ABSTRACT

PANDE, RASIKA HEMANT. Automating the Extraction and Evaluation of Comments in Source Code. (Under the direction of Dr. Collin Lynch).

Writing comments serves as the primary way to document source code internally and improve its readability. In Computer Science education, assignments in introductory programming courses may require students to explain the underlying logic and functionality of their program in the form of comments. This practice has been thought to help students' reasoning about the code and thus improve their ability to program effectively. Research has also shown that writing comments can help students organize their code logically and visually. However, there is no standardized mechanism of automatically providing feedback to students based on the semantic meaning of their comments. Since comments are written in natural language, such an evaluation would have to be performed manually by instructors. In this thesis, we present a tool to automate the extraction and evaluation of comments from code submitted in introductory programming courses. The focus of our work is on Java source code written by Computer Science undergraduate students at North Carolina State University. The tool first extracts comments from source code files and then tags them with the associated code context by traversing the program's Abstract Syntax Tree. We then evaluate these comments based on their descriptiveness and similarity with the code using natural language processing and machine learning techniques.

Automating the Extraction and Evaluation of Comments in Source Code

by
Rasika Hemant Pande

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2019

APPROVED BY:

Dr. Sarah Heckman	Dr. Christopher Parnin

Dr. Collin Lynch
Chair of Advisory Committee

BIOGRAPHY

Rasika Pande received her Bachelor's in Computer Engineering from College of Engineering Pune. After graduating in 2016, she worked at Vuclip India Pvt. Ltd. in data analytics. In 2017, she joined the Master's program in Computer Science at North Carolina State University to pursue her interests in machine learning and natural language processing.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Collin Lynch, for his support, encouragement, and guidance throughout the course of this thesis. I would also like to thank Adam Gaweda for all his help. Finally, I would like to acknowledge the constant support of my family and my friends.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

1.1 Motivation

Documenting code using comments is often considered as one of the best practices for understanding program functionality and maintaining software. As a result, programming assignments in introductory programming courses look to inculcate this habit in students by requiring them to supplement their code with appropriate comments. It is possible to automate grading certain aspects of these assignments, like checking their output or checking if test cases pass or fail. However, evaluating comments based on whether they effectively describe program logic usually requires manual intervention. We want to reduce the manual efforts required by instructors and TAs in providing feedback to students based on the quality of their comments. One of the challenges here is automatically interpreting the language used in comments. Although comments like Javadocs have a certain level of structure to them, there is significant natural language content that still needs to be parsed. A comment can also be written in multiple ways to express the same meaning, which can make the evaluation subjective. As a result, the process of evaluating the semantics of

student comments lacks a standardized approach.

1.2 Goals

Our overall goal is to address this problem of automating the evaluation of comments from programming assignments written by students. We aim to highlight submissions which are satisfactorily commented against the ones which are not, using supervised machine learning techniques. To achieve this goal, we first need to extract individual comments from programs, associate them with the lines of code they describe, and analyze natural language aspects of the comments.

1.3 Contribution

This thesis presents a preliminary but novel approach to classify student comments based on their semantic quality using natural language processing and supervised machine learning techniques. We demonstrate its effectiveness by assessing Java comments from two different programming assignments based on their descriptiveness and semantic similarity with code. A major contribution of our work also includes a generalized technique for deriving source code contexts for Java and Python comments. Using Abstract Syntax Tree traversals, we define program constructs which the extracted comments intend to describe.

CHAPTER

2

LITERATURE SURVEY

2.1 Source Code Explanation

To establish a relationship between students' ability to write effective code explanations and their programming skills, Corney et al. studied responses to questions from a CS-2 level data structures course exam, which asked students to describe the purpose of code snippets in simple English [Cor14]. Significant correlation was found between students' scores on these and on the questions which required code to be written. Based on these results, the authors recommend a mix of writing and explaining code as a part of the coursework in order to develop students' programming ability.

Additionally, research on students' thinking patterns has demonstrated the value of describing code in the form of comments. Mohammadi-Aragh et. al analyzed comments written by novice programmers as part of Writing to Learn (WTL) programming assignments [MA18]. When these were compared to traditional assignments (which did not require comments explaining the code), the authors found that the WTL assignments contained more comments

and divided the code into logical sections, displaying a better organizational strategy.

Both these studies try to improve the process of learning to program by establishing a need for writing comments and effective code explanations. We aim to analyze these comments written by students and determine if they are indeed effective and well-written.

2.2 Comment Analysis

Haouari et al. analyzed the commenting patterns of Java developers, who had experience with programming in Java [Hao11]. In the three-open source Java projects that they studied, the authors observed that developers usually explain the functionality of program constructs using instruction keywords rather than abstract terms. Khamis et al. [Kha10] and Schreck et al. [Sch07] developed several heuristics and metrics to measure aspects like readability and completeness of Javadoc comments to ensure that they serve as clear and understandable documentation for developers. Khamis et al., for example, developed an automated approach to measure the quality of the Javadoc language by calculating the number of nouns, verbs, abbreviations, and the average number of words per comment [Kha10]. They also checked if the correct names, types and descriptions were present while writing Javadoc tags for return values, method parameters and exceptions in the code. This check for comment-code consistency is restricted to the analysis of Javadoc tags. While we also aim to analyze the similarity between code and comments, we do a deeper, semantic-based comparison between them. Steidl et al. have presented a scheme to judge the quality of comments with an aim to identify the ones that can be refactored [Ste13]. This evaluation was based on four criteria — coherence between the comment and code, the completeness, consistency, and usefulness of comments in understanding the purpose of the code. The coherence metric uses Levenshtein distance to measure the relevancy between the text in the comment and its associated method name. However, it does not perform an analysis of their semantic similarity, which we aim to do. In case of method comments, we also compare the comment with the entire body of the method and not just the method name.

Other researchers have also studied the consistency between comments and code by employing Natural Language Processing (NLP) techniques. Liu et al. computed program readability by using WordNet to find matching keywords between comments and method names

and their return parameters [Liu15]. While we use WordNet to check for synonyms, our analysis is not restricted to method comments. Along with semantic similarity, we also use descriptiveness as a feature for evaluating comments. Chen et al. developed a Skip-gram model to check similarity between comments and individual statements of code [Che18]. They use this similarity as one of the features of determining scopes of comments, whereas our main aim is evaluating the quality of the comment. Deep sequence-to-sequence learning has been used to find redundant comments which are highly similar to the code and can be directly inferred from it [Lou18]. However, the tool developed in that study currently works only for method-level comments, while we aim to evaluate statement-level comments as well. Tan et al. have developed a tool called iComment to detect discrepancies between comments and the associated code using a combination of NLP, mixture model clustering, and rule extraction [Tan07]. According to Tan, Yuan, Krishna, and Zhou, such a discrepancy may either denote a software bug or a bad comment. The scope of this study is limited to checking the comment-code consistency, and it does not evaluate the quality or comprehensiveness of the comment. Additionally, the focus is on comments stating the developer's assumptions and requirements, while comments providing explanations of code segments are ignored.

The research discussed above focuses on code written by relatively experienced developers. On the other hand, there are limited studies which evaluate comments written by novice programmers. A preliminary analysis of student-written comments has been the focus of [DeP12], where the authors have developed a tool to report the commenting practices of undergraduate students and professional open-source developers. According to the authors, analyzing the differences between the two can help students assess and improve their own commenting style. The tool does not evaluate any natural language aspects of the comments, instead extracting only basic Javadoc metrics like (i) checking for the presence of Javadoc tags wherever applicable (ii) calculating the percentage of methods in a class that have Javadocs (iii) calculating the average number of words found in method documentation (iv) checking for correct spellings (v) checking for offensive words (vi) checking for pre and post conditions. Currently, the system simply reports the values of these metrics. For instance, the authors observed that students rarely used Javadoc-style tags for method-level documentation and barely commented 50% of all methods. The tool also does not label any comments as being good or bad in quality, which is the focus of

our work. Instead, the differences between the commenting styles found in coursework assignments and open-source projects are only meant to serve as guidelines to students about the general best practices adopted by professional developers.

2.3 Automated Grading for Short Answer Responses

Grading short answer responses has traditionally been a manual task performed by instructors, given the unstructured nature of the answers. Automating this process would require understanding the semantic meaning of responses, and similar grades would need to be assigned to answers similar in meaning [Kle11].

We believe there is considerable similarity between automating the grading of short answer responses and that of source code comments. Both involve parsing and evaluating concise explanations written in natural language. In many cases, there may not always be a model answer which can be used as a grading rubric. Additionally, their unstructured nature also means that short answers and comments can be written in a variety of different ways to express the same meaning. These similarities motivated us to also survey research in NLP techniques for automatically assessing short answer responses. However, the key distinction between short answers and comments is that the short answers address *one*, single topic with *many* possible answers from different students. By contrast, our scenario has *many* ways of writing comments for the corresponding code, which itself may *differ* every time.

Pulman et al. have compared pattern-based Information Extraction approaches with machine learning methods like Inductive Logic Programming, Naïve Bayes classifier, and Decision Trees to grade questions on Science exams [PS05]. Out of all approaches, Information Extraction achieved the best performance though it involved manual efforts in creating customized patterns. In [Kle11], a set of manually graded short answers to questions on Computer Science exams was used to automatically grade other responses that were similar in meaning. The similarity of the text was determined using Latent Semantic Analysis and a combination of cosine similarity and Euclidean distance. Cutrone et al. have also auto-graded single sentence responses with respect to a correct answer submitted by the instructor [Cut11]. Using WordNet, the terms in the responses, their synonyms, and their

parts of speech were compared to check how well they matched.

CHAPTER

3

METHODOLOGY

Our approach to evaluating comments consists of three major steps – First, we extract all comments and tag them with the code they document. Next, we use this data to create features related to the descriptiveness of comments and their similarity with source code. Finally, based on the extracted features, we classify comments as *satisfactory* or *unsatisfactory* using supervised machine learning techniques.

3.1 Extracting Comment-Code Pairs

This process reads a source code file and returns a list of all comment-code pairs in the file. Currently, it supports parsing Java and Python source code files. The general format of the module is based on the Comment Parser project [Avi], which stores the text of single and multiline comments along with the line numbers on which they start. We, however, extended this functionality to also determine the lines of code that the comments are intended to describe. We achieve this by traversing the program's Abstract Syntax Tree using custom Python and Java parsers that we have developed in Python.

3.1.1 Comment Extraction

First, we use a combination of regular expressions and tokenizers to extract single and multiline comments from a file. In Java, single line comments start with '//' and continue till the end of the line. Block comments begin with '/*' or '/**' (in case of Javadocs), with both ending in '*/'. These can either be single or multiline. In Python, single line comments start with the '#' character and continue till the end of the line. Documentation comments (docstrings) in Python are enclosed within triple double quotes or triple single quotes and can span multiple lines. While parsing comments, we also keep track of their start and end line numbers, which will be used later during the code tagging process.

In certain cases, multiple single line comments are written on consecutive lines instead of one multiline/block comment. Because there is no code between these comments, they can be assumed to function as a single, continuous block documenting the same piece of code. Hence during the extraction process, we combine such comments into one and consider the overall start and end line numbers to be the start and end line numbers of the very first and last comments respectively.

3.1.2 Source Code Tagging

One of the factors we use for determining comment quality is the degree of consistency between the comment and its corresponding code. Hence, after extracting comments from a program, there is a need to determine the lines of code with which they are associated. This can be a single code statement or a block of code (i.e. class/method definitions, conditionals, loops). The required information can be extracted by parsing the code as an Abstract Syntax Tree (AST), which is a simplified tree representation of a program's syntactic elements. In this process, the code is converted into a hierarchical tree structure, with each code statement being represented as a tree node object. As a result, entire code constructs such as class, method, field definitions, function calls, conditionals etc. exist as nodes of the AST and can be accessed through a simple tree traversal.

Python has a built-in 'ast' library [Pyt01] for parsing Python code and working with its AST. We have also used the Python library, 'javalang' [Thu], to parse Java source code in Python. For each of the relevant tree nodes, both libraries provide the added functionality of storing

line numbers of the corresponding code constructs. This attribute helps to identify the position where the construct occurs in the original file. As a result, comparing the comment line numbers saved during the extraction phase with the AST line numbers can yield the related source code context for each comment.

Usually, comments are located in close proximity to their corresponding code structures. Code statements can be described using inline comments as shown in Fig. 3.1 and Fig. 3.2, where short, single line comments are written on the same line as the code. In the majority of cases, comments are immediately followed by the code constructs they intend to describe [Hao11]. For example, Fig. 3.3 shows a comment documenting a method definition. Additionally in Python, docstrings for method and class definitions are written within the definition body and right below the line containing the 'def' keyword. We use these styling conventions as the basis for comparing code and comment line numbers and estimating the required code context. For each extracted comment in Java and Python, we traverse the AST to check the closest occurring node according to the following rules -

1. **Inline comments present on the same line as code -**

   We first check if there are any nodes in the AST whose starting line number equals the ending line number of the comment. The comment should only be tagged with the code block/statement from the same line and not with any others occurring above or below it. Fig. 3.1 and Fig. 3.2 show examples of this instance, where each comment is only intended for a single code statement.

```
String temp = scan.next(); // seat string
temp = temp.trim(); // removes any whitespace
int dash = temp.indexOf("-"); // make sure there's a dash
if (dash == -1) {
    scan.close();
    throw new IllegalArgumentException(); // if no dash
}
```

Figure 3.1 Java Inline Comments.

```
def delete_item(i):
    del num_list[i]     # delete i th element from list
    return


num_list = [1, 2, 3, 4, 5]  # define list of numbers
delete_item(1)
print(num_list)         # print list
```

Figure 3.2 Python Inline Comments.

2. **Comment immediately preceding code -**

   If the first case isn't satisfied, we proceed to check if there are any nodes whose starting line number is one greater than comment's ending line number. The comment is tagged with the node occurring below it. For example, the comment in Fig. 3.3 will be tagged with the method definition block for `resetNumber()`, and the comment in Fig. 3.4 will be tagged with the loop block.

```
/**
 * Resets the Reservation number to 1000.
 */
public static void resetNumber() {
    number = 1000;
}
```

Figure 3.3 Java Comment Preceding Code.

```
dictionary = {'a': 1, 'b': 2, 'c': 3}

# print all key value pairs from dictionary
for k, v in dictionary.items():
    print(k)
    print(v)
```

Figure 3.4 Python Comment Preceding Code.

3. **Comment immediately following code -** In Python, this scenario is generally applicable to docstrings, which are used to describe class/method definitions and are located between the signature and the body of the block (Fig. 3.5). Since these are used primarily for blocks of code, we only check for nodes which have a 'body' attribute. If the comment occurs between the start of the node and its body, the comment is tagged with the node representing the entire code block.

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
```

Figure 3.5 Python docstring. [1]

[1] https://www.python.org/dev/peps/pep-0257/

Once a node is found for each comment, we extract and store the type of code construct (class declaration, field declaration, expression etc.) and the actual source text. To get the source text back from a node, we use the library 'asttokens' [SG] for Python code. For Java, we use the node start and end locations to retrieve the corresponding source from the original file. In the case of code blocks, there is a chance that the source text contains other comments within it. Since these have already been extracted separately along with their respective code context, they need not be repeated here. Hence, we compare the text of the code block with previously extracted comments and remove any matching snippets.

In the end, we return all comments tagged with the type of the code construct they describe and the corresponding source text.

3.2 Feature Extraction

In this process, we extract features from code and comment texts, which will then be used for classifying comments and determining their quality. While extracting features, we aim to utilize the natural language aspects (both syntactic and semantic) of the comment text. We define two such features – i) comment complexity and ii) semantic similarity of comments with code. Comment complexity is related to the grammatical complexity of a comment's sentence structure, which we calculate by finding dependencies between its words. We determine the degree of semantic similarity between comments and code by checking for synonyms and words expressing similar concepts. We discuss the calculation of these features in detail in sections 3.2.1. and 3.2.2.

In all of the subsequent analysis, our focus will be on Java comments. We also filter out the following comment types, which we consider irrelevant to our analysis -

- **Copyright or License Comments** - Any comments that begin with the keywords 'copyright' or 'license' are excluded since they do not describe code structures or explain programming decisions in any way.
- **Task Comments** - Similarly, comments that begin with 'TODO' can be excluded since they are only meant to serve as notes to programmers about any remaining tasks.

- **Comments without Code Context** - Stand-alone comments can be ignored since there are no neighboring AST nodes to check the comment-code similarity.

3.2.1 Comment Complexity

This first feature is based on the syntactic relationships between words in a sentence. Dependency parsing allows extracting these relationships by identifying dependencies between words. For example, a dependency relation exists between two words if one modifies the other. Applying this concept to comments, we aim to determine how grammatically complex and descriptive the text is.

Dependency Parsing

Given a sentence, a dependency parser can create its dependency tree, with nodes representing the words in a sentence and the edges representing labelled dependencies. For example, consider the following comment -
`// The function adds two numbers given as inputs and returns the sum`

Using the Stanford Dependency Parser [CM14], the dependency tree created for the comment text is shown in Fig. 3.6. The arrow labels indicate the type of grammatical dependencies (nominal subject, direct object), and the arrow directions denote dependent words. The arrow from the word ‘`numbers`’ to ‘`two`’ shows that ‘`two`’ modifies ‘`numbers`’.

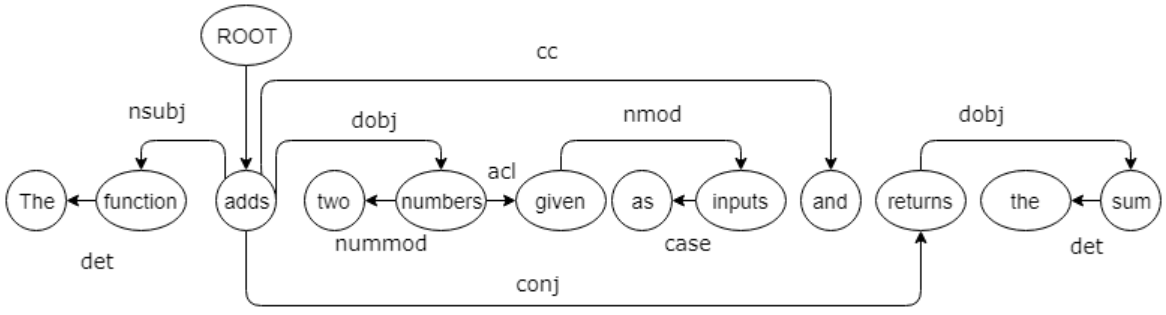

Figure 3.6 Dependency Parse Tree.

Average Dependency Distances

To convert these syntactic relationships into a quantitative complexity feature, we use the metric Average Dependency Distances (ADD) as proposed by [Oya11]. It uses distances between dependent words to find sentence complexity. For a dependency relation in the tree, the dependency distance is calculated by finding the absolute value of the difference between the two word positions. For example, in Fig. 3.6, a dependency relation exists between `‘adds’` and `‘returns’`. The dependency distance between them is $10-3=7$ since `‘returns’` is the tenth word in the sentence and `‘adds’` is the third. The final ADD value is the sum of all such dependency distances divided by the number of dependencies [Oya11].

Net Comment Complexity

We have applied this metric to Java comment data to find the complexity of comments. Since ADD is implemented at the sentence level, we first split each comment into separate sentences based on the delimiters ‘.’, ‘?’, ‘!’. In case of Javadoc comments, we have assumed that every line beginning with an ‘@’ tag is also a separate sentence. For each sentence, we construct its dependency parse tree using a Python wrapper [Guo] for the Stanford CoreNLP library [Man14] and calculate the ADD. The net complexity value for the comment is the maximum ADD across all sentences.

3.2.2 Comment Similarity with Source Code

The second feature deals with finding the degree of similarity between a comment and its corresponding source code. We first preprocess both the comment and code texts. After tokenizing, we further split tokens on camel case and underscores to separate out any identifier names. In case of code, we also split on dots to split instances of method calls/method chaining. Next, we convert all text to lower case. In order to retain only those tokens relevant to the analysis, we remove the commonly occurring English stopwords. However, we remove only those stopwords which are not Java keywords. Finally, we also remove all punctuation from tokens.

To compute the semantic similarity between the comment and code tokens, we use the WordNet lexical database. WordNet synonym sets group words with similar meanings and

expressing similar concepts together [Pri10]. The synonym sets themselves are linked with each other to convey complex relationships between concepts.
We extract these synonym sets for all word tokens in the comment and code using the nltk interface for WordNet. The degree of similarity between two synonym sets can be found using nltk's path similarity function, which returns a score ranging from 0 to 1. Higher scores between a pair of synonym sets indicate highly similar concepts. For each synonym set in the comment, we find a synonym set in the code that gives the highest path similarity score. The final value is the average of all these scores across all comment synonym sets.

While retrieving synonym sets, the nltk interface also requires a part of speech parameter to be provided along with the word token. Different synonym sets are returned for the same token based on the provided part of speech. nltk includes support for nouns, verbs, adjectives, and adverbs. Instead of using Part-of-Speech Taggers to predict a single part of speech for the token, we retrieve all synonym sets across all applicable parts of speech for comment and code tokens to increase the probability of finding a closer match. We choose the synonym sets with the highest path similarity scores.

3.3 Comment Classification through Machine Learning

Based on their complexity and code similarity, we aim to classify comments into two categories - satisfactory and unsatisfactory, using supervised machine learning techniques. We first apply Logistic Regression to the data since Logistic Regression usually performs well for the binary classification problem. We also perform classification using Support Vector Machines and compare the results.

CHAPTER

4

EXPERIMENTAL RESULTS

Since our focus is on student-written comments, the data we used for our experiments consists of submissions and grades for two Java programming assignments from the Fall 2017 course - 'Programming Concepts - Java' (CSC 216). In this section, we have evaluated the effectiveness of comment classification using these submissions.

4.1 Data for Experiments

Each of the assignments required students to write and submit multiple Java classes, along with comments for appropriate program constructs. The course TAs have then manually evaluated all files from each submission for their Javadoc content according to the following criteria -

1. The comments evaluated are for three types of declarations - classes, methods, and fields.

2. While grading, all comments of one type are evaluated together and given a single

score. This score is based on how well the comments address behaviors, abstractions, and states of the corresponding classes, methods, and fields.

3. For each type of comment, the score assigned can be 0, 1, 2, or 3, with 3 being the highest and denoting strong comment quality.

We built the comment dataset by running the comment extraction and feature creation processes on these student submissions. We then associated this data with the corresponding grades/scores, which we used as labels for the classification task.

Each assignment submission also included certain pre-written Java files, which were provided by the TAs and were usually required for creating the project GUI. We excluded such files from our analysis since they were not written by students.

4.1.1 Comment Extraction

We first extracted every comment-code pair from all submitted files. For the two assignments, we scanned a total of 3825 files and extracted 60181 comments. Table 4.1 shows the extraction results per assignment. Although students were asked to write comments wherever appropriate, the comments that were going to be graded were class, method, and field comments. As a result, they were among the most documented constructs. Over 70% of all comments were of these three types, as seen in Table 4.2

Table 4.1 Extracted comments per assignment.

	Number of submissions	Number of files	Total comments
Assignment 1	225	2475	34229
Assignment 2	167	1350	25952
Total	392	3825	60181

Table 4.2 Top six most commented constructs.

Statement type	Number of comments	Percentage of total comments
Method Declaration	28613	47.5%
Field Declaration	11041	18.3%
Class Declaration	4474	7.4%
Constructor Declaration	4280	7.1%
Statement Expression	1986	3.3%
If Statement	1899	3.2%

4.1.2 Dataset Preparation

To build a suitable dataset for training, we carried out the following data preparation tasks for each assignment -

1. We restricted the classification to class, method, and field comments due to the lack of grading data for any other comments. Although we filtered out all other types, we considered constructor comments as method comments due to their syntactic similarity. Similarly, we treated interface and enum declaration comments as class comments since they were usually defined in separate Java files.

2. Through the feature extraction process, we calculated complexity and similarity values for each individual comment. However, since all comments of a particular type were assigned a single score, we avoided training on individual comments. Instead, we aggregated comments of each type within a submission and found the means of their complexity and similarity values. We then associated the corresponding scores with these aggregated features.

3. To make the classification more effective, we grouped the scores assigned by TAs into two major classes -

   - Scores 0, 1, and 2 were relabeled as 0.
   - Scores of 3 were relabeled as 1.

   The submissions belonging to class '1' are considered to be satisfactorily commented, while those belonging to class '0' are not.

4. Finally, for each assignment, we separated out comments by type and created three distinct datasets of class, method, and field comments, as shown in Tables 4.3 and 4.4.

Table 4.3 Datasets for Assignment 1.

Dataset	Records with label '1'	Records with label '0'	Total records
Class Comments	144	74	218
Method Comments	134	87	221
Field Comments	186	28	214

Table 4.4 Datasets for Assignment 2.

Dataset	Records with label '1'	Records with label '0'	Total records
Class Comments	110	57	167
Method Comments	119	47	166
Field Comments	142	23	165

4.2 Model Details

We utilized Logistic Regression and Support Vector Machines (SVMs) to classify class, method, and field comments from i) Assignment 1, ii) Assignment 2, iii) Assignments 1 and 2 taken together.

We trained the two classifiers on each dataset by using 75% of the data for training and setting aside the rest for validation. We also performed 5-fold cross-validation and parameter tuning on the training sets to select the best models. While defining training-validation splits or cross-validation folds, we performed stratified splitting of the class labels. This

ensured that every set and fold had an equal proportion of each class label as the original dataset. We applied our models to the corresponding validation sets to classify assignment submissions as satisfactorily/unsatisfactorily commented.

4.3 Performance of the Models

We have used accuracy, precision, recall, and f1-scores to measure the performance of our models.

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \times 100 \tag{4.1}$$

$$Precision = \frac{TP}{TP+FP} \tag{4.2}$$

$$Recall = \frac{TP}{TP+FN} \tag{4.3}$$

$$F1\ score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{4.4}$$

where TP: True Positives (instances belonging to class '1' being classified as '1')
TN: True Negatives (instances belonging to class '0' being classified as '0')
FP: False Positives (instances belonging to class '0' being classified as '1')
FN: False Negatives (instances belonging to class '1' being classified as '0')

As a result, precision denotes the ratio of correctly classified satisfactory submissions to the ones that were predicted to be satisfactory. Recall denotes the ratio of correctly classified satisfactory submissions to the ones that were actually satisfactory. The f1-score takes both precision and recall into account by calculating their harmonic mean.

Tables 4.5, 4.6, and 4.7 show the classification results obtained on all datasets in each assignment. We have analyzed these to evaluate the effectiveness of our models.

Table 4.5 Performance for Assignment 1.

Comment type	Classifier	Accuracy	Class label	Precision	Recall	F1-score	Support
Class	Logistic Regression	69.09%	0	0.56	0.53	0.54	19
			1	0.76	0.78	0.77	36
	SVM	76.36%	0	0.75	0.47	0.58	19
			1	0.77	0.92	0.84	36
Method	Logistic Regression	73.21%	0	0.73	0.50	0.59	22
			1	0.73	0.88	0.80	34
	SVM	76.79%	0	0.85	0.50	0.63	22
			1	0.74	0.94	0.83	34
Field	Logistic Regression	88.88%	0	1.00	0.14	0.25	7
			1	0.89	1.00	0.94	47
	SVM	87.03%	0	0.00	0.00	0.00	7
			1	0.87	1.00	0.93	47

Table 4.6 Performance for Assignment 2.

Comment type	Classifier	Accuracy	Class label	Precision	Recall	F1-score	Support
Class	Logistic Regression	76.19%	0	1.00	0.29	0.44	14
			1	0.74	1.00	0.85	28
	SVM	76.19%	0	1.00	0.29	0.44	14
			1	0.74	1.00	0.85	28
Method	Logistic Regression	76.19%	0	0.67	0.33	0.44	12
			1	0.78	0.93	0.85	30
	SVM	71.42%	0	0.00	0.00	0.00	12
			1	0.71	1.00	0.83	30
Field	Logistic Regression	85.71%	0	0.00	0.00	0.00	6
			1	0.86	1.00	0.92	36
	SVM	85.71%	0	0.00	0.00	0.00	6
			1	0.86	1.00	0.92	36

Table 4.7 Performance using Assignments 1 and 2 together.

Comment type	Classifier	Accuracy	Class label	Precision	Recall	F1-score	Support
Class	Logistic Regression	75.25%	0	0.74	0.42	0.54	33
			1	0.76	0.92	0.83	64
	SVM	74.22%	0	0.79	0.33	0.47	33
			1	0.73	0.95	0.83	64
Method	Logistic Regression	75.25%	0	0.78	0.41	0.54	34
			1	0.75	0.94	0.83	63
	SVM	74.22%	0	1.00	0.26	0.42	34
			1	0.72	1.00	0.83	63
Field	Logistic Regression	86.31%	0	0.00	0.00	0.00	13
			1	0.86	1.00	0.93	82
	SVM	86.31%	0	0.00	0.00	0.00	13
			1	0.86	1.00	0.93	82

4.4 Analysis

Although the accuracy values of Logistic Regression and SVM were relatively consistent, SVM tended to perform slightly better when the data was more balanced, and support for the majority class ('1') was less than twice as that of the minority class ('0'). For example, in Assignment 1 (Table 4.5), SVM performed well on instances with label '1' and achieved high f1-scores of 0.84 and 0.83 for class and method comments respectively.

However, for unbalanced datasets, as was the case in Assignment 2 (Table 4.6), the SVM was more likely to classify all observations into the majority class. Here, Logistic Regression handled the data imbalance better for method comments. Although its f1-score for the minority class dropped to 0.44, it correctly identified more True Negatives than SVM.

Similarly, Logistic Regression achieved better accuracy and f1-scores when we combined data from Assignments 1 and 2 (Table 4.7). It attained an accuracy of 75.25% for class and method comments, with f1-scores of 0.83. Both Logistic Regression and SVM performed comparatively poorly while classifying field comments. However, this could be attributed to a significantly large imbalance in the data, where support for the majority class was around six times that of the minority class. Another limitation in the analysis could be that the TAs may not have been consistent while grading. Multiple TAs grading different assignments could make the grading subjective.

Overall, our approach can be considered effective for highlighting submissions with satisfactory comments, giving sufficiently high values of accuracy and f1-scores for the class labelled '1'. It also yielded consistent results using multiple classifiers and generalized to datasets from multiple programming assignments.

CHAPTER

5

CONCLUSION

Given the importance of writing effective comments, courses teaching introductory programming have developed assignments that require students to document their code using comments. The process of evaluating these and providing feedback to students is, however, a manual one. We have developed a machine learning based approach, which analyzes the natural language used in student-written comments and determines if their programming assignments are well-commented. As a part of this process, we have automated the extraction of comments and introduced a technique to associate comments with their corresponding lines of code.

Based on the results of applying Logistic Regression and SVM, our approach can be considered as an effective preliminary test to help filter out assignments which are likely to be well-commented. At the moment, this classification is restricted to two broad classes – satisfactory and unsatisfactory. However, in the future, we aim to assign distinct grades to commented programs. Having tested our approach with class, method, and field declarations, there is also scope for evaluating comments for other program constructs as well.

Finally, we plan to use more advanced natural language processing and learning techniques to further improve our classification accuracy.

BIBLIOGRAPHY

[Avi] Aviles, J.-R. *Comment Parser*. https://pypi.org/project/comment_parser/.

[CM14] Chen, D. & Manning, C. "A Fast and Accurate Dependency Parser using Neural Networks". *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, 2014, pp. 740–750.

[Che18] Chen, H. et al. "Automatically Detecting the Scopes of Source Code Comments". *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 01. 2018, pp. 164–173.

[Cor14] Corney, M. et al. "'Explain in Plain English' Questions Revisited: Data Structures Problems". *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*. SIGCSE '14. ACM, 2014, pp. 591–596.

[Cut11] Cutrone, L. et al. "Auto-Assessor: Computerized Assessment System for Marking Student's Short-Answers Automatically". *2011 IEEE International Conference on Technology for Education*. 2011, pp. 81–88.

[DeP12] DePasquale, P. J. et al. "// TODO: Help Students Improve Commenting Practices". *2012 Frontiers in Education Conference Proceedings*. 2012, pp. 1–6.

[Guo] Guo, L. *stanfordcorenlp*. https://pypi.org/project/stanfordcorenlp/.

[Hao11] Haouari, D. et al. "How Good is Your Comment? A Study of Comments in Java Programs". *2011 International Symposium on Empirical Software Engineering and Measurement*. 2011, pp. 137–146.

[Kha10] Khamis, N. et al. "Automatic Quality Assessment of Source Code Comments: The JavadocMiner". *Proceedings of the Natural Language Processing and Information Systems, and 15th International Conference on Applications of Natural Language to Information Systems*. NLDB'10. Springer-Verlag, 2010, pp. 68–79.

[Kle11] Klein, R. et al. "Automated Assessment of Short Free-Text Responses in Computer Science Using Latent Semantic Analysis". *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ITiCSE '11. ACM, 2011, pp. 158–162.

[Liu15] Liu, Y. et al. "Analyzing Program Readability Based on WordNet". *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. EASE '15. ACM, 2015, 27:1–27:2.

[Lou18] Louis, A. et al. *Deep Learning to Detect Redundant Method Comments*. arXiv: 1806.04616 [cs.SE]. 2018.

[Man14] Manning, C. D. et al. "The Stanford CoreNLP Natural Language Processing Toolkit". *Association for Computational Linguistics (ACL) System Demonstrations*. 2014, pp. 55–60.

[MA18] Mohammadi-Aragh, M. J. et al. "Coding the Coders: A Qualitative Investigation of Students' Commenting Patterns". *2018 ASEE Annual Conference & Exposition*. ASEE Conferences, 2018.

[Oya11] Oya, M. "Syntactic Dependency Distance as Sentence Complexity Measure". *Proceedings of the 16th International Conference of Pan-Pacific Association of Applied Linguistics* (2011).

[Pri10] Princeton University. *About WordNet*. https://wordnet.princeton.edu/. 2010.

[PS05] Pulman, S. G. & Sukkarieh, J. Z. "Automatic Short Answer Marking". *Proceedings of the Second Workshop on Building Educational Applications Using NLP*. EdAppsNLP 05. Association for Computational Linguistics, 2005, pp. 9–16.

[Pyt01] Python Software Foundation. *Abstract Syntax Trees*. https://docs.python.org/3/library/ast.html. 2001–.

[SG] Sagalovskiy, D. & Grist Labs. *asttokens*. https://pypi.org/project/asttokens/.

[Sch07] Schreck, D. et al. "How Documentation Evolves over Time". *Ninth International Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*. IWPSE '07. ACM, 2007, pp. 4–10.

[Ste13] Steidl, D. et al. "Quality Analysis of Source Code Comments". *2013 21st International Conference on Program Comprehension (ICPC)*. 2013, pp. 83–92.

[Tan07] Tan, L. et al. "/* iComment: Bugs or Bad Comments? */". *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. ACM, 2007, pp. 145–158.

[Thu] Thunes, C. *javalang*. https://github.com/c2nes/javalang/.