International Journal of Computer Vision and Signal Processing, 14(1), 78-88(2024)

The Integrity of Source Code Commenting: Benchmark Dataset and Empirical Analysis

Maksuda Islam*, Md Safayat Hossen, Ahsanul Haque, Md. Nazmul Haque, Lutfun Nahar Lota

Department of Computer Science & Engineering,
Islamic University of Technology, Gazipur, Bangladesh

Abstract

Code comments are a vital software feature for program cognition & software maintainability. For a long time, researchers have been trying to find ways to ensure the consistency of code-comment. While doing that, two of the raised problems have been dataset scarcity and language dependency. To address both problems in this paper, we created a dataset using C# projects; there are no annotated datasets yet on C#. 9,310 code-comment pairs of different C# projects were extracted from a data pool. 4,922 code-comment pairs were annotated after removing NULL, constructor, and variable. Both method-comment and class-comment were considered in this study. We employed two evaluation metrics for the dataset, one is Krippendorff's Alpha which showed 95.67% similarity among the rating of three annotators for all the pairs & other is Bilingual Evaluation Understudy (BLEU) to validate our human-curated dataset. An ensemble machine learning model with topic modeling is also proposed, which obtained 96.2% using the performance metric AUC-ROC after fitting the model to our proposed dataset.

Contribution of the Paper: A novel dataset, a set of rules for human annotation, and machine learning models to automate the evaluation process.

Keywords: Code-Comment Consistency, Coherence, Bilingual Evaluation Understudy, Krippendorff's Alpha

IJCVSP

ISSN: 2186-1390
http://cennser.org/IJCVSP

Article History:
Received: 10 August 2023
Revised: 23 September 2023
Accepted: 20 February 2024
Published Online: 25 February 2024

1. INTRODUCTION

To make software systems easy to modify or reuse, developers need to use comments in their code. Adding comments helps to explain the design's implementation details and the intention behind it. Compared to source code, comments are more straightforward, descriptive, and easy to understand [1, 2]. As a result, comments are the primary documentation source for a software system. They greatly assist in understanding the source code during the development and maintenance phases, ultimately reducing maintenance costs [1]. Comments provide developers with a valuable resource to maintain and enhance software applications.

For regular coding, debugging, and maintenance tasks, developers spend extensive time navigating and exploring existing code [3, 4]. If they could understand the code by reading the description, largely known as the "comment", this extensive amount of time could be saved. One way to eliminate all these blockages could be using the auto comment generator. Generating auto comments also comes with a problem. Since the software is an ever-evolving process, the system is constantly evaluated. Due to multiple programmers working on the same codebase, tracking the changes and descriptions of written functions and classes is often challenging. Developers usually comment with a code fragment that provides insightful information about a software system for managing software evolution and maintenance. Nowadays, there are many automation tools for comment generation, but comments are more legible when written by humans because they are written in natural language [1, 2]. Comments are essential to enhance the understandability of the source code. The reader gets misled when there are inconsistent comments in the source code. This can confuse and do more harm than good.

*Corresponding author
Email addresses: maksudaislam@iut-dhaka.edu (Maksuda Islam), safayathossen@iut-dhaka.edu (Md Safayat Hossen), Hoque.talha@gmail.com (Ahsanul Haque), nazmul.haque@iut-dhaka.edu (Md. Nazmul Haque), lota@iut-dhaka.edu (Lutfun Nahar Lota)

Researchers have worked for many years to mitigate the issue of inconsistent code and comments. In addition to other solutions, determining whether code and comment are consistent with one another at any point in the software lifecycle was identified as a crucial solution. To accomplish this, a standardized dataset is required. Although there is an insufficiency of the existing dataset, several works of literature have proposed various datasets. However, the majority of datasets are produced by Java Programming Language projects.

Alongside the Java programming language, C# is gaining popularity due to its extensive libraries, third-party software, large number of communities, etc. C# is the tenth most popular programming language according to a survey conducted in 2022[1]. C# is tightly coupled with the language.Net framework, one of Microsoft's most successful programming languages, is widely utilized by professional software developers. According to our exhaustive review of the existing literature, there is no dataset for the C# programming language that considers the coherence of code and its accompanying comment.

Java, an object-oriented programming language, supports more separation of concerns and good segregation of concepts; as a result, it produces a good number of classes [5]. Most of the previous work on code-comment consistency is done in Java language which does not mention working on class modules. In a systematic literature review, it was mentioned that 87% of the studies done in the last decade are from the Java system, and 50% of the studies focus on specific comment types like method or inline [6]. Multiple notable researchers worked on the method and comment pair only [7, 8, 9]. C# is also an object-oriented programming language that requires writing many classes. So, computing the coherence of both class comment and method comment should be considered because both class comment and method comment aid C# program and project understanding.

To solve the aforementioned issues, in this paper, we described the complete process of creating a dataset that contains an annotation of 4922 code-comment pairs of C# projects having both classes and methods. Our study is focused on determining whether the lead comment(3.2) of a method effectively describes the overall method or not. In the same way, whether the lead comment of a class effectively describes the general intent of the class as well as implementation details, for example, parameters type and returned values, etc. In-line comments are not considered in this work. Therefore, if a lead comment of a method or a lead comment of a class describes the intent of the method or class and its primary implementation detail, then that comment and the source code are coherent. To evaluate the human annotation process, we used two evaluation metrics. Krippendorff's Alpha & BLEU. First, Krippendorff's Alpha assesses the rating similarity among three annotators. Second, BLEU checks how much our final combined annotation matches the lexical similarity between code and comment. Our primary contribution to this paper is the proposed set of rules to determine whether or not the lead comment of the class is coherent with it, the annotated benchmark dataset, which holds 4922 code-comment pairs of C# projects.
To list down the contribution of this work is as follows:

- Took the class-comment pair into consideration and proposed a set of rules to annotate them
- Showed the qualitative analysis of the dataset through two evaluation metric
- Annotated class-comment and method-comment pair for C# programming language
- Enhanced topic modeling based model to automatically classify code-comment consistency
- Discussed possible threats to the work

The paper is structured as follows. In Section 2, we discussed related work, and background studies are presented in Section 3. We discussed the methodology in section 4. We reflected on the results and discussed them in section 5. The implication of this work to the practitioners is mentioned in section 6. In section 7, we validated the possible threats that could be raised by readers. The conclusion and future of the paper are in section 8.

2. 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 changes as program development progresses. However, occasionally developers fail to update comments to reflect progress. While most developers recognize the importance of software documentation, time constraints may lead to omitted comments [2]. Consequently, there were contradictory comments. A remark may contain information that is irrelevant or conflicting with the source code [10]. As a result, the costs associated with reviewing, updating, and testing source code will rise [11, 12]. All of these difficulties contribute to comment gaps and inconsistencies in source code.

Recent research for better software maintenance has yielded techniques for measuring the relationship between code and comment [13] and highlighted the need to maintain consistency between the two. Several methods for

[1]https://survey.stackoverflow.co/2022/#section-most-loved-dreaded-and-wanted-programming-scripting-and-markup-languages

identifying ancient document comments have been published [14, 11, 13, 15]. Because documented comments are well-structured for analysis, they were able to identify out-of-date document comments with high precision and recall [15]. For methods that only looked at block/line comments, the detection was done at the function/method level [11, 13]. For methods that looked at both, the detection was done at the topic level [14, 11].

A study created a technique for spotting obsolete comments during the evolution of code using a machine learning-based approach [16]. To find possible comment changes, use the source code change and the relationship between the code and the comment before and after the source code change. They were able to detect changes in 64 different properties of the software, as well as the state of comments and how they related to the code before and after the changes, using machine learning techniques. Typically, the developer who writes the code decides whether or not to include comments. How developers comment on code determines both the quality of the comments and the readability of the code. To better understand this term, the practice of adding comments to code in various programming languages is investigated [17]. The practice of commenting on code has evolved as a result of the development of natural programming languages and the evolution of ecosystems.

Researchers are seeking a long-term solution for the developers to comprehend this inconsistency. A method is based on information retrieval for locating traceability links between source code and free-text documents. In two case studies, probabilistic and vector space information retrieval strategies were used to track C++ source code on manual pages and Java code on functional requirements [18]. In another paper, a method is proposed for detecting code comment inconsistencies in locking and calling mechanisms. They were only interested in suggestions regarding programmers' assumptions and needs [11]. Another paper testing Javadoc's comments was published [12]. The authors analyzed method properties with null values and corresponding exceptions. They utilized Natural Language Processing to identify inconsistencies involving @param tags in Javadoc comments and the null parameter exception statement in codes and comments. Their efforts consist solely of commenting on null references and throwing exceptions.

In another study, the results of a manual analysis of how well the comments and implementations of 3,636 methods from three Java open-source software systems match up are shown [7]. The evaluations that resulted were compiled into a dataset that was made publicly available thereafter. This paper tries to find links between coherence and lexical information provided in source code and method lead comments, which is another important contribution.

All the above literature gives us a perfect picture that the problem of inconsistency between code and its comments exists and there is a lot to work on. One of the major drawbacks of all work is that they are done in Java programming language. So, the first problem that should be addressed is the scarcity of datasets in any different programming language other than Java. In our work, we aim to address that adequacy and work towards minimizing it.

3. Background Studies

Comments are the foremost source of code documentation. Due to this, their requirement is consistent with the written code. Computing code comment consistency is essential since a written method can be complex enough for another developer to understand. Complexity may occur for different reasons, for example, 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 labeled dataset for another programming language and by proposing a language-independent model. With that being the goal, a few terminologies and facts need to be addressed first. Those facts & terminologies are discussed in this section.

3.1. 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. Whereas, comments on top of the code generally describe the whole class or method. So, following the topological order of the comment is essential.

3.2. Lead Comment

Lead comment refers to the corresponding comment which is written just above the code.

3.3. Coherent and Incoherent

When a code snippet's meaning and respective comment are aligned correctly, the code-comment pair is recognized as a Coherent pair. On the contrary, when the meaning of a code snippet and respective comment is not appropriately aligned is labeled as Incoherent.

3.4. Bilingual Evaluation Understudy (BLEU)

A value for comparing a candidate sentence of text to one or more reference sentences of text. Complete match outcomes in a score of 1, and complete inconsistency results in a score of 0.

This metric is widely used to evaluate machine translation results [19, 20].BLEU scores have been employed as one of the evaluation metrics in recent source code summarizing research [21, 22]. A function to compare a candidate

```
//Removes all resources in the collection, and disposes every element.
public void Clear()
{
    foreach (var elem in _resources) elem.Dispose();
    _resources.Clear();
    _CUResources.Clear();
}
```

Figure 1: Implementation of the method 'Clear' and its corresponding lead comment

sentence to one or more reference sentences is offered by NLTK.

A BLEU score compares a tokenized predicted sentence to a tokenized reference sentence. The result is determined by comparing the predicted sentence's average precision to the reference sentence. In our scenario the reference text is code, and the candidate text is a comment. For a better understanding, an example from our dataset is discussed below.

In the code-comment pair in Figure 1, the reference text was the void method code. For clear implementation and candidate text, the comment "Removes all resources in the collection and disposes of every element" was used. The BLEU score of 0.516593 indicates that the candidate text (comment) matches the reference text (method) by 51.65%. A BLEU Score of 51-100% was considered as a threshold value to determine if a pair is coherent, and the pair in Figure 1 was found to be coherent.

4. Methodology

The complete workflow of the methodology is represented in Figure 2. At first, we extract the relevant code artifacts to generate a dataset having all the properties that satisfy our requirements (e.g., lead comment and class comment) in the 'Data Generation'[4.1] step. Then, we evaluate the dataset in the 'Validation and Evaluation of the Dataset'[4.2] step. Finally, we use this dataset in our proposed machine-learning model to predict the coherence of the code-comment pair in the 'Model training and prediction' [4.3] step.

4.1. Dataset Generation

The dataset generation process consists of three parts: Dataset crawling and extraction, Dataset Cleaning, and Data Annotation. Each part is described in the next three subsections.

4.1.1. Dataset Crawling and Extraction

To collect a set of naturally written comments by developers, we used the extracted code-comment pair by Gelman et al.[23]. This data pool of code-comment pairs holds extracted pairs from GitHub projects for five different programming languages. C# is only considered in the scope of this work. To pull the code-comment pair, the author of [23] chose the GitHub repository based on having a redistributable license and at least 10 stars. Table 1 represents the list of the C# projects from the data pool, selected for this research. To verify the projects, first, the data pool was explored, and later from that data pool, we found the list of C# projects from the given filename. 10 projects were chosen for annotating the code-comment pair from the data pool, considering the highest number of stars. This study was restricted to making allowances for comments in an exact topological order, which is comments being on top of class and method. The renowned term for this type of comment is 'lead comment'. The selected C# projects were manually examined from the GitHub repository to ensure the comments' topological order. All the code-comment pair for these projects was then pulled from the data pool.

Table 1: Selected projects & description

ID	Project Name	Project Description	No. of Stars
1	Managed-CUDA	NVidia's CUDA integration in .net applications programmed using the programming language 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	Free gaming server remote control (RCON) software	20
6	EmojiVS	EmojiVS is a Visual Studio 2013 and 2015 extension to display GitHub emojis in the editor	84
7	xunit-performance	Provides xUnit plugins for creating performance tests	186
8	SimpleAuth	Simple Auth embeds authentication into the API so one doesn'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

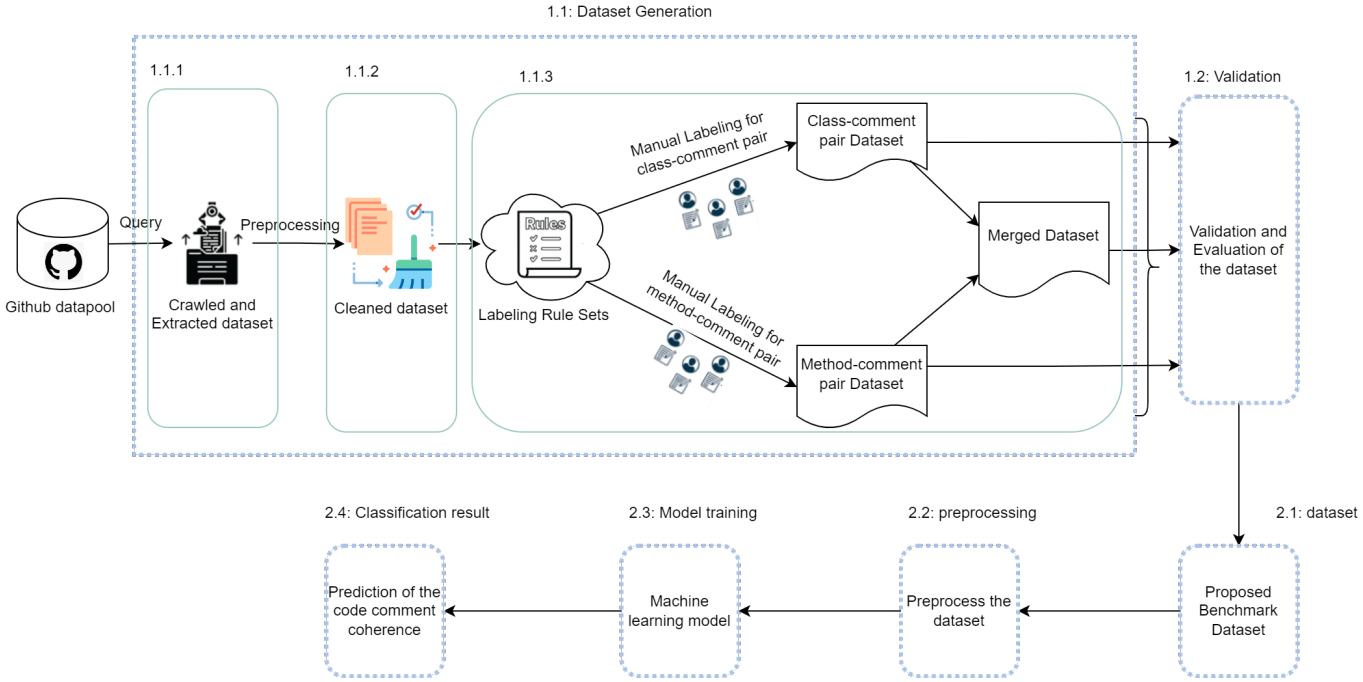

Figure 2: Complete workflow of the proposed methodology.

```
1
2   //A class that does the addition of two integer values
3   public class add
4   {
5
6       // decalaration of three integer variable
7       public int a,b,add;
8
9       //declaration a method which will print the addition of declared int
10      public void added()
11      {
12          add = a+b;
13          Console.WriteLine(add);
14
15      }
16  }
```

Figure 3: The implementation of the class 'add' and its corresponding lead comment

```
1   //Removes all resources in the collection, and
2   //disposes every element
3
4   public class Clear()
5   {
6       foreach (var elem in _resources) elem.Dispose();
7       _resources.Clear();
8       _CUResources.Clear();
9   }
```

Figure 4: Lead comment and the implementation of the method 'Clear'

As aforementioned, two types of code-comment pairs were considered for the dataset. The code can be either a class or a method, and the comment is placed at the very top of that class or method.

- **Class Level Code-Comment Pair Dataset:** Object Oriented Programming (OOP) classes are modeled after real-world entities. A user can designate a class as a blueprint or prototype from which further objects can be generated. Class declarations simply require the term class and the class identifier (name). Comments written immediately on top of class explaining the intent of the entire class are considered a lead comment. The lead comment of a class and the class itself are considered as a pair.

  The Comments we considered here is the comment which is just above the class. In Figure 3, we can see that there is a class declaration named 'codeComment' in line number 3, and just above that in line number 2 the statement is the comment which is describing the class 'codeComment'.

- **Method Level Code-Comment Pair dataset:** A method is a block of code that is only executed when called. A method can receive parameters or data. Also known as functions, specific action-carrying methods are also known as functions. Using the method's name followed by parentheses to define a method.

  The Comment we considered here is the comment which is just on the top of the method. In Figure 4, we can see that there is a method named 'Clear', and just above that, the statement is the comment which is describing the method. From Figure 4, an example of a method-comment pair can also be seen

written within the 'add' class. Line 9 has the lead comment of the method added and lines 10-15 have the code of the method.

4.1.2. Dataset Cleaning

Data cleaning must be done in the beginning to eliminate data noise and guarantee the correctness of the quality evaluation model. In this study, the code and comments contain a variety of identifiers and signs like punctuation marks, semicolons, brackets, and meaningless words with higher frequency. Hence, the data-cleaning procedure of this paper contains purifying special text characters, segmentation of words, etc. We removed stopwords for the English language, and words like 'public', 'class', and 'static' since these words add no extra value other than their multiple appearances in the dataset due to being a keyword of C# programming language. Symbols like '/////', '/n', '/t' were also removed from the dataset.

Table 2: Amount of distributed pairs

State of the Process	Class-Comment Pair	Method-Comment Pair	Total
Before Data Cleaning	1649	5432	9310 (Including Null)
After Removing Null	1171	5910	7081
After Data Cleaning	**1093**	**3829**	**4922**

Table 2 shows the number of pairs distributed as class-comment pair and method-comment pair. As seen, Before and After data cleaning the number of pairs was drastically reduced. Naturally, the method-comment pair is greater than the class-comment pair as multiple methods and its lead comment can be written within one class. At first from the selected projects, 9310 pairs of code and comments were taken. Since it was extracted using Doxyzen, which often fails to extract code or comment, as a result, the space stays as null space. So we removed the Null pairs from the dataset. Table 2 also shows how much data is remaining after removing the null pairs. Among 9310 pairs 23.51% pairs had missing values, hence, tagged as Null.

The primary content of this study is the method and class-level code comment pairs. So, After discarding the Null pairs, we discard the code comment pairs with any inline comments with their corresponding commented code or constructors and their short description of a class. This marks the last step of data cleaning, after which the total number of code-comment pairs becomes 4922 as seen in Table 2.

4.1.3. Data Annotation

Annotation is a procedure for appending details to a document at some level, a word or phrase, a section, or the entire manuscript [24]. In this research work, the initial investigation is whether the code snippet is based on classes or methods. If the code snippet is class-based, the rules for the class will be applied. If the code snippet is method-based, the rules of the method will be applied. If the code comment pair passes the rules, it will be annotated as Coherent; otherwise, it will be annotated as Incoherent.

To annotate the dataset, a human baseline approach was employed. Using the code-comment pairs of the selected projects, three annotators annotated those pairs with the defined rules for methods and classes differently. Those three annotators are Software Engineering graduates, who work in the software industry and contributed to different open-source projects. Detecting the relation between the code and the comment is a subjective task. With a substantial amount of experience and educational background, the annotators were entrusted with the annotation work. Two experts from the software industry were also present to check and ensure the quality of the set of rules and annotations.

Two sets of rules were utilized to annotate the data: Rules for method-comment and Rules for class-comment for annotating method-comment pairs already exist [7]. Table 3 contains the rules for annotating methods that were previously established and proposed by Corazza et al. aforementioned in the section 2. This set of rules is a good approach to letting others know about the intent & design implementation of the code, insights behind implementation decisions, etc. However, there are restrictions in this rule set. The rules can describe the behavior of a method and are generated exclusively for the Java programming language. This previously well-established set of rules has another setback, it is only given for method level and ignores the existence of class-level code, despite Java being an object-oriented programming language.

To annotate our data, we need two distinct rule sets, as we are focusing on both the method and class levels. The rules used to annotate classes are listed in Table 4. This set of rules was reviewed by two experts. The annotation procedure was identical to that for pairings of method code and comments. The rules specify the presence of the reason for the class declaration, implementation details of the class, information regarding the inherited class, polymorphism, and information regarding the declared constructor in the class.

Using the annotation process while keeping the rules sets for both method level and class, we have annotated the data. If the code-comment pair passes any 3 rules or

Table 3: Established rules for annotating method code-comment pair[7]

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 on the method describes all the expected behaviours 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.

```
// Perform color twist pixel processing.
// Color twist consists of applying the following formula to each image pixel using coefficients from the user supplied color twist
// host matrix array as follows where dst[x] and src[x] represent destination pixel and source pixel channel or
// plane x. dst[0] = aTwist[0][0] * src[0] + aTwist[0][1] * src[1] + aTwist[0][2] * src[2] + aTwist[0][3] dst[1] =
// aTwist[1][0] * src[0] + aTwist[1][1] * src[1] + aTwist[1][2] * src[2] + aTwist[1][3] dst[2] =
// aTwist[2][0] * src[0] + aTwist[2][1] * src[1] + aTwist[2][2] * src[2] + aTwist[2][3]


public static class ColorTwist
{
[DllImport(NPPICC_API_DLL_NAME, CallingConvention = CallingConvention.Cdecl)]
public static extern NppStatus nppiColorTwist32f_8u_AC4IR(CUdeviceptr pSrcDst, int nSrcDstStep,
NppiSize oSizeROI, [In, MarshalAs(UnmanagedType.LPArray, SizeConst = 12)] float[,] aTwist);
/// <summary>
/// 3 channel 8-bit unsigned planar in place color twist.
/// An input color twist matrix with floating-point coefficient values is applied within ROI.
/// </summary>
/// <param name="aTwist">The color twist matrix with floating-point coefficient values.</param>
/// <returns>
/// <see cref="NppStatus.StepError"/>, <see cref="NppStatus.NotEvenStepError"/>, <see cref="NppStatus.NullPointerError"/>
/// </returns>
[DllImport(NPPICC_API_DLL_NAME, CallingConvention = CallingConvention.Cdecl)]
public static extern NppStatus nppiColorTwist32f_8u_IP3R([In, MarshalAs(UnmanagedType.LPArray, SizeConst = 3)]
CUdeviceptr[] pSrcDst, int nSrcDstStep, NppiSize oSizeROI, [In, MarshalAs(UnmanagedType.LPArray, SizeConst = 12)] float[,] aTwist);
}
```

Figure 5: Implementation of the class 'ColorTwist' and its corresponding lead comment

for both code-comment and method-comment pairs if it passes the first 2 rules that pair is considered a coherent pair, otherwise it is labeled as incoherent.

Figure 5 displays code extracted from our dataset. This code snippet describes a class and its topological comments, but we only considered the lead comment. To make it an image, the code snippet was truncated from a long class. Three annotators have named this code-comment pair coherent. The lead comment here describes the class declaration's purpose and its basic implementation details, which relate to rules 1, 3, and partially to rule 2.

From Figure 6, a snippet of our annotated coherent code-comment pair. Code is the 'Update' method and lead comment is the comment written on top of the code. This pair was annotated as Coherent for satisfying rule no. 1, 2 but rule no. 4 & 5 is invalid for this method as this is a void method.

Figure 7 shows an example of an incoherent lead comment and void method. The reason behind labeling this

Table 4: Proposed rules for annotating class code-comment pair

ID	Rules for Classes
1	Lead comment for the class expresses 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 delivers the basic implementation information—for example, names and types of input parameters. The information is in orientation 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.

```
1  //Called from Update, checks for various lifecycle events that need to be
2  //forwarded to QCAR, e.g. orientation changes
3  public class Update()
4  {
5      if(SurfaceUtilities.HasSurfaceBeenRecreated())
6      {
7          InitializeSurface();
8      }
9      else
10     {
11         //if Unity reports that the orientation has changed, reset variable
12         //this will trigger a check for a few frames
13         if(Screen.orientation != mScreenOrientation){
14             ResetUnityScreenOrientation();
15             CheckOrientation();
16         }
17     }
18     mFramesSinceLastOrientationReset++;
19 }
```

Figure 6: Implementation of the method 'Update' and its corresponding lead comment

pair as incoherent is this pair does not abide by any of the rules.

Figure 8 is an example of an incoherent lead comment and class. The reason behind labeling this pair as incoherent is this pair also does not satisfy any of the proposed rules to be a coherent code(class)-comment pair.

3 annotators, all of them having at least 1 year of experience in the software industry, separately annotated the code-comment pairs checking both sets of rules. After annotating the dataset, Table 5 reveals that there are 1093 code-comment pairs for class and 3829 code-comment pairs for a method out of a total of 4922 code-comment pairs, of that 76.08% are coherent and 23.92% are Incoherent.

4.2. Validation and Evaluation of the Dataset

Since the three annotators annotated the dataset separately, it was essential to check how much the annotation for each data point of all three different annotators matched. To ensure that, we used Simpledorff - Krippendorff's Alpha [25]. Krippendorff's Alpha is a well-known

```
// Inplace exponential.
public void Exp()
{
status = NPPNativeMethods.NPPi.Exp.nppiExp_32f_C1IR(_devPtrRoi, _pitch, _sizeRoi);
Debug.WriteLine(String.Format("{0:G}, {1}: {2}", DateTime.Now, "nppiExp_32f_C1IR", status));
NPPException.CheckNppStatus(status, this);
}
```

Figure 7: Implementation of the method 'Exp' and its corresponding Lead Comment

```
1
2   //An index on a field
3
4   public class Index : Method{
5
6       ///The name of the index
7
8       public string Name{get;set;}
9   }
```

Figure 8: Incoherent lead comment and the implementation of the class 'Index'

Table 5: Description of the coherent and incoherent code-comment Pairs

Dataset	Coherent	Incoherent	Total
Class-comment pair dataset	1021 (93.41%)	72 (6.59%)	1093
Method-comment pair dataset	2719 (71.00%)	1110 (29.00%)	3829
Merged dataset	3745 (76.08%)	1177 (23.92%)	4922

inter-annotator reliability metric. It is devised to calculate the like-mindedness among observers, coders, reviewers, or raters, drawing distinctions among generally unformed spectacles or allocating computable weights to them. Values of Krippendorff's Alpha(α) range from 0 to 1, where 0 is perfect disagreement, and 1 is excellent agreement. Krippendorff suggests that it is conventional to require $\alpha \geq .800$. Where indecisive findings are still acceptable, $\alpha \geq .667$ is the lowest conceivable limit[26].

In addition to Krippendorff's Alpha, the bilingual evaluation understudy (BLEU) score is used to evaluate each code-comment pair and compare our annotated dataset. Many recent studies have used the BLEU score as their main evaluation metric. BLEU score estimation includes a brief sentence for predictions. Thus, the model is rewarded for predicting only n-grams in the reference sentence and for making long predictions. BLEU-1 was employed, implying the BLEU score estimation includes 1 gram and every smaller n-gram. Agreeing with Hu et al.[21], this in-work smoothing was also unused to settle the insufficiency of higher-order n-gram overlapping.

4.3. Model Training and Prediction

To measure the coherence (integrity) of the code-comment pair, we incorporated a variant of an ensemble machine-learning model[8] with topic modeling. The topic modeling approach is employed because it analyzes text data to select cluster words for a set of documents. Our modifications include adding natural language processing-based preprocessing techniques like lemmatization, removal of stop words, etc. Another modification our is employing logistics regression to infuse the extracted features from the multi-model random forest. As a cross-validator, a stratified k-fold method was applied.

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. Tokenized pairs were turned into a bag of word tokens. Later on, lemmatization was applied.
- **Feature 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. Afterwards, 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 fit into a random forest(RF) classifier. The RF model delivers a consistency score or probability score of being consistent for every pair of code comments using the 'proba' function. Since the topic numbers for both corpora of comment code might vary, it was necessary to obtain multiple numbers of topics to find the informative and compelling features. Due to necessity, 20 different feature sets were extracted, as seen in Fig 7, by changing the number of topics in the LDA every time. These extracted features were fitted into 20 different random forest classifiers.
- **Infusing with logistic regression:** Each random forest model produces a consistency score for a code-comment pair. These consistency scores originated from 20 different random forests required to be fused. Here, Logistic Regression combines the consistency scores supplied by 20 other random forests and predicts the coherence of the code-comment pairs.

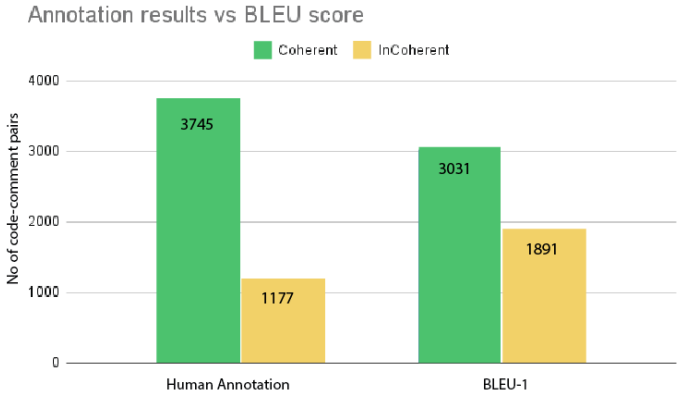

Figure 9: Comparison of the result based on the Human annotator and BLEU score

5. Results and Discussion

In this section, first, we evaluate the dataset based on the aforementioned two well-known metrics and then, compare the performance of our machine learning method with other for classification.

5.1. Evaluation of the dataset

We evaluate the dataset based on Krippendorff's Alpha. In the dataset, we find 95.76% similarity among the rating of 3 annotators for all the code-comment pairs. The percentage of similarity shows how much the rating matched among the raters. So, the result of this metric helped us to validate the human annotation. Upon investigation, it was found that the dissimilarity occurred for one rater with different labeling for a few small-size method-comment pairs than the other two raters.

Along with Krippendorff's Alpha metrics, we also compare the result with the BLEU score. From Figure 9, it is clearly visible that the BLEU labeled 714 pairs as Incoherent whereas, in our human-curated dataset, the annotators annotated those as consistent ones. The idea of the BLEU Score is, that it does not assume the definition of words, while it is permissible for a human to employ a different word with a similar meaning. The BLUE score also ignores paraphrasing.

5.2. Comparison of the performance in classification

After using the modified model on our prepared dataset, the performance score for multiple evaluation metrics (accuracy, precision, recall, and AUC-ROC) for three datasets can be observed from Table 6 and showed the strength of our machine learning model. One of the primary reasons behind the competitive result is that the code section has a high amount of docstrings (well documented) written within it. So when the model matches code and comments based on appearance and occurrence, they can find similar words present in the comment in the docstring.

Table 6: Performance of multiple evaluation metrics for 3 datasets

Dataset	**Accuracy**	**Precision**	**Recall**	**AUC-ROC**
Class-Comment	0.973	0.990	0.980	0.919
Method-Comment	0.974	0.975	0.989	0.963
Merged	0.973	0.981	0.983	0.962

In addition, we also compare the performance of our model with the same variant[8] on the merged dataset. Fitting the model on our annotated 4922 code-comment pairs of the merged dataset, we got the value 96.2% which is better than the previous model which gave 93.1% using the performance metric AUC-ROC. The value difference is shown in Table 7. Although these two models are similar, our preprocessing step makes the difference for better prediction results.

Table 7: Comparison between ours and a state-of-the-art model on our proposed dataset (merged)

Model	**AUC-ROC**	**Accuracy**	**Precision**	**Recall**
Rabbi et al.[8]	0.931	0.96	0.95	0.96
Our Proposed	**0.962**	**0.973**	**0.981**	**0.983**

6. Implication to the practitioners

The proposed C# dataset broadens the enthusiastic researchers of this field to work with a state-of-the-art dataset. In addition, we are the first one to provide a dataset having two types of code-comment pairs, class comment code pair and lead comment-code pair. From an industry perspective, these two types of comments are generated, and this dataset helps them to train a model that automatically detects the coherence of the code comment. We proposed a variant of a machine learning model that detects the coherence. Also, with the help of training this dataset, it is possible to build a model that automatically generates the comment or vice versa.

7. 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. The dataset proposal of our work can be

seen as an elongation work of Anna Corazza et al. work, they proposed a checklist-based public benchmark dataset for the Java programming language. As far as the model is concerned, it is an enhanced version of a previous study.

8. Conclusion & Future Work

The AI community is now moving towards data-centric AI. Due to this, building a suitable dataset is becoming far more acceptable than building complex models. We have presented a dataset of an analysis on the coherence between comment of methods and lead comments of class and their corresponding implementations. Human annotation and BLUE-1 label were different for 14.5% pairs among the total dataset. 95.76% similarity is found among the annotation of 3 annotators using the inter-annotator reliability metric, Krippendorff's Alpha. A description of the process, problems, and mitigating procedures while creating the dataset is also delivered. A step forward in this forthcoming research direction could be making it language-agnostic by extending the approach to other languages. Models can be fed datasets of different programming languages as well. Furthermore, the topological order of the comment should be analyzed more closely. Researchers have concluded in a review paper that studies may work more to generate comments in the correct place. [27] 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 was machine-generated. Exploring different metrics to evaluate this kind of dataset should be a priority. Using logistic regression in the ensemble model had the upper hand. We plan to experiment more with model training in the future.

References

[1] T. Tenny, Program readability: Procedures versus comments, IEEE Transactions on Software Engineering 14 (9) (1988) 1271–1279.

[2] S. N. Woodfield, H. E. Dunsmore, V. Y. Shen, The effect of modularization and comments on program comprehension, in: Proceedings of the 5th international conference on Software engineering, 1981, pp. 215–223.

[3] A. J. Ko, B. A. Myers, M. J. Coblenz, H. H. Aung, An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks, IEEE Transactions on software engineering 32 (12) (2006) 971–987.

[4] D. Piorkowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, M. Burnett, Foraging and navigations, fundamentally: developers' predictions of value and cost, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016, pp. 97–108.

[5] B. Joy, G. Steele, J. Gosling, G. Bracha, The java language specification (2000).

[6] P. Rani, A. Blasi, N. Stulova, S. Panichella, A. Gorla, O. Nierstrasz, A decade of code comment quality assessment: A systematic literature review, Journal of Systems and Software 195 (2023) 111515.

[7] A. Corazza, V. Maggio, G. Scanniello, Coherence of comments and method implementations: a dataset and an empirical investigation, Software Quality Journal 26 (2018) 751–777.

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

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

[10] F. Salviulo, 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, 2014, pp. 1–10.

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

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

[13] W. M. Ibrahim, N. Bettenburg, B. Adams, A. E. Hassan, On the relationship between comment update practices and software bugs, Journal of Systems and Software 85 (10) (2012) 2293–2304.

[14] S. C. B. de Souza, N. Anquetil, 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, 2005, pp. 68–75.

[15] N. Khamis, R. Witte, J. Rilling, Automatic quality assessment of source code comments: the javadocminer, in: Natural Language Processing and Information Systems: 15th International Conference on Applications of Natural Language to Information Systems, NLDB 2010, Cardiff, UK, June 23-25, 2010. Proceedings 15, Springer, 2010, pp. 68–79.

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

[17] P. Rani, S. Panichella, M. Leuenberger, M. Ghafari, O. Nierstrasz, What do class comments tell us? an investigation of comment evolution and practices in pharo smalltalk, Empirical software engineering 26 (6) (2021) 112.

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

[19] K. Papineni, S. Roukos, T. Ward, 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, 2002, pp. 311–318.

[20] C. Callison-Burch, M. Osborne, 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, 2006, pp. 249–256.

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

[22] S. Iyer, I. Konstas, A. Cheung, L. Zettlemoyer, Summarizing source code using a neural attention model, in: 54th Annual Meeting of the Association for Computational Linguistics 2016, Association for Computational Linguistics, 2016, pp. 2073–2083.

[23] B. Gelman, B. Obayomi, J. Moore, D. Slater, Source code analysis dataset, Data in brief 27 (2019) 104712.

[24] M. Petrillo, J. Baycroft, Introduction to manual annotation, Fairview research (2010) 1–7.

[1] https://github.com/kima063/code-comment-consistency

[25] K. Krippendorff, Computing krippendorff's alpha-reliability (2011).
[26] J. M. Ford, Content analysis: An introduction to its methodology, Personnel Psychology 57 (4) (2004) 1110.
[27] X. Hu, X. Xia, D. Lo, Z. Wan, Q. Chen, T. Zimmermann, Practitioners' expectations on automated code comment generation, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1693–1705.