Bachelor Thesis

August 22, 2021

"*Comment Quality Assessment and Classification*"

Tim Moser

of Kl. Andelfingen, Schweiz (15-916-117)

supervised by

Prof. Dr. Harald Gall

Dr. Sebastiano Panichella

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Bachelor Thesis

"*Comment Quality Assessment and Classification*"

Tim Moser

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Bachelor Thesis

Author: Tim Moser, tim.moser@uzh.ch

URL: https://www.ifi.uzh.ch/en/seal.html

Project period: 22.02.2021 - 22.08.2021

Software Evolution and Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank...

- Prof. Dr. **Harald C. Gall** for allowing and approving this thesis.
- Dr. **Sebastiano Panichella** for supervising its execution and giving continuous feedback and support. His help, availability, and responsiveness have been invaluable to the creation of this thesis.
- **Pooja Rani** from the University of Bern, **Andrea Di Sorbo** from University of Sannio and **Rafael Kallis** for their expert opinions, supply of related work, pointers to relevant technology and feedback on methodology and the shape of the thesis.

Abstract

Almost all tasks concerning the evolution and maintenance of software require a developer to understand the code. Multiple studies have shown in the past that commented code is more readable than code without any comments, indicating that comments are containing vital information about the implementation. However, not all comments are of equal quality: some are often incomplete, inconsistent with the code, hard to read and understand, or entirely missing.

In this thesis, we investigated how we can provide an approach to analyze comments and rate them with respect to their quality in four different programming languages: *Java, C, C++ and C#*. The goal of this thesis is twofold:

- *RQ1*: Propose a deep learning approach to classify comments into different categories, based on purpose and semantics. We show that our classification pipeline reached an accuracy of over 90% (F1-score), out-performing traditional machine learning models on the same data set in the same environment.
- *RQ2*: Develop a tool for assessing comment quality with respect to readability, coherence, usefulness, completeness and consistency. We will also demonstrate with said tool the evolution of comment quality in four repositories, written in different programming languages and suggest directions for future work, like an empiric evaluation with real developers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Comment Quality Assessment Challenges

Most software maintenance and evolution tasks require developers to understand the source code of their software systems. In fact the majority of the time spent by developers constitutes of code comprehension and adds up to a total of *58%* [1]. Software developers usually inspect class comments to gain knowledge about program behavior, regardless of the programming language they are using [2]. However, comments tend to be often incomplete, inconsistent with the source code, hard to read and understand or simply missing, which substantially complicate any program comprehension tasks [3]. It is no secret and has been shown on numerous occasions that commented code is easier to understand than code that is lacking any comments [4] [5]. This is also why reviewing comments and their quality is an important part of any code reviewing process, to ensure their readability, completeness, consistency, coherence, and usefulness [6].

1.2 Thesis Motivation

There are several tools to supplement the most common activities of code reviews, focusing on code documentation and its comments. Some enable collaboration by providing a platform for commenting on code[1], others on enforcing good-practices when coding, by suggesting common refactoring possibilities inside the IDE[2] [7] [8] [9]. Some of the latter can give very limited feedback on the quality of code comments (e.g., class comments): For example, they can detect when a method or class has no documentation at all. Another basic approach is evaluating the ratio of comments to lines of code or their general spacing and distribution [10].

While the before-mentioned metrics give an easy overview of the completeness and coverage of the comments, they can not give any true insight on the quality of the comments themselves.

The challenge within this lies in the many different dimensions of what makes a comment, and documentation through comments as a whole, to be considered of high quality: For example, is a comment that perfectly explains the code but is hard to read of high quality? Or is a documentation of high quality, if the comment coverage is high, but the comments themselves are too short and do not add anything valuable to code comprehension?

[1] e.g. CodeFlow, Collaborator, Crucible, etc.
[2] e.g. PMD, CheckStyle, etc.

1.3 Goal and Research Questions

Based on the motivation of this thesis and the proposed challenges in the field, we address the following **research questions** in this thesis:

- ***RQ_1: Does Deep Learning perform better than Machine Learning approaches in classifying code comments?***
  The focus of this research questions is to compare results of Deep Learning and Machine Learning approaches in the context of comment type classification. Results of this research question are useful to identify what is the best approach to classify comments of different languages.
- ***RQ_2: How to systematically assess code comment quality in different program languages?***
  This research question has the goal to propose an approach that by design can be used to classify code comment quality of different programming languages, namely Java, C, C++ and C#.

The main goal of this thesis is to answer the before mentioned two research questions and explore approaches not considered in previous work, iterate on known approaches, include proven methods from related studies and aggregate them under one tool. The quest for those answers resulted in an application that is able to categorize comment types and provide indications on the quality using different and complementary criteria:

- **Type** of comments with respect to their purpose:
  Summary, Expand, Rationale, Usage or *Warning*
- **Quality** of comments with respect to a sub-set of the following:
  Readability, Coherence, Completeness, Consistency and *Usefulness*

1.4 Summary of Main Contributions

This thesis makes the following contributions:

- **Pipeline for comment type classification**: Expandable and flexible machine learning pipeline for binary classification of comment types.
- **Python tool for comment quality assessment**: A tool for evaluating local project files for the quality of their comments. The software is developed in a way that can target potentially different scenarios:
  - **Code Review**: The program provides an overview of the overall quality of comments of a project hosted in GitHub comments, in a post-commit code review scenario, which can help identifying problematic or lacking comment coverage and quality. It is intended to be run locally, scraping and evaluating the comments in the local files of a project directory.
  - **Code Comments Evolution**: It may also assist pinpoint the lacking co-evolution of comments with their respective source code, which is a problem discovered and discussed in previous work [3]. Applying the tool on multiple iterations or releases of a project, one can easily observe any changes in the comments' quality.

  - **Local Development**: While well-documented and commented code simplifies any maintenance and development process, several developers overlook or delay commenting their code [11]. Through providing pointers to lacking comment documentation and ways to increase quality consistency, the tool at hand helps solving this problem and invokes awareness to said issues.

- **Initial research**: We investigated new methods of assessing comment quality and classification and aggregated and combined proven methods in a coherent tool.
- **Future direction**: We suggest relevant directions for future work in the area of comment quality assessment and how to further improve our approach and tool in the chapter 7.

1.5 Outline

After this introduction to the topic, we will provide insight into related work in the chapter 2. The following chapters 3 and 4 are dedicated to the first and second research questions respectively, where we define the methodology used to answer the appropriate research question and then present and discuss the results. Chapter 5 gives some more insight in the actual tool developed to assess comment quality. In the end we will discuss any threats to validity in chapter 6, conclude the thesis in 7 and propose directions for future work and improvements of the proposed application. In the appendix are the link to the GitHub repository of this thesis and some up-scaled figures that might be hard to read and interpret in the thesis's text.

Chapter 2

Related Work

2.1 Code Comment Classification

Related work on comment categories can be split into two main approaches:

- **Category based on syntax and location**: One way to classify comments is by looking at their position and syntax. For example, a comment above a class is a header or a comment inside a method is an in-line comment. Steidl et al. (2013), proposed a taxonomy based on a comment's syntax and position. Examples of the said comment category taxonomy include in-line-, member- or section comments [6]. They did this by manually tagging 830 comments and training a J48-model using predefined features like length of comments, the number of brackets, etc. Generally, these methods are based on pre-defined rules and require a deep knowledge about the domain [12].

- **Categories based on purpose**: The other way to classify comments into categories is to look at their semantics and purpose. For example, a comment that summarizes the code is a summary comment or a comment that explains the usage of the code, is a usage comment. Pascarella and Bacchelli (2017) proposed such a taxonomy for the java programming language [13], while Zhang et al. (2018) proposed a different one for the Python language [14]. The recent study of Pooja Rani et al. (2021) focused on creating a mapping between the taxonomies of different programming languages, proposing a language independent approach to classification [15].

In our study, we will focus on the latter regarding the classification of comments. All proposed approaches rely on a machine learning pipeline to classify the intent and purpose of comments. The algorithms learn to classify text by observing data [12]. As stated in the first research question, we propose a different approach to the same problem, using deep learning.

It is shown in related work about text classification that deep learning based pipelines have surpassed common machine learning based models in multiple different tasks like news categorization, sentiment analysis, etc. [12]. From this, we extracted the hypothesis that deep learning outperforms traditional machine learning approaches for comment type classification too, as it is also a text classification task.

It is worth noting that we will also record a comment's type based on location alone as a side effect of scraping for comments in files, when answering the second research question about comment quality assessment, elaborated further in the section 4.2. Instead of using a machine learning algorithm, comments were classified solely based on their occurrence in the code: If a comment precedes a function, it is a function comment. If a comment is preceding a class, it is classified as class comment. This was necessary, because some established quality metrics only

apply for certain types of comments. For example, in-line comments should be shorter than 30 words, while class comments have no hard upper limit in length [16].

2.2 Code Comment Quality Attributes

The first question that we need to answer is *What defines a comment of high quality and how can we measure it?*. This is not an easy question to answer, as related work concerning quality assessment of comments does not have a consensus on a definition or any metrics. To illustrate this point, we present a brief overview of some approaches from related work, who defined either top level assets of quality, metrics to represent them or both:

- Khamis et al. (2010) propose several metrics to assess the quality of javadoc comments with their tool JavadocMiner [17]:
  - **Token, Noun and Verb Count Heuristic**: These heuristics are the initial means of detecting the use of well-formed sentences within in-line documentation.
  - **Words Per Javadoc Comment Heuristic (WPJC)**: Average number of words in a Javadoc comment: detects over/under documentation.
  - **Abbreviation Count Heuristic (ABB)**: According to "How to Write Doc Comments for the Javadoc Tool"[1] the use of abbreviations in javadoc comments should be avoided.
  - **The Fog Index**: Indicates the number of years of formal education a reader would need to understand a block of text. The fog index must not be above 12, a score between 7-8 is optimal.
  - **Flesch Reading Ease Level**: Rates text on a 100 point scale. The higher the value, the easier to read. The optimal score would range from 60 to 70.
- Wang et al. (2019) and Corazza et al. (2018) both propose a way of rating the **coherence** of comments [18] [19]. Wang created a data set from javadoc method comments together with a tokenized version of the corresponding code, while Corazza tried vectorizing the comments of java methods based on their text only. Both trained a machine learning model for binary classification (coherent or not coherent).
- Steidl et al. (2019) defines several different top-level metrics to assess the quality of C++ comments [20]:
  - **Coherence**: How comment and code relate, i.e. method comments should contain method names. Coherent comments should provide details and explain the non-obvious.
  - **Usefulness**: Clarify the intent of code and be helpful. Deleting the comment should make the code harder to understand
  - **Completeness**: Global metric of coverage. Every file should have a license comment etc.
  - **Consistency**: Global metric of how uniform the comments are: Same language and same format of license comments in all files.
- D. Steidl (2012) [16] uses the same classification metrics in her master thesis, but focuses a lot more on usefulness, providing ways to assess it using the amount of question and exclamation marks in a comment: **Question marks** confuse readers instead of clarifying the code.

[1] https://www.oracle.com/ch-de/technical-resources/articles/java/javadoc-tool.html#:~:text=Writing%20Doc%20Comments-,Format%20of%20a%20Doc%20Comment,%40return%20%2C%20and%20%40see%20.

Therefore useful comments shall never contain question marks. As a metric for helpfulness, they deducted similarly, that an **exclamation mark** in a comment indicates poorly written code, personal involvement of the author and needs manual investigation. While they are helpful as to warn to user of hacky code, they should rather be tagged as a task comment.

- The paper of Moreno et al. (2013) proposes a tool for the automatic generation of natural language summaries for java classes [21]. For evaluation of the application, the generated comments were rated manually, based on three quality metrics by 22 programmers:
  - **Content adequacy**: Is all important information reflected in the comment?
  - **Conciseness**: Is there any information that could be omitted?
  - **Expressiveness**: How readable and understandable are the comments?

For this study, we mainly oriented our selves on the work of D. Steidl et al., regarding the definition and assets of quality in comments. This is because the studies are very precise in their explanations of the methodology used and extendable to be applicable to multiple languages and comment types. The quality assets coherence, usefulness, completeness and consistency are well defined and could be used to describe the other metrics of related work, as it is quite broad. For example, Khamis' WPJC can be mapped to completeness, or Moreno's conciseness can be mapped to coherence.

In addition to adopting Steidl's taxonomy, we put any metrics that indicate how easy a comment is to read into a new top level category named **readability**, as suggested by the future work section in their study [16]. Another novelty to the thesis at hand, is that we propose an approach for assessing comment quality on a set of programming languages, namely Java, C, C++ and C#.

We will introduce and further discuss the exact metrics, scores and calculations used, concerning the above-mentioned quality assets, in the section 4.2.

Chapter 3

Comment Type Classification based on Deep Learning

3.1 Overview

This chapter of the thesis is dedicated to the first research question: *Does Deep Learning perform better than Machine Learning approaches in classifying code comments?*. The section 3.2 will explain how the pipeline was built, which choices were made and the means of evaluation of the results. The latter will then be presented in the section 3.3. Finally, we will report the interpretation and discussion of the results to the above-mentioned research question in section 3.4.

3.2 Methodology

When setting up the Machine Learning pipeline for the first research question, there were five topics to agree upon that are discussed in the coming subsections. The figure 3.1 serves as an overview of the steps that were taken in order to set up a environment to answer the first research question.

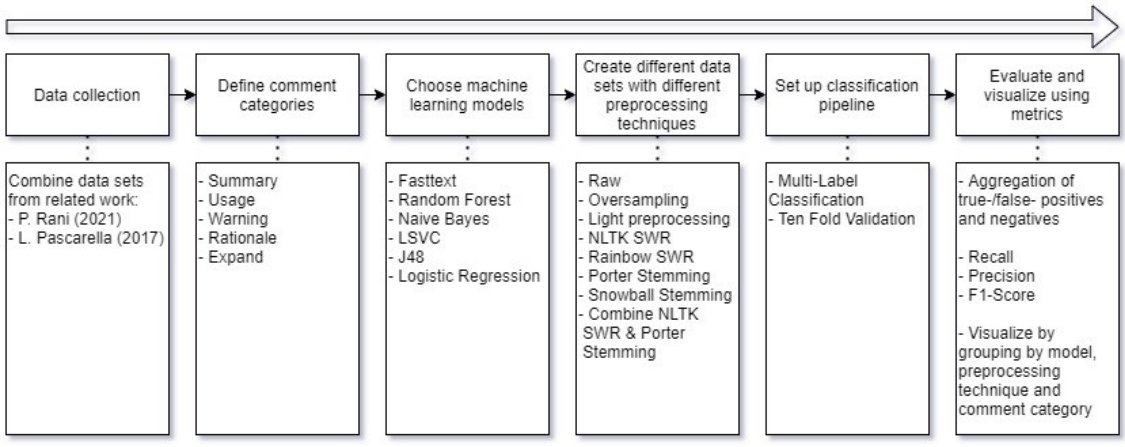

Figure 3.1: Overview of our approach to setup the pipeline to answer RQ1

3.2.1 Data Collection

Every Machine learning approach needs a proper data set. Luckily there are several readily available from related work. In this case, there was a combination of [15] and [13] used. This resulted in a labeled data set with 7,429 entries after reducing the labels to the after mentioned ones. Both data sets are labelled collections of java comments from real repositories.

3.2.2 Code Comment Categories

After combining the data sets, there were several resulting different code comment labels: *class, comment, summary, expand, rationale, deprecation, usage, exception, todo, incomplete, commented code, directive, formatter, license, ownership, pointer, auto-generated, noise, warning, recommendation, precondition, coding guidelines, extension, subclass explanation, observation.*

However, for this thesis we were interested in 1) Only the five most represented ones, as machine learning performs better the more data and the fewer labels are provided and 2) Labels that are neither all tagged in the comment itself, nor not rich with natural language. Examples for the latter were license, formatter, pointer and noise comments. In the end we decided on the small list of:

- **Summary**: Comments that contain a brief description of *what* the code does. This is by far the most represented label in this data set and the most represented type of comment in software projects period.

- **Expand**: Similarly to summary, expand comments explain *how* the code works. Contrary, they expand on the code, showing details that are not obvious from reading the program, opposed to summarizing what is written in the code.

- **Usage**: Usage comments give insight to how to use the code properly (e.g. parameters of functions). The usage comments in the data set of [13] are all annotated (@param, @usage, @return, etc.), while the ones in the data set of [15] are not, resulting in a mixture that still poses a challenge to any classification.

- **Rationale**: Comments that explain the reasoning behind a choice or approach in the code. It answers the question *why* the code is written as it is.

- **Warning**: As the name suggests, warning comments warn the user about edge-cases and help prevent misuse. It is worth noting that the data set of [13] does not contain an explicit *Warning* label, but we mapped the *Deprecation* category to it for the merge, as it originally *warns* the user about using a deprecated functionality.

The pie chart 3.2 shows the distribution of the resulting labels. The most represented one is *summary* with 4539 entries, while *Warning* is the least represented one, only consisting of 85 comments.

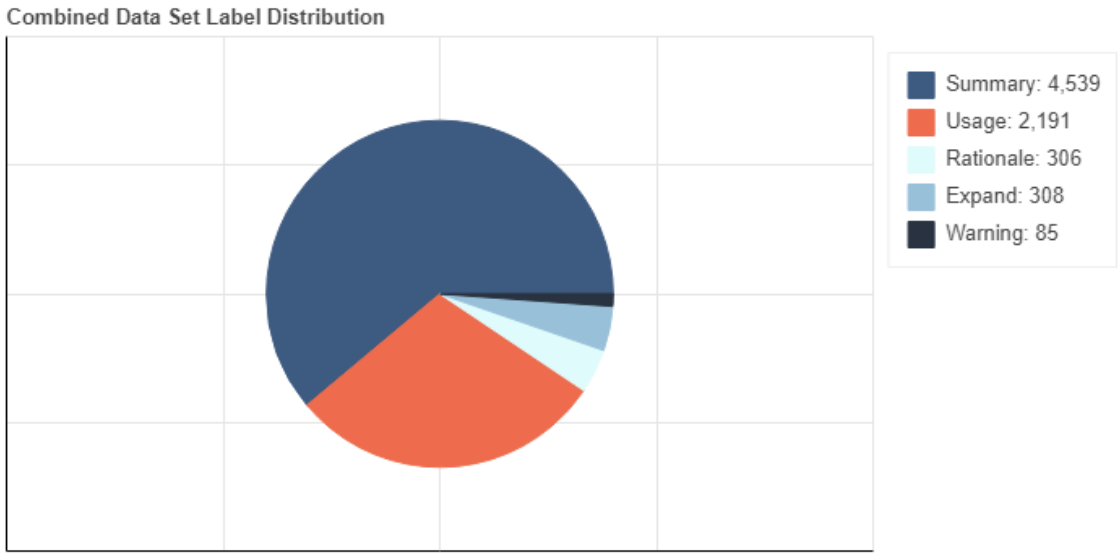

Figure 3.2: Distribution of different labels in combined data set.

3.2.3 Models and Classification

The model we chose to go with in this thesis is **fasttext**: An open-source and lightweight python framework that allows users to create text classifiers through supervised learning, developed by Facebook's AI Research (FAIR) lab. It uses a neural network for word embedding, which suits our cause of answering, if a Deep Learning technique is able to classify comment types consistently.

For direct comparison and discussion, the pipeline is also able to learn on a plethora of different supervised classification models:

- **Random Forest**: A model which constructs a multitude of decision trees (100 in our case) when training. During classification, the result is the label that was selected by the most trees.
- **Naive Bayes**: Naive Bayes is a simple probabilistic classifier based on applying Bayes' theorem. It is naive because it has strong independence assumptions between the features when minimizing the probability of misclassification. In this thesis, the multinomial variant was used to cater to our needs.
- **Logistic Regression**: A model, which is also based on the concept of probability. It uses a sigmoid function for estimating cost between 0 and 1.
- **Linear Support Vector Classification (LSVC)**: A faster Support Vector Classification for the case of a linear kernel, which suits our classification problem and accelerates training.
- **J48**: J48 is a decision tree based, statistical classifier using the concept of information entropy. It is originally a Java implementation of the C4.5 algorithm and popularized by WEKA, a Java framework for machine learning.

The implementations are provided by the python package **scikit-learn**, which is built on NumPy, SciPy, and matplotlib. Contrary to fasttext, these models need a separate TF-IDF word vectorizer. In this case it is set up with 1000 maximum features and a minimum and maximum

data frequency of 1.0. All algorithms used were executed with their default settings and evaluated using a randomly sampled ten-fold validation loop.

As for the classification itself, a multi-label binary approach was used. The labelled data set is split into one for each of the five categories. This means in the resulting temporary data set for *Summary* all non-summary entries are labeled as *Other*, leading to a binary classification problem (either summary or not). As a next step, the algorithm trains a separate model for each binary data set, resulting in five models (one for each category). When predicting an unknown label, all five models are asked to output their probability and the highest positive one is the resulting prediction and probability for the comment's text.

3.2.4 Preprocessing

In this thesis, we not only compared different models and their performance but also the impact of different kinds of preprocessing. This resulted in eight differently processed data sets:

- **Raw (no processing)**: The almost raw data set, with only any new lines and duplicate white spaces being removed.
- **Oversampling**: A data set where the under represented categories have been repeatedly randomly sampled until they have as many entries as the most represented label (summary). All further data sets are oversampled.
- **Light Preprocessing**: For this data set, a few regex based substitutions and changes have been made using the following patterns and in this order:
  - Remove any numbers:

    ```
    \d
    ```

  - Remove Special characters:

    ```
    [\-!$%^&*()_+|~='{}\[\]:\";´<>?,.\/]
    ```

  - Splitting camel/dromedary case into separate words:

    ```
    ((?<=[a-z])[A-Z]|(?<!\A)[A-Z](?=[a-z]))
    ```

  - Turn everything to lower case.

  The following data sets also were lightly preprocessed in addition to the mentioned technique.

- **Stop Word Removal (SWR)**: In this step, we removed any English stop words. These are words that occur very often but do not hold any value for word-by-word classification. Some examples include: "the", "is", "in", "for", "where", "when", "to", "at" etc. The sum of those words make up the stop word list. For comparison's sake, there were two different lists used in this thesis:
  - **Natural Language Toolkit (NLTK)**: Python's most popular natural language processing package comes with its own stop word list. It is very widely used in related work that uses a Python implementation.
  - **Rainbow**: Another popular stop word list based on Rainbow statistical text. This list is mostly used by Java developers as it is part of the WEKA API.

- **Stemming**: Stemming is the process of reducing words to their root form, in order to reduce the amount of unique words in a text and in theory reach a better classification accuracy after training. There are several different ways and algorithms for stemming in English, so two different implementations from NLTK were used in the pipeline's evaluation:
  - **Porter stemming**: The most popular stemmer known for its non-aggressiveness. However, it is also a rather old concept and more computationally intensive than its competitors.
  - **Snowball stemming**: This algorithm is also called Porter2, as it is based on Porter, but with slightly faster computation time than porter, and a fairly large and growing community around it.
- **Porter stemming & NLTK SWR**: As a final dataset, we combine SWR and stemming in a single data set, as is usual with preprocessing pipelines.

3.2.5 Evaluation Metrics

The last thing to define is which metrics to use when discussing and evaluating the performance of the models in conjunction with the data sets. As the classification problem is split into five binary ones, we can calculate a confusion matrix for each label and model. For this, precision and recall are calculated by aggregating all true positive, false negative and false positive predictions of each models in each fold:

$$\text{Precision} = \frac{\text{true positives}}{\text{true positives} + \text{false positives}}$$

$$\text{Recall} = \frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$$

Precision answers the question *How often is the classifier correct, when it predicts positive?*, while a high recall helps minimizing false negatives. Regarding visualization and discussion of results, the F1 score suits our needs best, as it combines precision and recall in one metric to showcase accuracy like so:

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

The F1 score of a classifier is considered perfect when it hits 1.0. This translates to it correctly identifying labels and you not being disturbed by outliers. Contrary, the model is considered worse, the closer its F1 score is to 0.0.

3.3 Results

In this section, we will show multiple plots that are the result of several ten-fold validations to compare the performance of different models, preprocessing techniques, and oversampling. The same figures but upscaled for further inspection can be found in the appendix chapter 8.

In figure 3.3 we can clearly see that fasttext outperforms the other models by considerable margins in regard to the overall F1 score, followed by Random Forest, Logistic Regression, LSVC J48 and lastly Naive Bayes.

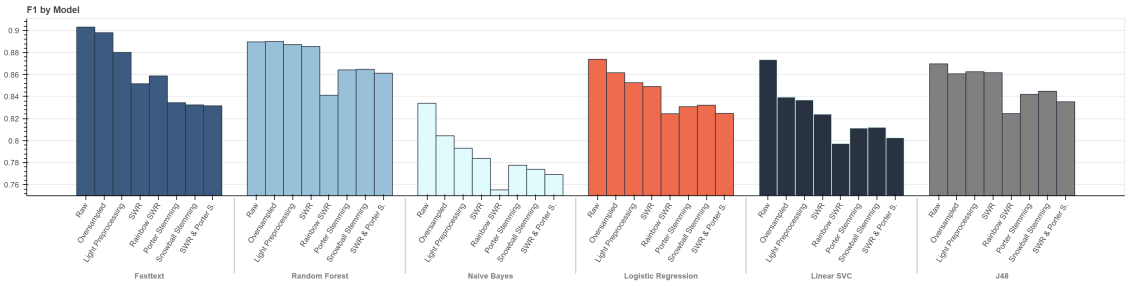

Figure 3.3: Comparison of F1 score of different models

While fasttext was the most accurate over all, this is not always true when looking at specific data sets and preprocessing techniques in isolation, illustrated by the plot 3.4: The more preprocessing, the worse fasttext performs and is gradually outclassed by the other models. However they never outperform Fasttext's peak accuracy on the raw data set, no matter the amount or kind of preprocessing.

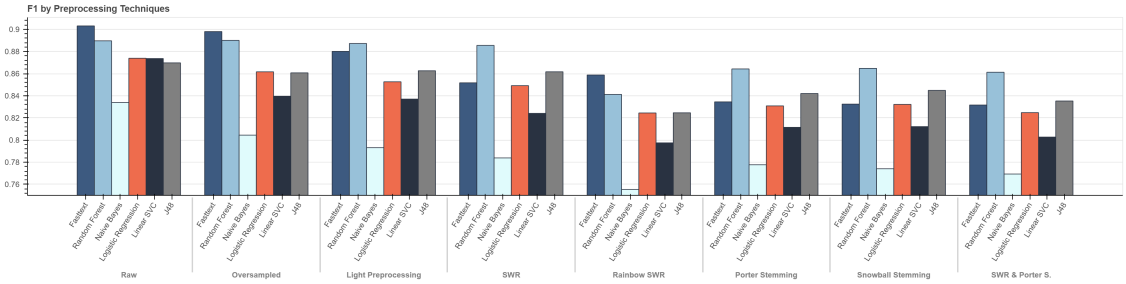

Figure 3.4: Comparison of F1 of different preprocessing techniques

The following figure 3.5 illustrates the effects of oversampling split up by label and model. The dashed bars represent the oversampled data sets, while the normally colored ones are trained on the unprocessed comments. One can observe how oversampling has a negligible negative impact on the highly represented labels, while boosting the underrepresented ones quite a bit.

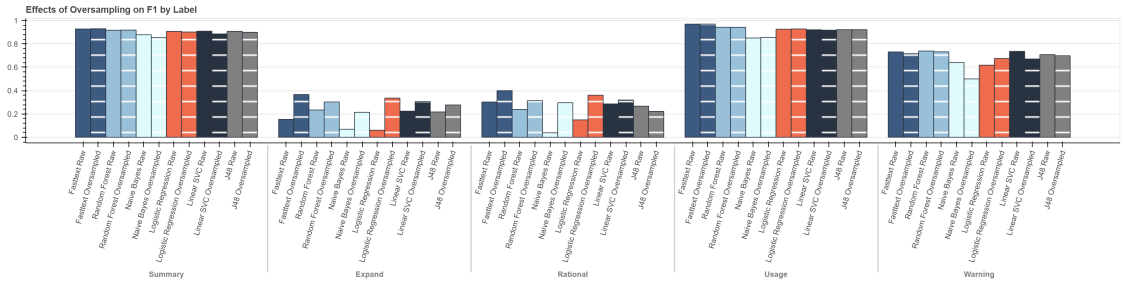

Figure 3.5: Effect of Oversampling on Labels

It is also worth noting that the average ten-fold validations took about one minute, with Fasttext being in the lead at around 40 seconds, true to its name. Random Forest is a big outlier here, as it took almost 50 times as long to train and evaluate with the given settings. The pie chart 3.6 shows the relative differences in processing time of the different models on the author's computer with the specifications showcased in the table 3.1.

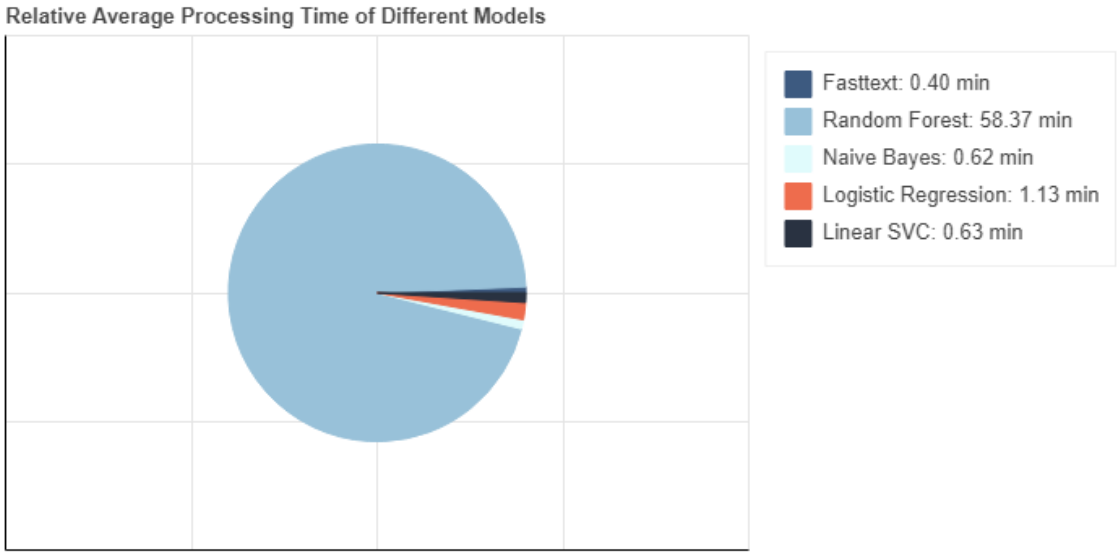

Figure 3.6: Relative Processing Time of Different Models

Table 3.1: System specifications of the used computer for reference of processing speeds

System Specifications	
Processor	AMD Ryzen 5 3600 6-Core 3.59 GHz
RAM	16 GB
Graphics	NVIDIA GeForce RTX 2060
OS	Windows 10 Home 64-bit, Build 19043.1165

3.4 Discussion

The graphs and data from the results section 3.3 gives plenty of material to discuss. In this section, we will evaluate if research question one can be answered and discuss interesting findings. They can be summarized into four main takeaways:

3.4.1 Performance of Deep Learning

We can answer the research question one *Does Deep Learning perform better than Machine Learning approaches in classifying code comments?* with a yes, when it comes to overall accuracy and with fasttext as an example for Deep Learning. However, Random Forest and J48 can perform better than fasttext under certain preprocessing conditions such as SWR and Stemming. While this might be the case, Fasttext clearly performed best, while having the shortest computing time and needing next to no processing of the data set. This makes it a very versatile, quick, and easy to use tool that might be of value in future work.

In related work, studies have reached higher accuracy with smaller data sets, by using set heuristics and features that suit their certain data set and classification tasks. In order for fasttext to be able to hold its candle to those algorithms, we would need a much bigger data set, which Deep Learning is notorious for. But since this data must be labelled and validated by a human being, creating such a data set would be very costly, time intensive and for most studies not feasible.

3.4.2 Processing Time and Model File Size

Moreover, it is worth commenting on Random Forest performing better than the other scikit-learn models. It is not clear if this advantage is gained by using a superior algorithm or due to the fact that it trained 50 times longer than the other models. This raises the question if models with a bigger file size that trained longer would change the rankings, as there were no limits set or attention given on model size or training time in this paper and comparison. However, would model size also have been a consideration for performance, the contest would probably still lean in fasttext's favour, as it is also known for its small models, fast training speed, and potent compression, if one wanted to save even more space [22].

3.4.3 Effects of Oversampling

Another finding is the effect of oversampling on underrepresented labels. Looking at figure 3.3 one might think at first glance that oversampling has a negative impact on performance: in four out of five algorithms we see a drop-off in F1 score from the raw data set to the oversampled one. On the contrary in figure 3.5, we can see that the summary and usage comments are almost not affected at all, while the expand and rationale comments doubled or even tripled in performance due to oversampling. Therefore, why does the overall performance decrease? This is due to the fact, that the aggregated F1 score in figure 3.3 is weighted. Summary and usage together have more than ten times bigger representation of the remaining labels combined. This means that the slightest decrease to performance of the labels with the most comments (which is hardly visible in figure 3.5) out-weights the big increase in accuracy of the other labels. This leads to the conclusion that oversampling can help increase accuracy in comments with low representation in the data (and therefore real life), at the cost of overall weighted accuracy. This decrease is in our case worth the trade-off, as we are interested in e.g. expand comments just as much, as we are in summary comments. Anything to help combat the greatly and inherently biased data set is welcome in our case.

3.4.4 Preprocessing Decreases Accuracy

The last point of discussion is that preprocessing consistently worsens the accuracy across all models and data sets. While we can explain the decrease of the F1 score when oversampling the data, other preprocessing techniques used should in theory increase performance. One possible

explanation is that fasttext, in particular, is primarily made for natural language text without preprocessing and functions best with it. Techniques such as stemming and stop word removal remove several natural language properties, which seem to worsen the performance.

Another explanation could be that the models trained on the raw data weight special characters a lot. This leads to the algorithms associating usage comments, for example, with certain special characters. These then get lost in the light preprocessing data set when removing all special characters, making different comment types harder to distinguish, and ultimately decrease accuracy.

The initial results are promising and a more accurate classification than with traditional approaches seems within reach, given a bigger data set.

Chapter 4

Comment Quality Analysis

4.1 Overview

This upcoming chapter is dedicated to answering the second research question: *How to systematically assess code comment quality in different program languages?*. We will first talk about the methodology of our approach in section 4.2. This is also where the reader is introduced to the metrics and scores to assess code comment quality, how the tool proposed in this thesis gets the comment data, evaluates and aggregates it for a presentation. We will also discuss how we selected the repositories for the comment evolution analysis plots in the section 4.3. In the last section 4.4, we will discuss our findings concerning RQ2.

4.2 Methodology

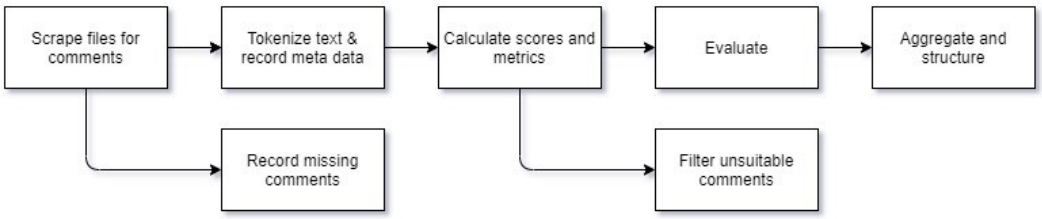

Figure 4.1: Pipeline setup for assessing code comment quality.

This section shows how the quality assessment pipeline is built for this study. Figure 4.1 shows its flow from left to right and acts as an overview. For more detailed information about the implementations in the proposed pipeline, refer to the chapter 5. The following subsections then explain or discuss certain steps (such as the scraping of comments) of the pipeline or the choices made to create it (such as the metrics and scores).

4.2.1 Metrics and Scores

In order to analyze, evaluate, and discuss the quality of a comment, one needs to define metrics and scores to describe the different aspects of quality. As is shown in the related work section 2.2, studies do not have a definite consensus on how to define and measure quality in code comments. In regard to the thesis at hand, we focus on readability, usefulness, coherence, completeness, and consistency of comments to measure their quality. For further analysis, visualization, and discussion, the tool also records meta-data about the comments in a project. In this subsection we are introducing the mentioned quality attributes and metrics to express them, while giving insight on their calculation or acquisition.

Meta Data

The following metrics do not hold any numeric values. They are for identifying a certain comment and provide some background information about its contents.

- **Path**: This attribute holds the file name and path of the file where the comment is found.
- **Position**: The position describes the location of a certain comment within its file. It consists of the line and column number of the first character of said comment, separated by a colon.
- **Type**: The type describes which kind of comment you are looking at. The type value can either be *class, function, constructor, interface, enum* or if none of those apply, a comment is classified as *in-line*. This depends if the comment precedes a declaration of the corresponding type in the code of the file. Comments that precede other comments are merged into a single one.
- **Label**: This value is set by the pipeline created to answer research question one. It classifies a comment based on its text as either *summary, usage, warning, rationale* or *expand*. The prediction probability is also recorded, to be able to discard any classifications under a certain threshold.
- **Text**: This field holds the unmodified and raw textual contents of a comment.
- **Code Language**: A value to describe the language of the code. It is applied on a file scope based on srcML's parser, meaning it belongs to one of *C, C++, C#* or *Java*. This will be expanded in the section 4.2.2 about scraping comments.
- **Commented Code**: A true or false value if a comment contains any commented code. While it is a rather bad practice, developers often keep old code around inside a comment "just in case" [23]. As they do not consist of natural language, we need to be able to filter them before applying any evaluations. In order to do that, a basic approach was used by us to match certain special characters with a regex like so:

```
"=+&|;
```

  If more than three of those characters that are common in all of the programming languages we consider are found in the comment, it is classified as commented code.

  It is worth mentioning that this approach is far from perfect, as there definitely exist natural language texts that might be wrongly accused of being commented code. The same is true for the opposite: Commented code exists that is not caught by this approach, especially simple or short statements such as variable declarations. The approach proposed by D. Steidl (2012) of matching java method call patterns and similar would be problematic in

our case, as we want to make the pipeline assess quality regardless of programming language [16]. Trying to detect commented code with a higher accuracy by checking if every comment's content could be executed, was just way outside of this study's scope and time frame. However, future work would definitely profit from a more advanced solution.

Readability

A high readability makes a comment easier to read and therefore understand. Several of the following proposed metrics need information such as the amount of words or sentences. This is not as easy as splitting a string on every period for sentences or white spaces for words, which would be a too naive approach. Instead let us take a look at how we get the amount of words in a string:

1. The tool uses the following regex to get all raw words:

   ```
   ([a-z][-'a-z]*)
   ```

   The words we consider for this thesis consist only of letters, hyphens (a dog-friendly hotel) and apostrophes for indicating omitted letters (I've coded), possession (the man's dog) or the plurals of lowercase letters (p's, x's, etc.).

2. Then we filter all matches without any vowels, as words without vowels do not exist in the English language besides abbreviations and onomatopoeia (e.g. shh, brr, tsk, etc.) with the following regex:

   ```
   [aeiouy]
   ```

   Note that "y" is considered a vowel in some words such as pygmy and lynx, contrary to German.

3. Lastly, all words that contain a hyphen "-" are checked for their validity. Valid hyphens cannot be earlier than two letters into a word and need at least two letters after them. This is matched with:

   ```
   [a-z]{2,}-[a-z]{2,}
   ```

For tokenizing sentences and counting them, NLTK's sent_tokenize() function was leveraged, which takes any string (in our case the comment text) and a target language (English) and splits it into sentences consistently.

- **Flesch Reading Ease Level (FREL)**: FREL is a readability metric on a 100 point scale. The higher the score, the easier it is to understand the text. A score of 60 to 70 is considered to be optimal. Other ranges and their meanings are illustrated in table 4.1. The reading ease level itself is calculated as follows:

$$\text{FREL} = 206.835 - (1.015 \cdot \text{Words per Sentence}) - (84.6 \cdot \text{Syllables per Word})$$

Table 4.1: Flesch Reading Ease Level Score Overview

Score Range	Description & Level	Examples
0-10	Extremely difficult: Professional level.	Internal Revenue Code
10-30	Very difficult: University graduate level	Standard auto insurance policy
30-50	Difficult: College level.	Wall Street Journal, New York Times
50-60	Fairly difficult: 10th to 12th grade level	Time, Newsweek
60-70	Plain English: Easy to read by most 13 to 15 year olds.	Sports Illustrated, Reader's Digest
70-80	Fairly easy: 7th graders and up.	Movie Screen
80-90	Easy to read: Conversational English of 6th graders.	Consumer ads in magazines
90-100	Very easy: Average 11 year olds understand this.	Comics

- **Flesch Kincaid Grade Level Score (FKGLS)**: Is another readability score, which represents an U.S. grade school level of difficulty to read. A score of 8.0 means that the document can be understood by an eighth grader, for example. An optimal score lies between 7.0 and 8.0. It is calculated with using the following formula:

$$\text{FKGLS} = (11.8 \cdot \text{Syllables per Word}) + (0.39 \cdot \text{Words per Sentence}) - 15.59$$

- **Fog Index (FI)**: This is a metric indicating the number of years of formal education a reader would need to understand the text on the first reading. The fog index must not be above 12 and score between 7-8 is considered optimal for readability.

$$\text{Fog Index} = (\text{words per sentence} + \text{complex words percentage}) \cdot 0.4$$

  Complex words encompass any words with more than two syllables. In order to calculate the number of syllables in a word, another regular expression approach was used:

  1. Since we only care about the number of syllables and not the exact syllables themselves, words are split using:

```
[bcdfghjklmnpqrstvwxz]*
[aeiouy]+[bcdfghjklmnpqrstvwxz]*
```

  2. In a second step we need to handle the edge-case that the trailing syllable is a lonely "e", which is always silent and cannot be a syllable. It is therefore appended to the second to last one and omitted.
     - sitting $\rightarrow$ sitt | ing = 2
     - language $\rightarrow$ lang | uag | e $\rightarrow$ lang | uage = 2

  It is worth noting that this approach is only an approximation and not fool-proof in certain scenarios, especially when a vowel forms its own syllable. For example the word "area" would match as two syllables, but should actually be three. Future work would profit from a safer and less naive method.

- **Number of Abbreviations**: Research agrees that abbreviations make it harder to read and understand any text. Therefore the tool matches any used abbreviations using a list provided by the Oxford fourth edition classical dictionary[1]. Matches are recorded and counted and substitutions for the full words are recommended.

[1] https://public.oed.com/how-to-use-the-oed/abbreviations/

Usefulness

Useful comments clarify intent of the code and are helpful to the reader that tries to understand the code. Deleting a useful comment should make the code harder to understand.

- **Number of Question Marks**: Research deducted from a survey that question marks confuse readers instead of clarify [16]. Therefore useful comments shall never contain question marks. For this metric they are simply counted.
- **Number of Exclamation Marks**: As a metric for helpfulness, the same study deducted similarly, that an exclamation mark in a comment indicates poorly written code, personal involvement of the author and needs manual investigation [16]. While they are helpful as to warn to readers of hacky code, they should be tagged as a task or todo comment.
- **Length**: Related work agrees that a comment that consists of less than three words is not useful and should be expanded or deleted. Similarly a comment might be too long: Studies indicate that comments above 30 words are not as useful. This upper limit mainly applies to *in-line* comments, as other types such as class comments and license comments can reach this limit easily in a complex project. These too long in-line comments can also be an indicator for the need of extracting its corresponding code into a separate method or function.

Coherence

Coherence describes how comment and code relate. A comment that is unrelated to its code does not hold any value. It also can be *too coherent* if for example a comment just repeats the method name without explaining or expanding the code. For example:

```
// Calculates Square
public int calculateSquare(int n) {
    return n*n;
}
```

- **Coherence Coefficient**: This metric matches the words in a comment and its corresponding handle of the following code element (e.g. class comments matched to class names, function comments matched to function names). The words in the given handle are extracted by splitting the camel case of the name using the method described in the preprocessing subsection in 3.2 and casting both the comment text and handle contents to lower case. In order to calculate the coherence coefficient, we need to calculate the Levenshtein distance between each words of the two strings. It describes the minimum amount of additions, deletions and changes of characters to transform one string into another, resulting in a metric for similarity. For example the Levenshtein distance between *sitting* and *kitten* is 3:

  1. sitting $\rightarrow$ sittin
  2. sittin $\rightarrow$ sitten
  3. sitten $\rightarrow$ kitten

  Next, we count the similar words. A word is similar to another if the Levenshtein distance is less than two. This means our two example strings would not be considered similar, but *ball* and *wall* would. In the end the coherence coefficient is calculated as follows:

$$\text{Coherence Coefficient} = \frac{\text{Number of similar words}}{\text{Total words in comment}}$$

It follows that a coherence coefficient of 0.0 indicates that the comment and name of code element have no similar words and nothing in common. This leads to a coherence that is too low. To rectify this, one could expand the comment, in order to make the relation to the method name more obvious. Another interpretation is that the code element (e.g. function) needs to be refactored into a different name that fits its explanation in the comment more. On the other side of the spectrum, if the coefficient is higher than 0.5, a comment is too trivial as it adds little explanation of the code and almost only contains the code element's name.

Completeness

Completeness is a global metric that describes the total coverage of comments. *Global* means here that one can not evaluate completeness by looking at a comment in isolation. In theory every function, class, etc. should be commented and all files must contain a license comment at the top. The more of those comments are missing, the less the comments of a project are complete and the lower its completeness.

- **Missing Comments**: While the program is analyzing existing comments, it also considers any *class, function, constructor, interface or enum* that is not preceded by a comment or any files that do not start with a comment. It is then recorded as a missing comment, together with its supposed path, position, type (as in header, class, function, etc.) and name of the element without a comment (e.g. function name). How this was achieved, is explained in the following subsection 4.2.2.

Consistency

The second global metric that can not be evaluated on single comments and needs context. It describes how uniform the comments are in their presentation and nature. As an example, all license comment in a project should have the same layout and order of fields.

- **Synonym Analysis**: The purpose of this analysis lies within the idea that if one talks about the same thing, one should use the same words. This increases consistency in the comments and helps avoid confusion and misunderstandings.

```
public class Rectangle{
   // this attribute describes the rectangle's height
   int h;
   // this field describes the rectangle's width
   int w;
   ...
}
```

In the above example, the terms *attribute* and *field* are used to describe the same thing. To increase consistency and decrease the potential for confusion, the author should have used one of them in both comments.

To achieve this, the tool gets the synonym of every word of every comment in a file, using *wordnet*[2], a large lexical database of the English language that is organized in sets of cognitive synonyms, also called synsets. If a word in one comment is in a word's set of synonyms of another comment, the match is recorded together with a pointer to the other comment.

[2]https://wordnet.princeton.edu/

As the complexity of this algorithm is very high and processing time grows exponentially with the amount of files and comments, the pipeline offers the option to omit this analysis for large projects. This also is the case for the comment evolution results in section 4.3.

- **Consistent Language**: Another aspect of consistency is a consistent language across comments. It is highly advisable to have all comments be of the same natural language and not some for example in German and others in English. Analyzing and recording the language of a comment also serves a second purpose, as in order for some metrics and techniques to work properly, it is crucial that the comments are in English (e.g. English stop-word removal, word/sentence tokenization, synonym analysis, etc.). Luckily fasttext provides a free model for this task, trained on data from Wikipedia, Tatoeba and SETimes [24]. It is able to identify up to 175 different languages consistently. Similarly to the *label* attribute, we also record the probability for any predictions to make cuts at a threshold of 0.75.

4.2.2 Scraping Comments using srcML

One of our goals in developing this quality assessment pipeline was to make it accessible to anyone and easy to use. So in order to circumvent users having to create a data set or table of their projects' comments, the tool works on unmodified local project files. This means that the only requirement for anyone wanting to evaluate and analyze comments is to provide a path to any directory containing the project, files and comments.

The first approach considered to achieve this, was to read all files in a directory line by line, extracting comments by matching Java comment delimiters ("/**", "//","*", "**/" etc.). This approach however, poses several challenges and inconveniences. The following arbitrary Java code snippet covers a hand full of examples that a single regular expression could not tackle easily:

```
// foo
class SomeClass {
   /* foo1 */
   // /* foo2
   foo3();
   // foo4 */
   foo5();
   /* // foo6 */
   System.out.print("Is this a /* comment */ ?");
   int/* some comment */foo = 5;
   // /*
   foo7();
   // */
}
```

The maybe biggest con of such an approach is that we severely limit the amount of languages that can be processed, as most of them have unique syntax and semantics for declaring comments, further increasing the amount of edge-cases. Also it would make it very hard to extract information besides the comments' content. One such example would be that *//foo* is the comment for the class named *SomeClass*. The same problems apply for collecting classes, functions, interfaces, etc. that are missing preceding comments.

This is where **srcML**[3] comes into play: an open-source software for creating an XML format

[3]https://www.srcml.org/

for the representation of source code in Java, C, C++ or C#. It does this by parsing the Abstract Syntax Trees (AST) of a file by applying the appropriate language's grammar. During this conversion, nothing is lost and one can easily recreate and comprehend the original source code. The following is a snippet of our example edge-cases from above after being parsed by srcML:

```
...
  <comment type="line">// foo</comment>
  <class>
    class
    <name>SomeClass</name>
    <block>
      {
      <comment type="block">/* foo1 */</comment>
      <comment type="line">// /* foo2</comment>
     ...
      }
    </block>
  </class>
...
```

This srcML file solves all problems the first approach struggles with and comes with added advantages:

1. SrcML comes with the capability of handling multiple different languages.

2. We can consistently find a complete list of *all* comments in any language supported by srcML, without having to worry about edge-cases. This is because srcML uses the language's original abstract syntax grammar and simply allows us to find all <comment> elements in the XML representation.

3. Since the resulting tree can freely be walked up and down or inward and outward, finding any comment's corresponding element and its name is an easy task. This also allows us to classify a comment as a class-comment, function-comment or similar.

4. Finding classes or functions etc. without a comment above them in the source code, is as easy as checking the comment's preceding sibling element in the srcML tree. This method also makes it possible to merge comments that precede other comments into one, because in some styles, multiple single-line comments in succession are used instead of a singular block comment. For parsing the XML tree, the python native XML *xml.dom.minidom*[4] was used, as this minimal implementation of the Document Object Model (DOM) interface served our purposes while being especially lightweight.

5. SrcML has the functionality of taking not only files as input, but scraping directories, including their sub-directories and appropriate files, too. This eliminates the need for any recursive crawling of the local file system in our application itself.

6. The time srcML takes for this process of creating the XML format is negligibly small.

7. The application can be run such that all elements in the XML also hold a field for their position in the original source code, allowing us to record meta data for identifying a particular comment.

[4]https://docs.python.org/3/library/xml.dom.minidom.html

The only real down side to using srcML is having to use srcML: It adds an application that needs to be installed on the user's computer in order to run the pipeline and heightens the difficulty of the install process of the comment evaluator. In our case this was considered a valid trade-off.

4.2.3 Evaluating Comments

In the previous step, the tool scrapes for comments, tokenizes them and calculates most metrics. This gathered data is then saved as a .csv file. In order to interpret the quality of the comments, we need to give this raw information meaning through evaluation, which will be discussed in this section.

The pipeline at hand leverages the python native tool *pandas*[5] in order to read our created data. Pandas is perfect for this task, as it is a flexible, fast and powerful open source package. It allows the pipeline to manipulate, filter and extend thousands of rows in our table of comments within split seconds.

Filtering Comments for Evaluation and Aggregation

First, one needs to understand that there exist multiple reasons why one should ignore the scores of certain comments and not consider their values for aggregation and evaluation. The figure 4.2 illustrates an overview of this idea:

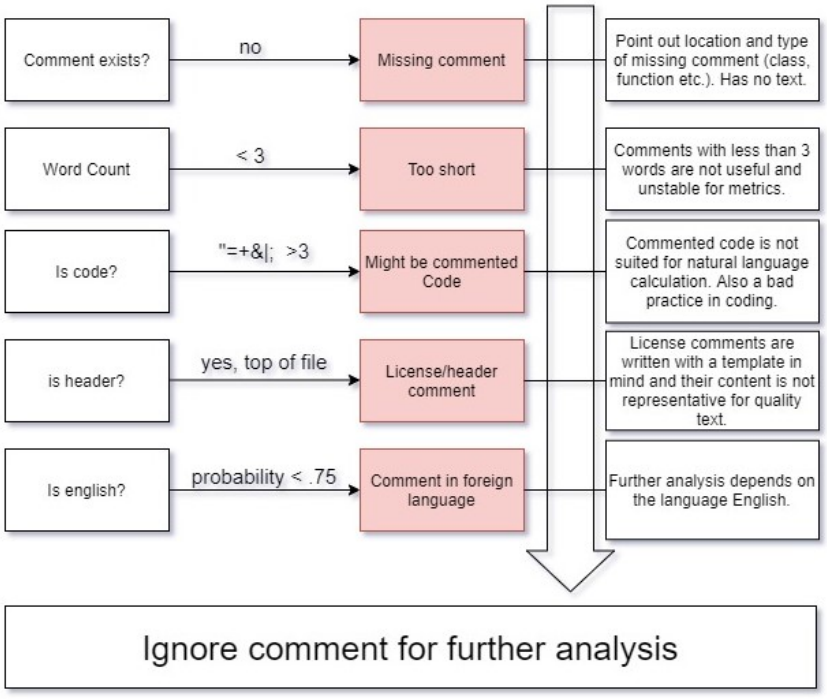

Figure 4.2: Values making a comment being ignored for further readability and usefulness assessment

An ignored comment should be omitted from any readability evaluation (FKGLS, FREL, FI).

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

The amount of question marks, exclamation marks, abbreviations and synonyms used are also of no concern. A comment shall be ignored, if any of the conditions in figure 4.2 are met:

1. If a comment in the data set is tagged as missing, it is obviously not eligible for any textual analysis, as it does not hold any text.

2. If a comment is too short, readability metrics become unstable and unreliable. For example, if we calculate the FKGLS of the simple (and bad) comment *//Run Tests.*, we get a score of -3.1.

3. Commented code should not be evaluated, as it contains no natural language under normal circumstances, but actual code. As they do not function as documentation, they also can never be tested for coherence etc.

4. Similarly, header or license comments should be ignored for the above mentioned evaluations. They are written after a template and do not represent the actual use of language of the developer. Taking the readability of license comments into account would also dilute the metric when aggregating, as they contain similar phrases to written law and contracts, which are notorious for their bad readability as shown in the table 4.1.

5. The last thing we need to keep in mind for evaluation, is that all comments should be in English. As mentioned in the section 4.2.1, the language of a comment's text is classified using fasttext and any comments that are not classified as such with at least 75% probability are ignored. This is a precaution because several metrics and tokenization techniques inputs in the English language to function properly. As a side effect, this filter also helps catch commented code, as the classification pipeline struggles classifying them.

Mind you that those ignored comments are *not deleted* from the data. For example, a comment that is too short might still be considered as English and summed up with other English comments in aggregation. It just wouldn't make sense to calculate certain scores for it and weight them in any evaluations.

The Meaning of the Metrics

Now that we have all this numeric data from our scores and filtered potential threats to a meaningful analysis, we need to give it meaning. Some metrics, such as the amount of question/exclamation marks, are easy to give meaning: the less the better, zero would be optimal. The amount of abbreviations and matched synonyms fall into the same category. However, most metrics need a little bit more interpretation as to their meaning:

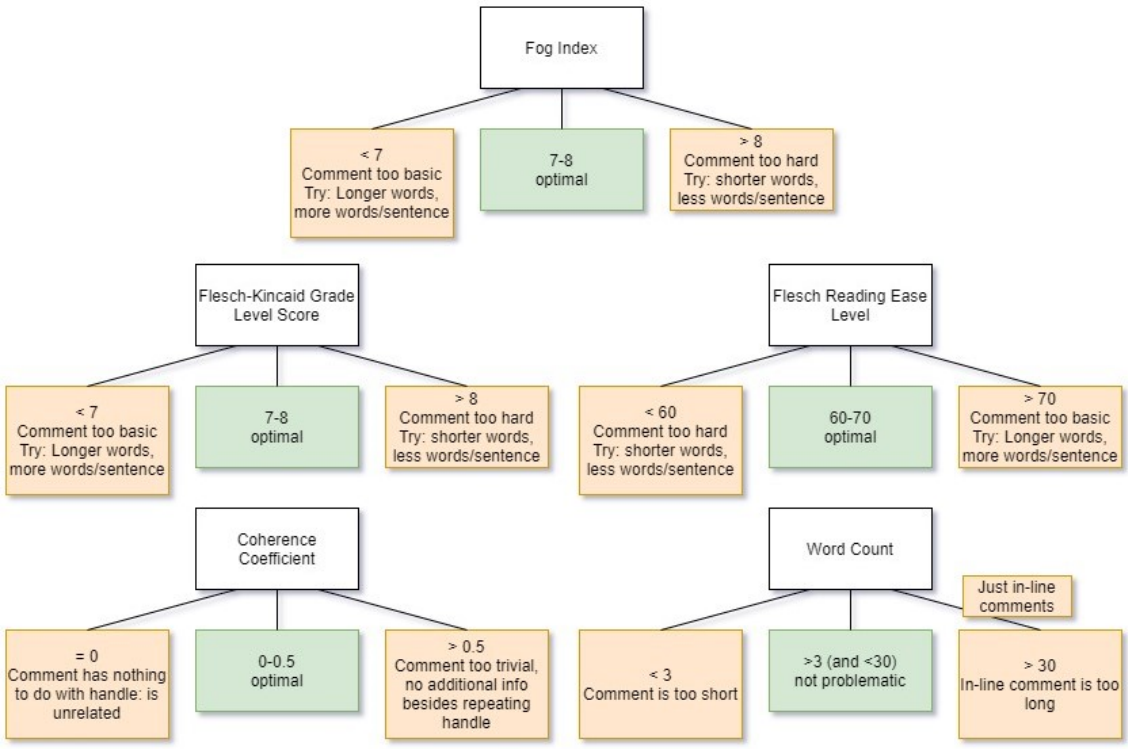

Figure 4.3: How to interpret the scores of the filtered comment data.

The optimal values depicted in figure 4.3 are introduced by Steidl et al. (2019) [20] and Dubay (2004) [25].

4.2.4 Aggregation and Presentation

The final step in our pipeline is aggregation and presentation. In order to allow future work to create a front-end visualization of the data, the tool outputs a big JavaScript Object Notation (JSON) file. This allows our results to be readable by humans, as well as offer easy integration for any JavaScript based implementation (REACT for example).

The idea for presentation was to mimic the local file system to create a direct link to the project files where the comments initially were extracted from. After we have calculated and evaluated all comment's data, we aggregate all comments in the same file. Next, the tool aggregates all files in the same directory. This is then recursively repeated until we have aggregated everything under the most top level directory of the analyzed project: This results in a tree with the root node being the root directory or input directory of the project. The figure 4.4 visualizes this idea.

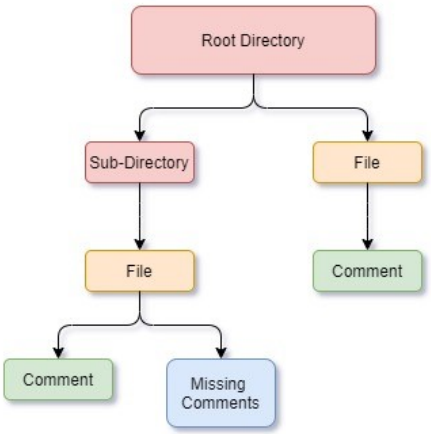

Figure 4.4: Directory tree structure found in the JSON file.

The resulting tree consists of 4 different elements or nodes:

1. **Comment**: This element holds all data gathered and evaluated in the previous section. It can only be a leaf node of the directory tree. It also holds the detailed information for some metrics that is omitted in aggregation. For example comment elements hold the complete dictionaries of matched synonyms or abbreviations, while the parent file node only sums up the amount of matches (N_matched_synonyms).
2. **Missing Comment**: Missing comments are similar to comment elements, as they act as leaves of the tree. However, they do not hold as much data.
3. **File**: File nodes hold the aggregated values of all comments in them under an array called *comments*. A file can only be the child of a directory or the root node. The field *name* holds the name of the file.
4. **Directory**: Similarly directory aggregates all values of its sub-directories and files under the field *children*. Directories can not be leaves, as directories without any files are of no interest to the analysis and omitted. The field *name* holds the name of the directory.

A complete break down of all elements in the resulting JSON and their fields can be found in the table 4.2.

When aggregating the comments' data under a file and files or directories under directories, some values are summed up and others meaned. For example, if two comments in a file have the fog index of 7.0 and 8.0, the field *FI* in the file node would say 7.5. Fields that describe values that have been counted are always described with the prefix *N_* and are summed up for the aggregation. This allows users to get a picture of the total amount of question marks in a file or directory or whole project, for example.

Values with the prefix *is_* are described with either 0 or 1 on a comment level instead of a truth value. This allows us to sum up and aggregate those values more easily. The same idea applies to *count* (all found comments have count = 1) and *count_missing* (all missing comments have count_missing = 1), which allows files and directories to have the fields *count* and *count_missing* to represent the amount of found comments and missing comments in them. The table 4.2 also shows which aggregation method has been chosen for which metric. Fields without a value in the column *Aggregation* are omitted in the aggregation and can only be found in the indicated node elements (shown with *X*).

Table 4.2: All JSON attributes in the resulting file, their occurences and their aggregation techniques.

Metric	Aggregation	Directory	File	Comment	Missing Comment
FKGLS	mean	X	X	X	
FREL	mean	X	X	X	
FI	mean	X	X	X	
is_english	sum	X	X	X	
is_code	sum	X	X	X	
is_too_short	sum	X	X	X	
is_too_long	sum	X	X	X	
N_matched_synonyms	sum	X	X	X	
matched synonyms				X	
N_exclamation	sum	X	X	X	
N_question	sum	X	X	X	
N_abbreviation	sum	X	X	X	
abbreviations				X	
is_trivial	sum	X	X	X	
is_unrelated	sum	X	X	X	
count	sum	X	X	X	
count_missing	sum	X	X		X
children		X			
comments			X		
type				X	X
path				X	X
name		X	X		
position				X	X
text				X	
code_language				X	
label				X	
label_proba				X	
handle				X	X
ignore				X	

4.2.5 Comment Evolution Repository Selection

In the upcoming results section 4.3 we will have a look at 4 different open source repositories off of GitHub. As we are still in the methodology section of this study, we take the opportunity to talk about how those projects were selected.

A list of conditions was set up, to be able to find appropriate repositories with GitHub's advanced search feature:

1. Get a project from every language the tool can handle for comparison. Namely: Java, C, C++ and C#. The repositories should be mainly written in one of said languages.
2. The project has recent activity (issues, commits, etc.) on the repository, namely in the last 3 months.
3. The repository must have had at least ten different releases/tags over its lifespan.
4. At least 1000 comments must be found in the most recent version with the tool.

5. The language for comments must be English. Several repositories were discarded for consideration, because the documentation was written in an Asian language.

6. After applying these filters, sort the search results by forks and pick the top one that is suitable and satisfies all previous conditions.

When executing the queries with the previously mentioned conditions, we ended up with a list of four repositories presented in the table 4.3.

Table 4.3: Overview of the chosen repositories for the comment evolution demonstration.

Repository	Language	Commits	Forks	Stars	Releases	Comments	Description
googletest[6]	C++	3.7k	7.8k	23.5k	15	7753	Testing and mocking framework
ijkplayer[7]	C	2.6k	7.6k	29.3k	78	1462	Android and iOS video player
retrofit[8]	Java	1.9k	6.9k	38.5k	54	1038	Android and JVM HTTP client
shadowsocksr-csharp[9]	C#	740	4.5k	14.3k	102	1617	Encryption protocol

4.3 Results

This section is dedicated to a demonstration of what can be done with the proposed quality assessment tool: We depicted a scenario, where a developer wants an overview of the evolution of his or her project's comments and their quality. In order to illustrate this, the tool was run on the latest ten major releases of the four repositories mentioned in the section 4.2.5 above.

As the evolution and change of quality metrics over time is the focus, any y-axis scale has been omitted in favour of multiple lines that fit in the same plot. This has been achieved by running the y-data arrays through the upcoming method in the plotting script:

[6]https://github.com/google/googletest
[7]https://github.com/bilibili/ijkplayer
[8]https://github.com/square/retrofit
[9]https://github.com/shadowsocksr-rm/shadowsocksr-csharp

```
def normalize(input_list):
    norm_list = list()

    if isinstance(input_list, list):
        sum_list = sum(input_list)

        for value in input_list:
            try:
                tmp = value / sum_list
            except ZeroDivisionError:
                norm_list.append(0)
                continue
            norm_list.append(tmp)

    return norm_list
```

As one can see, every value of a set of y-data for a plot has been divided by the sum of the values in this list. This lets us emphasize the changes and evolution in the values, without creating a dozen plots for each repository.

Said figures are containing four plots each:

1. **top-left**: This plot illustrates the changes in the amount of comments that are either too long, too short or missing in relation to the total found comments in this version.

2. **top-right**: In this plot, one can investigate the different readability metrics and their evolution over time. The blue and orange zones represent the optimal value range for FKGLS and FREL respectively.

3. **bottom-left**: The third plot shows the amount of found question and exclamation marks in relation to the total comments found over the different versions. This can be used to interpret the relative usefulness of the comments. The dashed line at the bottom indicates the goal of zero question or exclamation marks.

4. **bottom-right**: The last plot depicts the evaluated coherence coefficient and shows the amount of trivial and unrelated comments in relation to all English comments in a version of the project. Similarly to above, the dashed line represents the goal of zero trivial or unrelated comments.

The upcoming subsections will present and discuss the said figures. The same figures but upscaled for further inspection can be found in the appendix chapter 8.

4.3.1 Googletest

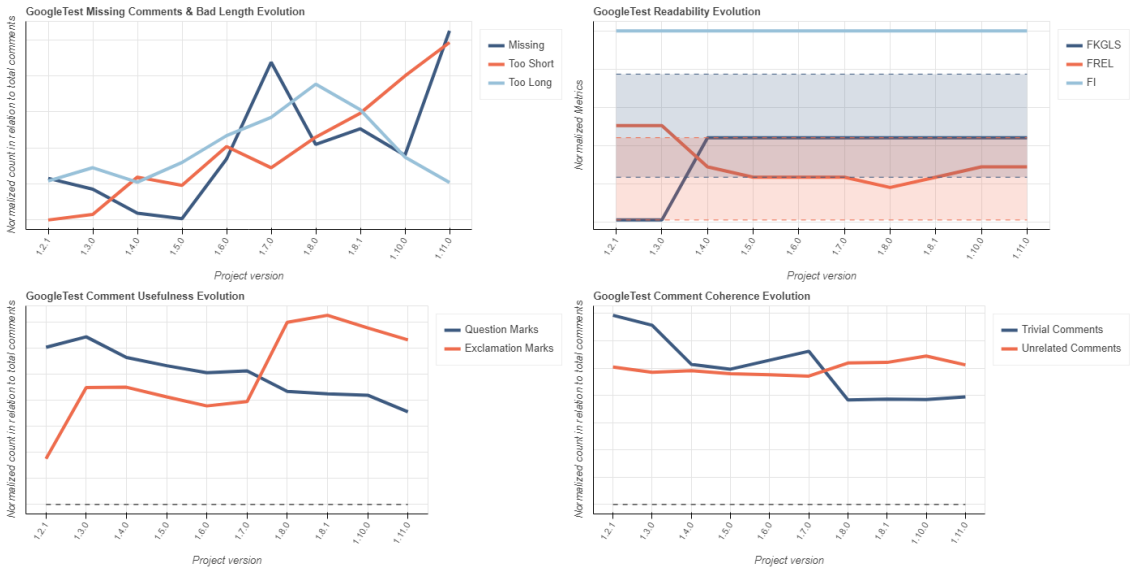

Figure 4.5: Evolution of Googletest's Quality Metrics

In the figure 4.5 above, we can see the comment analysis of the googletest repository. It shows a steady increase in the amount of comments that are too long, too short or missing entirely. Contrary to that, the readability is very constant and right in the optimal zone for bot FREL and FKGLS, which no other project in this comparison achieved. This can be connected to the fact that the repository is developed by Google, which follows very strict and expansive documentation guideline. It advocates for high readability and accessibility in comments and even states specifically: *Use shorter sentences. Try to use fewer than 26 words per sentence.* [26]. This guideline directly reflects in FREL and FKGLS calculations, which use the amount of words per sentence as a factor for readability.

4.3.2 Retrofit

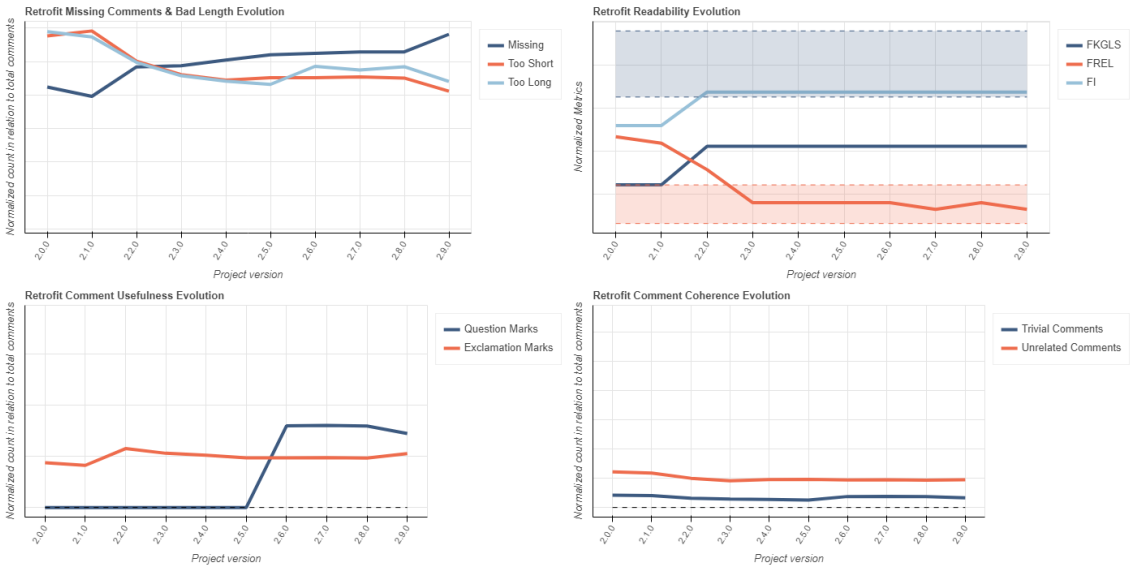

Figure 4.6: Evolution of Retrofit's Quality Metrics

The most notable feature of retrofit's comment quality plots is certainly its constantly high coherence. One can derive this from the low amounts of trivial or unrelated comments. This speaks for a very good and intuitive naming of code elements such as methods and classes without just repeating this name in the comment, which is one of the metrics for a high quality comment. The length of the comments and the amount of missing ones in this project are not its strength. Also the project's amount of question marks was raised considerably by the version 2.6.0, which should be revisited by the developers and corrected.

4.3.3 Ijkplayer

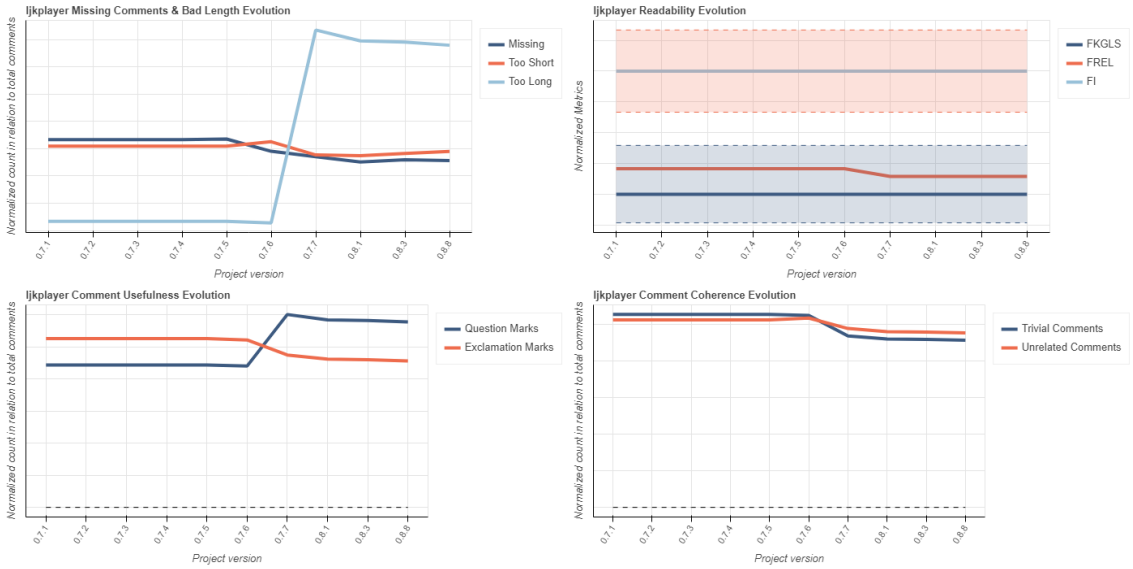

Figure 4.7: Evolution of Ijkplayer's Quality Metrics

Ijkplayer's evolution of their comment's quality looks remarkably constant in the above plot. This either means that newly added code was not documented, or that the developers very strictly followed quality guidelines, as every added comment has the same quality as the already existing ones. The only note worthy version that seemed to change up the quality of comments is 0.7.7, where a lot of too long comments were added (or normal/short ones were made longer) in relation to the total comments.

4.3.4 Shadowsocksr-csharp

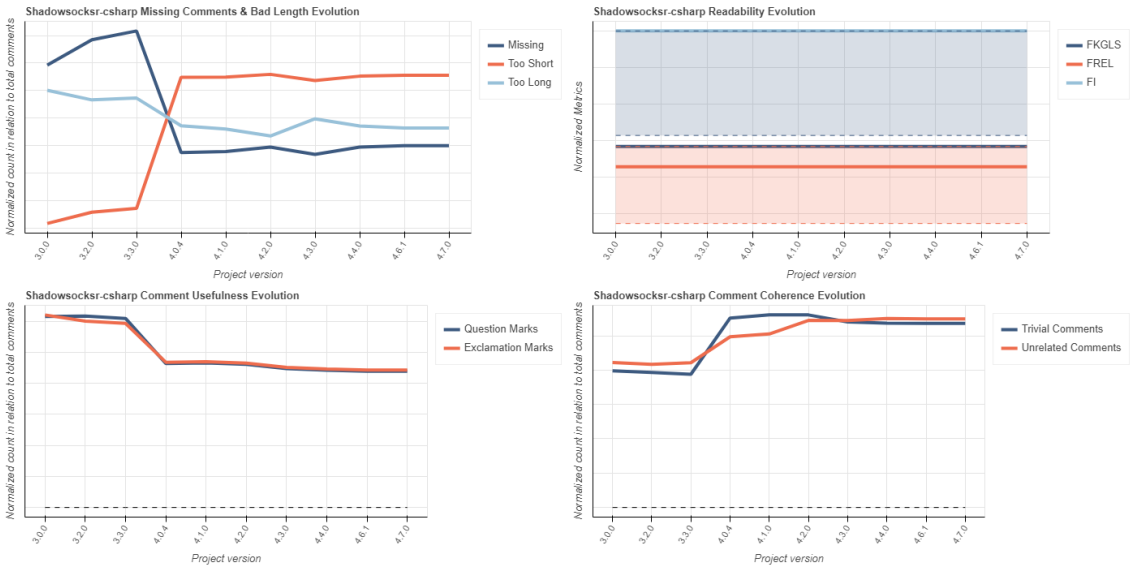

Figure 4.8: Evolution of Shadowsocksr-csharp's Quality Metrics

A note worthy thing about the above plot is the steady increase and then constant hold in the amount of question marks and exclamation marks. This can be explained with the reason that shadowsocksr is a truly open source project, maintained and developed by a lot of volunteering independent developers. As a consequence, several of those developers use comments to document issues or point others to problematic portions in the code: Comments such as *Do the version info bits match exactly? done.* or *Sanity check!* are common place.

In version 4.0.4 one or multiple of those contributors seemed to try to document some code that was missing comments. As a result, the missing comments have dipped a lot, as did the amount of question-/exclamation marks in relation to the total comments, indicating they refrained from using them. However, the comments added were probably rushed because they ended up too short in several cases.

4.4 Discussion

In our quest to answer the second research question *How to systematically assess code comment quality in different program languages?*, a tool was created that allows us to assess five major aspects of quality in code comments:

1. **Readability** through *Flesch Reading Ease Level*, *Flesch Kincaid Grade Level Score*, *Fog Index* and *Abbreviation Matching*.

2. **Usefulness** through *matching question and exclamation marks* and *analyzing word count*.

3. **Coherence** through a similarity analysis using Levenshtein distance to calculate and evaluate the *coherence coefficient*

4. **Completeness** by searching and recording *missing comments*
5. **Consistency** through *synonym analysis* and *natural language classification*

The results of this analysis are then evaluated and aggregated in a way that allows for a quick glance overview by looking at the root directory's values or an in-depth review on comments level and anything in between. Walking this JSON-tree allows to identify problematic sections or files in a project with lower quality than the rest.

Because the tool takes only a very sensible amount of time to be run on projects of any size, it allows users also to run multiple analysis on different versions of the same repository to evaluate the evolution of their documentation. Another use case would be to compare a new project of a developer team to an established one to see if the documentation standard set by the older project is upheld, even across code-language barriers.

While we can not claim that the tool assesses quality from all its aspects, it is a good start and covers the most important aspects with a hand full of metrics. The same can be said for the covered code languages and that the tool only operates on the English language. But since the pipeline itself is well documented and built to be extendable, this could change in future work.

Chapter 5

Framework Architecture & Components

To answer the two research questions of this study, a tool was created capable of classifying comments into types and assessing a local project's code comments. It is written in Python 3.8[1] and was developed and tested on Windows 10. Because of the nature of the scripts, there should be no problems when ran on different operating systems, but due to time constraints and limited resources, the pipeline was not tested on them.

It is designed to run locally for now but could be extended in the future to work on a server just as well, making the tool available to be accessible on a website, for example.

The interface is strictly over the command line for now, with the needed paths to input directories, output files, and miscellaneous settings as parameters that are well documented and can be viewed with a help command.

In total, there are six scripts that can be run from the command line, three for each research question:

5.0.1 trainer.py

:

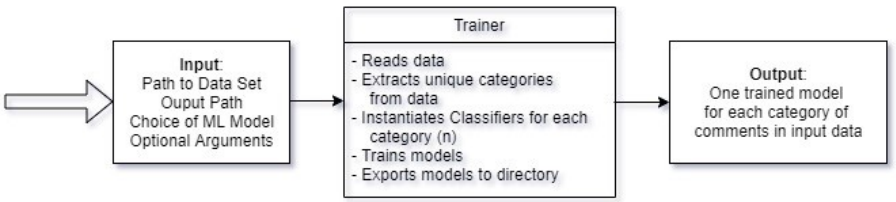

Figure 5.1: Trainer Implementation

This is the training script, that creates a binary classification model for every label found in

[1]https://www.python.org/downloads/release/python-380/

the input data that satisfies the minimum representation. The training script of the pipeline takes up to 6 arguments:

- data: This mandatory argument is the path to your data set in fasttext format: Every line is a data entry formatted as:

```
__label__summary this is a summary comment.
```

- output: The second mandatory argument is the path to the path to the output directory where the trained models are saved in as:

```
__label__category_name.model
```

- oversampling: This optional argument enables/disables the oversampling of underrepresented labels. It is enabled per default.

- model: This parameter allows the user to choose from the implemented machine learning algorithms, namely:

```
fasttext, naive_bayes, logistic_regression,
lsvc, random_forest or j48
```

  If none is set, fasttext is defined as the default one, as it performs best for our analysis.

- representation: This argument lets users choose the minimum representation of a comment category in a data set, for it to be considered as a training target. The default value is set to 50, which seemed sensible for our data.

- log: A value for switching the log level of the application. Users can choose form the following common list of levels:

```
debug, info, warning, error, or critical
```

  Every script has this argument and will not be further mentioned in this overview. The default logging level is *info*.

5.0.2 predictor.py

:

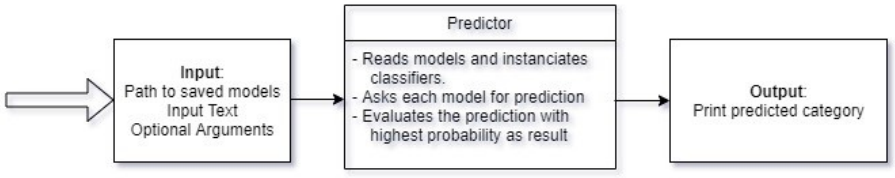

Figure 5.2: Predictor Implementation

This is the script that takes the path to the directory with the models created by the trainer script and an input string to print the predicted category to console. It takes two mandatory arguments and one optional one:

- models: The path to the trained models.
- text: The text to be classified as a string.
- verbose: This argument allows the user to not only print the predicted label, but also its probability.

5.0.3 validator.py

:

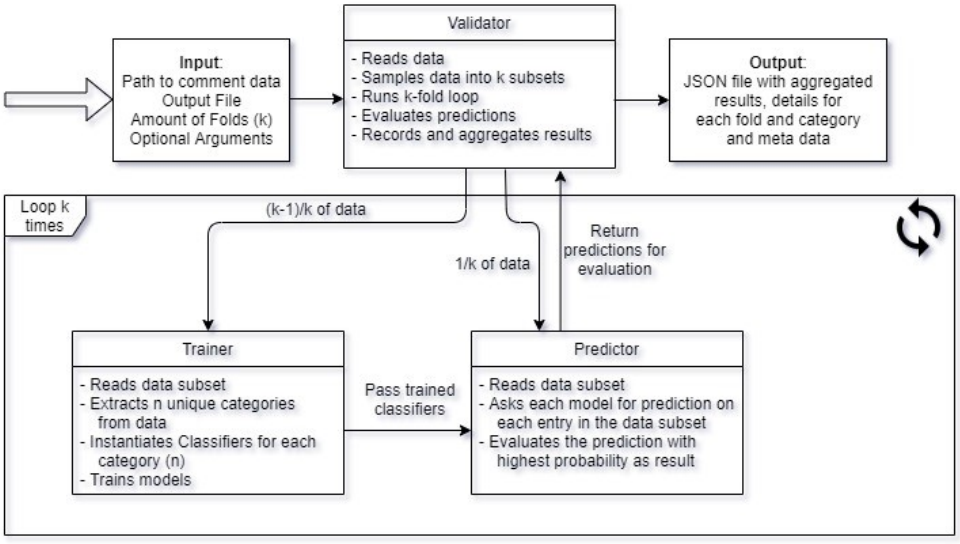

Figure 5.3: Validator Implementation

This is the k-fold validation pipeline that combines the training and predictor scripts to test a model's accuracy on a certain data set. It runs with up to seven arguments:

- data: The input to the data set that shall be used for testing and training.
- output: The output file with all detailed data of the validation.
- kfolds: This optional argument allows users to change the amount of folds in the k-fold validation. The default is set to 10, as is common practice.
- Oversampling, model, representation, and logging are also arguments of this script as it combines the previous two.

5.0.4 comment_rater.py

:

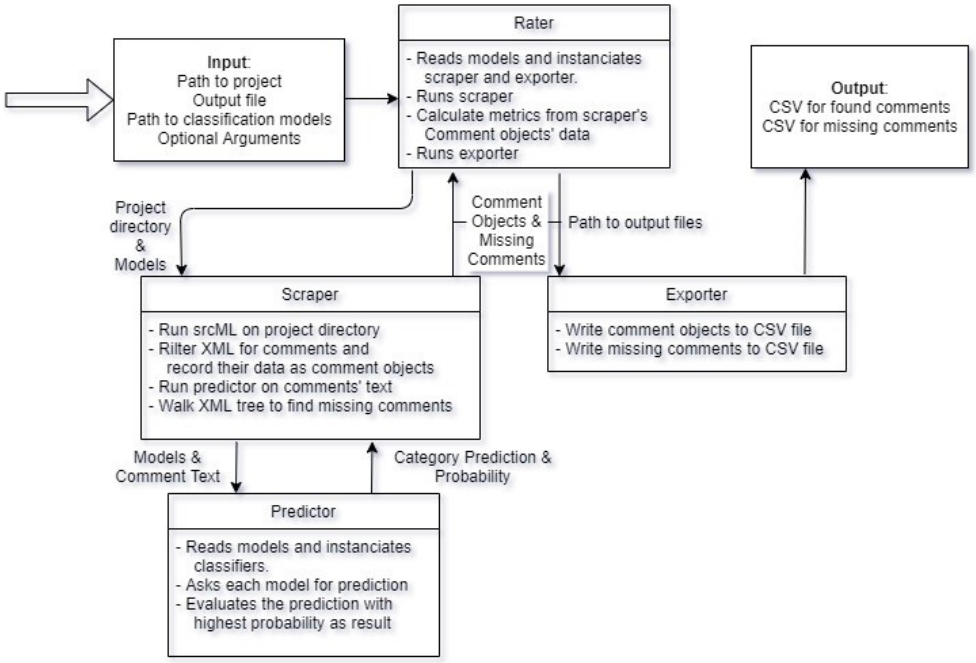

Figure 5.4: Rater Implementation

Running this file with a project directory as input creates a .csv file that holds all found comments and the calculated metrics and one .csv file with all the detected missing comments. This file exists because users may want to only scrape for all their comments and create a data set without the evaluation. The arguments are as follows:

- project: This is the path to the project directory that is going to get its comments analyzed.

- output: This argument contains the path for the resulting .csv file containing all found comments. A second file is also created for the missing comments, by appending *_missing* to the file name, so there is no need for inputting two paths.

- models: The directory to the trained models for classification is also needed, as comments are classified when gathering data.

5.0.5 comment_evaluator.py

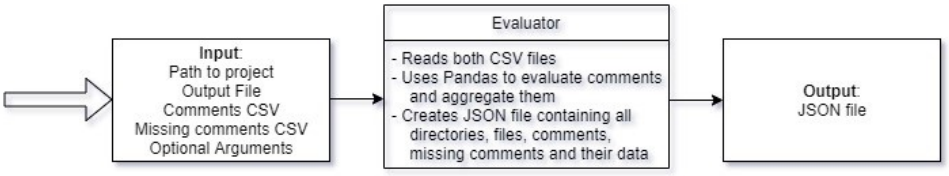

Figure 5.5: Evaluator Implementation

This script takes the two .csv files generated by the comment_rater and reads, evaluates, aggregates and presents the result in JSON tree. The arguments consist of:

- project: The path to the project's directory that will be evaluated.

- output: This is the location where the .json file will be saved to.

- (missing_) comments: These two arguments contain the paths to the two files generated by the rater script.

- synonyms: A flag that allows users to turn the synonym analysis on or off, as the processing time for this algorithm increases exponentially with project size. As it is only recommended for small projects, the flag is set to be disabled by default.

5.0.6 main.py

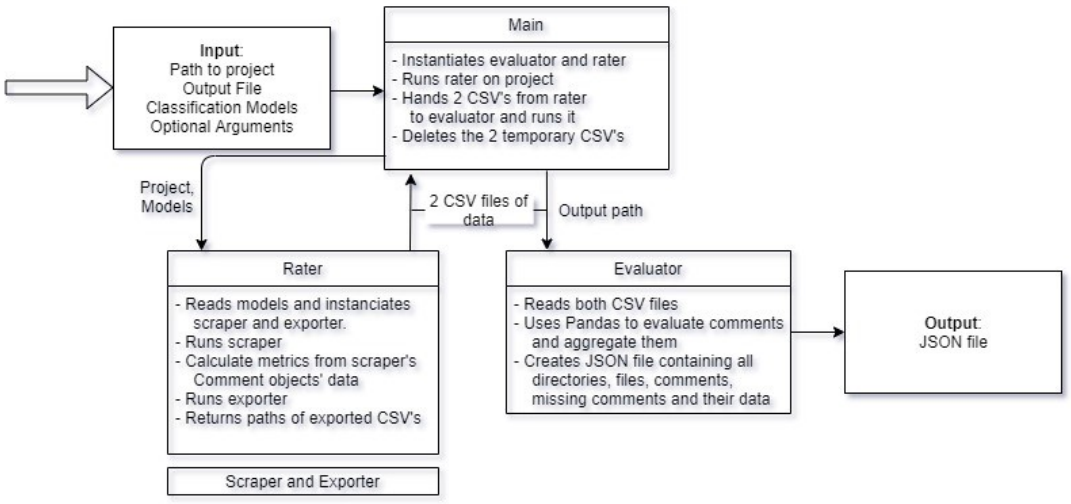

Figure 5.6: Main Implementation

This is the main script of the tool. It combines rating and evaluation into one big pipeline, taking a project directory's path and generating a full quality assessment of its comments, saving the results to a .json file. As it combines the two scripts above, it takes the same arguments, minus having to save the intermediate .csv files.

The repository also comes with the differently preprocessed data sets and all other data needed (e.g., abbreviation list). The pre-trained models could sadly not be included due to their big file size and GitHub's restrictions for free users. However, it is easy enough to run the training script to replicate the study, as the data is all here.

The tool and data can be viewed and downloaded by anyone on GitHub[2], pull requests are welcome. In its ReadMe file, one can also find more information about the correct usage and install instruction. The documentation of the code also contains details of the specific implementations and methods mentioned in this thesis.

[2]https://github.com/TimDeanMoser/coality

Chapter 6

Threats to Validity

When it comes to threats to validity of the comment type classification pipeline, the biggest one is that we have used a data set for training the models that was originally created by extracting comments from java projects, combining them in a single data set. As a consequence, our pipe line inherits the same threats as the two original studies, like the validity of the samples, meaning that the studies have been conducted on a small sample of projects and may therefore present little general knowledge, and the taxonomy validity [15] [13].

Additionally, only a subset of the proposed taxonomies and comment categories was used for training our machine learning models, increasing the threat of generalization and weakening the external validity.

Our quality assessment tool utilizes these models trained on java comments to classify comments into categories, which can lead to a threat to validity when used on non-java (e.g. C#) projects: Related work suggests that different programming languages have different sets of categories. As we do not actually use the information gained by classifying comments for the evaluation of quality, this hardly matters, but still must be kept in mind.

Related work also suggests that different programming languages have different requirements for quality in comments and so do their best practices for documentation. However, we circumvent this threat to validity by focusing mainly on the natural language qualities of comments and evaluate comments in isolation (without their corresponding source code). The exception to this is the coherence coefficient, which is calculated using by matching the comment's content to the name of the method, class, or similar, which srcML's parsing provides. Additionally we are filtering any threats with our ignored flag, described in 4.2. This leads our evaluation of quality to be independent from the programming language.

The biggest threat to validity in general is that, while this thesis has taken different proven approaches from related work, any iterations on them and new metrics or approaches found are based only on theory and sensible choices alone and not any empirical real world validation. The true usefulness and validity stands to be proven in future work by evaluation with real developers.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

In this thesis, we proposed a tool that can help developers assess the quality of their project's comments. We presented an approach to automatically assesses the quality of comments in respect to their readability, usefulness, completeness, coherence and consistency. This can help developers documenting their code properly by pointing them to problematic comments or files, while giving an aggregated overview for comparison across different versions of any project to analyze the evolution of its comments.

We have also shown that deep learning can out perform traditional machine learning algorithms when it comes to comment type classification when it comes to accuracy and speed. It could prove a valid approach for the future of comment classification and, should future research invest in a data set of a much bigger scope, hold its candle to approaches of related work.

7.2 Future Work

The biggest thing in general that can be done in future work is evaluating and validating the tool with real developers, like mentioned in the chapter 6: The tool's usefulness and validity are not based on any empirical real world validation: An experiment where a human's quality rating of comments is compared to the result of this thesis' tool would be an invaluable confirmation.

On the same note, future work could use this tool for topics like the evolution of comment quality. While we created a pseudo analysis in the section 4.3 for demonstration purposes, actually setting up an empirical study with hypotheses and research questions that can be analysed and answered using the proposed tool would be very insightful.

Next we are going to talk about possible improvements of the two parts of our tool more specifically:

7.2.1 Comment Category Classification

Collect more Data

In this thesis we have shown that a deep learning algorithm like fasttext can out perform traditional machine learning when it comes to classifying comments types. However when we compare the accuracy of this approach to those of related work can be over 95% [15]. However, the

hypothesis is that fasttext could reach higher amounts of accuracy the bigger the data set, as deep learning approaches are notorious for needing a big data set to perform best. For comments, creating such a big data set is no easy task, as they are not inherently labelled. This means that humans would need to manually assign comments to different comment types, which requires a lot of resources and time.

Expand supported Comment Types and Models

In this study, we limited the amount of comment types of concern to a sub set: There exist many more types like todo-, exception-, dependency- or responsibility comments. Classifying comments into those miscellaneous categories is not feasible with out approach as their representation is too low in an already too small data set. However, future work with a much bigger data set, as proposed in the section above, might make it possible to categorize comments into many more or even all types with respectable accuracy. For this study, fasttext was chosen to represent the deep learning party of classifiers. It could also be interesting in future work to try a different approach. The same is true for all machine learning models used in the thesis: While we have a big enough list for comparison, extending the amount of models might bring new insights. After all. the classification pipeline is built in a way that switching or extending data sets, comment types or models would be an easy enough task.

7.2.2 Comment Quality Assessment Tool

Improve the Tool

One thing that we would love to revisit in future work is the tool itself. While it does its job and does it fairly well, it can still be improved in multiple aspects:

- Increase the general stability of the tool. As of now it is not very safe to use, as it is developed to be used for this study only. Would someone use it in a way that was not directly intended, weird and unaccounted for behaviour might occur. Unit tests are definitely a must have addition for example, which have been omitted due to time and resource constraints.
- The synonym analysis is very high in its complexity and makes the feature nearly unusable on bigger projects, as it takes too long. It is definitely the bottle neck in performance of the tool and should be revisited, as the other evaluations are very fast in comparison, no matter the project size.
- On a similar note, the pipeline would greatly benefit from a synonym and abbreviation library or list that is more in the context of software development. The language of the IT world is often different from everyday English and this difference should not be ignored, as comments live in a different context than for example a newspaper article.
- Another aspect of the pipeline that needs improvement is the detection of commented code. Our approach described in the section 4.2 is far from perfect and could not have been validated properly.
- While we have covered every major quality aspect for comments with at least one metric or score, there still exist several different approaches from related work that could be of use for our analysis and evaluations. For example, another aspect of the consistency of comments would be that header and license comments have the same layout in all files. The more aspects our tool can cover, the better it can give insight of the quality of a project's comments.

Front-End and Visualization

The pipeline does not really have a front-end for visualization yet. JSON, while readable by humans, is not very usable for quick analysis. Hence why we would like to extend the tool in the future with a way to visualize the results and since the output is a JSON file, we are already half way there to be able to create a front-end with something like REST inside a browser. The logical next step would be to move away from a command line interface (CLI) and allow users to select their inputs and settings over a graphical interface, which is a leads us into the next section:

Accessibility, Ease of Use and Install

Besides improving the actual inner workings of the quality assessment tool, it would benefit a lot from improving the user experience, Multiple aspects come to mind:

- Increase the availability of the application for more operating systems. The tool was developed and tested entirely on Windows 10 and while it should work in other operating systems like OSX and Linux, it was never tested due to time constraints.
- Right now the install process is rather tedious and could be simplified by releasing the application as a package on PyPi[1], where it then could be easily installed and updated using pip.
- Another way of increasing accessibility is by releasing the tool as part of a web based application, forgoing the install process as a whole. This directly ties into the idea of creating a visual interface inside a browser and would increase the availability drastically. Another aspect of this could be to allow users to either upload their projects for evaluation or pulling it from GitHub directly, bypassing any downloads or uploads.
- Integrating the application directly into the development cycle could also be an interesting approach. For example, GitHub allows users to install Apps to automate and streamline workflow[2]. In this case, the tool could be extended to be installed as such an app, creating a report of the comment quality in the project after every commit, for example.
- The last proposed way of moving away from a CLI, would be to integrate its functionality into an integrated development environment (IDE). Most modern IDE's like Visual Studio and IntelliJ allow users to install extensions, which would be a perfect fit for our quality assessment tool, while still working locally on the machine.

[1] https://pypi.org/
[2] https://docs.github.com/en/developers/apps

Chapter 8

Appendix

8.1 Repository

The tool developed in this thesis and the data used can be viewed and downloaded by anyone on the following GitHub repository:

https://github.com/TimDeanMoser/coality

8.2 Zoomed in Figures of Sections 4.2 and 4.3

To increase the readability of the plots in the sections 4.2 and 4.3, one can find the plots with increased size in this section.

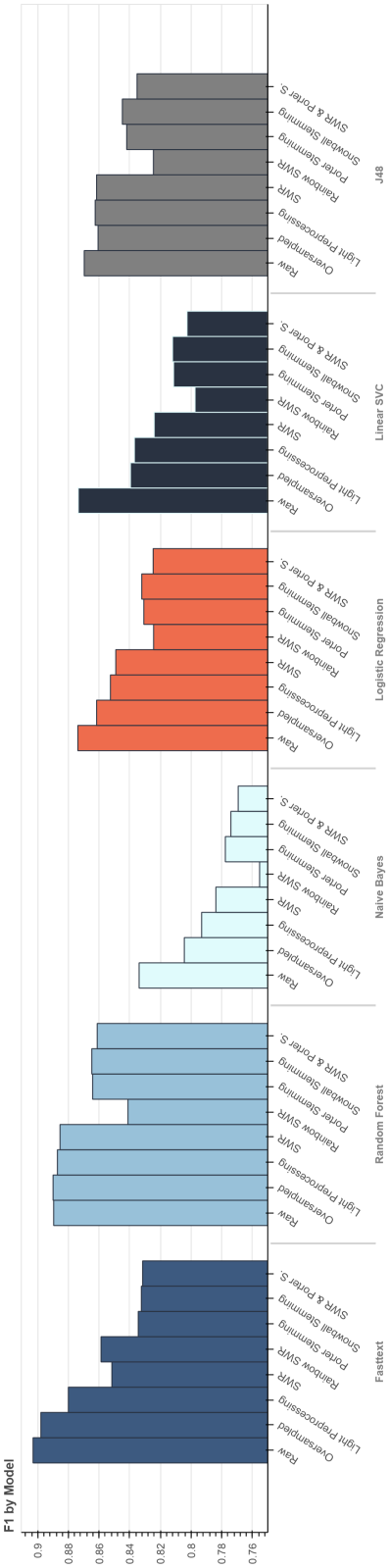

Figure 8.1: Zoomed Plot of Comparison of F1 score of different models

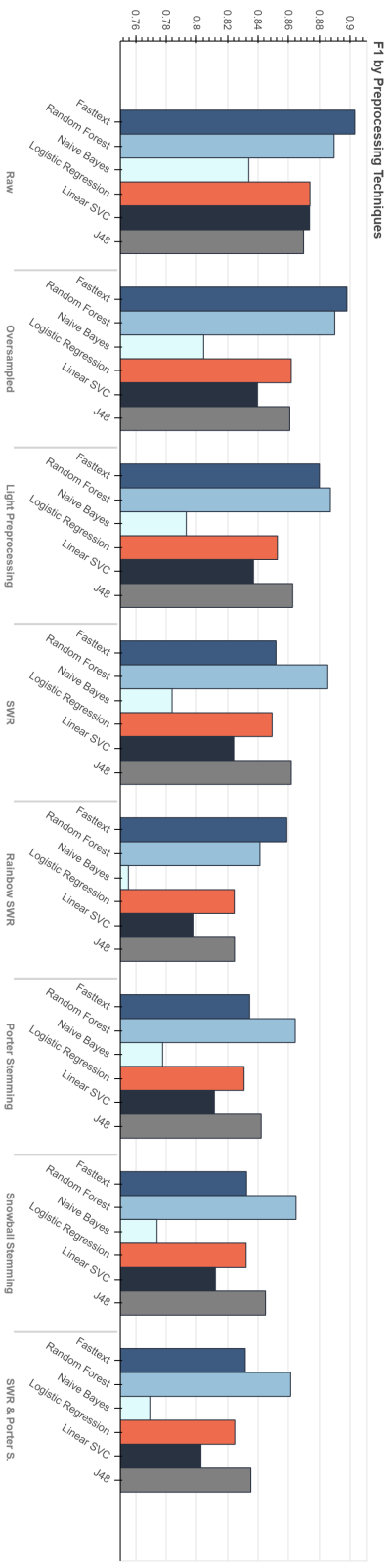

Figure 8.2: Zoomed Plot of Comparison of F1 of different preprocessing techniques

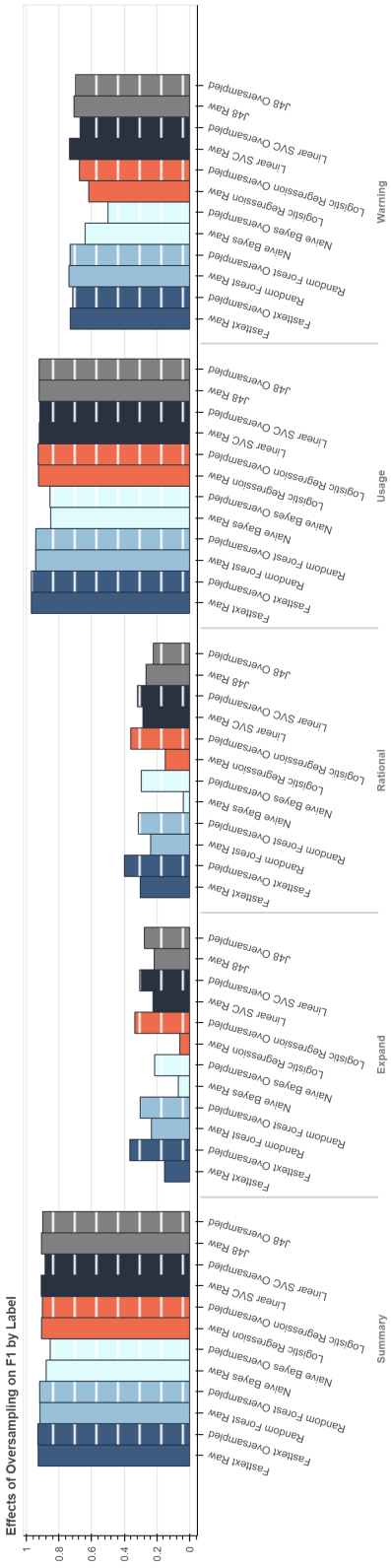

Figure 8.3: Zoomed Plot of Effect of Oversampling on Labels

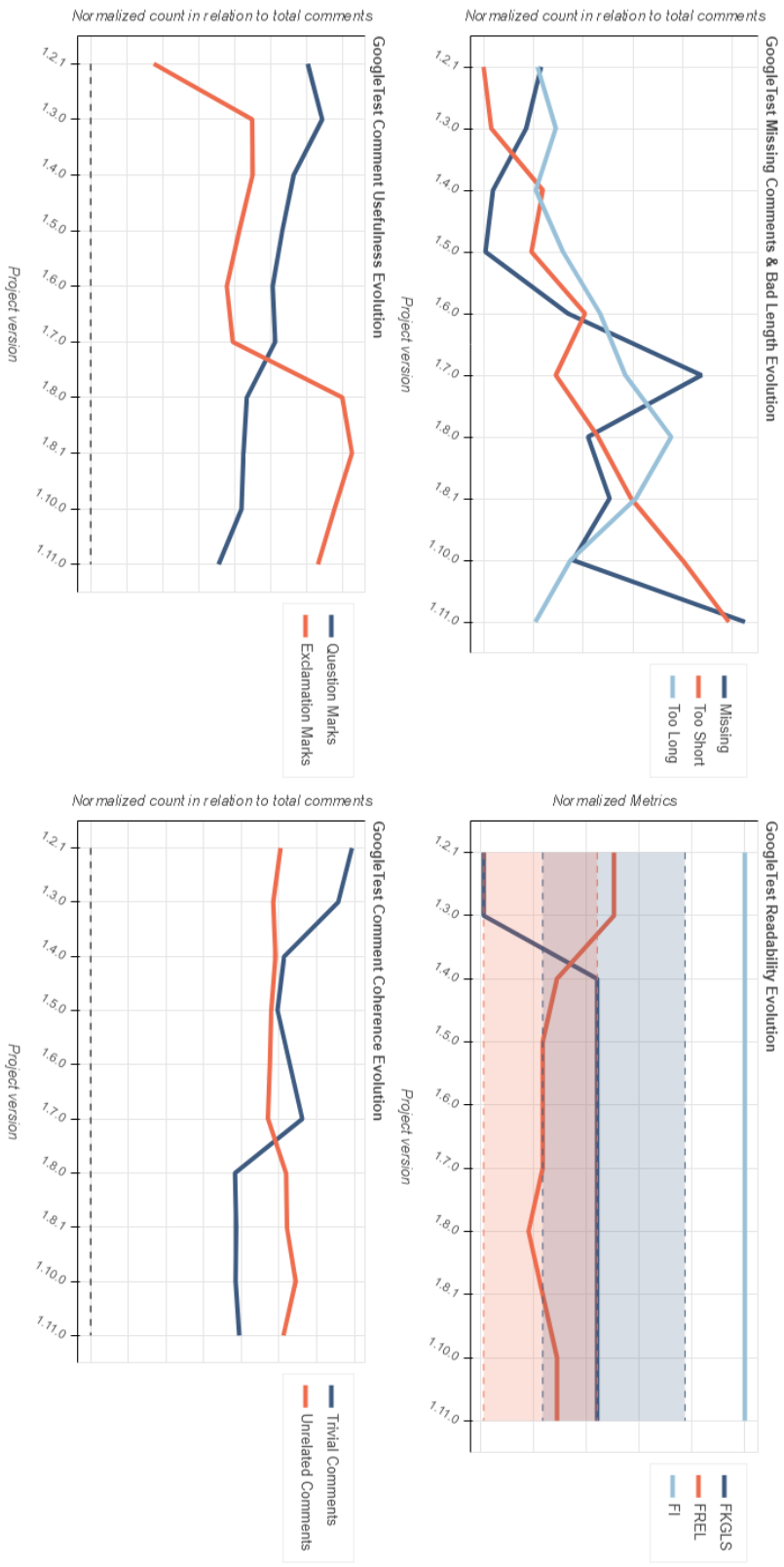

Figure 8.4: Zoomed Evolution of Googletest's Quality Metrics

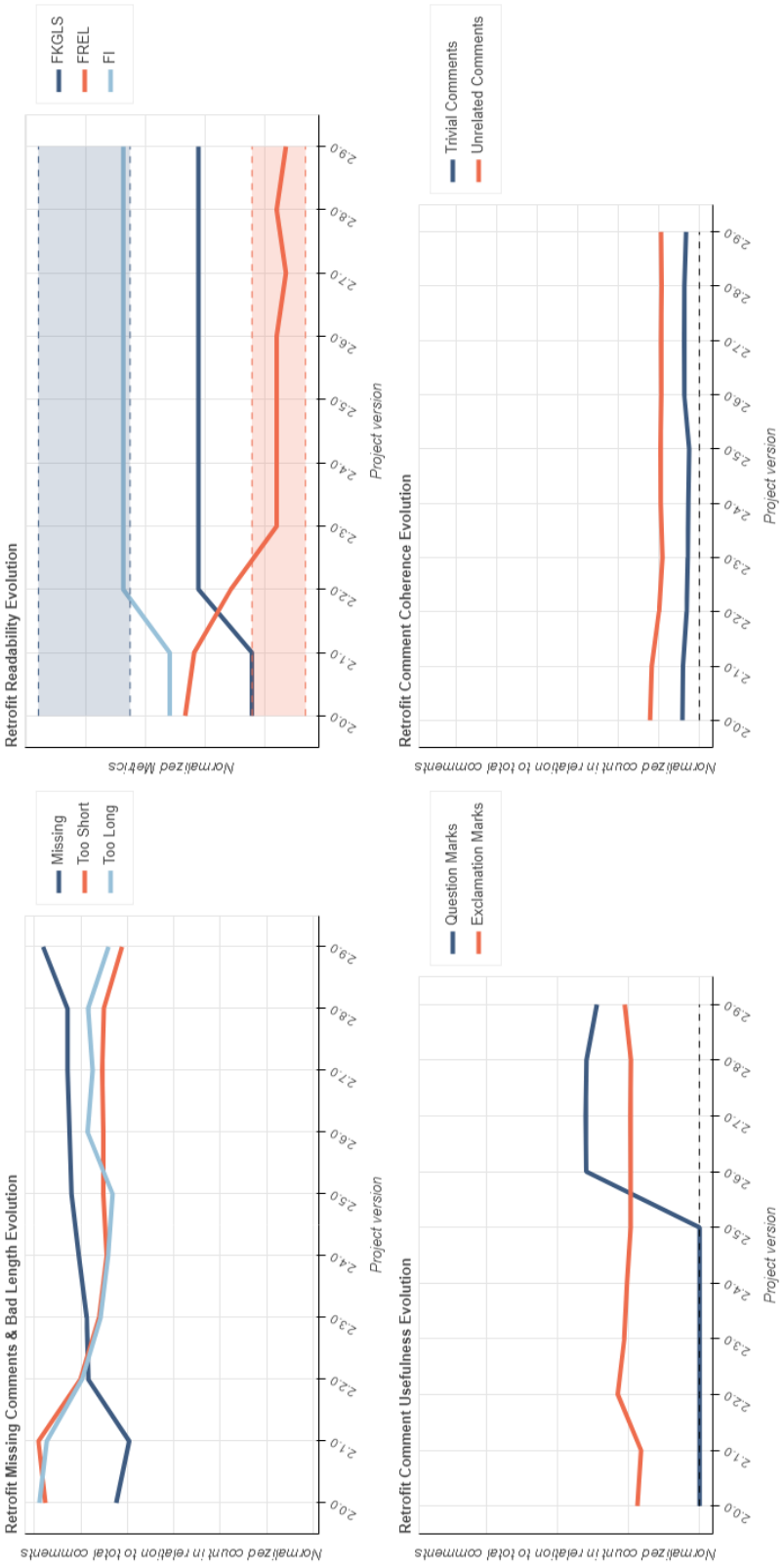

Figure 8.5: Zoomed Evolution of Retrofit's Quality Metrics

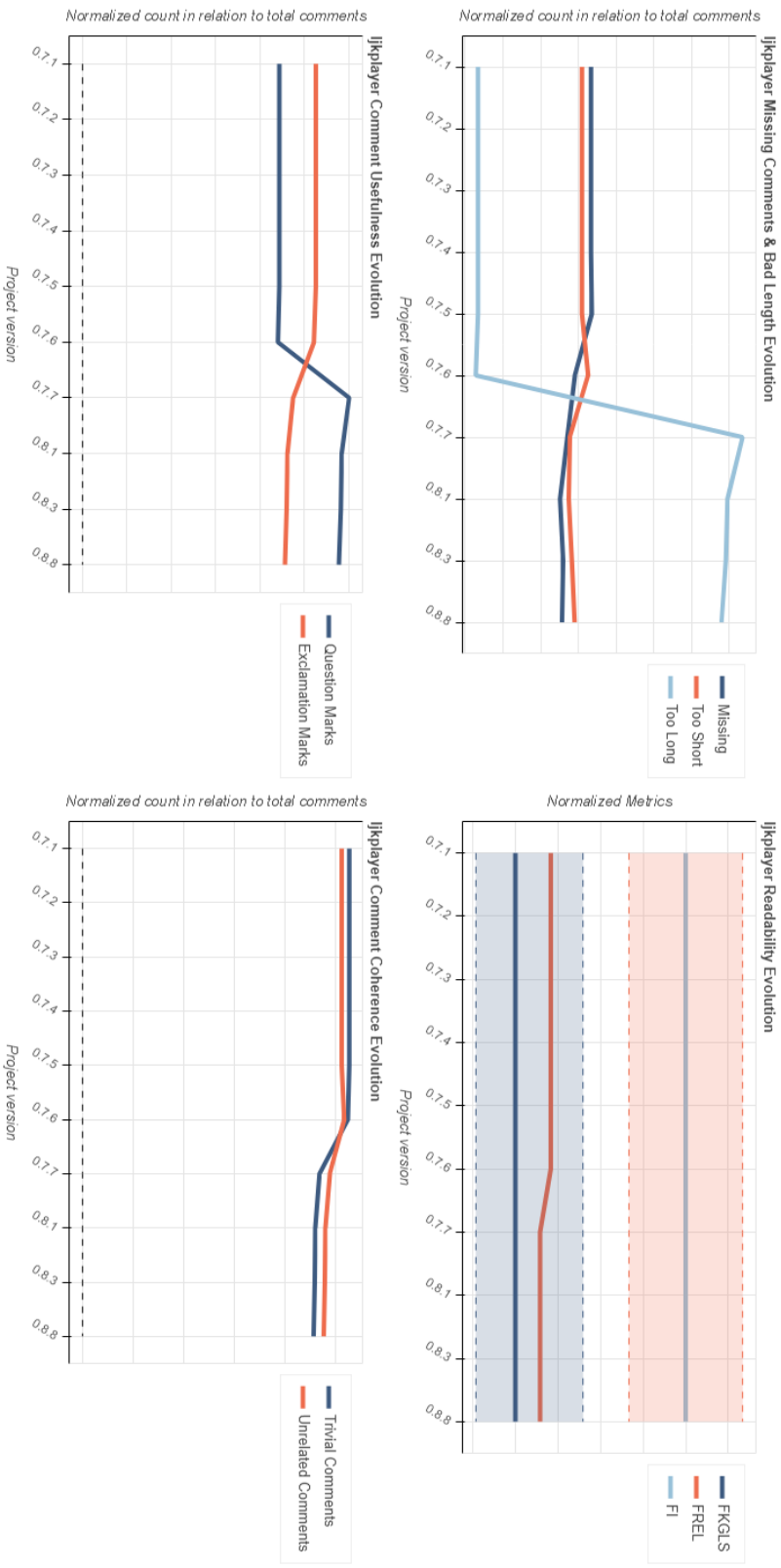

Figure 8.6: Zoomed Evolution of Ijkplayer's Quality Metrics

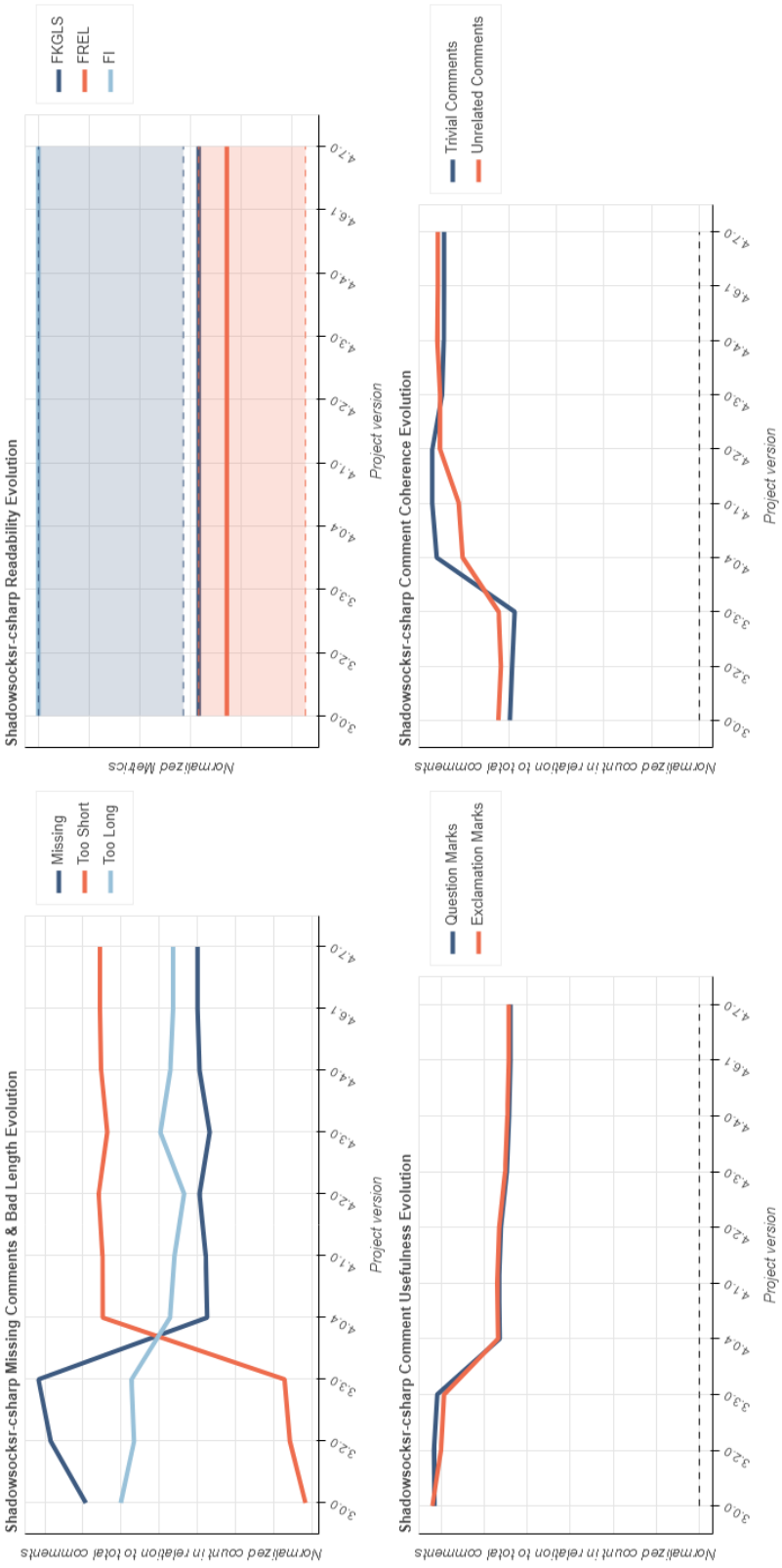

Figure 8.7: Zoomed Evolution of Shadowsocksr-csharp's Quality Metrics

Bibliography

[1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, 2018.

[2] S. C. B. de Souza, N. Anquetil, and K. Oliveira, "A study of the documentation essential to software maintenance," in *SIGDOC '05*, 2005.

[3] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*, pp. 70–79, 2007.

[4] T. Tenny, "Program readability: procedures versus comments," *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1271–1279, 1988.

[5] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, p. 215–223, IEEE Press, 1981.

[6] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 83–92, 2013.

[7] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 161–170, 2015.

[8] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, no. 2, p. 1419–1457, 2019.

[9] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, "Context is king: The developer perspective on the usage of static analysis tools," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 38–49, 2018.

[10] P. Oman and J. Hagemeister, "Metrics for assessing a software system's maintainability," in *Proceedings Conference on Software Maintenance 1992*, pp. 337–344, 1992.

[11] A. Curiel and C. Collet, "Sign language lexical recognition with propositional dynamic logic," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, (Sofia, Bulgaria), pp. 328–333, Association for Computational Linguistics, Aug. 2013.

[12] S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu, and J. Gao, "Deep learning–based text classification: A comprehensive review," *ACM Computing Surveys (CSUR)*, vol. 54, no. 3, pp. 1–40, 2021.

[13] L. Pascarella and A. Bacchelli, "Classifying code comments in java open-source software systems," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 227–237, 2017.

[14] J. Zhang, L. Xu, and Y. Li, *Classifying Python Code Comments Based on Supervised Learning: 15th International Conference, WISA 2018, Taiyuan, China, September 14–15, 2018, Proceedings*, pp. 39–47. 09 2018.

[15] P. Rani, S. Panichella, M. Leuenberger, A. Di Sorbo, and O. Nierstrasz, "How to identify class comment types? a multi-language approach for class comment classification," *Journal of Systems and Software*, vol. 181, p. 111047, 2021.

[16] D. Steidl, "Quality analysis and assessment of source code comments," 2012. .

[17] N. Khamis, R. Witte, and J. Rilling, "Automatic quality assessment of source code comments: The javadocminer," vol. 6177, pp. 68–79, 11 2010.

[18] D. Wang, Y. Guo, W. Dong, Z. Wang, H. Liu, and S. Li, "Deep code-comment understanding and assessment," *IEEE Access*, vol. 7, pp. 174200–174209, 2019.

[19] A. Corazza, V. Maggio, and G. Scanniello, "Coherence of comments and method implementations: a dataset and an empirical investigation," *Software Quality Journal*, vol. 26, 06 2018.

[20] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 83–92, 2013.

[21] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *2013 21st International Conference on Program Comprehension (ICPC)*, pp. 23–32, 2013.

[22] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "Fasttext.zip: Compressing text classification models," 12 2016.

[23] K. C. Dodds, "Please, don't commit commented out code," 2015.

[24] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," *arXiv preprint arXiv:1607.01759*, 2016.

[25] W. Dubay, "The principles of readability," *CA*, vol. 92627949, pp. 631–3309, 01 2004.

[26] Google, "Writing accessible documentation," 2021.