken down into four distinct categories: distributional, frame-based, model-theoretical and interactive learning. [8] Each of these has its advantages and disadvantages, and as a result, situations that they are most suited to. In the context of analysing commit messages, it would be preferable to use an approach that is broad (to consider the variety of projects that use Git) and somewhat specialised (just to the general context of a commit message), as well as one that does not require much, if any, human supervision.

There are several types of subtasks involved in natural language processing. Five types identified by Matthew Mayo [17] are: text classification tasks (for categorising words or sentiment analysis), word sequence tasks (for predicting next or previous words and text generation), text meaning tasks (for representing meaning of words), sequence to sequence tasks (which find and use hidden representations of texts) and dialogue systems (which have two main categories: goal-oriented dialogue and conversational dialogue). For commit messages, the first three types are the most relevant, as messages are not particularly involved with dialogues or hidden meanings.

2.3.1 Distributional approaches

The idea behind distributional approaches to natural language processing is the distributional hypothesis: "linguistic items with similar distributions have similar meanings." [11] What this means is that words used in similar contexts will usually have the same meaning, so distributional approaches rely on the relationships between words rather than their meaning, and often use mathematical analysis as well. Usually they involve forming tables or matrices of words and contexts and then using mathematical reductions on those matrices [9]. In other words, distributional approaches use statistical analysis to determine the meaning of a word. Figure 2.3.1.1 shows an example visualisation of how such an approach may derive the meaning of a word, as well the type of the word, by its relationship with other words.

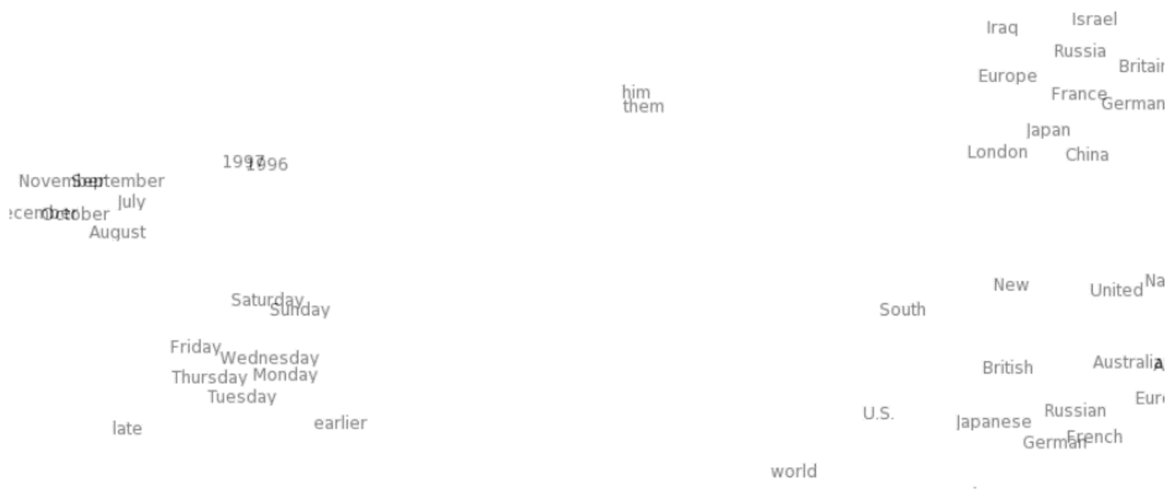


Figure 2.3.1.1 – example visualisation of an analysis produced through a distributional approach [9]

These approaches lend themselves to machine learning due to them heavily relying on mathematical, statistical analysis, though as a result, they require training or access to premade machine learning models. This reduces the supervision needed, although may not be as accurate as a more supervised approach. Distributional approaches “perform quite well at tasks such as part-of-speech tagging […], dependency parsing […], and semantic relatedness.” [8] This might be useful for identifying parts of commit messages – verbs, tense of verbs, features (through nouns) and reasons for the commit, which can then be used to make suggestions to change these messages. An example of how this may be useful is illustrated in Figure 2.3.1.2.

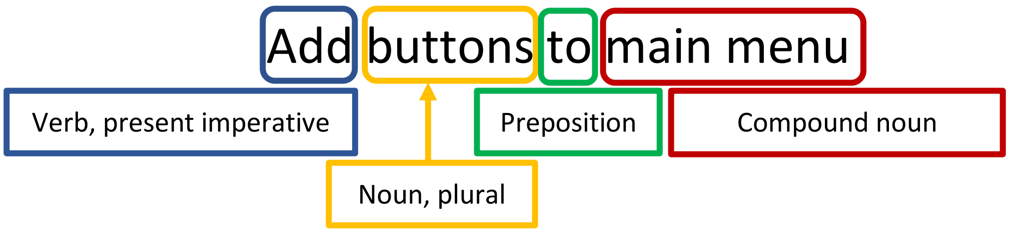


Figure 2.3.1.2 – an example of how distributional approaches may be used for part-of-speech tagging

However, because of the idea that words used in similar contexts have similar meanings, this can also result in words and their antonyms being considered similar, for example. [9] Still, there is "no such thing as pure unsupervised learning" [9], so it is important to consider the context of language while analysing it.

Distributional approaches achieve breadth, flexibility and scalability at the cost of depth. [8] While they can be used for a wide range of situations, they generally cannot answer more complex questions, and they do not have a true understanding of real-world semantics. This limits their utility somewhat for larger texts, however in the context of analysing Git commit messages, a distributional approach may be suitable given the simplicity and short length of most messages.

2.3.2 Frame-based approaches

"A frame is a data-structure for representing a stereotyped situation," writes Marvin Minsky. [10] These contain information about a certain context and may be used for a variety of situations which are not necessarily limited to natural language processing. Minsky elaborates that a frame can be thought of as "a network of nodes and relations." There are "top levels" which represent invariants, terminals or "slots" which must be filled in by specific data, and there are also conditions that may be specified for the data assigned to the terminals, such as its type. A "matching process" will then try to assign values to each slot of data which are consistent with the structure of the frame. [10]

A frame for commit messages might look like the following diagram:

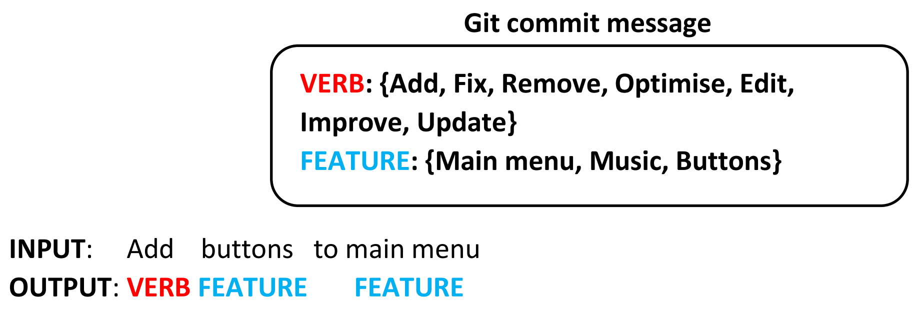

Git commit message
VERB: {Add, Fix, Remove, Optimise, Edit, Improve, Update}
FEATURE: {Main menu, Music, Buttons}
INPUT: Add buttons to main menu
OUTPUT: VERB FEATURE FEATURE

Figure 2.3.2.1 – A frame for Git commit messages

A disadvantage of frames is that they require supervision [8] – they cannot be automatically generated but must be created by humans who have enough understanding of the context to make it viable. They are also incomplete and cannot be used to analyse sentences outside of the scope of the frame. This makes it potentially unviable to create a solution to the problem of analysing Git commit messages using only a frame-based approach.

While Git commit messages are a specific context, what they deal with varies broadly depending on the project. So, while frame-based approaches work well with specific contexts, the need to construct and update frames to account for new information might be too cumbersome to consider using such an approach on its own. However, using a more generalised frame along with other approaches may prove viable. This frame might only have the invariants of there needing to be a noun and a verb, with the message having to also start with a verb but otherwise the structure does not matter, as long as it is grammatically correct.

2.3.3 Model-theoretical approaches

Model theory is "the study of the interpretation of any language […] by means of set-theoretic structures[.]" [12] It deals with adding information to incomplete sentences such that the sentence is made true – the added information is called an interpretation of the sentence. This combined with the concept of compositionality – that determining what parts of a sentence mean can be used to determine the meaning of the whole sentence [13] – can be combined to produce another approach to natural language processing.

Model-theoretical approaches are compared to turning language into computer programs. [8] Before answering a given query, the query must first be broken down to each individual concept, then recombined to deduce the meaning and then the answer, which should satisfy the query. For example, the query "what is the largest city in Europe" must first be broken down into what the measure of largeness is (e.g. population, area), what a city is, what the cities in Europe are and then which one of those cities has the largest value for the chosen measure of largeness. The latter parts would be determined by sorting through the appropriate data. Figures 2.3.3.1 and 2.3.3.2 demonstrate this concept more clearly.

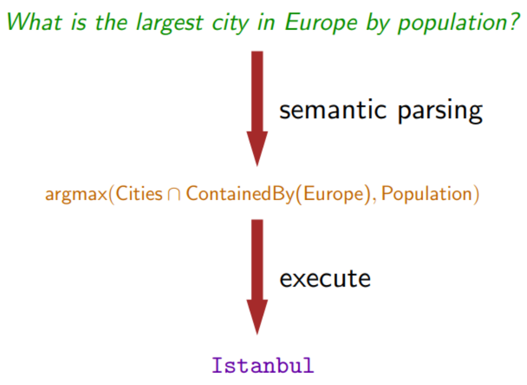

What is the largest city in Europe by population?
semantic parsing
argmax(Cities ∩ ContainedBy(Europe), Population)
execute
Istanbul

Figure 2.3.3.1 – how a model-theoretical approach might parse the query "What is the largest city in Europe by population?" to reach the answer of Istanbul *[9]*

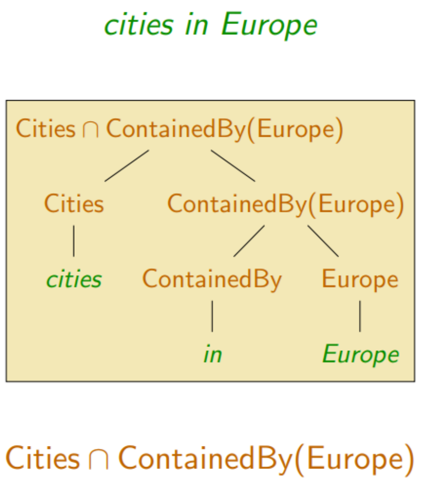


Figure 2.3.3.2 – compositional semantics of the phrase "cities in Europe" [9]

Advantages of model-theoretical approaches are full-world representation and rich semantics, which enable them to answer more complex questions. These advantages come at a cost of a need for supervision, which is labour intensive, and narrow scope. Model-theoretical approaches have potential to be both broad and deep, however in practice a trade-off needs to be made. [8]

Also, model-theoretical approaches appear to mostly deal with queries, which is not necessary for this project as commit messages are not queries, but statements of what has been changed. However, such approaches (or similar) may be used to extract contextual information about a commit, not just from the message itself but from the changes that accompany it, such as information on what really has been changed, to allow suggestions to be made to “vague” commits to make them more informative.

2.3.4 Interactive learning approaches

Interactive learning approaches to natural language processing intend to have breadth and depth by having humans teach computers gradually, without explicitly building any models of the world. [8] They are based on the idea that “language derives its meaning from use,” [9] as opposed to more theoretical approaches. Nadav Lidor and Sida I. Wang write in a research blog of the Stanford Natural Language Processing Group that allowing users to interact with computers with natural language might be key to having more natural and usable natural language interfaces, while most such interfaces today are trained only once, resulting in limitations on how useful they are. [14]

Exemplar attempts towards interactive learning approaches have been produced, one such attempt being SHRDLURN[2]. The objective is to teach a computer a language to facilitate moving of virtual blocks from a start state to a goal state.

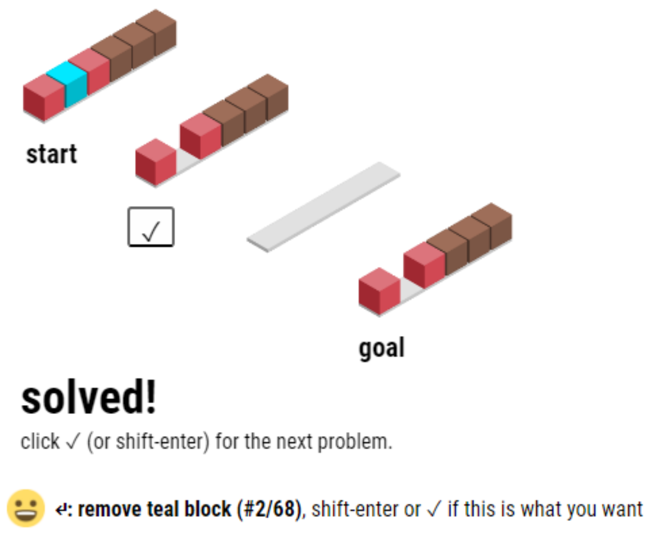

start
goal
solved!
click ✓ (or shift-enter) for the next problem.
↵: remove teal block (#2/68), shift-enter or ✓ if this is what you want

Figure 2.3.4.1 – SHRDLURN, an interactive learning program, learning to perform tasks through natural language

[2] teach a computer a language - http://shrdlurn.sidaw.xyz/acl16/

As noted in the research blog [14], pilot users of SHRDLURN used various languages, from English, Arabic and Polish to custom programming languages, showing both the breadth and depth which interactive learning may accomplish, but also how convoluted such approaches might be with many users. Consistency on the users' part is also necessary to allow computers to learn most efficiently. [8]

While interactive learning approaches may be useful for teaching computers how to perform specialised tasks, an obvious disadvantage of these approaches is that they would require far more supervision than even model-theoretical approaches and may not be suitable for all contexts. Pilots of such approaches have been described as "unnatural and un-scalable for large action spaces," and good coverage might not be achieved if every individual interacting with a computer had to teach it from scratch. [14]

Interactive learning approaches, therefore, might not be useful for a project like this. Using interactive learning would necessitate first creating an interface through which a computer is taught what a good commit message looks like, creating appropriate models of the world, getting enough people to teach the computer and then interpreting the results. This is far too cumbersome given the time scale of this project, and the vast availability of Git commit messages both good and bad, which could be used to train a machine learning model that uses a distributional or model-theoretical approach instead, or even an entirely different approach not using the established methods for natural language processing.

2.3.5 Existing tools for natural language processing

There is already a wide variety of available tools for natural language processing, most of which utilise machine learning and distributive approaches in some way to do this.

One of these is the Stanford CoreNLP toolkit. [15] The toolkit claims to provide most of the common core natural language processing operations and claims to be one of the most used toolkits for this purpose. The advantages of the toolkit, as described in the corresponding paper, is that the components are stable and robust, while also considering that most people who use it "are unlikely to be an NLP expert, and are hence looking for NLP components that just work." The CoreNLP toolkit also provides a simplified version of the API, which will enable easier implementation. However, the machine learning models of the toolkit must first be downloaded before the toolkit itself can be used.

Another toolkit for natural language processing is the Apache OpenNLP toolkit. [16] This toolkit also claims to support the most common natural language processing operations. It is machine learning-based and provides a great deal of functions that allow the training of machine learning models on given data. However, as with the CoreNLP toolkit, it appears that the machine learning models must first be downloaded, and the toolkit does not appear to be usable in any language other than Java which, while not a problem for this project, may limit its utility in other projects.

Both the CoreNLP and OpenNLP toolkits are general-purpose and appear to use distributional approaches to natural language processing, as they both use machine learning models, which distributional approaches lend themselves to. These toolkits are able to identify parts-of-speech, which is one of the most important tasks where Git commit message analysis is concerned.

Aside from the Stanford CoreNLP toolkit and Apache OpenNLP toolkit, there are other tools available for natural language processing. However, they are either specialist or can only support one programming language. The Stanford CoreNLP toolkit appears to be a better fit for this project, as given the short length of most commit messages, it is unlikely that most features of the Apache OpenNLP toolkit will be used.

Apart from NLP toolkits, natural language processing is already used in one way or another in a wide range of applications. For example, spell checkers, spam filters and smart home solutions such as Amazon Echo or Google Assistant use natural language processing in some form. [22] This shows that natural language processing can solve a wide range of problems through the use of a variety of approaches. Presumably, a spell checker or a spam filter would use a distributional approach to detect common key terms and their context, while a solution such as Amazon Echo or Google Assistant might use a model-theoretical approach to deal with queries a user may have or tasks a user may want them to perform.

Chapter 3 – Specification

3.1 Problem description

The main objective of this project is to create an application which can obtain commit message histories from Git, analyse them against the existing guidelines and produce suggestions on how to change each message to make them follow the guidelines if they are not already. For this project, although there are many different websites that host Git repositories, the focus will be on GitHub.

As stated earlier, the problem that the application aims to solve is the problem of poorly written Git commit messages. In particular, the main issues that Git commit messages have are:

- The header of the message does not start with a verb,
- The verbs in the header of the message are not in the present imperative, where appropriate,
- The message is vague and does not describe what change has been made sufficiently,
- The header is too long and gets cut off while reading the commit message history,
- The header has punctuation at the end, although it shouldn't as it is a title.

The application should focus on solving these problems through a distributional approach to natural language processing, as that is particularly useful for part-of-speech tagging and getting the lemma – base form, which is usually in the present imperative – of a word. These are key tasks in identifying verbs and their tense, as well as identifying any punctuation marks without needing an exhaustive list of every verb or punctuation mark to be hard-coded into the program. Particularly for identifying whether a message is vague, it is also important to identify whether the message has any verbs and any nouns at all, as well as whether the nouns themselves give enough information about what the commit achieved.

A user of the application should be able to know when their commit messages have violated the message guidelines, and then have a suggestion on how to improve that commit message, for future reference when they write new messages.

3.2 Requirements

The requirements to create this application can be split into user, functional and non-functional requirements.

3.2.1 User requirements

The user requirements for this application are as follows:

- The user should be able to type in a single commit message for analysis.
- The user should be able to link a GitHub repository for analysis of all commit messages, or only a range of commit messages.
- The user should be able to extract meaningful information from the analysis of the messages.
- The user should know when their messages follow guidelines and when they do not.
- The user should know which changes to make if their messages do not follow guidelines.

3.2.2 Functional requirements

The application should be able to:

- Provide an interface for the user to analyse commit messages.
- Provide an interface for the user to link a GitHub repository and select a range of commit messages to analyse.
- Connect to a GitHub repository to extract commit messages from.
- Perform analysis of messages using the Stanford CoreNLP toolkit.
- Detect when messages violate the commit message guidelines.
- Produce suggestions on how to alter the message to make it follow guidelines.
- Notify the user of these suggestions, as well as notifying them of which messages do follow guidelines.

Time permitting, the application could also:

- Use contextual information from the GitHub repository to see if a commit message matches what has been changed or provide a more detailed commit message in case of vague messages.

- Notify the user of any duplicate or near-duplicate messages.
- Produce statistics on which guidelines are being violated, and how many commit messages violate them, from the analyses of the messages.

3.2.3 Non-functional requirements

The application should be:

- Usable – the application should be easy to use and understand, and it should be usable on multiple operating systems. The application should not produce any runtime errors.
- Secure – the application should not expose any part of the user's system, or any repositories that the user might use the application for, to any threats. This is an especially important consideration as a user needs to know that an application is secure in order to trust it.
- Efficient – the application should be capable of analysing several messages in the order of seconds, so that the user can get feedback as soon as possible.
- Maintainable – it should be relatively easy to implement new features and improve and fix existing features.
- Extendable – the application should leave room to be adapted to a wide range of situations.

Chapter 4 – Design

This chapter describes the design of the application.

4.1 General description

The application will be written in Java, to allow it to run on multiple operating systems and to allow it to be reused. It will be built using Gradle. The application will allow the user to enter singular commit messages for analysis and provide appropriate suggestions for these messages. The application will allow users to link a GitHub repository and select a range of commits to analyse. The application will also users to view suggestions in form of a report, showing each message they entered, which guidelines were violated and suggestions to change the messages to make them more appropriate. This report will be produced in the form of several windows, to display a list of commit messages as well as a list of guideline violations for each commit message.

The application will be designed particularly with efficiency and maintainability in mind, to enable the application to be extended and improved for further use.

4.2 Use cases

The general idea of the application is described by the following use case diagram.

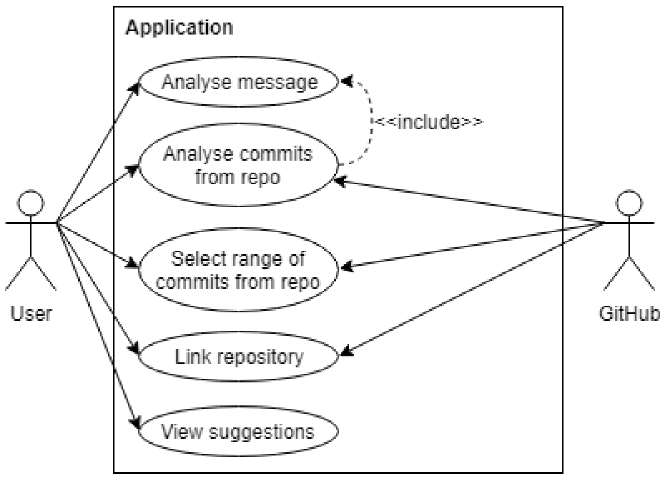


Figure 4.2.1 – a use case diagram of the proposed solution

Initially, the user will be able to either enter a message for analysis or link a GitHub repository for analysis of the commits in that repository. A repository must be linked to allow a range of commits to be chosen and analysed. The application will have to interface with GitHub to access the repository, the range from which commits can be chosen and the commits (and their respective messages) themselves. Analysing the commit messages from the repository is essentially a repetition of the singular "analyse message" use case. The user will then be able to view suggestions on the message(s) that they have allowed the system to analyse.

4.3 System architecture

The architecture of the system is described by the following three-layer system architecture diagram, with the direction of the arrows representing data flow.

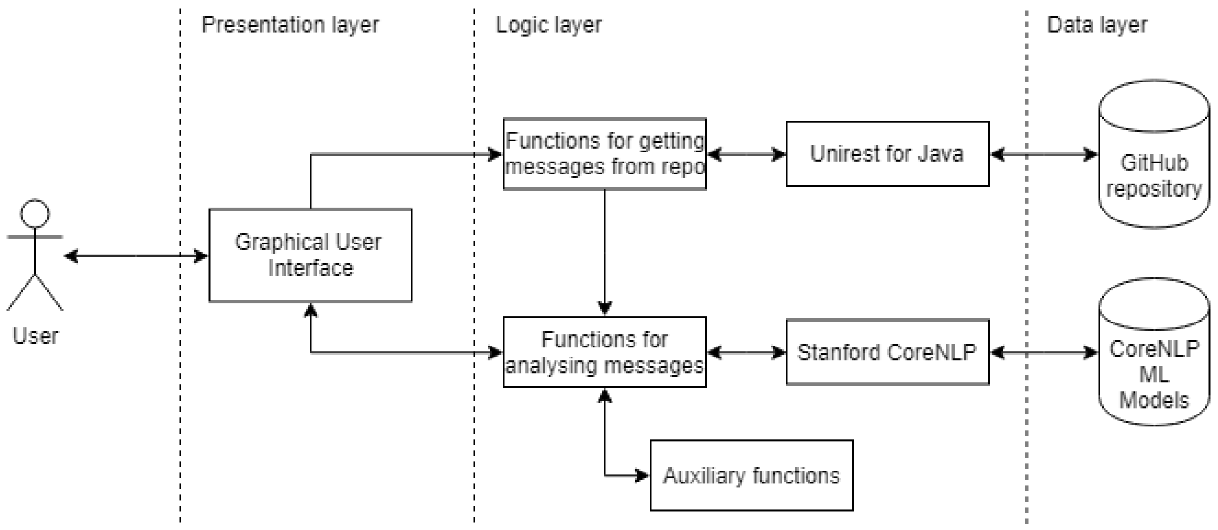


Figure 4.3.1 – a system architecture diagram of the proposed solution

The user will access the system through a graphical user interface. One of the inputs the user can make will be a single commit message, which will need to be processed by the functions and algorithms that analyse the message. These functions will make use of the Stanford CoreNLP library to tokenise the message (that is, split the message into parts), get the part-of-speech tags of the message and to get the base form of verbs in the message.

A second input the user can make is the web address of a GitHub repository. This will eventually lead to the user being shown a list of commit messages and their respective analyses. The application will make use of the Unirest for Java library to send a HTTP request to the GitHub REST API to obtain a list of commits and the data associated with them. This list of commits will then be processed into a list that is readable by the program, and each message on the list of commits will go through the same process as for a single commit message.

4.4 External libraries and tools

The external libraries and tools used for this application will be:

- Stanford CoreNLP – the main input to this API will be a sentence (a commit message) and the main output will be the part-of-speech tags associated with each word in the message. For example, verbs in the message might receive the tag "VB" which would then allow other parts of the program to identify them as a verb. This will also be useful in identifying other parts of the commit message, and writing algorithms based on where these parts of the message are. Another input to this API will be a verb, and the output will be its base form, which is usually in the present imperative and thus should allow solving the problem of incorrect tense of a message.
- Unirest for Java[3] – this library will be used for making HTTP requests to the GitHub REST API, to make code more readable and understandable.
- GitHub REST API – this API will be used for accessing the commits of a public repository. The input is the address of the repository and the output is a list of commits with their associated data in JSON format.
- Java Swing will be used for the graphical user interface.
- JUnit will be used for unit testing the application.

[3] Unirest for Java – Simplified, lightweight HTTP Request Library - http://unirest.io/java.html

4.5 Graphical user interface

The application will include a graphical user interface for ease of use. This will be made using Java Swing.

4.5.1 Main view

The main view of the application will be simple. It will allow the user to input a single commit message for analysis and then press the “analyse” button to get the feedback for that commit message, through a feedback view. It will also allow the user to input the URL of a GitHub repository in a specific format and then press the “analyse repository” button to get feedback on all commit messages in that repository, through a list showing each commit message. It should allow the user to select a range of commits if they do choose to analyse a repository.

4.5.2 Feedback view

The feedback view will need to allow a user to see the original commit message that was analysed. There will be a list showing how the commit message guidelines have been met or violated, through messages such as “the length of the message may be too long” or “the verbs must be in the present imperative tense.” These messages will be identified as positive or negative by adding a “+” or a “- “, respectively, as a bullet point. The feedback view will also need to show an appropriate suggestion for the commit message which takes into account the Git commit message guidelines.

4.5.3 Feedback list

The feedback list view will allow a user to see their chosen range of commit messages in chronological order. The user will then be able to double click a message on this list and be shown a feedback view which concerns their selected commit message. This feedback view will function exactly like the view described in subchapter 4.5.2.

4.6 State machine diagram

The operation of the proposed solution is described by the following state machine diagram.

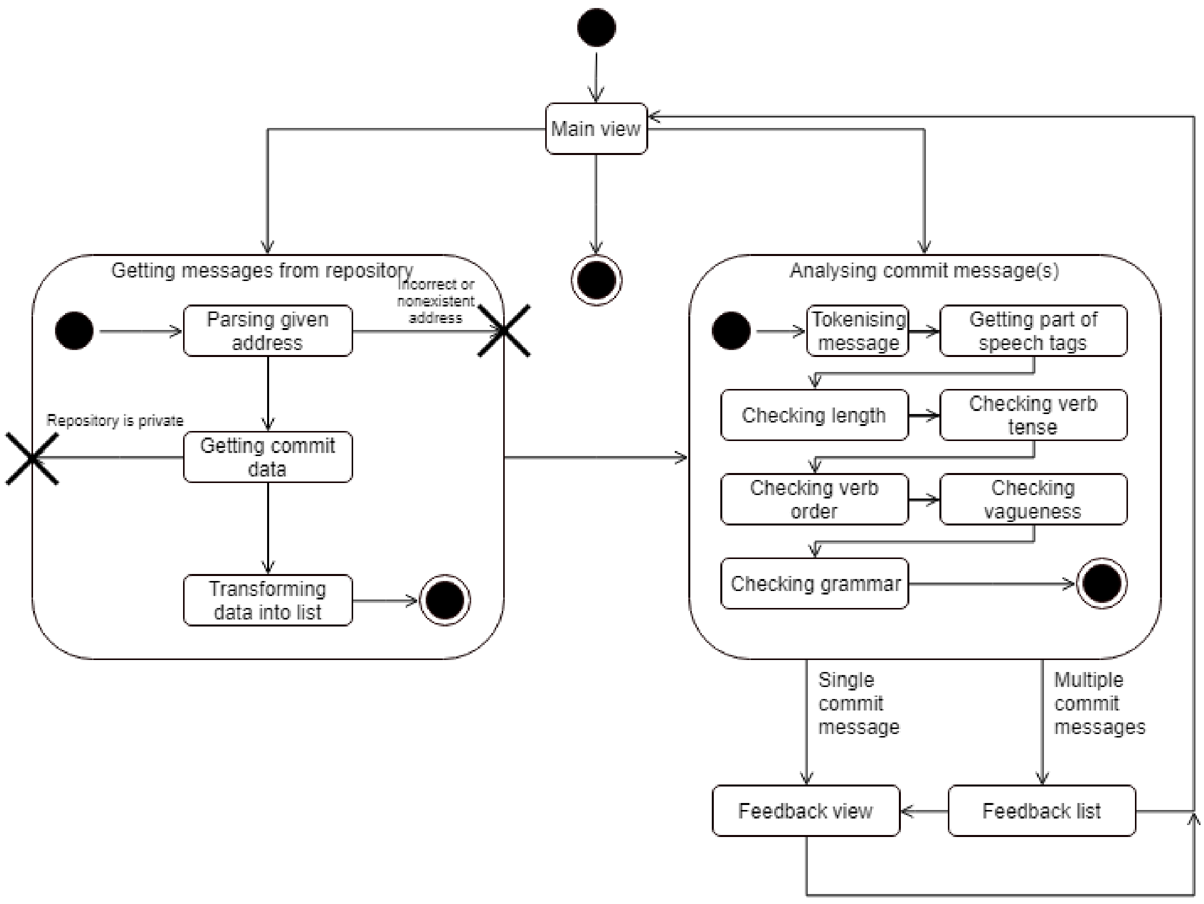


Figure 4.6.1 – state machine diagram of the system

This state machine diagram shows the main states of the program, as well as their relevant sub-states. From the main view, the user can either analyse a single commit message directly or type the address of a repository to analyse.

If the user chooses to analyse a single commit message, the message will first need to be tokenised and the part-of-speech tags identified. Afterwards, several algorithms will be run on the message to ensure that the message is correct, and if it is not, to change the message and let the user know. After the message is analysed, the feedback view will appear.

If the user chooses to analyse a GitHub repository, the program will first need to ensure that the address they type in is a valid repository. If it is invalid or there is no repository at that address, the program should terminate gracefully. If it is a valid repository, the program will then attempt to get commit data from the repository, unless it is a private repository, in which case the program should also terminate gracefully. The commit data will then be transformed into a simple list of commit messages without any information identifying the author, or other irrelevant information, such that each message on the list can then be analysed. Each message will then go through the same message analysis process as the singular commit message. Once every message is analysed, the feedback list appears, and the user will be able to double click on any message to get its respective feedback view.

The user will be able to close out of the feedback views and return to the main view. From there, they will be able to choose to analyse another commit message, analyse another GitHub repository or exit out of the application.

Chapter 5 - Implementation

This chapter describes the implementation of the program, firstly by manner of a flowchart and then by describing each class in the program. The chapter also focuses on development issues and testing of the application.

5.1 Flowchart

The process the application performs on a commit message is described by means of a flowchart.

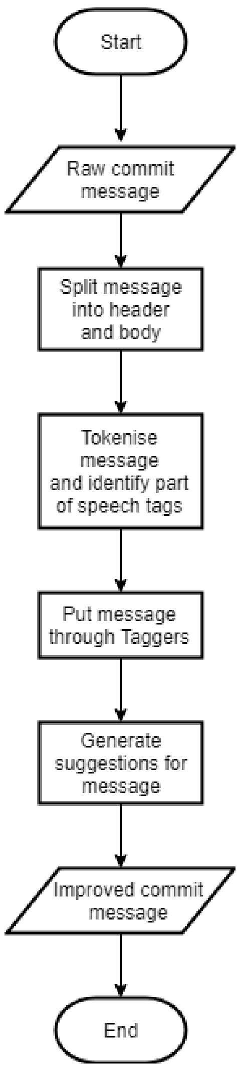

Start
Raw commit message
Split message into header and body
Tokenise message and identify part of speech tags
Put message through Taggers
Generate suggestions for message
Improved commit message
End

Figure 5.1.1 – a flowchart of the program's operations on a commit message

5.2 Classes

This sub-chapter will describe the classes present in the design of the application, and the rationale behind the design of the classes.

5.2.1 CommitMessage and MessageTag

The CommitMessage class describes a typical Git commit message, consisting of a header, body and SHA for accessing more details about the message, should it be necessary. There are further fields to enable a suggestion to be generated for the commit message using the function generateSuggestions(). The constructor of CommitMessage takes a maximum of two strings, the first string being the message itself (composed of both header and body, being split by the constructor in the process) and the second string being the SHA of the commit, which is optional. The first constructor would be used in cases of analysing a single potential commit message that is not linked to a repository, while the second constructor would be used in most other cases, which involve commit messages from Git repositories. Further fields describe the individual tokens and part-of-speech tags of the commit message, which are obtained by using the Stanford CoreNLP library's Sentence class and its associated words() and posTags() functions respectively.

Each CommitMessage has MessageTags which are associated with it. These MessageTags consist of a message, a suggested change for the commit message and whether the tag is positive. They are generated using the Tagger classes, which are described in Subchapter 5.2.2. The purpose of these MessageTags is to provide feedback to the user about a CommitMessage, so that the user can understand what the message does well and what improvements could be made.

Figure 5.2.1.1. describes the relationship between the CommitMessage and MessageTag classes more clearly.

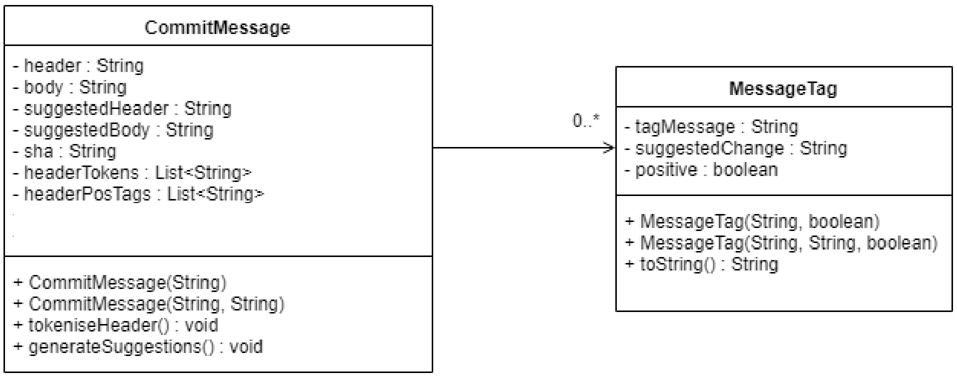

CommitMessage
- header : String
- body : String
- suggestedHeader : String
- suggestedBody : String
- sha : String
- headerTokens : List<String>
- headerPosTags : List<String>
+ CommitMessage(String)
+ CommitMessage(String, String)
+ tokeniseHeader() : void
+ generateSuggestions() : void
0..*
MessageTag
- tagMessage : String
- suggestedChange : String
- positive : boolean
+ MessageTag(String, boolean)
+ MessageTag(String, String, boolean)
+ toString() : String

Figure 5.2.1.1 – a class diagram of the CommitMessage and MessageTag classes

5.2.2 Tagger classes

The Tagger interface describes an interface to give CommitMessages MessageTags, which were described in subchapter 5.2.1. It provides the essential method, tagMessage() which produces MessageTags and adds them to a CommitMessage, as well as the methods getCount() and resetCount() which are used for analytical purposes (i.e. getting how many messages were tagged by a certain tagger). There are several classes which implement this interface, each with a specific series of checks it performs to tag the message with a relevant MessageTag. An example of this is provided in the class diagram below, with the Tagger interface and the HeaderVerbTenseTagger class shown below.

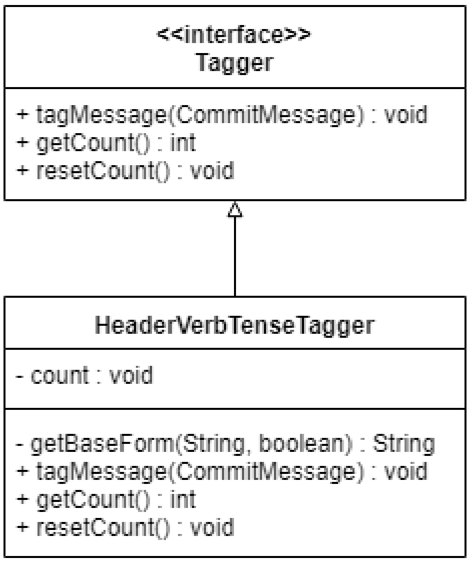


Figure 5.2.2.1 – a class diagram of the Tagger interface and HeaderVerbTenseTagger class

The Tagger classes change the headerTokens and headerPosTags of a CommitMessage depending on their specialisation. For example, the HeaderVerbTenseTagger changes all verbs in the CommitMessage to the present imperative wherever possible, and if appropriate. These changes are performed in a specific order by a class MessageTagger, described in Subchapter 5.2.3, to allow a final suggestion to be generated for the message.

More specifically, the Tagger classes perform these tasks:

- HeaderLengthTagger – checks that the message header is below 72 characters, and ideally, below 50.
- HeaderVerbOrderTagger – checks that the message starts with a verb. If it does not, this Tagger will attempt to find the first verb in the message and move it to the front of the message.
- HeaderVerbTenseTagger – checks that the tense of the verbs in the message is present imperative, and if they are not, and if it is appropriate to do so, changes the verb to its base form.
- HeaderGrammarTagger – checks that the words in the message are correctly capitalised and changes them if they are not.
- HeaderPuncuationTagger – checks that the message header does not end with a punctuation mark, as message headers are titles. If it does, the offending punctuation mark is removed.
- HeaderVaguenessTagger – checks whether the message is vague according to these criteria, and tags the message accordingly:
    - Does the message contain any verbs?
    - Does the message contain any nouns?
    - Are the only nouns in the message present in the list of vague nouns? (e.g. “bug” and “feature” are considered vague)

5.2.3 MessageTagger

The MessageTagger class is used to tag a CommitMessage according to a specific order of Tagger class operations, described by the following flowchart.

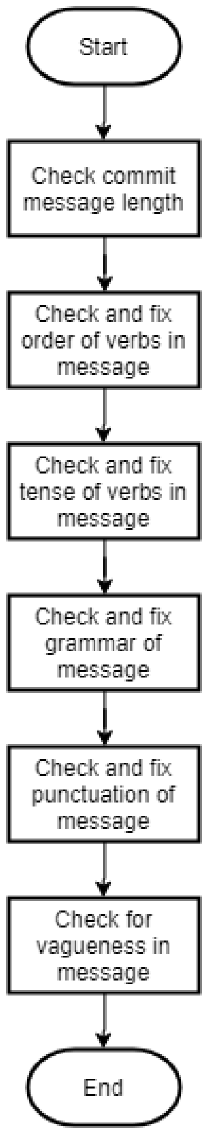

Start
Check commit message length
Check and fix order of verbs in message
Check and fix tense of verbs in message
Check and fix grammar of message
Check and fix punctuation of message
Check for vagueness in message
End

Figure 5.2.3.1 – a flowchart describing the order of operations of the MessageTagger class

The operations in the MessageTagger class are ordered in this way to allow for a suggestion for the message to be generated without having to overcomplicate the implementation. As each Tagger class changes the headerTokens and headerPosTags of the message where appropriate, with different transformations applied on the tokens such as reordering or changing the capitalisation or tense of a word, it is important to ensure that the general order of operations is correct as well.

5.2.4 MessageGetter

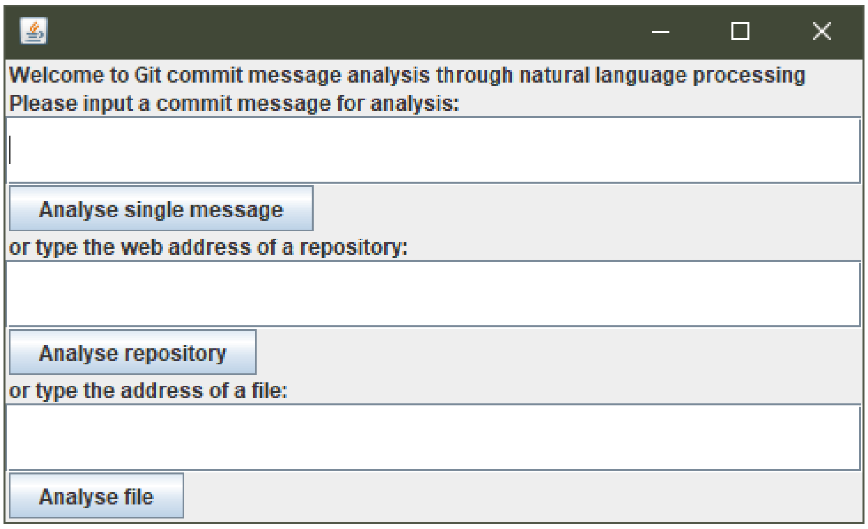
The MessageGetter class is used to obtain commit messages from a GitHub repository and convert them into objects of the CommitMessage class for tagging by the MessageTagger class. It does this by using Unirest, a library for Java that makes HTTP requests. Unirest is used to make writing the code relating to HTTP requests easier. In order to obtain commit messages from GitHub, the use of GitHub's REST API is required, as is the use of an API key. The MessageGetter class uses the appropriate functions of this REST API to obtain commit messages, and it is also capable of getting further data about a commit if it is required, as may be in the case of commits that are very vague and suggestions are needed.

The MessageGetter class also obtains commit messages from text files, where each message is on a new line. This is for the purpose of analysing a set of commit messages like the one in Appendix B.

5.2.5 Views

The design of the application aims to follow the MVC paradigm as far as Java allows it, although in Java the view and controller are, for all intents and purposes, one and the same. There are four view classes, each with a different purpose in the application.

The MainWindow class is the first window that the user will see when they run the application. It prompts the user to enter a commit message for analysis, the web address of a GitHub repository or the address of a file with commit messages.

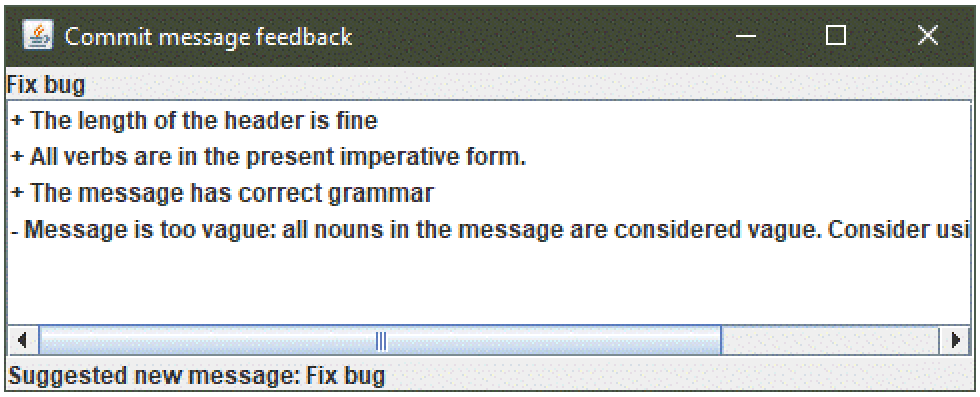

Welcome to Git commit message analysis through natural language processing
Please input a commit message for analysis:

or type the web address of a repository:

or type the address of a file:

Figure 5.2.5.1 – an instance of MainWindow

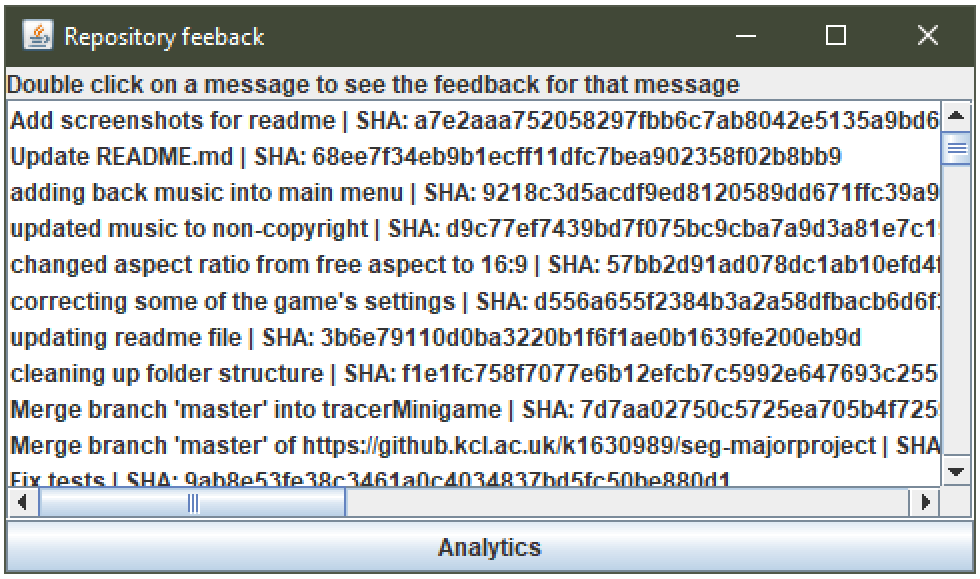
The FeedbackWindow class is used to show feedback on a single commit message. This is directly accessed through using the “analyse single message” function, or through double-clicking on an item on the FeeedbackList. Each feedback item (i.e. MessageTag) is displayed in the FeedbackWindow, with a “+” or “-“ showing whether it is a piece of positive or negative feedback.



Fix bug
+ The length of the header is fine
+ All verbs are in the present imperative form.
+ The message has correct grammar
- Message is too vague: all nouns in the message are considered vague. Consider usi
Suggested new message: Fix bug

Figure 5.2.5.2 – an instance of FeedbackWindow

The FeedbackList class shows a list of commit messages from either a GitHub repository or a file. Each item in the list can be double-clicked to allow the user to see feedback on the commit message, and there is a button to allow the user to view the analytical data for the repository or file as well.

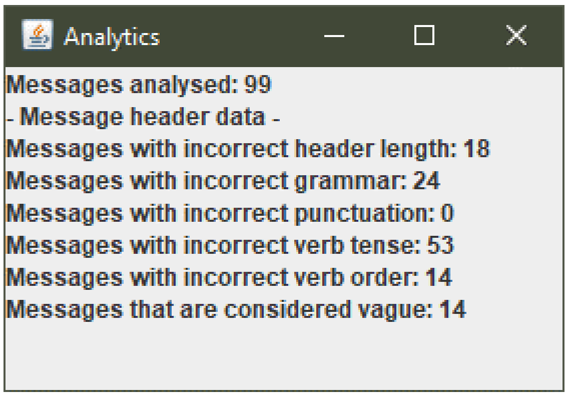


Double click on a message to see the feedback for that message
Add screenshots for readme | SHA: a7e2aaa752058297fbb6c7ab8042e5135a9bd6
Update README.md | SHA: 68ee7f34eb9b1ecff11dfc7bea902358f02b8bb9
adding back music into main menu | SHA: 9218c3d5acdf9ed8120589dd671ffc39a9
updated music to non-copyright | SHA: d9c77ef7439bd7f075bc9cba7a9d3a81e7c1
changed aspect ratio from free aspect to 16:9 | SHA: 57bb2d91ad078dc1ab10efd4
correcting some of the game's settings | SHA: d556a655f2384b3a2a58dfbacb6d6f
updating readme file | SHA: 3b6e79110d0ba3220b1f6f1ae0b1639fe200eb9d
cleaning up folder structure | SHA: f1e1fc758f7077e6b12efcb7c5992e647693c255
Merge branch 'master' into tracerMinigame | SHA: 7d7aa02750c5725ea705b4f725
Merge branch 'master' of https://github.kcl.ac.uk/k1630989/seg-majorproject | SHA
Fix tests | SHA: 9ab8e53fe38c3461a0c4034837bd5fc50be880d1

Figure 5.2.5.3 – an instance of FeedbackList with an example repository[4] *loaded in*

[4] https://github.com/mprzewlocki98/second-year-major-project

The fourth and final view is the AnalyticsWindow, which provides analytical data on a repository or file. This data shows how many commit messages were analysed and how many were tagged by each tagger as having something wrong with them.



```
Messages analysed: 99
- Message header data -
Messages with incorrect header length: 18
Messages with incorrect grammar: 24
Messages with incorrect punctuation: 0
Messages with incorrect verb tense: 53
Messages with incorrect verb order: 14
Messages that are considered vague: 14
```

Figure 5.2.5.4 – an instance of AnalyticsWindow

5.2.6 Other classes

There are several smaller classes that assist in the running of the application. These are BracketsFixer, TokenChecker and VagueNounChecker.

Through design or otherwise, when tokenising a sentence, the Stanford CoreNLP API turns brackets such as "(" into a representation such as "-LRB-", in this example presumably meaning "left round bracket." The BracketsFixer class aims to remedy this by replacing all occurrences of such representations back into the original bracket. A HashMap that maps Strings to Strings is used to achieve this for efficiency reasons, as indexing a HashMap is an O(1) operation and the program will deal with many strings at once, so the data structure used should not be the limiting factor in terms of time complexity. This class just goes through the tokens of a message, checks if each one is in the HashMap, then replaces it with what is in the HashMap is there is an entry for it.

The TokenChecker class is used for checking whether a part-of-speech tag belongs to a certain class (e.g. verb, noun, proper noun or non-proper noun) as well as checking whether a token is a bracket or a closing bracket. For the same reason as the BracketsFixer class, HashMaps are used to check whether a token is in a certain class. However, this time the HashMaps map Strings to Booleans as the only purpose of this class is to check whether or not a given token belongs to a certain class, not to change them in any way.

The VagueNounChecker class is used for checking whether a given noun is considered vague or not. A vague noun, for the purposes of this project, is a noun that on its own does not provide much of a description of what was accomplished. These include “bug”, “thing” and “stuff.” Again, a HashMap of Strings to Booleans is used for efficiency reasons. The HashMap for this is hard-coded as opposed to being in a separate text file to ensure that the application does not depend on any external files, and for further efficiency reasons – having to load the HashMap from a text file could, if the map was extensive enough, slow down the initialisation of the program significantly. The map of vague nouns is not exhaustive, as the aim of introducing this class was more to check that this additional step in checking for vagueness is viable.

5.3 Development issues

As with any project, there were issues during the development of the program.

The initial plan was for the application to be written in Ruby instead of Java. There are hooks for the Stanford CoreNLP API from Java to other programming languages, but the Ruby hooks proved too problematic to work with. There was an issue with the .jar files downloaded by the Ruby gem early on, which halted development, so Java was chosen instead to avoid any further unnecessary delays which could have hindered a functional application from being produced.

Early on in development, further issues and problems were encountered. One problem involved an initial approach to changing the tense of a verb which is not in the present imperative. The initial approach was simply to remove the suffix of the word, such that “fixing” becomes “fix” by removing “-ing,” or “fixed” becomes “fix” by removing “-ed.” This proved not to be the best solution as there were verbs such as “removing” which, when “-ing” was removed, became “remov” [sic]. This was solved through using the lemma() function of the simple Stanford CoreNLP API, which always produces a verb in its base form, which in most cases is functionally identical to the present imperative form. This still produces some issues, however, for example, changing the verb “is” to “be.”

Another problem was obtaining messages from a GitHub repository. Initially, only the SHAs of a commit were obtained in one HTTP request and the commit message was then obtained by individually getting the data about a commit using its SHA in another HTTP request. This proved to be more time-consuming than necessary, since most data about a commit save the content of the message itself is not used in most cases. This was solved by getting both the commit message and the SHA of a commit and only using the SHA of a commit if absolutely necessary and resulted in a massive improvement in the running time of the application.

A third problem was producing suggestions for a commit message. Initially, the CommitMessage class had fields for suggested tokens and part-of-speech tags in addition to the fields for tokens and part-of-speech tags produced by tokenising the message, which would then be changed by each Tagger class individually. This was problematic as the Tagger classes looked at the fields for tokens and part-of-speech tags to detect errors in the message and would then change the fields for the suggested tokens and part-of-speech tags, but this would sometimes mean that changes would be applied to the incorrect tokens. The ordering of the Taggers in MessageTagger was also important and sometimes resulted in incorrect suggestions – for example, the grammatically incorrect message "bug fix" would first be changed to "Bug fix" and then "fix Bug" which is still incorrect. To solve the problem of producing suggestions, it was decided that the fields for suggested tokens and part-of-speech tags were unnecessary. Instead, each Tagger would modify the initial tokens and part-of-speech tags sequentially and in the order dictated by the MessageTagger class to allow for more consistent suggestions to be generated.

5.4 Testing

The application underwent extensive manual and unit testing.

Manual testing was the primary approach used during the development of the application, as new examples of commit messages that violate guidelines appeared throughout development, and a fix for the application that might solve a problem with some commit messages might break a solution for other commit messages at the same time. To perform manual testing, the application was run, and different commit messages were entered to test whether the application detected the applicable guideline violation and whether the application produced a suitable suggestion for the message. If it did not, the code would be rewritten, and the same message would be tested again until the application did so.

Unit tests were also written for the application, although with the volatility of natural language and the Stanford CoreNLP they may not always produce the same results. The unit tests focused on testing the Tagger classes individually and as a whole as these are the basis of the program. Some classes, such as the CommitMessage and MessageTag classes, were also tested indirectly through these unit tests. Other classes, such as the views, are better suited to be manually tested as they will be used by a human user, not a computer.

Unit testing showed some bugs that were not caught in manual testing. For example, there was a bug with excess spaces being added in a commit message after it was analysed – as spaces are invisible, this was not visible on the interface, but a space could be the difference between a message exceeding the recommended length and meeting the recommended length, so this bug was fixed as soon as it was caught.

5.5 Changes from design

Although the design was followed as closely, some changes were made to the application.

Initially, an analytics window was not part of the design of the application. This is because most users would not need analytics, only suggestions on commit messages. The analytics window was added to allow for a greater insight on how the application performed against manual analysis. This also has an additional benefit of providing more information to users who are interested in it.

A feature to allow reading commit messages from a text file, as opposed to a repository, was also added. This is also to allow an insight into how the application performs and has the side benefit of allowing users to input more than one commit message without having to go back to re-enter and submit each message. A further benefit is that a user does not require an Internet connection to use the application in this way, which is especially useful in cases where an Internet connection is not available.

As stated in Chapter 2, most commit messages are limited to the subject line only. Although the solution was intended to analyse both the subject line and the body of a commit, as very few commit messages include a body, this feature was also omitted. Most of what a commit accomplishes can already be described using only a subject line, and the contents of the body of the commit message are only required to provide more information if the subject line cannot do this sufficiently. For the few commit messages that do have a body, the program could be extended to include Taggers for the body as well, although these may be limited to only simple grammar and spelling checks.

The feature to allow selecting a range of commits from a repository was removed. This is discussed further in the Evaluation section. However, the primary reason was to avoid the GitHub REST API from being overloaded with requests and to disallow prohibitively large ranges of commit messages from being chosen. Instead, the first 100 commit messages are always chosen for analysis, and this range should allow most users to still see what they need to change in terms of their commit messages.

Chapter 6 – Results / Evaluation

6.1 Quantitative evaluation

To evaluate the application quantitively, the set of messages in Appendix B was analysed by the application. The messages were analysed sentence-by-sentence to produce a precision and recall analysis for each Tagger in the application. In this case, a positive result means that the application was expected to, or did produce a tag of the relevant category, while a negative result means that the application was not expected to or did not produce the relevant tag.

Results for the header length tagger:

		Actual results	
		Positive	**Negative**
Expected results	**Positive**	39	0
	Negative	0	162
Precision: 100%, Recall: 100%			

Table 6.1.1 – precision and recall analysis of the header length tagger

These results suggest that the header length tagger performs as expected, as the length of the header should not pose any difficulty to a program like this.

Results of the header grammar tagger, which checked for improper capitalisation of the header:

		Actual results	
		Positive	**Negative**
Expected results	**Positive**	76	1
	Negative	32	92
Precision: 70%, Recall: 99%			

Table 6.1.2 – precision and recall analysis of the header grammar tagger

These results show that the grammar tagger performs reasonably well, detecting most grammar issues but having trouble with more nuanced messages, such as those which contain file names, which may be capitalised but not detected by the CoreNLP library as being proper nouns. One grammar issue went undetected, although the message in context was “Add go benchmarks (#240)” and although this presumably referred to the language Go, it is reasonable that in this case the word “go” is considered as a verb.

Results for the header punctuation tagger, which checked for improper punctuation in the header (i.e. any punctuation marks at the end of the message):

		Actual results	
		Positive	**Negative**
Expected results	**Positive**	15	0
	Negative	0	186
Precision: 100%, Recall: 100%			

Table 6.1.3 – precision and recall analysis of the header length tagger

This tagger also performed as expected.

Results for the header verb order tagger, which checked that the message starts with a verb:

		Actual results	
		Positive	**Negative**
Expected results	**Positive**	43	0
	Negative	37	121
Precision: 54%, Recall: 100%			

Table 6.1.4 – precision and recall analysis of the header length tagger

Although the recall of this tagger is 100% (meaning all relevant messages were highlighted as not having correct verb order), the precision is only 54% indicating that this tagger is not as successful as it should be. One reason for this is that the CoreNLP library failed to detect verbs such as "update" or "fix", which are common verbs in many commit messages, as verbs, instead detecting them as nouns.

Results for the header vagueness tagger:

		Actual results	
		Positive	**Negative**
Expected results	**Positive**	34	10
	Negative	27	130
Precision: 56%, Recall: 77%			

Table 6.1.5 – precision and recall analysis of the header vagueness tagger

Both the precision and the recall of this tagger are quite low. From these results, it can be inferred that the vagueness tagger does not perform sufficient checks to see whether a message is vague or not – the criteria for vagueness are: the message has no verbs, the message has no nouns or the only nouns are considered "vague" (so, for example, "bug" and "feature" on their own do not sufficiently describe what was changed). The list of verbs which are considered "vague" is hardcoded into the program and may not sufficiently represent all verbs which would be considered vague in a commit message. Messages such as "fix typo" might be undetected by the tagger despite such messages not being very informative. Also, similarly to the verb order tagger, the CoreNLP library sometimes failed to detect certain verbs as verbs, meaning that the tagger tagged the message as vague for not containing any verbs.

Results for the header verb tense tagger:

		Actual results	
		Positive	**Negative**
Expected results	**Positive**	48	0
	Negative	18	135
Precision: 73%, Recall: 100%			

Table 6.1.6 – precision and recall analysis of the header verb tense tagger

The verb tense tagger performed relatively well, although still with some issues. The algorithm this tagger uses to check for correct verb tense is quite simple but goes beyond checking every single verb in every single message. Still, the algorithm clearly does not cover all cases appropriately. Messages such as “Fix clippy warnings returning the result of a let binding from a block” were incorrectly tagged as having incorrect verb tense due to verbs other than the first being in other forms.

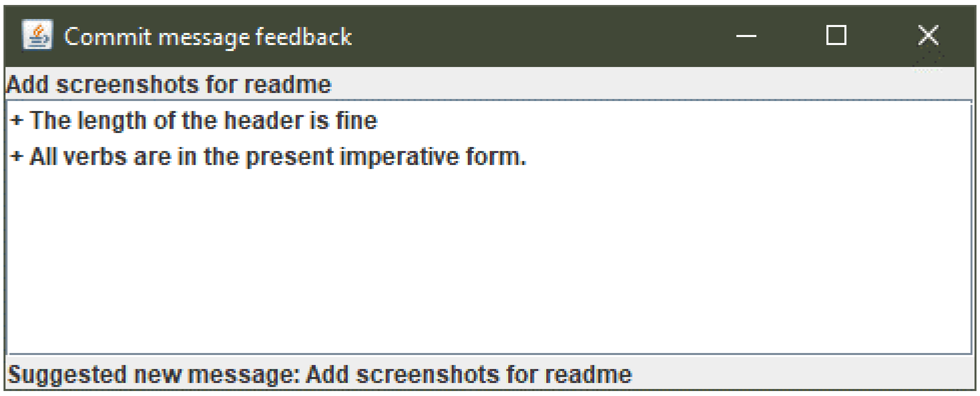
From these quantitative results, it can be inferred that the application performs reasonably well enough in the task of detecting commit message guideline violations, although it is too strict and usually tags messages which are otherwise fine as violating some guideline. This is especially a problem with the verb order tagger and vagueness tagger, although most of the problems are caused by the CoreNLP library failing to detect verbs as verbs, instead detecting them as nouns. In short, these results show that there is room for improvement in the application.

6.2 Qualitative evaluation

For the qualitative evaluation, the suggestions generated by the program for the commit messages are considered, and the program itself is evaluated on a broader scale.

6.2.1 Qualitative evaluation of message analysis

In general, the suggestions generated by the program for commit messages are fair. Grammar mistakes are changed (such as capitalisation of words or punctuation at the end of the message) and verbs are generally changed (in terms of tense and being the first part of the message) correctly. Messages that are already correct seem to be unchanged for the most part. In addition, the verb tense tagger can deal with some simple contexts with relative ease. Some examples of successful changes are provided below.

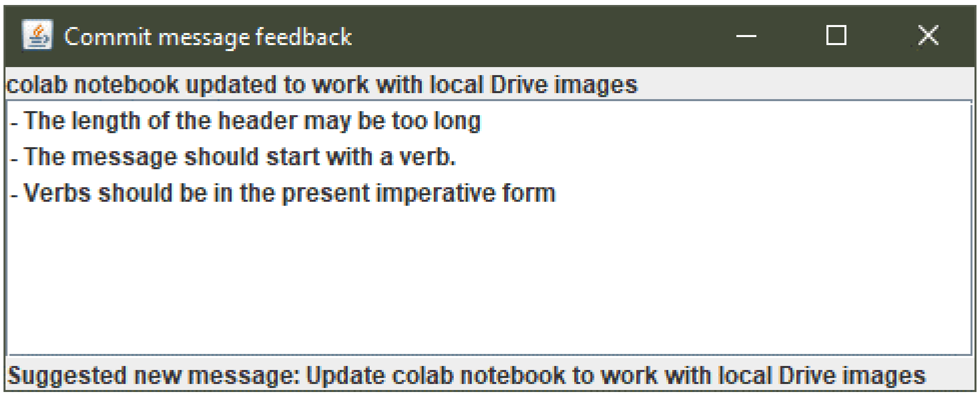


Add screenshots for readme

+ The length of the header is fine
+ All verbs are in the present imperative form.

Suggested new message: Add screenshots for readme

Figure 6.2.1.1 – messages that are already correct are left unchanged

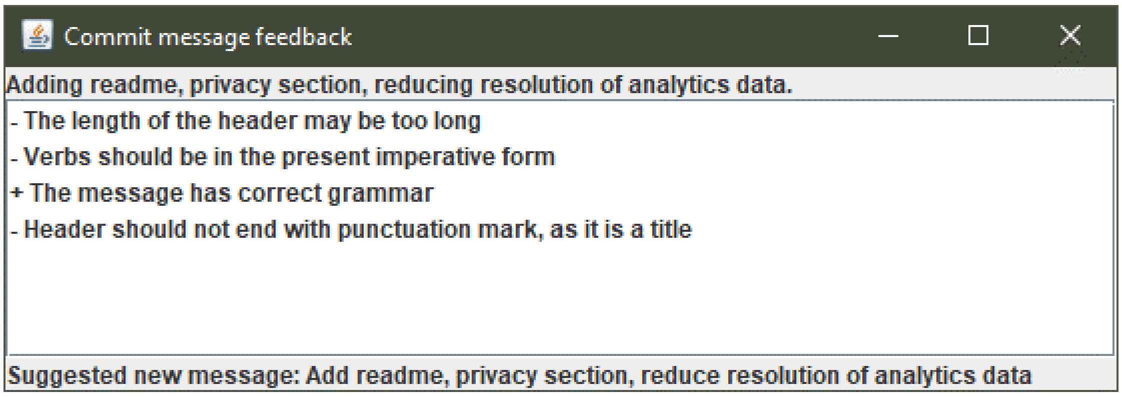


colab notebook updated to work with local Drive images

- The length of the header may be too long
- The message should start with a verb.
- Verbs should be in the present imperative form

Suggested new message: Update colab notebook to work with local Drive images

Figure 6.2.1.2 – the application successfully re-ordered and changed the tense of the verb "update" in this message

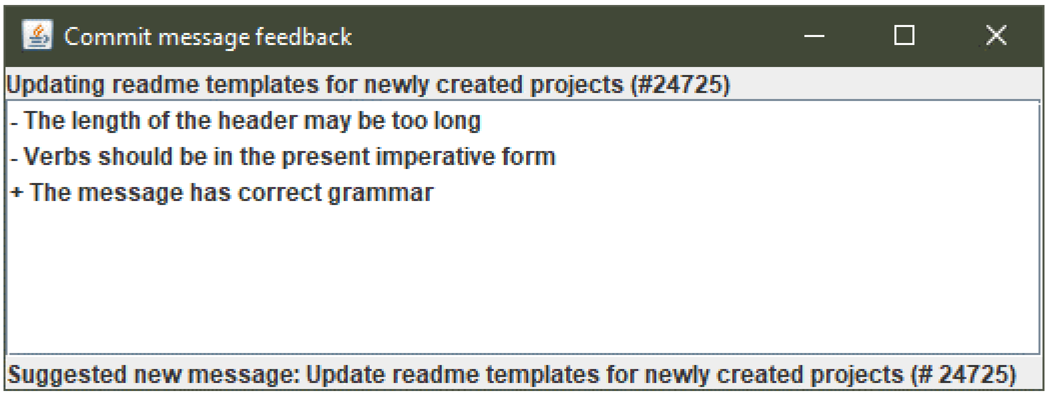


Adding readme, privacy section, reducing resolution of analytics data.

- The length of the header may be too long
- Verbs should be in the present imperative form
+ The message has correct grammar
- Header should not end with punctuation mark, as it is a title

Suggested new message: Add readme, privacy section, reduce resolution of analytics data

Figure 6.2.1.3 – the application successfully changed the tense of the verbs to the present imperative.


Commit message feedback

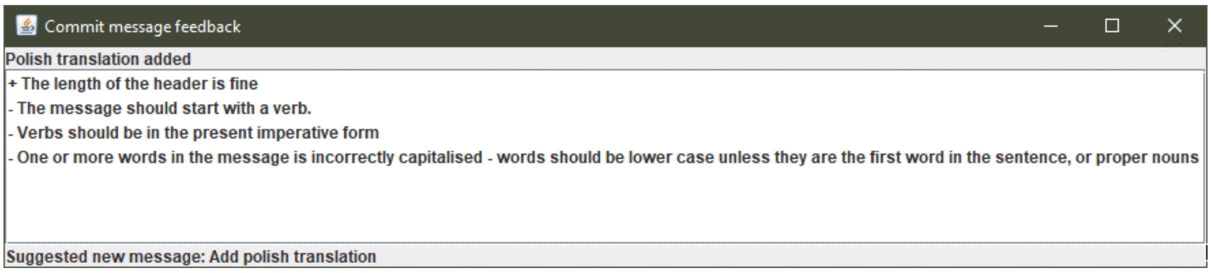
Updating readme templates for newly created projects (#24725)

- The length of the header may be too long
- Verbs should be in the present imperative form
+ The message has correct grammar

Suggested new message: Update readme templates for newly created projects (# 24725)

Figure 6.2.1.4 – the application correctly changes the tense of the word "updating" but leaves "newly created projects" alone

There are, however, some issues which are caused by both procedural faults of the application and the Stanford CoreNLP API failing to correctly detect verbs, nouns and proper nouns. Examples of less successful changes are provided below.

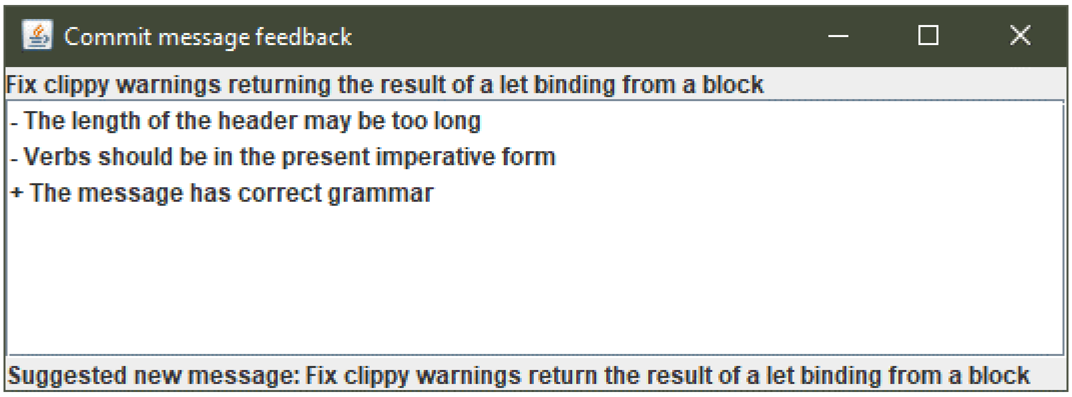

Commit message feedback

Polish translation added

+ The length of the header is fine
- The message should start with a verb.
- Verbs should be in the present imperative form
- One or more words in the message is incorrectly capitalised - words should be lower case unless they are the first word in the sentence, or proper nouns

Suggested new message: Add polish translation

Figure 6.2.1.5 – though the application correctly reorders and changes the tense of the verb "added", it fails to detect "Polish" as referring to the Polish language and thus incorrectly capitalises the word "Polish."

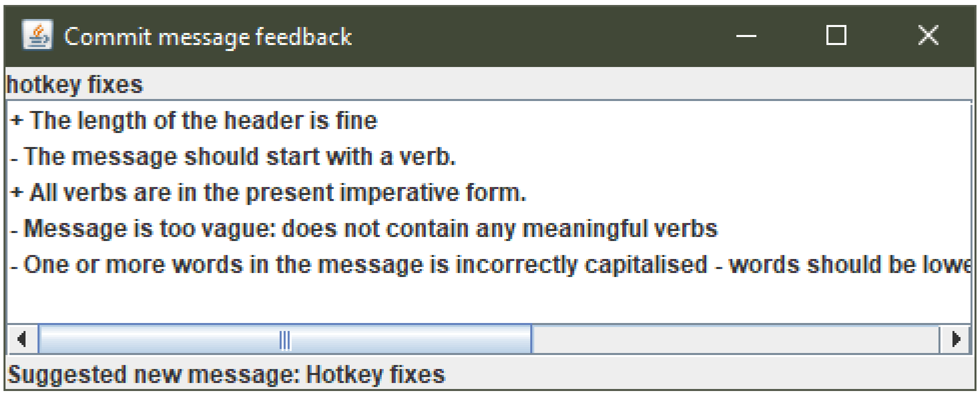

Commit message feedback

Fix clippy warnings returning the result of a let binding from a block

- The length of the header may be too long
- Verbs should be in the present imperative form
+ The message has correct grammar

Suggested new message: Fix clippy warnings return the result of a let binding from a block

Figure 6.2.1.6 – the application incorrectly changes the tense of the verb "returning"

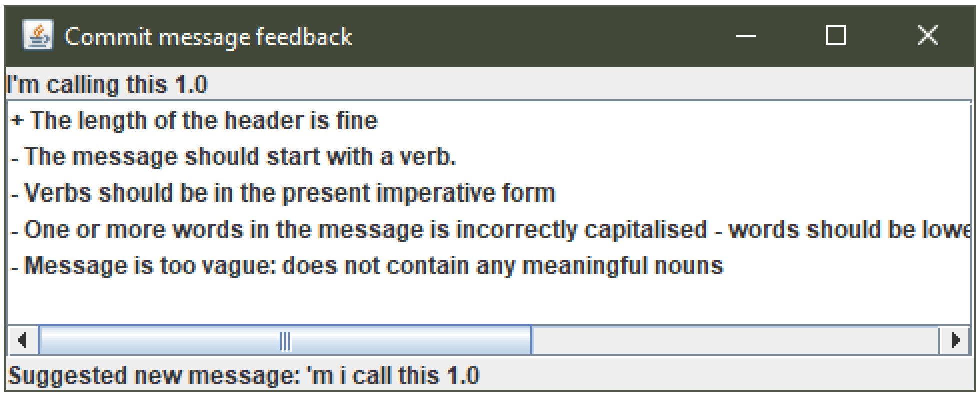

Commit message feedback

hotkey fixes

+ The length of the header is fine
- The message should start with a verb.
+ All verbs are in the present imperative form.
- Message is too vague: does not contain any meaningful verbs
- One or more words in the message is incorrectly capitalised - words should be lowe

Suggested new message: Hotkey fixes

Figure 6.2.1.7 – failure to detect "fixes" as a verb instead of a noun means that this message cannot be changed to "fix hotkey"


Commit message feedback

I'm calling this 1.0

+ The length of the header is fine
- The message should start with a verb.
- Verbs should be in the present imperative form
- One or more words in the message is incorrectly capitalised - words should be lowe
- Message is too vague: does not contain any meaningful nouns

Suggested new message: 'm i call this 1.0

Figure 6.2.1.8 – an example of an entirely unsuccessful change

This qualitative evaluation of the application's suggestion-making capability shows that it can perform simple fixes with relative ease. It also shows that the application can detect breaches of the commit message guidelines. From detecting these breaches alone, a user should be able to make the appropriate changes to a commit message manually should the suggestion feature fail. However, for more difficult fixes that require consideration of context, or for fixes that are outside of the limits of the Stanford CoreNLP machine learning models, the application is not as successful. This could be remedied with more complex algorithms, or through using machine learning models specifically trained on GitHub commit messages as opposed to more general-purpose models.

6.2.2 Qualitative evaluation of application

In general, the application produced meets most of the points in the specification in Chapter 3. However, the application does not allow the user to select a range of commits to analyse, instead always analysing the first 100 commit messages of a repository. This is because the GitHub REST API provides information in pages of up to 100 items to prevent their servers from being overloaded with requests. For this reason, and to prevent cases of the user selecting prohibitively large ranges of commit messages, the feature of selecting a range of commits to analyse was left out from the application.

Further, the application does not draw on the context of a Git commit in case of a vague commit message. This is because the information provided about a commit by the GitHub REST API for commits[5], while large in content, turns out to be quite limited. As a side note, most information provided by this API is discarded to prevent breaches of anonymity and for efficiency reasons. The most information that can be obtained from this API alone is how many lines of code were added, deleted and which files were affected. This would only enable the application to produce suggestions of limited utility, such as "add file1.txt" or "modify file2.txt." While potentially more useful than messages such as "fix bug", these still are not very informative so the process of obtaining this context is more hassle than what it is worth. To be able to have even more context about a commit message would require the use of static code analysis, which was omitted due to time constraints and because using static analysis on every vague message may reduce the efficiency of the application.

Otherwise, the application does meet most points the specification. A user can enter a commit message for analysis, or the address of a GitHub repository for analysis of commit messages. The application can produce feedback on whether the message meets guidelines or not and can produce suggestions on how to change the message. This analysis is conducted using the Stanford CoreNLP simple API. Whether this information is meaningful to the user will depend on the user, however, every effort has been made to ensure that most users would be able to understand the faults of their commit messages.

[5] https://developer.github.com/v3/repos/commits/

As for non-functional requirements, the application is usable. The interface appears to be simple enough for a user to use, although less technical users may have trouble with it. It is intuitive, as all the user must do is type a commit message and press a button, or type the address of a GitHub repository, press a button and then double-click on an item on the list that appears. This may not be immediately obvious to the user; however, they should be able to figure out how to use the application in a short enough time.

As for security concerns, every effort has been made to ensure that the application is secure, and any security faults may not necessarily be the result of the programming of the application. The only way the application can access a repository is through an API key which permits reading of public repositories and nothing else. This prevents unauthorised reading of private repositories as well as unauthorised modifications to the contents of any repository should the user attempt to do so. The application also does not have any code which would allow for the modification of the contents of a repository, only to get the contents of it. However, a malicious user could change the API key to one with the permissions of reading and modifying all repositories, although they would first have to decompile the application. This does pose a security risk, but if they wished to do anything else with that API key, they would have to first reprogram the application to allow for malicious use, which would be time consuming.

The Unirest API is well-documented and its creators, Kong Inc.[6] (formerly known as Mashape Inc.) are a legitimate company, although caution is still required as it is a third-party API and there are security risks involved with using such APIs. For example, the company responsible for the API might be compromised by an internal or external malicious party and sensitive data such as the key for the GitHub REST API would need to be re-generated.

The Stanford CoreNLP API does not appear to make any interactions through HTTP requests and was produced by a reputable source, Stanford University, although again, as it is a third-party API, risks

[6] Kong: Next-Generation API platform for Microservices - https://konghq.com/

remain. Having said that, the use of third-party APIs prevents “not invented here syndrome”[7] and so avoids favouring first-party solutions only for the reason that they are first-party when more efficient third-party solutions are available.

As for efficiency, the application aims to be as efficient as possible. HashMaps were used extensively throughout this project due to their O(1) accessing and adding time complexities, which is essential when dealing with a large amount of data and to avoid O(n) time complexity algorithms becoming $O(n^2)$ time complexity algorithms, for example. Where possible, parallel programming is used as analysing commit messages like the application does appear to be an embarrassingly parallel task, i.e. analysing one message does not rely on analysing another message in any way. A drawback to this is that this can have a negative effect on the accuracy of the results shown in the analytics view. Limiting factors in terms of efficiency include the time taken to get the results of HTTP requests and the time taken by the Stanford CoreNLP API to tokenise and identify part-of-speech tags, although these are inevitable delays, and they are necessary for the running of the application.

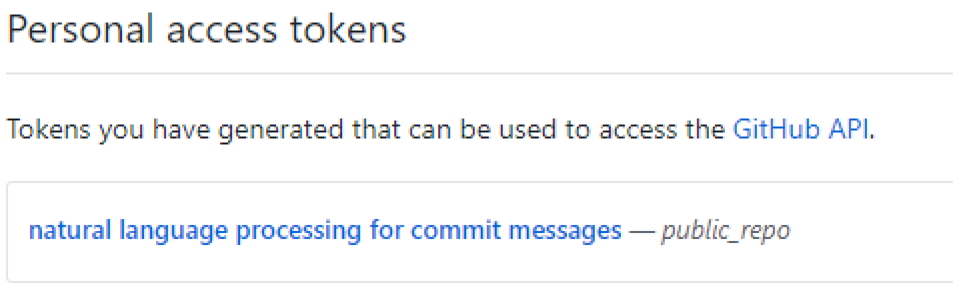
Some algorithms in the application may not be as efficient as they should be. For example, the CommitMessage class has the function generateSuggestions() and as part of that function, there is a procedure to determine whether a space should be added after the current token. This accesses the token that comes after the current token, and decides based on that, instead of simply grafting the tokens into a sentence and then removing excess spaces. This efficiency trade-off and others like it may, however, be necessary to ensure accuracy and ensure the user gets a more readable, “pretty” commit message suggestion, thus enhancing user experience.

[7] What is not invented here syndrome? Webopedia Definition - https://www.webopedia.com/TERM/N/not_invented_here_syndrome.html

Chapter 7 – Legal, Social, Ethical and Professional Issues

The project draws on publicly available data for evaluation, that is, public GitHub repositories from which commit messages are taken for testing and evaluating the software. Every effort is made to only use the data concerning the commit in question, without using data about the authors themselves. As the data is in the public domain, there is due regard for the privacy of others by only using that which has been made public.

Further, the application produced by the project deliberately only allows access to public GitHub repositories to prevent unauthorised access to private repositories, which could breach both ethical and legal boundaries. This is enabled by GitHub's personal access tokens (i.e. API keys) having customisable permissions, or scopes. In the case of this application, the API key used only allows public repositories to be accessed, as seen in Figure 6.1, and attempting to access a private repository will result in a 403 Forbidden error.


Personal access tokens

Tokens you have generated that can be used to access the GitHub API.

natural language processing for commit messages — *public_repo*

Figure 7.1 – the personal access token used for this application only has the public_repo scope.

The BCS Code of Conduct[8] states "you shall only undertake to do work or provide a service that is within your professional competence" and "develop your professional knowledge, skills and competence on a continuing basis." Java was chosen to develop the application as it is a programming

[8] Code of Conduct for BCS Members - https://www.bcs.org/upload/pdf/conduct.pdf

language I am familiar with, and therefore, it is within my professional competence. The development of this application has resulted in the development of professional knowledge and skills as well.

The BCS Code of Conduct also states, “you shall have due regard for […] privacy [and] security […] of others.” As such, every effort has been made to make the application as secure as possible, from using a restrictive API key to not including any code that makes HTTP POST requests that could potentially result in the modification of a repository. The security implications are discussed further in the Evaluation chapter.

Other parts of the BCS Code of Conduct are noteworthy although not necessarily applicable to this project, such as the part concerned with bribery or anything relating to direct interaction with others as this an individual project within an academic setting. As such, there is no reason why bribes would be taken or offered during the course of the project, and interaction with others is at a minimum. Of course, it is important that for future work with others, bribes should not be taken at all.

All libraries and code used that is not mine is explicitly stated, as with the Unirest and Stanford CoreNLP APIs as well as the JUnit framework. Great care has been taken to cite the sources of claims made if they are not already general knowledge to avoid plagiarism, which is unethical as it deprives the author of a source of the credit they deserve, as shown through the use of a bibliography and footnotes.

As for the program itself, because the program deliberately omits a feature to directly edit a repository’s commit history, this should show users that modifying the commit history of a repository is not condoned. However, users still may be tempted by the program to modify the commit history to fix all commit messages that do not follow guidelines. This is poor practice, as once a commit is made to a repository, it should be considered final. Therefore, if the program is to be released to the general public, it should be made clear in either the program itself or any documentation material that the history of a Git repository should not be changed, not only because it is bad practice but also because of a host of issues associated with it (e.g. other programmers having to pull the entire history of the repository again).

Chapter 8 – Conclusions and Future Work

8.1 Conclusions

Natural language analysis – or at least, a distributive approach to it – is reasonably successful in the context of Git commit messages. For the most common mistakes, such as incorrect verb tense, verb order or grammar, these problems are easily remedied by applying a machine learning, distributive approach to detect the parts of speech and change them where necessary. However, as natural language is not very clear-cut, having many different rules and exceptions as there is an inevitable human factor, there are always going to be problems with implementing such approaches, not only to Git commit messages but also to a wider range of contexts in general. For example, the Stanford CoreNLP machine learning API sometimes incorrectly identifies a verb such as "fixes" as a noun instead, or fails to correctly detect proper nouns, considering them as nouns instead. This is shown by the 54% precision of the verb order tagger and 56% precision of the vagueness tagger, both caused in part due to this fault of CoreNLP. As such, for more complex Git commit messages, these approaches must be used carefully.

More generally, strides have been made in the field of natural language analysis - the existence of a library like Stanford CoreNLP today to allow inexperienced users to perform natural language processing is a testament to that, whereas the earliest experiments in natural language processing, taking place as early as the 1950s, were rudimentary and limited. Even just analysing long sentences with the best algorithms and technology available at the time took speeds upwards of 7 minutes, [21] whereas tasks such as part-of-speech tagging and detecting the tense of a verb can now be carried out and be mostly accurate within a matter of milliseconds. However, despite extensive research into the field, there is still not yet a single approach to natural language processing that will work 100% of the time with 100% accuracy, and such an approach may not even exist. The quantitative results in Chapter 6.1 show that the precision and recall of distributional approaches can vary between the two extremes of perfect precision and recall to being only slightly better than a random approach (which would theoretically provide 50% accuracy).

To solve the problem of incorrectly written Git commit messages, it may be easier to teach the guidelines to programmers who use Git before they start writing commit messages, as opposed to trying to fix pre-existing incorrect messages, as analysing commit messages through natural language processing is not a trivial task. A system like the one developed for this project would best be suited to showing users where commit messages go wrong for future reference, rather than to be used directly to change commit messages to be correct. Although the recall for most of the taggers produced for the application is near 100%, the precision varies from 54% to 100%, so it is important to exercise careful judgement when it comes to the suggestions given by programs like this one.

8.2 Future work

One limitation of the program is that it only works in the English language. Because every language has different syntax, different semantics, different tenses, different alphabets etc., there would be some difficulty in translating the approach this program takes to other languages, for which the guidelines for commit messages may also be somewhat different. Making the program work for multiple languages is, however, one possible extension of the program, although it would require native knowledge of these languages to ensure that the solution provides the most accurate suggestions for messages.

Another possible extension of the solution is to turn it into a RESTful API or service which anyone could access in order to receive suggestions for their commit messages as part of a more substantial program. This would allow greater flexibility for those who wish to take an automated approach to commit message analysis, as they could, for example, use a programming language of their choice other than Java, and they could incorporate it into more practical projects such as an automatic commit message marker for software engineering as an academic activity. This does have the additional security implication of possible third-party interference, however, and that should be dealt with accordingly.

A third possible extension would be for the program to consider commit messages in the context of the code changes associated with the commit, for cases where the message is considered vague. This would allow a more meaningful suggestion for the commit message to be generated rather than just "add a verb" or "add a noun," or even "file X was modified." However, this may require more research into static code analysis despite the scope of this project being only natural language processing.

Yet another possible extension for the program would be a scoring system for commit messages. This would be so that the user has a quantitative indicator of how good a commit message is. For example, each commit message would start at 100 points, and 10 points would be subtracted for every infraction of the guidelines detected by the Taggers until the message reaches 0 points, which would be a minimum. This could be used for, amongst other things, grading repositories in academic settings, particularly modules aiming to teach students about the basics of software engineering and Git.

To determine the efficacy of different approaches to natural language processing, this project could be done again although through a model-theoretical, mainly frame-based or even interactive learning approach. The results of those approaches could then be compared to the results obtained by analysis of the same data set used for this project, or different, more expansive data sets. This would provide a more concrete foundation on which approach to natural language processing is best for Git commit messages.

Overall, this is a very versatile project which provides a basis from which it can be adapted for many different situations. By design, it is extendable and has room for modification and improvement, and these examples for future work are only some of the many potential extensions and uses of the program.

Bibliography

[1] Ben-Yosef, A. (2015). *Bad Commit Messages Hall of Shame - codelord.net.* [online] Codelord.net. Available at: https://www.codelord.net/2015/03/16/bad-commit-messages-hall-of-shame/ [Accessed 10 Dec. 2018].

[2] help.github.com. (2018). *Changing a commit message - User Documentation.* [online] Available at: https://help.github.com/articles/changing-a-commit-message/ [Accessed 10 Dec. 2018].

[3] Beams, C. (2018). *How to Write a Git Commit Message*. [online] chris.beams.io/posts/git-commit/. Available at: https://chris.beams.io/posts/git-commit/ [Accessed 29 Oct. 2018].

[4] Amaza, S. (2018). *How to Write Proper Git Commit Messages – Medium*. [online] Medium. Available at: https://medium.com/@steveamaza/how-to-write-a-proper-git-commit-message-e028865e5791 [Accessed 29 Oct. 2018].

[5] Pope, T. (2008). tbaggery - *A Note About Git Commit Messages.* [online] tbaggery.com. Available at: https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html [Accessed 29 Oct. 2018].

[6] git-scm.com. Git - Contributing to a Project. [online] Available at: https://www.git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project [Accessed 29 Oct. 2018].

[7] Oxford Dictionaries. (n.d.). *natural language processing | Definition of natural language processing in English by Oxford Dictionaries*. [online] Available at: https://en.oxforddictionaries.com/definition/natural_language_processing [Accessed 10 Dec. 2018].

[8] Yao, M. 2017. *4 Approaches To Natural Language Processing & Understanding*. [online]. Available at: https://www.topbots.com/4-different-approaches-natural-language-processing-understanding/ [Accessed 29 Oct. 2018].

[9] Liang, P. (2017*). "Foundations of ML" Bootcamp.* [online]. Available at: https://simons.berkeley.edu/sites/default/files/docs/5950/2017.02.01-21.15.12-simons-nlp-tutorial.pdf [Accessed 29 Oct. 2018].

[10] Minsky, M. (1974). *A Framework for Representing Knowledge.* [online] web.media.mit.edu. Available at: http://web.media.mit.edu/~minsky/papers/Frames/frames.html [Accessed 29 Oct. 2018].

[11] en.wikipedia.org. (2018). *Distributional semantics.* [online] Available at: https://en.wikipedia.org/wiki/Distributional_semantics [Accessed 29 Oct. 2018].

[12] plato.stanford.edu. (2001). *Model Theory (Stanford Encyclopedia of Philosophy)*. [online] Available at: https://plato.stanford.edu/entries/model-theory/ [Accessed 13 Dec. 2018].

[13] plato.stanford.edu. (2004). *Compositionality (Stanford Encyclopedia of Philosophy).* [online] Available at: https://plato.stanford.edu/entries/compositionality/ [Accessed 13 Dec. 2018].

[14] Lidor, N. and Wang, S. (2016). *Interactive Language Learning - The Stanford Natural Language Processing Group.* [online] nlp.stanford.edu. Available at: https://nlp.stanford.edu/blog/interactive-language-learning/ [Accessed 10 Dec. 2018].

[15] Manning, Christopher D., Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. *The Stanford CoreNLP Natural Language Processing Toolkit* In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 55-60.

[16] opennlp.apache.org. (2017). *Documentation - Apache OpenNLP*. [online] Available at: https://opennlp.apache.org/docs/ [Accessed 29 Oct. 2018].

[17] Mayo, M. (2018). *The Main Approaches to Natural Language Processing Tasks.* [online] kdnuggets.com. Available at: https://www.kdnuggets.com/2018/10/main-approaches-natural-language-processing-tasks.html [Accessed 13 Dec. 2018].

[18] Stack Overflow. (2015). Stack Overflow Developer Survey 2015. [online] Available at: https://insights.stackoverflow.com/survey/2015#tech-sourcecontrol [Accessed 18 Mar. 2019].

[19] dictionary.cambridge.org. (n.d.). NATURAL LANGUAGE | meaning in the Cambridge English Dictionary. [online] Available at: https://dictionary.cambridge.org/dictionary/english/natural-language [Accessed 18 Mar. 2019].

[20] interactivepython.org. (2019). 1.11. Formal and Natural Languages — How to Think like a Computer Scientist: Interactive Edition. [online] Available at: http://interactivepython.org/runestone/static/CS152f17/GeneralIntro/FormalandNaturalLanguages.html [Accessed 18 Mar. 2019].

[21] Jones, K. (2001). Natural language processing: a historical review, p.2. Available at: https://www.cl.cam.ac.uk/archive/ksj21/histdw4.pdf [Accessed 25 Mar. 2019].

[22] Slingerland, C. (2018). 12 NLP Examples: How Natural Language Processing is Used. [online] Wonderflow. Available at: https://www.wonderflow.co/nlp-examples/ [Accessed 3 Apr. 2019].