Islamic University of Technology
Department of Computer Science and Engineering

Consistency of Comments to Source Code: An Empirical Investigation and A Dataset

Submitted by
Maksuda Islam 170042063
Ahsanul Haque 170042087
Md Safayat Hossen 170042090

Supervised by
Lutfun Nahar Lota
Assistant Professor, Department of CSE

A thesis submitted in partial fulfilment of the requirements of
the Islamic University of Technology for the degree of
Bachelor of Science in *Software Engineering*

May 13, 2022

Abstract

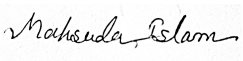
Comments in the code are a primary source for system documentation. These are indispensable for the work of software maintainers as a foundation for code traceability, maintenance activities, and the use of the code itself as a library or framework in other projects. However, the quality of the comment has been overlooked for various reasons. Also, comments are doubtful to change with the evolution of the source code. The source code gets updated whenever the changes occur, but the comments are ignored. It leaves a new developer even more confused. So the coherence between the comments and the source code must be ensured and maintained. This paper aims to provide a dataset consisting of code-comment pairs through our research work. We have annotated 9,311 classes and methods of different C# projects. 4,953 code comment pairs were taken after removing NULL, constructor, and variable. We employed a metric called Bilingual Evaluation Understudy (BLEU) to validate our human-curated dataset. This paper also includes a comparative analysis and discussion between the human-curated annotation and annotation provided by the BLEU score. A modified model from a previous study is also proposed, which obtained an accuracy of 96.56% using the performance metric AUC-ROC after fitting the model to our annotated 4,953 code-comment pairs. In contrast, the previous model gave 93% accuracy using a similar performance metric on this same dataset.

Keywords: Coherence, Bilingual Evaluation Understudy, Code-Comment pair

Declaration of Authorship

This is to certify that the work presented in this thesis is the outcome of the analysis and experiments carried out by Maksuda Islam, Ahsanul Haque, Md Safayat Hossen under the supervision of Lutfun Nahar Lota, Assistant Professor of Department of Computer Science and Engineering (CSE), Islamic University of Technology (IUT), Gazipur, Dhaka, Bangladesh. It is also declared that neither this thesis nor any part of it has been submitted anywhere else for any degree or diploma. Information derived from the published and unpublished work of others have been acknowledged in the text and a list of references is given.

Authors:

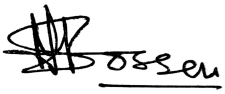


Maksuda Islam

Student ID: 170042063

Ahsanul Haque

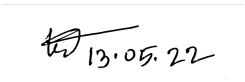
Student ID: 170042087



Md Safayat Hossen

Student ID: 170042090

Supervisor:



Lutfun Nahar Lota

Assistant Professor

Department of Computer Science and Engineering

Islamic University of Technology

Acknowledgement

The moment is right for us to submit our Bachelor's thesis. We would want to thank Allah Almighty for his blessings, which enabled us to conclude our thesis research successfully. We would not be here without Allah's mercy. Allah be glorified.

We want to thank Lutfun Nahar Lota, Assistant Professor in the Department of Computer Science and Engineering at the Islamic University of Technology, for serving as our advisor and mentor. Her inspiration, suggestions, and insights have been invaluable for this thesis. Without her assistance and proper direction, this thesis would not have followed the correct research route. Her valuable opinion, time, and input throughout the thesis work, from the initial phase of thesis topic introduction, research area selection, and implementation, assisted us in rightly completing our thesis work. We would like to thank Mohammad Anas Jawad Sir and Md. Mezbaur Rahman Sir lecturer in the Department of Computer Science and Engineering at the Islamic University of Technology for their exceptional contribution. Lastly, for our work, insights from industry experts were necessary. Teachers from the Software Engineering Lab of the Department of Computer Science and Engineering at the Islamic University of Technology helped by giving their valuable opinion as former industry experts. We want to express our gratitude towards them as well.

We want to extend our vote of thanks to the jury members of my thesis committee for the many interesting comments and criticism that helped improve this manuscript. Lastly, we are deeply grateful to our friends and family for their unconditional support. This work would have never been completed without the consistent support and encouragement from them throughout the program.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Software engineering is different from other engineering disciplines. Where most engineering projects follow a strict no-change policy regarding changes, in software engineering, changes are a natural phenomenon. Software evolves with time and the development process is never-ending. Apart from new features, changes may come for slight modifications, small bugs, refactoring, etc. All of these changes have an impact on the source code. As software development is a long process, it is very hard to track why or how a function or method, or class is written in source code. And to keep track of this, we need documentation. For documenting the source code, comments in the source code are the primary source [2]. Developers usually make comments with a code fragment that provide insightful information about a software system for the management of software evolution and maintenance. Nowadays, there are many automation tools for comment generation but Comments are more legible by humans because they are written in natural language [3, 4].

The value of software documentation, such as comments, is widely acknowledged among developers. However, it is not uncommon for this material to be ignored. This can happen for a variety of reasons, including a tight deadline, pressure from the management, human error, and so on. Sometimes comments can be vague or inconsistent with the class or the methods.

Also, there are cases when programmers write the code and related comments properly. Then there are some changes in the requirements thus the source code changes. These changes are not updated in the comments and the comments become inconsistent [5].

We need comments to enhance the understandability of the source code. By reading comments we get a basic idea about the functionality and implementation of the code. So when there are some inconsistent comments in the source code, the reader gets misled. This can confuse and do more harm than good. So, maintaining consistency in the source code comments is a very important part of the documentation. Several different ways have been offered for tracking the inconsistency of source code and associated comments. The majority of ways use Information Retrieval (IR) techniques to collect lexical content, assuming that the source code and comment textual information are the same.
This assumption, however, can be challenged [6] in several ways.

Due to a lack of standard datasets, there isn't enough literature to track this inconsistency. With a suggestion to quantify the coherence between source code and comment, a benchmark dataset has been supplied [1]. The lexical similarity was determined using the Vector Space Model to classify the text using tf-idf [7], and the code-comment inconsistency was determined using the Support Vector Machine (SVM).

1.2 Our Contribution

To date, the datasets that exist are language-dependent in terms of the coherence between the comments and the implementation. Particularly, all datasets are constructed using Java projects. In this paper, we will attempt to create a database containing information about the consistency of the lead comment in terms of class or method implementation of C# projects. We will describe the process by which the dataset was created. This study aims to determine whether the lead comment of a method or class conveys the method's or class's primary intent and implementation details, such as the types of parameters and returned values, accurately.

We disregard in-line comments in this scenario because they only provide information about very specific parts of the code base, but we accept the docstring. Thus, coherence exists between a method's or class's lead comment and its source code if the comment describes both the method's or class's intent and its actual implementation. Our dataset contains annotations on the coherence of 4953 methods and classes from 10 C# projects implemented as open-source projects. This dataset was created using a protocol that we also describe in this paper. Another contribution of this study is that we have modified a previously proposed model which helped us to gain better accuracy.

1.3 Thesis Structure

The rest of the thesis is organized into chapters given as follows,

In chapter 2, we included all the necessary background studies that would be used further in the paper, it describes Lead Comment, Coherent, InCoherent, Doxygen, Topic Modeling, LDA.

In chapter 3, we presented all the related works that have influenced our study directly or indirectly.

In chapter 4, we described the whole methodology, starting from the data preparation, and the evaluation of the dataset.

In chapter 5, we discussed about evolution.

In chapter 6, we discussed Result Analysis & Discussion of Dataset.

In chapter 7, we mentioned all the threats and the explanation, validating all the threats from our study.

In chapter 8, the thesis work's future pursuit and our plan to follow it through. And with that, it concludes the thesis work.

Chapter 2

Background Studies

Comments are the foremost source of code documentation. Due to this, their requirement is being consistent with the written code. Computing code comment consistency is essential since a written method can be complex enough for another developer to be understood. There can be dependencies among the functions. The project can be one of the open-source libraries. Our primary focus is to reduce the language dependency of this research topic by providing a dataset for another programming language and by proposing a language-independent model. With that being the goal, a few terminologies need to be addressed first. Those terminologies are discussed in this section.

2.1 Importance of checking the consistency of code comments over generating consistent comments

Previously many studies have been conducted on generating consistent comments, one widely known example of that is Javadocs which works as a documentation tool for java language. Generating consistent comments by analyzing the code is indeed very large progress, although one mentionable lacking of it is not considering the evolution process of a software project. One example [8] is added below to highlight this lacking:

```
/**
 * Checks if one of the graphs is from unsupported graph type
 * and throws IllegalArgumentException if it is. The current
 * unsupported types are graphs with multiple-edges.
 * @param graph1
 * @param graph2
 * @throws IllegalArgumentException
 */
protected static void assertUnsupportedGraphTypes(
  Graph graph1,
  Graph graph2)
  throws IllegalArgumentException
{
  Graph [] graphArray = new Graph [] {
   graph1, graph2
  };
```

Figure 2.1: Method body with the written lead comment

```
- protected static void assertUnsupportedGraphTypes(
- Graph graph1,
- Graph graph2)
+ protected static void assertUnsupportedGraphTypes(Graph g)
    throws IllegalArgumentException {
- Graph [] graphArray = new Graph [] {
- graph1, graph2
- };
- for (int i = 0; i < graphArray.length; i++) {
- Graph g = graphArray[i];
```

Figure 2.2: Snippet from version control after the code changed due to evolution

From figure 2.1 we can see a method-code and lead comment can be seen and from figure 2.2 we can see the changed code of the previous snippet (figure 2.1), previously passed two parameter is now changed into one, whereas from the both figure 2.1 & figure 2.2 it is noticeable that the comment has remained unchanged. So, the comment which was consistent with the previously written code is not inconsistent with the modified code. This show that being able to detect the current state whether the code is consistent or not is equally important if not more than generating consistent comment.

2.2 Importance of Topological Order of comment

Comments can be written on top of the code, inside the code, and at the end of the code. Usually, comments written between class and method explain or describe the fragment of that code. The same goes for comments written below. Whereas comments were written on

top of the code generally describe the whole code. So, following the topological order of the comment is essential.

2.3 Lead Comment

The corresponding comment is written just above the code.

```
/// <summary>
/// An opaque structure holding the description of an activation operation.
/// </summary>
public ActivationDescriptor()
{
    _desc = new cudnnActivationDescriptor();
    res = CudaDNNNativeMethods.cudnnCreateActivationDescriptor(ref _desc);
    Debug.WriteLine(String.Format("{0:G}, {1}: {2}", DateTime.Now, "cudnnCreateActivationDescriptor", res));
    if (res != cudnnStatus.Success) throw new CudaDNNException(res);
}
```

Figure 2.3: Snippet of Method Code-Comment pair from GitHub of the project ManagedCUDA

In figure 2.4, a comment was written on top of the method or class which explains briefly the corresponding code known as the lead comment.

2.4 Coherent and InCoherent

When the meaning of a code snippet and respective comment is properly aligned on the other hand When the meaning of a code snippet and respective comment is not aligned properly.

2.5 Doxygen

Doxygen is a familiar mechanism for rendering documentation. It can also be configured to extract the code structure and its corresponding comment with few modifications. It primarily supports c++ code files. However, it also works for other prevalent programming languages such as C, C, PHP, Java, Python, IDL, Fortran, etc.

2.6 Bilingual Evaluation Understudy (BLEU)

A score for comparing a candidate sentence of text to one or more reference sentences of text. An ideal match results in a score of 1.0, and an ideal mismatch results in a score of 0.0.

This metric is widely used to evaluate machine translation results [9,10]. Recent works in source code summarization have used BLEU scores as the primary evaluation metric [11,12]. NLTK provides the function for evaluating a candidate sentence against one or more reference sentences.

In our case, the reference text is code, and the candidate text is a comment. An example of computing BLEU Score is given below:
Tokenized Reference Sentence: [the', 'quick', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
Tokenized Candidate Sentence: ['the', 'fast', 'brown', 'fox', 'jumped', 'over', 'the', 'lazy', 'dog']
The BLEU Score is: *0.75*
Because two words, "quick" & "fast" were not a match.

2.7 Topic Modeling

Topic Modeling aims at automatically finding topics in a given textual document that represents it as a cluster of words. It does not use tags and training data. Clusters of words are often used together based on the frequencies of words and frequencies of co-occurrence of words within one or more documents.

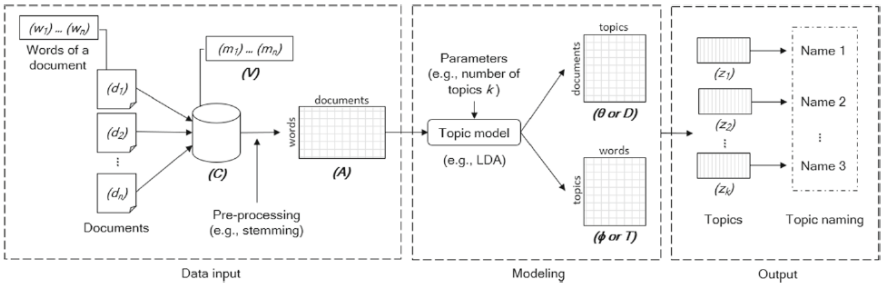


Figure 2.4: General Topic Modeling Process

Here,

w = words, d = documents, C = corpus (set of documents),

V = set of m unique words that appear in a corpus,

A = Term-Document Matrix

Z = Topics (collection of terms that co-occur frequently in the documents of a corpus), Theta or D= Document Topic matrix (contains the possible topics (represented by K above) that the documents can contain)

2.8 Latent Dirichlet allocation(LDA)

Latent Dirichlet allocation is an often-used algorithm for topic modeling, which is driven by two principles:

- Every document is a mixture of topics: In LDA, each document is assumed to contain words from several topics in individual balances. For example, in a two-topic model, if "Document A is 90% topic X and 10% topic Y, while Document B is 30% topic X and 70% topic Y."

- Every topic is a mixture of words: For instance, we could picture a two-topic model of Asain news, with one topic for "politics" and one for "entertainment." The hugely occurred words in the political topic might be "President", "Congress", and "government", while the in entertainment topics might be words like"movies", "television", and "actor". Notably, words can be shared between topics; a word like "budget" might occur in both equally.

Chapter 3

Related Works

Commenting is the most crucial aspect of making source code comprehensible. Because comments are more direct and detailed, developers like them. More information regarding the implementation of the source code may be found in the comments. Source code, on the other hand, changes as program development progresses. Again because of deadline issues, we sometimes avoid comments. So, for a variety of reasons, connected comments do not appear. For example, a remark may contain information that is irrelevant or contradictory to the source code it refers to [13]. When this happens, comments can be confusing and misleading to developers, raising the cost of understanding, maintaining, and testing source code [14, 15]. A discrepancy between the method comment and the body, on the other hand, might suggest a flaw in the body or a flaw in the remark, which could lead to method callers being misled and so creating flaws in their code [15]. Although developers commonly agree on the importance of software documentation, comments are often overlooked due to release deadlines and other time pressures during development [4]. All the aspects create gaps and inconsistencies in comments with respect to its source code.

Recent research for better software maintenance has yielded techniques for measuring the relationship between code and comment [16] and highlighted the need to maintain consistency between the two. Several methods for identifying ancient document comments have been published [14, 16–18]. Because document comments are well-structured for analysis,

we were able to identify out-of-date document comments with high precision and recall [18]. While detection was performed at the function/method level for methods that examined only doc or block/line comments [14, 16], it was performed at the topic level for methods that examined both types of comments [14, 17].

A study created a technique for spotting obsolete comments during the evolution of code using a machine learning-based approach [19]. On the basis of the source code change and the code-comment relationship before and after the source code change, potential comment adjustments can be recognized. Using machine learning techniques, they were able to detect changes in 64 distinct software properties, as well as the state of comments and their connection to the code before and after changes.

The developer who writes the code typically decides whether or not to include comments. The manner in which developers comment on code determines both the quality of the comments and the legibility of the code. The practice of adding comments to code in various programming languages is investigated to better understand this term [20]. The practice of commenting on code has evolved over time as a result of the development of natural programming languages and the evolution of ecosystems. To test this hypothesis, they will look at how developers' commenting practices change over time, with a focus on the modern Smalltalk environment Pharo.

To understand this inconsistency, researchers are trying to find out something that will help the developers for a long time. A method for retrieving traceability linkages between source code and free-text documents based on information retrieval. In two case studies, they used both a probabilistic and a vector space information retrieval approach to trace C++ source code on manual pages and Java code on functional requirements [21]. An approach for detecting code comment inconsistencies in locking and calling mechanisms was proposed in another paper. They were only interested in suggestions about programmers' assumptions and requirements [14]. For testing Javadoc's comments another paper was published [15]. The

authors looked into null value method properties and related exceptions. They used Natural Language Processing to discover conflicts between codes and comments utilizing @param tags in Javadoc comments and the null parameter exception statement. Their effort is confined to merely making remarks about null references and throwing exceptions.

All the above literature gives us a perfect picture that the problem of inconsistency between code and its comments pretty much exists and there is a lot to work on. Many methods and models have been applied for a long time. But use of Topic Modeling is rare particularly in this topic. So we are trying to use this methodology on this specific topic.

Camila Costa Silva et al. analyzed 111 publications from ten highly regarded software engineering sites and examined subject modeling from four distinct perspectives in a single research paper. Which topic models and modeling techniques have been used, which textual inputs have been used for topic modeling, how textual data was prepared via pre-processing for topic modeling, and how generated topics, such as word clusters, were named to give them a human-understandable meaning are the four elements of their analysis. On a majority-vote basis, every aspect was answered. [22].

Recently in 2020, a methodology was created using topic modeling to check the similarity of comments with respect to its source code [23]. This paper describes a topic modeling-based method for examining the comments' congruence with the source code. Because comments should be consistent with the source code to which they refer, a model was created to evaluate the consistency of comments and, by extension, their quality. All comments and source code in the software classes were retrieved for each of the analyzed projects. Another study, which is related to our research topic, was published in 2021 [2]. In this research, an effective method for automatically extracting dominating topics and determining the coherence of a code snippet and its related comment is proposed. This approach was done using a benchmark dataset containing 2.8K Java code-comment pairings, which revealed that the suggested approach outperformed the existing state-of-the-art Support Vector Machine on vector space

model on several evaluation measures.

The findings of a manual assessment of the coherence between comments and implementations of 3636 methods taken from three Java open-source software systems are presented in another study [1]. The evaluations that resulted were compiled into a dataset that was made publicly available thereafter. This paper also tries to find links between coherence and lexical information provided in source code and method lead comments, which is another important contribution.

Chapter 4

Methodology of Dataset Creation

4.1 Dataset Preparation

To collect a set of naturally written comments by developers, we used the extracted code-comment pair using Doxygen by Gelman et al [24]. They chose the GitHub repository based on having a redistributable license and at least 10 stars. We owned reverse engineering here by first exploring the extracted code comment pair data pool, later from that data pool we find out the list of c# projects from the given file name. From the found project list we selected 10 projects for annotating the code-comment pair from those. After that, we extracted the code-comment pair of those selected projects only and, annotated those using our defined set of rules for methods and classes differently. We take the Comment written on top of the class or method. We only consider the code written for class and method and manually checked the comments written on top of the code of extracted pairs.

Table 4.1: Selected Projects & Description

ID	Project Name	Project Description	No of Stars
1	ManagedCUDA	An integration of NVidia's CUDA in .net applications written in C#	331
2	RoboSharp	RoboSharp is a .NET wrapper for the awesome Robocopy windows application	174
3	hpack	Header Compression for HTTP/2 written in C#	11
4	intense	Controls, templates, and tools for building Universal Windows Platform apps for Windows 10	90
5	Potato	A free remote control (RCON) tool for game servers, currently supporting Battlefield: Bad Company 2, Battlefield 3, Medal of Honor: Warfighter, and Battlefield 4	20
6	EmojiVS	EmojiVS is a Visual Studio 2013 and 2015 extension to display GitHub emojis in the editor	84
7	xunit-performance	Provides extensions over xUnit to author performance tests	186
8	SimpleAuth	Simple Auth embeds authentication into the API so one don't need to deal with it	158
9	BDInfo	BDInfo collects video and audio technical information from unencrypted Blu-ray movie discs	18
10	Gitter	Developed as a standalone repository management tool for Windows with powerful GUI	24

Table 4.2 includes the rules used for annotating methods which were also used in the previously discussed paper of Corazza et al [1]. in the related work section. In our work, by lead comment we meant the corresponding comment with the code. Table 4.3 includes the rules we have used to annotate the classes.

Table 4.2: Established Rules for annotating Method Code - Comment Pair [1]

ID	**Rules for Methods**
1	The comment of the method describes the intent of the source code of this method.
2	The intent described in the lead comment of the method corresponds to the actual implementation of this method.
3	The comment of the method describes all the expected behaviors of the actual implementation of this method.
4	If the comment of a method provides implementation details (e.g., names and types of input parameters according to JavaDoc), this information is aligned with the implementation of this method.
5	If the comment of the method provides details about input parameters, their intended use is properly described.

Table 4.3: Proposed Rules for annotating Class Code - Comment Pair

ID	Rules for Classes
1	The lead comment for the class describes the purpose of the declaration of that class.
2	The lead comment for the class gives the idea of the constructor of that class along with the parameters used in the constructor according to the alignment to the implementation of the class.
3	Comment of a class provides the basic implementation details(e.g., names and types of input parameters). The information is aligned with the implementation of the class.
4	The lead comment contains information about the inheritance properties of the class.
5	The lead comment contains information about the polymorphic methods of the class.

Using the defined set of rules we have annotated the data into four different labels: Those are:

1. Coherent
2. Not-Coherent
3. Not Sure
4. Null

To annotate the dataset we have used a human baseline. Detecting the relation between the code and comment is a subjective task. To see how good humans perform, we have conducted an experiment where we took 20 coherent code-comment pairs(generated by JavaDoc, a widely used source of coherent comments) and mixed them randomly with 30 self-written not-coherent comments for both classes and methods, and gave them to 3 human annotators who are undergraduate students(Software Engineering). They were told to read the code-comment pairs one by one and annotate those according to the rules mentioned before. We excluded code-comment pair samples with null values, constructors, and variables.

The first question is whether the code snippet is class-based or method-based.

- If the code snippet is class-based then they will use the rules of Table 4.3(Rules for class)
- If the code snippet is method based then they will use the rules of Table 4.2(Rules for method)

If the code comment pair passes the rules, then that will be labeled as coherent otherwise annotation will be Not-Coherent. During the process we are also weeding out the code comment pairs which are missing either value, we labeled them as NULL. As we are dependent on human judgment which is very prone to confusion, there is another label for that. Individual annotators will label those code-comment pairs as Not-Sure. After the whole annotation process, we separate the code-comment pairs which are labeled as Not-Sure. Then annotators sat together and decided whether one Not-Sure is more inclined to be coherent or non-coherent. Then labeled that accordingly. When the discussion got futile and annotators were still confused, we decided to exclude those code-comment pairs.

Chapter 5

Evaluation

We used a bilingual evaluation understudy (BLEU) score for each code-comment pair to check the validity and compare our annotated dataset. Many notable recent works have used the BLEU score as their primary evaluation metric.

A BLEU score compares a tokenized predicted sentence to a tokenized reference sentence. The score is based on the average precision of the predicted sentence compared to the reference sentence. The BLEU score calculation incorporates a succinctness penalty that penalizes overly short predictions. Thus, the two components balance each other, awarding the model for predicting only n-grams contained in the reference sentence and awarding the model for making appropriately lengthy predictions. We use BLEU-2, meaning the BLEU score calculation includes 2-grams and every smaller n-gram. Consistent with Hu et al. [11], we also did not use smoothing to settle the lack of higher-order n-gram overlap. Mathematically, we calculate BLEU-2 as,

$$BLEU = Be^{\sum_{n=1}^{2} w_n \log(p_n)} \tag{5.1}$$

$$B = \begin{cases} 1, & if c > r \\ e^{1-r/c}, & otherwise \end{cases} \tag{5.2}$$

here,

p_n = is the ratio of n-grams in the prediction that are also in the reference,

w_n = is the weight associated with those n-grams,

c =is the length of the prediction, and

r = is the length of the reference.

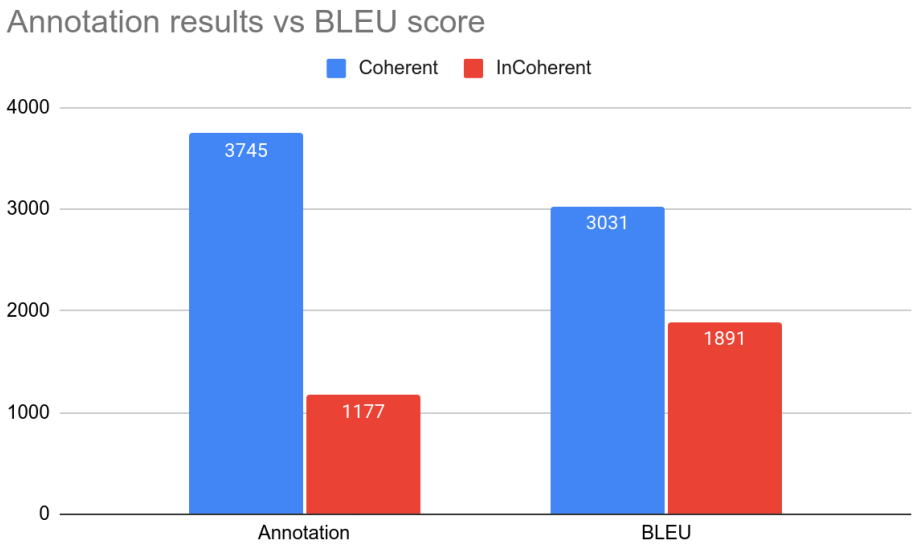


Figure 5.1: Chart for Human Curated Annotation vs BLEU Score

From figure 5.1, it is visible that the BLEU labeled 741 pairs as Incoherent whereas in our human-curated dataset the annotators annotated those as consistent one. The reason is this, we think is BLEU does not consider the meaning of words, while It is perfectly acceptable for a human to use a different word with the same meaning. The BLUE score also ignores paraphrasing.

Chapter 6

Result Analysis & Discussion of Dataset

At first from the selected projects, we have taken 10000 pairs of code and comments. then we annotate that 10000 pairs in the mentioned 4 categories. After removing the duplicates there are 9311 pairs of code and comments. We annotated as Coherent if the comment is aligned with the given rules. We annotated as InCoherent if the comment is not aligned with the given rules. We annotated as Null if there is no content either in the comment or in code. After annotating in this way, we found that 58% pair is Coherent. 18.1% is InCoherent and the remaining is Null.

Table 6.1: Amount of distributed classes

Coherent	58.0%	5400
InCoherent	18.1%	1683
Coherent	23.9%	2228
Total		9311

We can see that from the table 6.1 that there were a huge number of null pairs in the extracted pair from the data pool. Since it was extracted using Doxyzen often fails to extract code or comment as a result the space stays as null space. Then we remove the Null data from the dataset which makes the percentage 76.3 and 23.7 for Coherent and InCoherent and the total is now 7083.

Table 6.2: Amount of distributed classes after removing data with null

Coherent	76.3%	5400
InCoherent	23.7%	1683
Total		7083

As we want to consider only class and method, we exclude everything apart from that. After that, the percentage remains the same the total pair of code comments is now 4922.

Table 6.3: Amount of distributed classes after removing data with code of constructor and variables

Coherent	76.1%	3745
InCoherent	23.9%	1177
Total		4922

6.1 Threats to Validate

One of the external threats of this work can be using selected systems and the programming language. To validate this external threat, we have used checklist-based annotation work, so that the dataset can be easily extendable for other languages. Our work can be seen as an extension work of Anna Corazza et al work, they proposed a checklist-based public benchmark dataset for Java programming language.

Another internal threat can be raised about the qualification of the annotators. All three annotators are fresh graduates of the Software Engineering program with the experience of real-world projects for being involved in the industry for 4-6 months. The used set of rules was also validated by a 15 years industry expert.

Chapter 7

Model training Detail & Result

In the previous studies different kinds of approaches have been proposed to detect code comment consistency. We exercised the topic modeling approach because automatically analyzes text data to determine cluster words for a set of documents. As we chose to employ the topic modeling approach, naturally we went through the previous studies which employed the same. While doing that, a previously published model that employed a topic modeling approach along with an ensemble machine learning technique caught our attention because of the use of a multi-model technique which we immediately thought of adapting with our modifications [2].

7.1 Model Training Detail

Our modifications include adding natural language processing-based preprocessing techniques like lemmatization, removal of stop words, etc. Another modification of our is employing logistics regression to infuse the extracted features from the multi-model random forest. We used stratified k-fold as a cross validator.

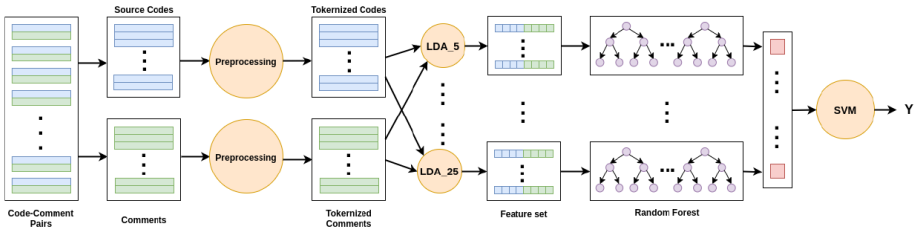


Figure 7.1: Overall process of the method

Along with the modification the overall process of the method can be summarised as follows:

- Preprocessing of code-comment pairs: The newlines, tabs and special characters, Extra whitespaces, and stop words were removed at first. Words were also split based on camel cases. Through this, we found tokenized pairs which were turned into a bag of word tokens. Later on, lemmatization was applied.

- Vectors from code-comment: We used Count Vectorizer to turn the tokenized code-comment into an index vector. Then index vectors of code/comment were separated from each other and passed into two identical LDA models for training corpora separately. For n number of topics, both LDA models provided different probability distributions. Afterward, the achieved probability distribution for a code-comment pair is concatenated to produce a feature vector.

- Multiple Classifier: Extracted features of n number of topics were fed into a random forest classifier. The random forest model delivers a consistency score or probability score of being consistent for every pair of code-comment using the "proba" function. Since the number of topics for both corpora of comment-code can vary, it was necessary to include a different number of topics to find the informative and compelling features. Due to necessity, 20 different feature sets were extracted as seen from fig 7 by changing the number of topics in the LDA model every time, and these were fitted into 20 different random forest classifiers.

- Infusing with Logistic Regression: Each random forest model yields a consistency score for a code-comment pair. These consistency scores emanated from 20 different random

forests were needed to be soldered. Here, Logistic Regression is used to solder the consistency scores supplied by 20 different random forests and predict the result.

7.2 Result Analysis

After fitting the model on our annotated 4953 code-comment pair we got the value 96.56% which is better than the previous model which gave 93.3% using the performnace metric AUC-ROC. The value difference is shown in table 7.1 .

Table 7.1: Performance Comparison in terms of Area Under Precision-Recall Curve

Dataset	**Previous Model**	**Proposed Model**
C# Dataset	.933	.9656

Table 7.2: Attained class-wise accuracy

	Precision	**Recall**	**F1-Score**	**Support**
Coherent	0.98	0.98	0.998	770
Not-Coherent	0.64	0.88	0.74	221
Accuracy			0.97	991
Macro Avg	.95	.95	.95	991
Weighted Avg	.97	.97	.97	991

Figure 7.2: Class-wise Confusion matrix result

The class-wise value received after using our proposed modified model on our prepared dataset can be observed from figure 7.2. We can see the model correctly identifies 730 pairs from test data as coherent pairs whereas incorrectly labeled 24 coherent pairs as non-coherent ones. For non-coherent or in-coherent classes, the model correctly identified 215 pairs as non-coherent ones where wrong labeled 16 non-coherent pairs as coherent ones.

Chapter 8

Conclusion & Future Work

We have presented a dataset of an investigation on the coherence between comment of methods and lead comments of class and their corresponding implementations. Description of the process, problems and shortcomings, and the mitigating way faced while creating the dataset is also delivered. This work can be seen as an extensional work of Corazza et al. [1]. As a step forward in this future research direction, making it language-agnostic should be the prime focus. The topological order of the comment should be analyzed more closely. As far as the model is concerned, more ways other than topic modeling can be explored. Even though the BLEU score has been used before to evaluate these kinds of datasets, none of the code or comments were machine-generated, so exploration of different metrics to evaluate this kind of dataset should be a priority.

References

[1] A. Corazza, V. Maggio, and G. Scanniello, "Coherence of comments and method implementations: a dataset and an empirical investigation," *Software Quality Journal*, vol. 26, no. 2, pp. 751–777, 2018.

[2] F. Rabbi, M. N. Haque, M. E. Kadir, M. S. Siddik, and A. Kabir, "An ensemble approach to detect code comment inconsistencies using topic modeling.," in *SEKE*, pp. 392–395, 2020.

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

[4] 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*, pp. 215–223, 1981.

[5] 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, IEEE, 2007.

[6] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing source code vocabulary," in *2010 17th Working Conference on Reverse Engineering*, pp. 3–12, IEEE, 2010.

[7] R. Baeza-Yates and P. Raghavan, "Next generation web search," in *Search Computing*, pp. 11–23, Springer, 2010.

[8] N. Stulova, A. Blasi, A. Gorla, and O. Nierstrasz, "Towards detecting inconsistent comments in java source code automatically," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 65–69, IEEE, 2020.

[9] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pp. 311–318, 2002.

[10] C. Callison-Burch, M. Osborne, and P. Koehn, "Re-evaluating the role of bleu in machine translation research," in *11th conference of the european chapter of the association for computational linguistics*, pp. 249–256, 2006.

[11] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred api knowledge," 2018.

[12] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2073–2083, 2016.

[13] F. Salviulo and G. Scanniello, "Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals," in *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pp. 1–10, 2014.

[14] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments?*," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pp. 145–158, 2007.

[15] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@ tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 260–269, IEEE, 2012.

[16] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and software bugs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2293–2304, 2012.

[17] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pp. 68–75, 2005.

[18] N. Khamis, R. Witte, and J. Rilling, "Automatic quality assessment of source code comments: the javadocminer," in *International Conference on Application of Natural Language to Information Systems*, pp. 68–79, Springer, 2010.

[19] Z. Liu, H. Chen, X. Chen, X. Luo, and F. Zhou, "Automatic detection of outdated comments during code changes," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 154–163, IEEE, 2018.

[20] P. Rani, S. Panichella, M. Leuenberger, M. Ghafari, and O. Nierstrasz, "What do class comments tell us? an investigation of comment evolution and practices in pharo smalltalk," *Empirical Software Engineering*, vol. 26, no. 6, pp. 1–49, 2021.

[21] D. Lucia *et al.*, "Information retrieval models for recovering traceability links between code and documentation," in *Proceedings 2000 International Conference on Software Maintenance*, pp. 40–49, IEEE, 2000.

[22] C. C. Silva, M. Galster, and F. Gilson, "Topic modeling in software engineering research," *Empirical Software Engineering*, vol. 26, no. 6, pp. 1–62, 2021.

[23] M. Iammarino, L. Aversano, M. L. Bernardi, and M. Cimitile, "A topic modeling approach to evaluate the comments consistency to source code," in *2020 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, IEEE, 2020.

[24] J. Moore, B. Gelman, and D. Slater, "A convolutional neural network for language-agnostic source code summarization," *arXiv preprint arXiv:1904.00805*, 2019.