UNIVERSITY OF L'AQUILA

MASTER THESIS

Effects of implicit and sequential meaning of words in code-comment coherence prediction

Author:
Michael Dubem IGBOMEZIE

Supervisor:
Dr. Phuong T. NGUYEN

Corso di Laurea Magistrale in Informatica

Department of Information Engineering Computer Science and Mathematics

Academic Year 2022/2023

Acknowledgements

I am entirely grateful to Prof. Phuong T. Nguyen for his guidance and his patience through this research. His dedication to seeing me through this phase of my program was a big motivation. I have learnt so much working under his supervision, and I have had to push the boundaries of what I thought I already knew, which is very important for me.

I am also grateful to my colleagues who at one point or the other aided me in achieving my academic targets. They all made these two years worth the while.

Finally, I would like to thank my family and close friends for the support through this period of study (financially and emotionally amongst others), as I would not have been able to complete it without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Code comments play a crucial role in software development, as they provide programmers with practical information, allowing them to better understand the intent and semantics of the underlying code. Essentially, comments are valuable pieces of information when it comes to code understanding and maintenance [22, 28], revealing various aspects of the underlying method [16, 23]. Among other purposes, developers use comments to communicate their intent and explanations about the implementations. This helps developers spend less time understanding the existing implemented solutions.

Nevertheless, developers tend to leave comments unchanged after updating the code, resulting in a discrepancy between the two artifacts. Such a discrepancy may trigger misunderstanding and confusion among developers, impeding various activities, including code comprehension and maintenance. As pointed out by Corazza *et al.* [5] *"information provided in the comment of a method and in its corresponding implementation may not be coherent with each other (i.e., the comment does not properly describe the implementation)."* Software evolution, among others, is a factor that could break the coherence between a comment and the block of code it originally intended to describe.

As a motivating example, we depict two code snippets[1] with comments below. The first snippet (see Fig. 1.1) throws an XMLParseException if the opening and closing tags do not match. By checking the comment, we realize that the intent of the code is well explained, corresponding to a *coherent* relationship between code and comment. Meanwhile, the code in Fig. 1.2 is used to check if the first character of the input string matches with the one specified by the *entityChar* variable, and if this is the case, then an error will be thrown. Looking at the comment, we see that it does not properly explain the behaviour of the code. Thus, we identify an *incoherent* case, i.e., the comment implies a completely different story compared to the one contained in the code. Such inconsistency may cause misunderstanding, leading to various issues, including difficulties in software maintenance.

It is crucial to identify if, given a code snippet, its corresponding comment is coherent and reflects well the intent behind the code. Locating instances where there is a lack of coherence between code methods and their accompanying comments could significantly improve software practice, and reduce the time spent in source code integration. Corazza *et al.* [6] were among the first to address the issue, and they curated a dataset where code and comment were manually selected and evaluated. Since then, there have been various studies proposed to the detection of coherence between comments and code [24, 26]. Nevertheless, existing approaches to this problem, while obtaining an encouraging performance, either rely on heavily pretrained models, or treat input data as text, neglecting the intrinsic features contained in comments and code, including word order and synonyms.

[1] Both snippets are extracted from the dataset curated by Corazza *et al.* [6].

```
/**
* Throws an XMLParseException to indicate that the closing tag of an
* element does not match the opening tag.
*
* @param systemID     the system ID of the data source
* @param lineNr     the line number in the data source
* @param expectedName the name of the opening tag
* @param wrongName  the name of the closing tag
*/
static void errorWrongClosingTag(String systemID,
int  lineNr,
String   expectedName,
String   wrongName)
throws XMLParseException
{
 throw new XMLParseException(systemID, lineNr,
 "Closing tag does not match opening tag: ‘"
 + wrongName + "’ = ‘" + expectedName+ "’");
```

FIGURE 1.1: Examples of coherent code comment pair.

```
/**
* Reads a character from the reader disallowing entities.
*
* @param reader     the reader
* @param entityChar   the escape character (&amp; or %) used to indicate
*             an entity
*/
static char readChar(IXMLReader reader,
char     entityChar)
throws IOException,
XMLParseException{
 String str = XMLUtil.read(reader, entityChar);
 char ch = str.charAt(0);
 if (ch == entityChar) {
  XMLUtil.errorUnexpectedEntity(reader.getSystemID(),
  reader.getLineNr(), str);
 }
 return ch;
```

FIGURE 1.2: Examples of incoherent code comment pair.

This work presents CO3D as a practical approach to the detection of code comment coherence. We pay attention to internal meaning of words and sequential order of words in text while predicting coherence in code-comment pairs. We deployed a combination of Gensim word2vec encoding and a simple recurrent neural network, a combination of Gensim word2vec encoding and an LSTM model, and CodeBERT. The experimental results show that CO3D obtains a promising prediction performance, thus outperforming well-established baselines.

The main contributions of our work are as follows:

- We propose CO3D–a practical approach to the detection of code comment coherence using word embedding, LSTM, and CodeBERT.
- We conduct an evaluation using a real-world dataset, and compare CO3D with well-established baselines.
- The tool developed through this paper together with metadata is published to allow for future research.

Structure. The thesis is organized in the following chapters. In Chapter 2, there is the mathematical background concerning word embeddings, recurrent neural networks, and CodeBERT. Chapter 3 reviews the related work about code-comment

coherence detection, as well as related works that have implemented a combination of the methods (word embeddings, recurrent neural network) in other research areas. Chapter 4 presents in detail CO3D–the proposed approach. Afterward, we report the experiment conducted to evaluate CO3D on real-world datasets in Chapter 5. This chapter also analyzes the experimental results, and compares CO3D with existing baselines. Finally, Chapter 6 sketches future work, and concludes the thesis.

Chapter 2

Background

This chapter reviews the mathematical background concerning word embeddings, recurrent neural networks (RNNs), long short-term memory neural networks (LSTMs), and CodeBERT.

2.1 Word Embedding (Word2Vec)

This is an encoding method that seeks to represent words in a continuous vector space by combining the bag of words method with a neural network. To give an overview of the concept, the word embedding method used in this research simply generates the vector representation of a given word using conditional probability [19].

$$P(w_t|context) = P(w_t|w_{t-c}, \cdots, w_{t-1}, w_{t+1}, \cdots, w_{t+c}) \tag{2.1}$$

in which

- w_t represents the target word at the position t.
- $w_{t\pm c}$ represents the word at the position c steps away from t in either direction.
- c is what is referred to as the context window, and is a fixed size for all words to be predicted in the given vocabulary.

Given a set of words, a training set is generated using each word as a target variable and the words that appear within its context window as the independent variables. This context window can be arbitrarily chosen to be a certain number of words on each side of the target word, however, the wider this window, the more distinctive features of each given word is captured. This context window length is the same for all the words being trained on and determines the length of the vector embedding of each word.

In order to illustrate the concept, assume we have a the sample sentence;

> "Due to the downward trend in the economic condition of the country, the cost of living keeps worsening and inflation is at an all time high."

Regular expressions (regex) like punctuations and so on are removed from the text which is then tokenized before being processed. At this stage stop words can be removed or not depending on the desired preference. However if stop words are left in the text and the window size is small, many words might just simply contain mostly stop words in their context window which creates a false sense of similarity between semantically different words. On the other hand if stop words are removed, this is also some information from the text being lost in transformation. The aim is

to find a right balance, where the window size is big enough to retain stop words and still maintain a distinct context between semantically different words, but not choosing a window size too big that the model becomes cumbersome to train. Due to the limited vocabulary in this example, stop words aren't considered.

$$v = [\text{'Due'}, \text{'downward'}, \text{'trend'}, \text{'economic'}, \text{'condition'}, \text{'country'}, \text{'cost'}, \text{'living'}, \text{'keeps'}, \text{'worsening'}, \text{'inflation'}, \text{'all'}, \text{'time'}, \text{'high'}] \tag{2.2}$$

For the sake of this example, we can choose a context window length of 2 words, which refers to two words on either side of the target word being considered. We will also choose three random words from the text (economic, cost, worsening)

For the three given words, the list of context words for the three chosen words from the text are,

$$w_t^1(economic) = [\text{'downward'}, \text{'trend'}, \text{'condition'}, \text{'country'}] \tag{2.3}$$

$$w_t^2(cost) = [\text{'condition'}, \text{'country'}, \text{'living'}, \text{'keeps'}] \tag{2.4}$$

$$w_t^3(worsening) = [\text{'living'}, \text{'keeps'}, \text{'inflation'}, \text{'all'}] \tag{2.5}$$

With the selected target words and their context words, their respective inputs (composed of their context words) which will be fed into the neural network model is generated using the bag of words method, with the vocabulary size as the feature length.

$$i_t^1(economic) = [0,1,1,0,1,1,0,0,0,0,0,0,0,0] \tag{2.6}$$

$$i_t^2(cost) = [0,0,0,0,1,1,0,1,1,0,0,0,0,0] \tag{2.7}$$

$$i_t^3(worsening) = [0,0,0,0,0,0,0,1,1,0,1,1,0,0] \tag{2.8}$$

In this initial bag of words encoding approach, the words in the vocabulary are used as features and for a given target word, the value 0 is assigned to positions occupied by words in the vocabulary not in its list of context words and 1 otherwise.

This generated input is fed into the neural network that will be trained to predict the target word given a set of context words. The context window length is used to define the size of the first layer of the neural network having its activation function as a simple average. All the words in the vocabulary used for training are fully connected to this first layer and the respective weights of these connections are randomly initiated or gotten from pre-trained data depending on the trainer's needs. From the illustration in Figure 2.1, it can be observed that at first, there doesn't seem to be any closeness between the word 'economic' and the word 'cost' despite, both words having a significant overlap of context word.

$$a_f = (\sum_{k=1}^{N^{(i)}} w_{if} \cdot a_i)/N \tag{2.9}$$

in which

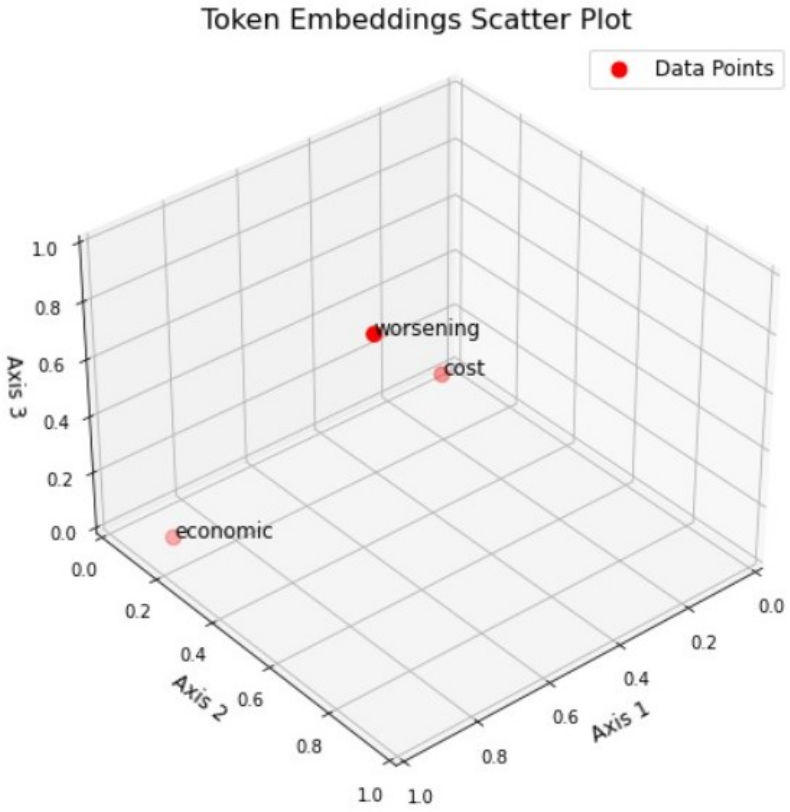

FIGURE 2.1: An illustration of plotting the vector embedding of each token when the weights are randomly initialized.

- a_f is the output of neuron f in the first layer.
- w_{if} is the weight connecting input node i to neuron f in the first layer.
- a_i is the value at the input node i in layer *l-1*.
- $N^{(i)}$ is the number of neurons in the input layer.

This first layer is then fully connected to a second layer which has a softmax activation function. This second and final layer is responsible for outputting the probability of each word in the vocabulary being the target word given a set of context words

$$a_s = softmax(\sum_{k=1}^{N^{(f)}} w_{fs} \cdot a_f + b_s) \tag{2.10}$$

in which

- a_s is the output (probability of a target word) of neuron s in the second layer.
- w_{fs} is the weight connecting neuron f in the first layer to neuron s in second layer.
- a_f is the value of node f in the first layer.
- $N^{(f)}$ is the number of neurons in the first layer, which is defined using the decided context window length.

This neural network is then trained using back propagation on the training data generated from our vocabulary, in order to adjust the weights of each connection to minimize error. As earlier explained, each input will be comprised of a bag-of-words representation of the context words of a given target word, with the value 1 for all the words in the vocabulary present in the list of context words in a given instance, and the value 0 for all the words in the vocabulary not in the list of context words for that given instance. This bag-of-words representation is then fed in as input to the neural network to predict the target word using the output with the highest probability. After the weights are properly trained for the target variable, the weights of the neural network can then be used to either predict a target word given a set of context words, or represent the vector embedding of the context words of a given target word.

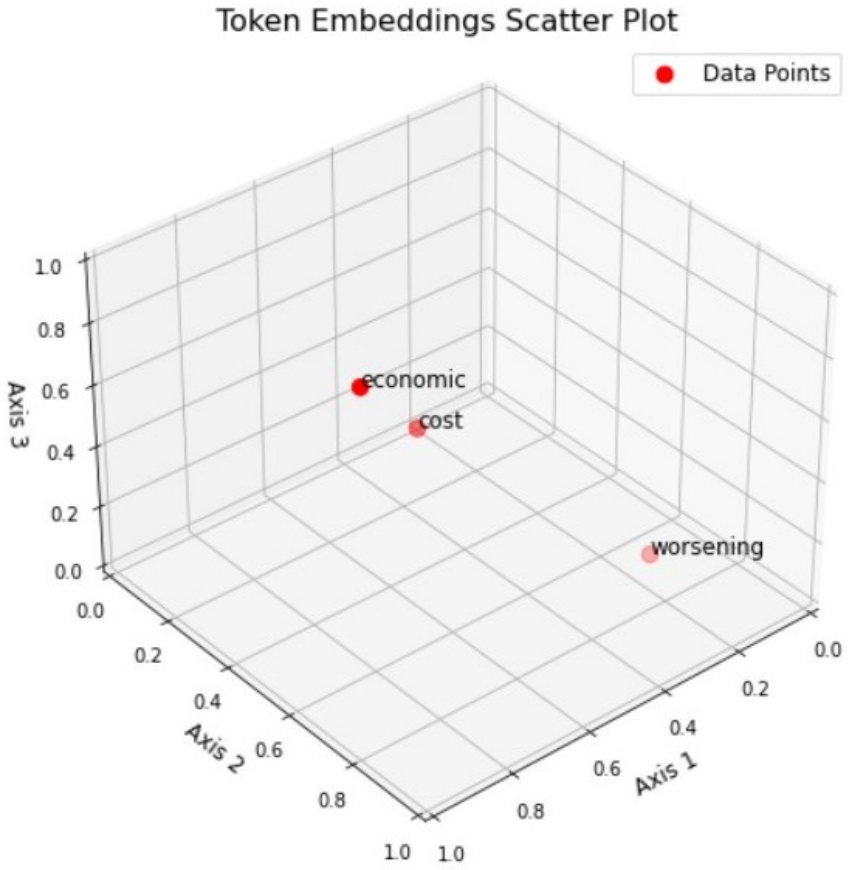

FIGURE 2.2: An illustration of plotting the vector embedding of each token after training the model.

Due to this whole concept and architecture hinging on the idea that similar words will appear in similar contexts thereby having similar context words, weights of the connections of the context words of similar words are eventually close in value due to much overlap between their context words (as can be seen in Figure 2.2) which is reflected during training. This also helps to calculate similarity amongst words.

2.2 Simple Recurrent Neural Network (RNN)

The recurrent neural network model is a variation of the deep neural network model that was specifically designed to account for the order in which input is fed into

a deep learning model. This architecture is more suited for sequential prediction in which Natural Language Understanding falls under. When reading sentences, humans do not process the meaning of each individual word in a given sentence as an isolated element, but rather as an element dependent on the words that come before or after it, which comes together to form the context encased in the sentence. In essence, past words play a role in understanding the meaning and context of current words, as in most sequential type predictions. This is one amongst many types of sequential actions humans are capable of carrying out [14].

The traditional multi-layer perceptron isn't built to handle this kind of input structure. Multi-layer perceptron takes in all the inputs of a given row in what could be considered a parallel manner, with each input not influencing the interpretation of another in a time dependent manner. The inputs from the input layer could eventually have a joint impact on a node in the hidden layer based on their weights after back propagation is carried out, however they do not interact with each other. This is where the recurrent neural network architecture comes in. Unlike the multi-layer perceptron, each hidden layer in the recurrent neural network is connected to both the input and the previous hidden layer. This makes it so that the previous input influences the weight of the current input during model optimization, which is what accounts for sequential input.

To better describe the recurrent neural network mathematical model, it is best to first describe the mathematical model for the multi-layer perceptron so as to draw a clear distinction. For the multi-layer perceptron, its mathematical description of a single layer is as follows [14];

$$a_j^{(l)} = \sigma\left(\sum_{k=1}^{N^{(l-1)}} w_{jk}^{(l)} \cdot a_k^{(l-1)} + b_j^{(l)}\right) \tag{2.11}$$

in which

- $a_j^{(l)}$ is the output of the *j-th* neuron in layer *l*.
- $w_{jk}^{(l)}$ is the weight connecting neuron. *k* in layer *l-1* to neuron *j* in layer *l*.
- $a_k^{(l-1)}$ is the output of neuron *k* in layer *l-1*.
- $b_j^{(l)}$ is the bias of neuron *j* in layer *l*.
- $N^{(l-1)}$ is the number of neurons in the layer *l-1*.

In this particular vanilla type of deep neural network architecture, there is only a linear flow of information where the next layer *l* only has inputs from the previous layer *l-1*. Hence, nodes do not have a cyclic interaction with each other to account for any order in input, but simply focus on the value at each position (index) of the given input data [14].

To explain this with a simple example,

In the figure above, assuming the x_1 and the x_2 are the activation values of the nodes from previous layer, the node in the next layer then takes in the activation values from the nodes in the previous layer, along with their weight (which are both -2 in this example) and its own bias (which is 3 in this example) to compute its own activation value, which is also passed in a linear manner to the nodes of the next layer after it.

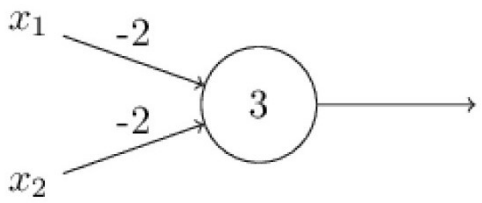

FIGURE 2.3: An illustration of a vanilla multi-layer perceptron [21].

However, this is the fundamental difference between the multi-layer perceptron and the simple recurrent neural networks. The mathematical representation of a simple recurrent neural network can be given below [14].

$$h_t = \sigma(W_{hx} \cdot x_t + W_{hh} \cdot h_{t-1}) \tag{2.12}$$

$$y_t = softmax(W_{yh} \cdot h_t) \tag{2.13}$$

in which

- x_t is the input at time step t.
- h_t is the hidden state at time step t.
- h_{t-1} is the previous hidden state, at time step *t-1*.
- y_t is the output at time step t.
- W_{hx} is the weight matrix for the input.
- W_{hh} is the weight matrix for the hidden state.
- W_{yh} is the weight matrix for the output.

Here, an obvious difference can be observed between the architecture of the simple recurrent neural network and that of the vanilla multi-layer perceptron. In the architecture of the simple recurrent neural network, the activation value of a given state is not only a function of the weights and values of the input, but also the weights and activation values of itself at the previous time step. The input data is also introduced into the system at time steps, which allows the system to observe how each input at each time step affects the interpretation of the input in the next time step.

In the figure above, x_1 can be seen as the input at a given time step, while x_2 can be seen as the node in a hidden state at a given time step, the w_{21} is the weight linking the input to the node in given time step and the symbol μ represents the product of the activation value of the node in the hidden layer for the previous time step and the weight between the node in the hidden layer for the previous time step and the current time step.

In this more stretched out version of a simple recurrent neural network as seen in the figure above, it becomes somewhat clearer to understand the recurrence relation the hidden state has with itself with respect to a given time step. At each time step, an input and its accompanying weights is fed into the current hidden state along with the activation value of the hidden state for the previous time step and its accompanying weights. This process produces the activation value for the current state which will then serve as the previous time step hidden state activation value to

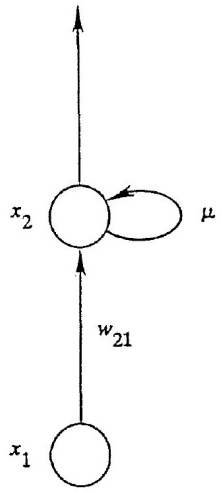

FIGURE 2.4: An illustration of a simple recurrent neural network [14].

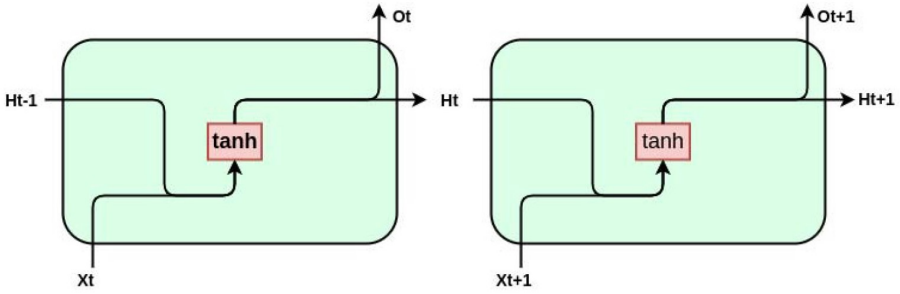

FIGURE 2.5: Another illustration of a simple RNN [24].

be fed into the hidden state for the next time step along with the input and weights for the next time step.

2.3 Long Short Term Memory (LSTM)

The long short term memory neural network is a variation of the simple recurrent neural network designed to fix a problem in the recurrent neural network architecture. It also takes input in time steps and each hidden state of the model is dependent on the input of its given time step and its own values for the previous time step. Due to the hidden state in a given time step in the simple recurrent neural network only paying direct attention to its most recent previous time step, this tends to give priority to the immediate preceding inputs, while subsequently diminishing the impact of the inputs from much earlier time steps regardless of whether these are more important to the context for interpreting the meaning of the entire input or not. In more mathematical terms, this is as a result of the vanishing gradient effect which grows with the increase in time steps [11]. This phenomenon has been generally referred to as the short term memory of the simple recurrent neural networks.

This particular architecture plays around with the sigmoid (σ)and tanh activation functions to simulate long and short term memory effects on the model. It does this using three main gates (the forget, input and output gate) and two states (cell state and hidden state). To better explain this, below are the mathematical representations of the gates and the states, followed by a pictorial representation and a simplified explanation.

$$f_t = \sigma(W_f \cdot [H_{t-1}, x_t] + b_f) \tag{2.14}$$

$$i_t = \sigma(W_i \cdot [H_{t-1}, x_t] + b_i) \tag{2.15}$$

$$C_t^n = \tanh(W_c \cdot [H_{t-1}, x_t] + b_c) \tag{2.16}$$

$$C_t = f_t \cdot C_{t-1} + i_t + C_t^n \tag{2.17}$$

$$o_t = \tanh(W_o \cdot [H_{t-1}, x_t] + b_o) \tag{2.18}$$

$$H_t = o_t \cdot \tanh(C_t) \tag{2.19}$$

in which

- W_f is the weight matrix for the forget gate at the current time step t.
- W_i is the weight matrix for the input gate at the current time step t.
- W_c is the weight matrix for the new cell state candidate.
- W_o is the weight matrix for the output gate at the current time t.
- H_{t-1} is the previous hidden state at time *t-1*.
- C_{t-1} is the previous cell state at time *t-1*.
- x_t is the input at the current time step t.
- b_f is the bias of the forget gate at the current time step t.
- b_i is the bias of the input gate at the current time step t .
- b_c is the bias of the new cell state candidate at the current time step t.
- b_o is the bias of the output gate at the current time step t.
- f_t is the forget gate (value) at the current time step t.
- i_t is the input gate (value) at the current time step t.
- C_t^n is the candidate for the new cell state at the current time step t.
- C_t is the updated cell state at the current time step t.
- o_t is the output gate (value) at the current time step t.
- H_t is the updated hidden state at the current time step t.

For a given time step in the LSTM architecture, there seems to be a lot to unpack, so here is a brief overview before going more in-depth in explanation. The forget gate is basically used to determine what to forget (and in turn what to remember) from the previous iteration, which is largely dependent on the relationship between the previous inputs and the output. The input gate is responsible for deciding what to retain from the input for the current time step. The cell state contains the representation of the relationship between the input and the output and is recomputed on

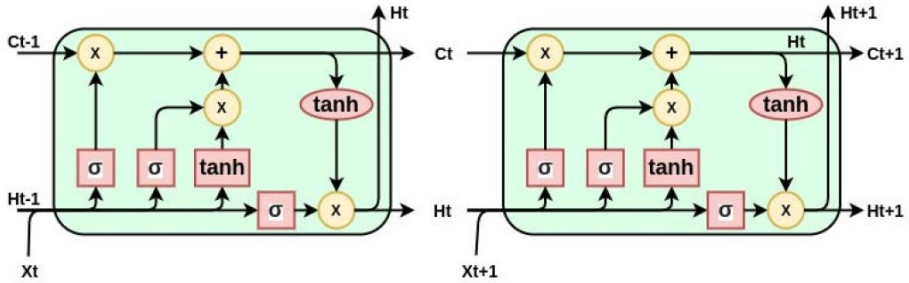

FIGURE 2.6: An illustration of a Long Short Term Memory (LSTM) neural network [24].

every time step. Hence, for each time step the previous cell state is combined with the current candidate for the new cell state and is used to give a final cell state for the given time step. This is very similar to the role of the hidden state, however, the hidden state can be considered as a filtered version of the cell state which eventually serves as the final output to be displayed by the model.

Going more in-depth into these explanations, the forget gate (Equation 2.14) takes in the previous hidden state and the input of the current time step, along with its own weights and bias and passes these through the sigmoid (σ) function to produce an output between 0 and 1. As a result of this, the forget gate contains the relevance of the previous hidden state with respect to the current input. The next gate following the forget gate is the input gate. The input gate also takes in the previous hidden state, the input of the current time step and its own bias and passes them through the sigmoid (σ) function. Despite the similarity between the forget gate and the input gate in terms of their formation, their difference is majorly in their respective applications, which will be explained shortly. The next component that comes after this is the new cell state candidate, which also takes in the previous hidden state, the current input and its own bias but passes it through a tanh function instead of a sigmoid (σ) function. The major difference between the sigmoid (σ) function and the tanh is in their range of output, with the tanh having an output range between -1 and +1, while the sigmoid (σ) function has an output range between 0 and +1. The difference between both functions determines the nature in which they are used in this architecture. Due to the ability of the tanh to preserve both positive and negative relationships (not only in magnitude but in direction as well) between input and output, it is used in this capacity to define the cell states retaining more information about how the inputs affect the outputs. The sigmoid (σ) function on the other hand is used instead to retain the relevance of the information captured, which values closer to 0 meaning lower importance and therefore discarded and values closer to 1 meaning higher importance and therefore retain during combination of these elements.

The forget gate is applied on the previous cell state, which causes a memory effect on the previous cell state, with irrelevant relationships (values) being forgotten and relevant relationships (values) being retained. The input gate is applied on the new cell state candidate, with the input gate having a similar memory effect on the new cell state candidate. This effect retains relevant values in the new cell state candidate and discards the irrelevant values. The previous cell states that have been modified by the forget gate and the new cell state candidate that has been modified

by the input gate are then combined to form the new updated cell state for the current time step. This current cell state isn't outputted as is but is rather filtered (simply put) before being outputted. The filtered form is what is known as the hidden state which is presented as the output instead. The hidden state of the current time step is a product of the updated cell state, after it has been modified by the output gate. The output gate, similar to the forget gate and the input gate, takes in the previous hidden state and the current input and passes it through a sigmoid (σ) function in order to retain relevant relationships and discard irrelevant relationships. The updated cell state is passed through a tanh function before being combined with the output gate to produce a filtering effect on the irrelevant relationship. This filtered version becomes the new hidden state for the current time step and also serves as the output for the given time step.

The current cell state and the current hidden state are then carried over to the next time step to serve as the previous cell state and hidden state for the requirement computations. This process is repeated in a cyclic manner until all time steps are accounted for. This combined effect from the forget gate, input gate, and output gate help in mitigating the short term memory of the simple recurrent neural network to a reasonable extent, and allows the model to remember relationships between inputs with more time steps between them.

2.4 CodeBERT

Code Bidirectional Encoder Representations from Transformers (CodeBERT) is a model for learning from an input sequence to produce an output sequence. To understand CodeBERT, it is important to first understand the BERT model and its underlying architecture. This is important because the CodeBERT model is a variant of the BERT model which was specifically built to account for NLP tasks including programming languages. To better understand the BERT model, it is also important to understand the Transformer model as explained in the paper entitled "Attention is all you need" by Ashish *et al.* in the year 2017 [27]. The transformer models are made up of an encoder component and a decoder component as seen in Figure 2.7. There are many models based on a combination of these two components, like the BART model, however, the BERT model uses only the encoder component, therefore the explanation in this research will focus on the architecture of the encoder part of the Transformer architecture.

The encoder part of the model takes in an input embedding. This input embedding is a combination of the vector representation of each token from the tokenized raw input (the text) and their positional encoding. Due to the parallel nature of the transformer model, there is a need to hard code the position of each token. This way the model is able to take the sequence order of the input into account. This can be considered a substitute method for the processing of inputs in time steps as in the case of the simple recurrent neural networks and the long short-term memory neural network.

Given a vector representation of the tokenized version of the raw input sequence $T_t = \{t_1, t_2, t_3, ..., t_n\}$, and the hard coded position vector representations $P_t = \{p_1, pt_2, p_3, ..., p_n\}$, the final input embedding is given as the element-wise addition of the the two vectors [27],

$$e_i = t_i + p_i \tag{2.20}$$

Hence, the final input embedding is simply,

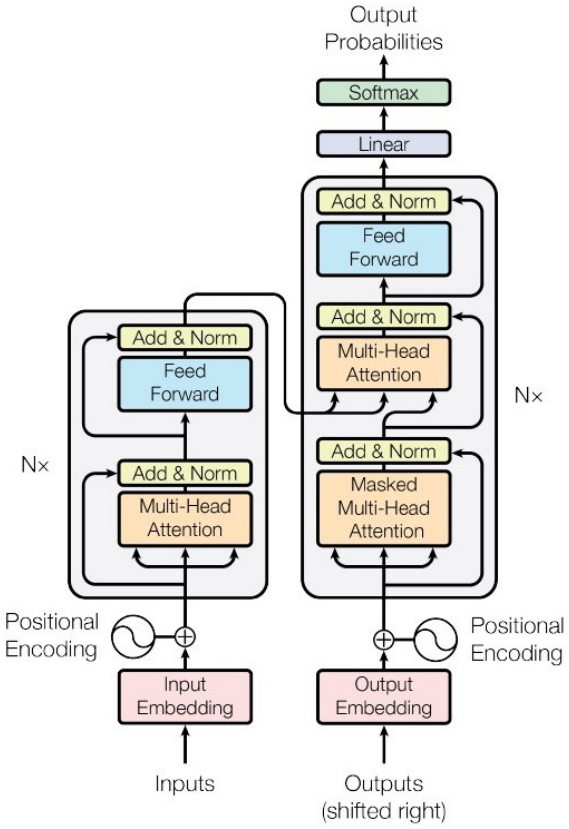

FIGURE 2.7: The Transformer Model Architecture [27].

$$E = T + P \tag{2.21}$$

This final combined input embedding is then fed into the encoder component. The encoder combined can be split into two major sub-components which are the multi-headed attention and the feed-forward neural network. The multi-headed attention comprises of the query matrix, key matrix and the value matrix. The combined input embedding is used to create the query, key and value matrices using their respective weights (query weight, key weight, and value weight). These weights are learned during the training of the model. To give a non-technical explanation of what these matrices represent, the matrices can be seen as storing information about the relation between the tokens to other tokens and the information contained within a token itself, which are then later combined to determine the amount of attention to be paid on each token and the amount of attention each token pays on other tokens.

$$Q = W_Q \cdot E \tag{2.22}$$

$$K = W_K \cdot E \tag{2.23}$$

$$V = W_V \cdot E \tag{2.24}$$

After deriving these matrices, the query and key matrices are combined using dot product and then normalizes using the dimension of the key matrix. This normalization is to prevent exploding gradients during training. The result of this union

represents the representation of the relationship between tokens coming from the input and it accounts for both directions of influence. After the dot product and normalization is carried out, the result is then passed through the softmax function which translates all the values of the matrix to a value between the range of [0 -1].

$$A = softmax(Q \cdot K^T / \sqrt{d_k}) \tag{2.25}$$

The purpose of this is to amplify the important relationships while discarding or paying much less attention to the irrelevant relationship amongst tokens. This is similar to the function of the forget [2.14], input [2.15] and output gate [2.18] in the long short-term memory neural network. The output of the softmax function executed on the normalized dot product of the query and key matrices is then used to multiply the value matrices to get the final output of a single head of the multi-headed attention component.

$$Z = A \cdot V \tag{2.26}$$

This mathematical information flow is visualized in the figure below,

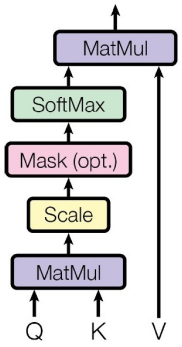

FIGURE 2.8: The Output of a single attention head [27].

The reason it is called a multi-headed attention is because it comprises of multiple single heads stacked and trained in a parallel manner, with each distinct head better representing a part of the details contained in the input and its context. The results from all the single heads are then concatenated to produce the input for the feed forward neural network.

The outputs of all single attention heads are concatenated along their last dimension. Assuming there are a total of h attention head, and the output for attention head i is H_i with a dimension of $[m, n]$, the concatenated output is given as,

$$H_c = [H_1, H_2, H_3, ..., H_h] \tag{2.27}$$

Simply put, the result of the separate attention heads are stacked horizontally, which is similar to concatenating two pandas data-frame tables in a python environment. The final output H_c will have a dimension $[m, h \cdot n]$.

This final output is then fed into the feed forward neural network which then uses the weights learned during training to carry out the task required of it, which in this case is the binary classification of a given code-comment pair. The feed forward consists of two fully connected layers, with the first layer taken in the input H_c from the multi-headed attention component, and the second layer producing the output desired.

$$H_{i+1} = \text{ReLU}(H_i \cdot W_i + b_i) \tag{2.28}$$

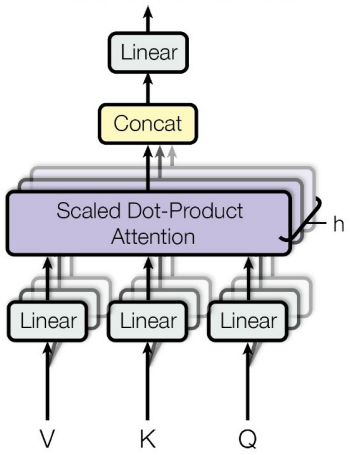

FIGURE 2.9: Multi-headed Attention [27].

in which

- H_i is the input coming from multi-headed attention, in the case of the first layer, or the input coming from the first layer in the case of the second layer (final layer).
- W_i is the weight connecting the input layer to the first layer, or the weights connecting the first layer to the second layer (final layer).
- b_i is the bias of the first or second layer.
- H_{i+1} is either the output of the first layer going into the second layer, or the output of the second layer which is the final output of the feed forward neural network.

It is also important to note that large language models like BERT aren't made up of a single encoder components, but multiple encoder components stacked on top of each other with the outputs of one encoder (which is the final output of the feed forward neural network) serving as the input of the encoder coming after it, and the final encoder carrying out the final task like classification of sequence prediction. The base form of the BERT model for instance has about 12 stacks of these encoders, while the large version of the BERT model has up to 24 stacks of these encoders. The more encoders you have stacked, the more context, internal meaning and nuance your model is able to represent. However, this will significantly increase the size, and expense needed to train the said model. This means that like all other hyper-parameters, the number of encoders stacked on each other, or even the number of single-heads to make up the multi-headed attention component needs to have a trade and balance between the time, cost, space complexity and the desired efficiency.

Chapter 3

Related Work

In this chapter, we review the related work with focus on those dealing with the detection of code-comment coherence. Moreover, we also recall different techniques for word embeddings as a base for further presentation.

3.1 Code-Comment Coherence

The first work to address the issue of code comment coherence was published by Corazza *et al.* [6]. Due to lack of adequately labelled data to use for this research, Corazza *et al.* collated and annotated the data used in their original work themselves and published this in another paper [5]. In this paper Corazza *et al.* annotated code-comment pairs obtained from 3 different open-source projects written in Java. The data gathering and annotation was done this way to encourage other researchers to extend the idea if interested. In their approach, code-comment pairs obtained from these projects were given to annotators to label, and in the case of conflict amongst annotators, the conflict would be resolved by the experts. If the experts were at conflicts themselves, then the code-comment pair in question is discarded. They then used the kappa index to measure the reliability of the labels obtained from the annotators.

After the dataset was adequately prepared, three different approaches were used to investigate code-comment coherence:

- Checking if the lexical similarity between the code and comment calculated using cosine similarity had any correlation with the coherence of each pair.
- Executing a dimensionality reduction method (PCA) on the vector encoded dataset using the vector space model (VSM), and visualizing the result to check if there is some obvious divide between coherent and non-coherent pairs.
- Using support vector machines (SVM) to train and predict on the encoded dataset.

This encoding was done using the tf-idf encoding schema. TF-IDF stands for term frequency - inverse document frequency, and is an encoding method which combines the occurrence of a term(word) in a given document (code-comment pair) and its rareness in the entire set of documents to assign a vector representation to this term. The authors stated that the encoding method used would have some disadvantages in terms of the amount of information it captured from the dataset, but decided to explore this option due to the fact that despite its simple appearance it has shown potential in similar tasks.

The result of the investigations carried out in this paper [6] showed that;

- The lexical similarity (calculated using cosine similarity) between the code-comment pairs was low in both the coherent and non-coherent pair, despite the fact that the similarities were slightly higher in coherent pairs. Further visualization and analysis involving hypothesis test to check if there was a significant different between the lexical similarity of coherent and non-coherent pair shows that there is a significant different between the lexical similarity of these two categories which could play a significant role when differentiating these two categories given an appropriate threshold.
- The visualization done after the PCA dimensionality reduction method was carried out on the encoded dataset didn't do much to help differentiate between coherent and non-coherent pairs.
- The classification method explored using a combination of the SVM and VSM methods turned out to be the most reliable of the three methods explored with an average accuracy of 83.5% and F1 score of 86.9%.

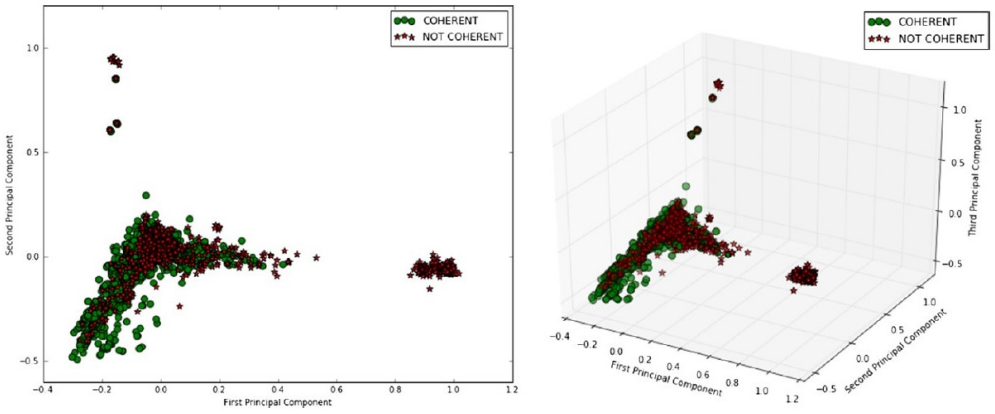

FIGURE 3.1: Visualizing the vector representation of the input data after PCA by Corazza *et al.* [6]

They also noted that the difficulty in the annotation of the JHotDraw dataset given its kappa index seemed to reflect in the performance of the model trained on it as the model trained on this particular dataset during the research yielded the lowest performance amongst all other models.

In a more recent paper [3], Alfonso *et al.* having noted the possible limitation of using the tf-idf encoding schema in the earlier paper [6] done by Corazza *et al.*, explored more advanced encoding options in combination with a couple other classification methods. In this research paper, the authors stated their concern with the simplicity of the VSM (tf-idf) method which led to their decision to explore more advanced methods in text encoding. They implemented various vector embedding methods, which were all variants of the Word2Vec ref and the Global Vector ref models. These embedding methods were then combined with three classification engines, which were Support Vector Machine (SVM), Random Forest and K-Nearest Neighbours. The experiment was carried out on the same dataset used by Corazza *et al.* [6]. The results from their experiments show that the baseline model from [6] Corazza *et al.* gave a better performance when compared to the methods explored, despite being very close in performance. They also mentioned the difference in embedding time for the baseline compared to the methods they explored, with the baseline having a much larger embedding time. While this paper attempted to capture

the inherent meaning of the words that make up each code-comment pair, this was their only focus leaving out the sequential order of these words.

Shortly after this, another work [24] tried to improve Corazza *et al.* [6] by capturing the sequential order in words. In this paper, Abstract Syntax Tree (AST) was used to create the embedding to be fed into the classification engines. AST is a tree-like representation of the syntax of a method, which includes elements like type of method (in the case of programming languages that explicitly define the method type), declaration of variables etc.

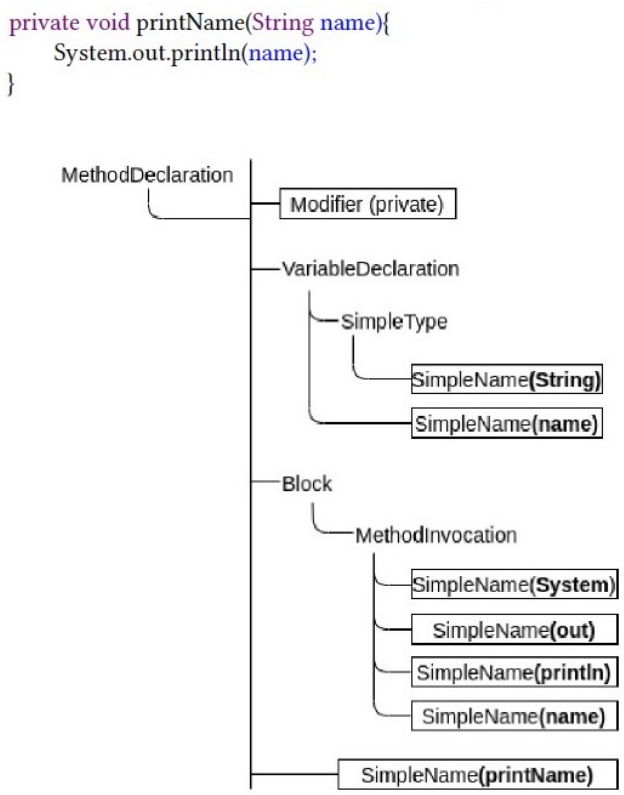

FIGURE 3.2: An example of an Abstract Syntax Tree for a method [24].

This tree was then extracted from the source code for each code-comment pair, and the elements of all the trees combined with elements from comments were used to create a vocabulary which was used to vectorize the code-comment pairs by index representation in the vocabulary.

These vectors are then fed into the classification engines which were the RNN and LSTM models. These models were used in order to account for the sequential order of words in code-comment pairs. The results from this research weren't compared with any baseline, but rather compared with itself. It was noted that the approach paid more attention to recall rather than precision and accuracy. After the models were built and tested, the word order of coherent code-comment pairs were randomly rearranged and fed into the network and it recognized them as incoherent, which was a way of testing if sequence really did play a role in the model recognizing coherent and incoherent code-comment pairs.

Similar to the previous paper discussed [3], this research [24] only paid attention to capturing the sequential ordering of words while leaving out the generalized inherent meaning of the words that make up the code-comment pairs. The encoding method explored was still very similar to the bag of words method which doesn't recognize the inherent meaning of words or their similarity to other words, hence

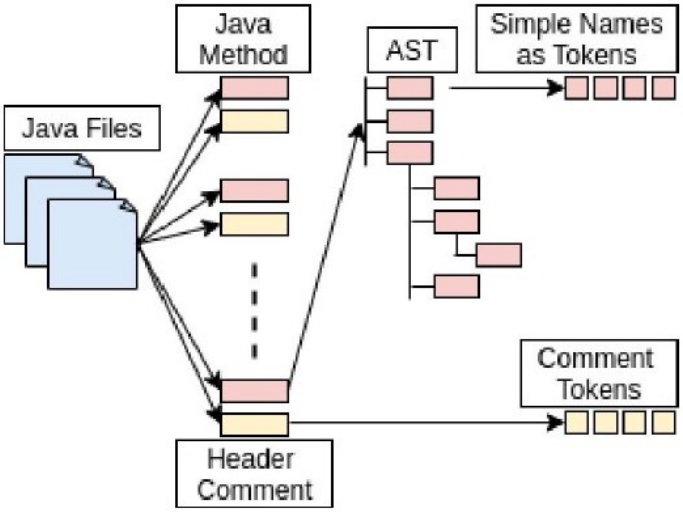

FIGURE 3.3: AST-based Tokenization [24].

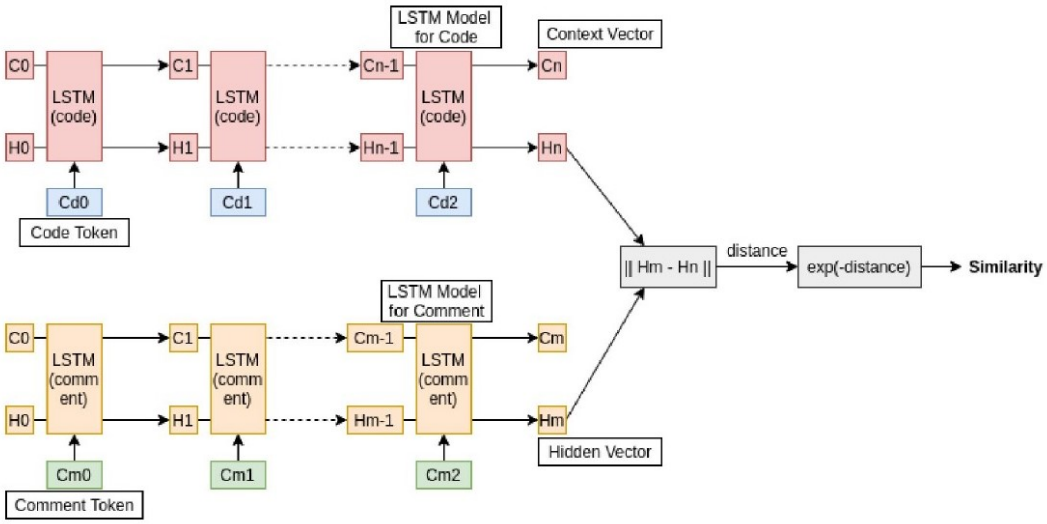

FIGURE 3.4: AST-RNN Combined Architecture [24].

might not be able to capture coherence or incoherence of similar code-comment pairs made up of synonymous words.

Two years after the last paper discussed [24], another research [25] was conducted under the same subject matter, which employed ensemble learning to detect code-comment coherence to improve on the vector space encoding approach used in the first paper by Corazza *et al.* [6]. In an effort to reduced the size of the input vectors and possibly reduce the time it takes to encode the raw text to vector representations, the authors used Latent Dirichlet Allocation (LDA) to summarize similar words in the codes and comments and represented them with the same topic.

The code-comment pairs were used to create a bag of words, with one bag of words separate for the codes and another for the comments. The vector representation for the code and comment for each code-comment pair was then fed into an LDA model which produced probability distribution for these vectors. For a given K-number of topic, the LDA model outputs the probability of the code and comment falling under a given topic. This probability distribution for the code and the comment of a given code-comment pair was then concatenated to form the final vector

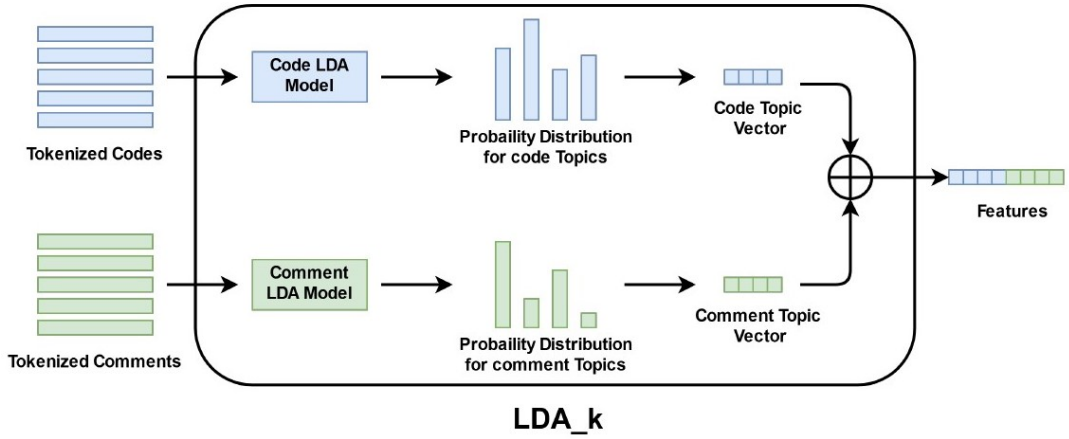

FIGURE 3.5: Summarizing Code-Comment using LDA [25].

representation for a code-comment pair. The final vector was first fed into a Random Forest model having 10 features, and later fed into an ensemble model which was a combination of Random Forest and Support Vector Machine.

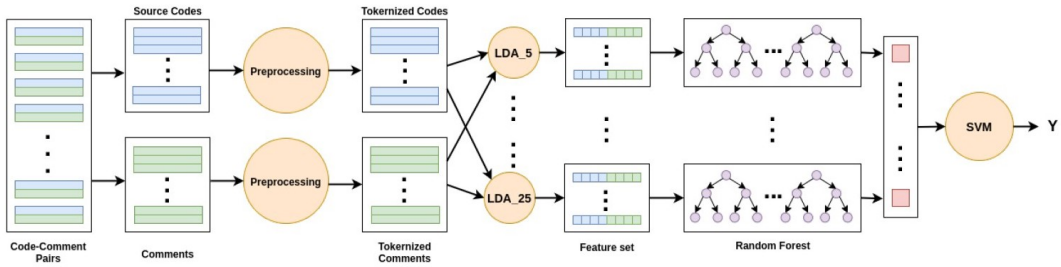

FIGURE 3.6: Ensemble Architecture (Random Forest and SVM) [25].

The results of this research showed an unsatisfactory accuracy from the first random forest model with 10 features. However, there seemed to be significant improvement in the model accuracy after combining the Random Forest model with the Support Vector Machine. This approach went a step further in trying to capture some internal meaning in the word by using a summarization technique, which still relied on a bag of words method. This means that the information captured from the data is dependent on the dataset used to build the corpus, and the classification methods used don't pay attention to the sequence order of the words that make up the code-comment pairs.

The most recent work similar to ours is the one by Steiner and Zhang [26], which considered the use of large pre-trained models, i.e., BERT and Longformer, for code-comment coherence check. In this research the authors tried to also account for real-time detection of code-comment coherence. The authors decided not to rely on external auto-encoders to convert their code-comment pairs to their vector embedding forms, but decided to use the in-built encoder that comes with the BERT model. Due to the BERT model having a maximum input length of 512 tokens which could lead to loss of information in long methods, the authors also consider the Longformer model which has a maximum input length of 4096 tokens. However, considering the significantly longer encoding time given the effort to accommodate much longer code-comment pairs, they decided to truncate the input length of the Longformer to 1024 tokens. The results of the research show that the two model (BERT and Longformer) considered in this paper significantly outperform the baseline considered

in terms of accuracy and F1 score with respect to the post-hoc prediction (which refers to pre-existing inconsistencies in code-comment pairs), while in terms of the real-time comparison, the BERT and Longformer model were outperformed by the baseline models.

The work done by Steiner and Zhang [26] was quite an interesting one. However, unlike the approach considered in their research, the methods considered in this paper covered more than checking performance amongst proposed methods and baseline, it also focuses on an in-depth look into the factors that could differentiate these pre-trained models from the traditional ones. The predictors of Large attention model architectures themselves aren't sequence dependent, however, it is important to state that although their embedding models aren't trained like the Word2Vec models considered in this paper, they still pay attention to context and word order in sentences which is crucial in Natural Language fields. This also places these large pre-trained models at a time-related advantage when it comes to word embeddings. Large pre-trained models can also be limited in how much they can be customized. However, what they lack in flexibility, they make up for in performance due to the underlying architecture and their sheer size. Even though this can be seen as a good attribute, it could also serve as motivation, because if simpler and smaller models while leveraging domain knowledge and specific embedded attribute in these textual data can match or even outperform these large models as well as baseline model, then it would be redundant to use large models in instances where these small model could come in. This is the case with our approach, in this work we show that even simple architectures can have a better performance compared to the approach built on heavily pre-trained models [26].

3.2 Word Embedding and Deep Learning

There are several embedding methods developed in the field of machine learning. Machine learning models, regardless of the type of prediction (image, numerical, textual, sound etc) they are deployed to handle, can only take input in numerical form. The field of Natural Language Processing is no different and as such have had various embedding methods developed over the years, with each method aiming to solve specific problems or account for certain aspects of the input data being fed into the model.

Some of the simplest embedding methods include the Bag of Words method and other slightly more advanced methods like the TF-IDF method. These models although faster than a lot of embedding methods, can generate a very large amount of data, and also tends to miss out some inherent meanings in the input data like the sequence or the contextual meaning of the elements that make up the input data. It is for these reasons that the Word2Vec embedding method was proposed by Mikolov *et al.* [19] in 2013. This method proposed in Mikolov *et al.*'s paper represents words by their context. This means that words that often occur in the same context have similar meaning and hence have similar vector embedding. This helped in capturing some nuance and internal meaning with the vector representation of words. However, this method, due to being much more complex than many of those that came before it, takes much more time to transform data for training as well. Despite the time required to convert data using this method of transformation, it was also limited to only words which existed in its corpus at the time of training. This caused a break in continuity whenever words outside its training corpus were encountered which also led to loss of information. A few years after the Word2Vec method was

developed, an improvement on this method was proposed, which was the FastText method by Bojanowski *et al.* [2] in 2016. The proposed improvement made by Bojanowski *et al.* sought to also account for the n-gram of various words in order to be able to construct a vector representation of word which aren't explicitly included in the original training corpus. This led to being able to capture more information from the input data for better pattern recognition by the trained models.

Recurrent Neural Networks in a similar manner was developed to account for data which comes in sequential order. One of the earliest works released on this method was by Jordan [14] in 1983. It achieved this purpose by processing input in time steps, linking one input to the next coming after it. This became more efficient than the regular multi-layer perceptron in sequence processing but was vulnerable to what was referred to as "short term memory". Given that each input is only directly linked to the one before it, any relations between inputs that extend beyond these direct connections are less and less accurately represented as the difference in time step increases. This then led to development of the Long Short Term Memory model (LSTM)[11] in 1997. This model was a variant of the recurrent neural network model which used gates and cell states to simulate a long and short term memory effect in the information processing by the model.

A machine learning model isn't only the optimization architecture but also the input transformation model, and when both these aspects of the model are properly selected, much of the information contained within the dataset can be captured by the model. Several research methods have combined these methods due to their combined ability to retain information within the input data which leads to higher prediction accuracy. In 2020, Santos *et al.* [18] combined the Word2Vec and the Long Short-Term Memory Neural Network to detect self-admitted Technical Debt in software, and found that this combination outperformed all other models explored in their research. This combination has proven to be quite efficient in the field of Natural Language Processing including in other languages as in the case of Muhammad *et al.* [20] where it was used to detect positive and negative hotel reviews in Indonesian language.

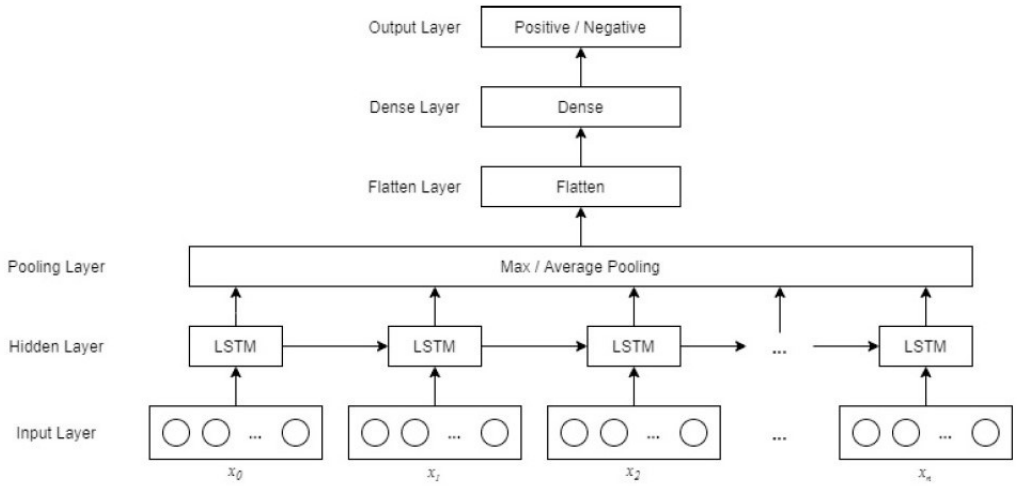

FIGURE 3.7: Architecture for LSTM combined with Word2Vec by Muhammad *et al.* [20]

This combination has also been applied in other more recent research works like Mallik *et al.* [17] where this combination was used to detect context-free fake news. A solution which has become increasingly necessary in today's digital world. In their

research, they compared this combination to many other baselines and methods (including large pretrained models like BERT) and found out that this combination significantly outperformed all these other models. Similar to the results gotten by Mallik *et al.*, a combination of Word2Vec and Bi-LSTM (a slightly more advanced version of the LSTM model) outperformed all other models when used to conduct sentiment analysis by Jaca-Madariaga *et al.* [13].

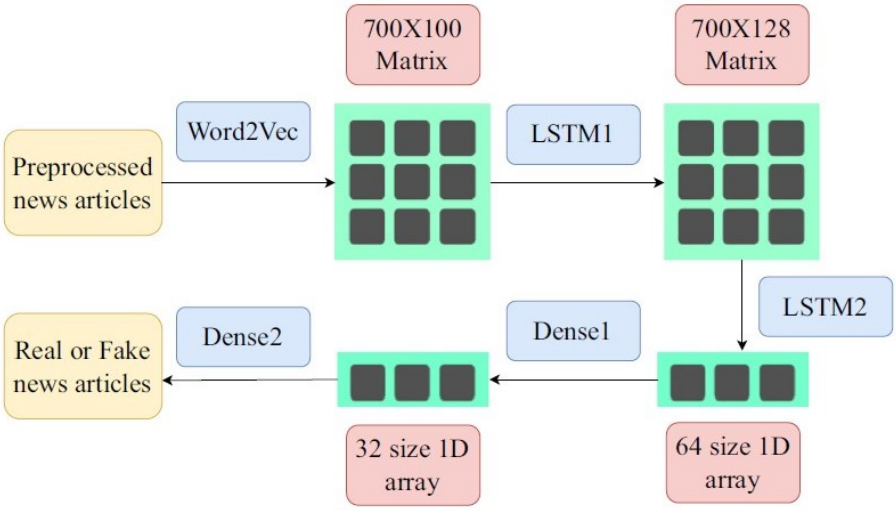

FIGURE 3.8: Architecture for LSTM combined with Word2Vec by Mallik *et al.* [20].

These simpler models have challenged much larger and advance models like the BERT [7] and CodeBERT models [8] which were built based on the proposed solution by Vaswani *et al.* in their 2017 paper titled "Attention is all you need" [27]. This paper by Vaswani *et al.* created the foundation for many advanced models used in the field of Natural language today. Some of the notable models include the BART (BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation), BERT (Bidirectional Encoder Representations from Transformers), RoBERTa (A Robustly Optimized BERT Pretraining Approach), MBERT (Multilingual BERT), T5 (Text-to-Text Transfer Transformer), and even the GPT (Generative Pre-trained Transformer) Series. In this research two of these large pretrained models (BERT and CodeBERT) along with the simpler models like Simple RNN+Word2Vec, LSTM+Word2Vec and Support Vector Machine (SVM) were implemented and compared in order to answer the research questions posed in Chapter 1.

Chapter 4

Proposed Approach

4.1 Motivation

4.1.1 Synonym

The internal meaning of words makes it possible for two sentences–made up of entirely of synonyms–to be interpreted as having similar or same intent, e.g.,

"*The baby dog crawled below the fence,*"

"*The puppy crept under the wall,*"

despite not containing the exact same words, should be interpreted as having the same information or intent.

Methods like Bag-of-Words encoding or the tf-idf encoding were designed to be relatively simple but in turn overlooks the inherent meaning that exists in words used to form sentences as understood by humans. If two synonymous words were used to build the corpus for the bag of words encoding, despite being synonymous would each fall under entirely different columns, which would lead to use of more space than needed in data representation and would lead to poor generalization on unseen data. These methods like the bag of words methods hinge on the idea that humans in general use limited words out of all the words at our disposal to apply our lingua. Despite this being true to an extent, any increase in this used vocabulary or evolution of the set of words used significantly impedes this idea.

4.1.2 Word Order

The order of words in a sentence differentiates two sentences made up entirely of the same words, but with different orders. For example,

"*The student listens to the teacher,*"

"*The teacher listens to the student,*"

have different meanings, though they share the same set of words.

In code comment, we encounter a similar phenomenon. Consider a method that takes base2 values as input, and produces as output base10 values, then the method is associated with the following comment:

*/**Takes base2 values as input and outputs base10 values*/*

There is the second method that takes base10 values as input, and outputs base2 values, however it has the following comment

*/**Takes base2 values as input and outputs base10 values*/*

An encoding method that does not take into account word order, will not be able to recognize that the second code-comment pair is actually incoherent, as the correct version should be

*/**Takes base10 values as input and outputs base2 values*/*

As stated earlier, methods like the bag of words approach pays no attention to the sequence in which these words occur in a given text input. This could create a false generalization, where an unseen data is mistaken for an already seen data due to this phenomenon.

4.2 Encoding Methods

Considering the level of emphasis put on the importance of the methods used to transform the dataset, four popular encoding methods were considered before selecting the one that suited the features our approach sought to account for. These four methods are the Bag-of-Words, TF-IDF, Word2Vec, and FastText.

4.2.1 Bag-of-Words

This could be considered the simplest method of encoding a dataset despite the fact that it generates a significantly large feature set. In this method, the most frequent words in the entire dataset are selected after having removed the stop words. The most frequent words before a certain threshold (e.g. first 1,500 most frequent words) are used to constitute the feature vectors. Each new entry is checked against this feature vector and is encoded by inputting ones (1) for features (words) that exist in this entry and zeros (0) for features that do not exist in the said entry.

Given a set/bag of words,

$$v = [\text{'}ant\text{'}, \text{'}bat\text{'}, \text{'}cat\text{'}, \text{'}dart\text{'}] \tag{4.1}$$

the bag of words representation for an input containing only the words "*bat*" and "*cat*" will be,

$$i = [0, 1, 1, 0] \tag{4.2}$$

As stated earlier this is a relatively easy encoding method, however, it requires a large feature set to capture relevant information, and it also creates sparse input data which could negatively affect the model performance.

4.2.2 TF-IDF

This is a slightly more complex encoding method than the Bag of Words method, and it considers the mathematical significance of words in its own entry as well as the entire dataset. Tf-IDF stands for Term Frequency – Inverse Document Frequency. This value for a given word is obtained by combining the importance of the said word in its entry with its importance in the entire dataset, therefore unlike the Bag-of-Words method the values for any cell in the input matrix isn't restricted to either zero or one. The first part of this TF-IDF encoding method is the Term Frequency which simply refers to the ratio of frequency count to the total word count of the

said word in its own entry. While the Inverse Document Frequency is the logarithm of the ratio of the number of documents in which the said word occurs to the total count of documents in the entire dataset [6].

Given a set of document, the tf-idf value is defined by the following mathematical formula;

$$tf = N_t / N_T \tag{4.3}$$

$$idf = \log(N_D / Nd) \tag{4.4}$$

$$tf - idf = tf \cdot idf \tag{4.5}$$

where

- N_t represents the number of occurrences of term t in a given document.
- N_T represents the total number of terms in the same document.
- N_D represents the total number of documents in the dataset.
- N_d represents the number of documents with the term t present in them.
- Note: "Document" in this context represents a code-comment pair, while the dataset is the group of code-comment pairs.

Similar to the Bag-of-Words approach the most frequent words before a given threshold is used to constitute the feature vector which will be used to encode the dataset. For a given entry, the TF-IDF value of each word in the feature vector made up of the most frequent words in the entire dataset is calculated and used to generate the vector representation for that particular entry. Again, similar to the Bag-of-Words approach, this encoding method pays attention to neither the inherent meaning of words nor the sequence order of words in a given text entry.

4.2.3 Word2Vec

In this encoding method, the entire dataset is used to encode each entry. The vector of each word being encoded is determined by checking which words it occurred more often with, which in turn tends to capture some inherent synonyms amongst words (or their usage at least). Words that tend to always occur amongst similar sets of words will have vector representations that indicate this closeness [19]. When this corpus is built using only the dataset to be predicted on, there tends to be a limit on how well this encoding method will help the model generalize to unseen data. However, this must not be the case as the corpus can be built independently of the dataset. In a lot of practical implementations of this approach, the corpora are built using very large pools of standard textual data so as to capture a much more generalized encoding of most words that can be used to form sentences. Following this approach, encoding a given word can be seen as looking up a word in a dictionary. This captures the inherent meaning of words defined by their synonyms to other words and the context in which these words are used, provided these words were included in building the corpus used for the encoding. The building of the corpus used for encoding is done using artificial neural networks. Finally, this does not produce sparse vector representations like the Bag-of-Words and the TF-IDF methods, as the feature vector is not dependent on the number of words in the vocabulary.

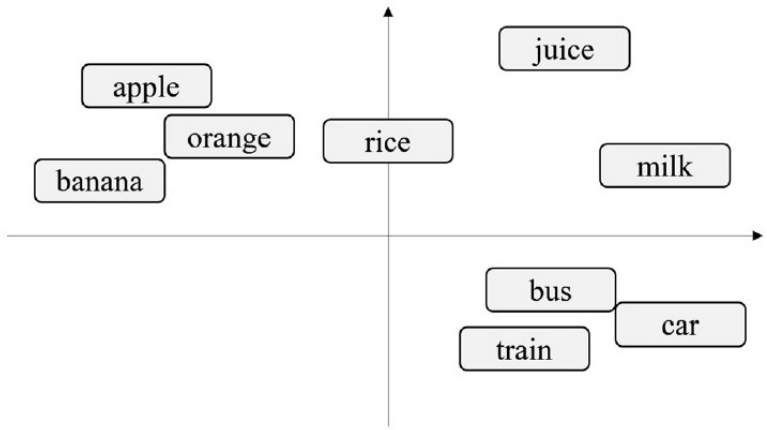

FIGURE 4.1: Illustration of closeness between vector representation of similar and dissimilar words [15].

The Word2Vec method was described more in-depth in Section 2.1.

4.2.4 FastText

This encoding method is very similar to the one described in the Word2Vec encoding method. As stated in the description for the Word2Vec method, words that were not included in the building of the corpus are not recognized by the corpus therefore will not yield any results if looked up in the corpus. This is the major difference between the Word2Vec and the FastText encoding method. The FastText corpus is built while including n-gram splits of words, therefore not only is there a vector representation of words but their n-gram splits as well. With this additional consideration, it takes a significantly longer time to build a FastText corpus, but words that aren't explicitly included in the corpus while building can be split into n-grams and represented by the combined representation of its n-grams as defined in the corpus [2]. This leads to more information captured from a given dataset during encoding as more words have a representation that can be fed into the model. This embedding method was also discussed further in Section 6.1.

4.3 Model Architecture

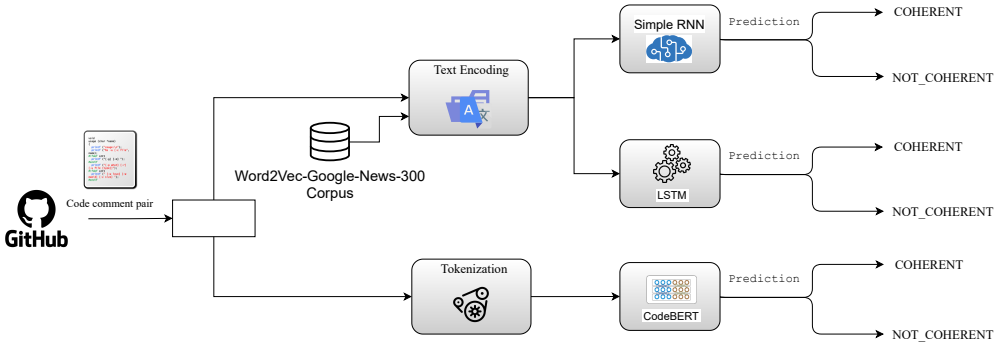

FIGURE 4.2: The overall architecture.

The workflow depicted in Figure 4.2 represents the system architecture for the solution proposed in this research work. The data to be categorized is gotten from GitHub and other code repositories and is cleaned and transformed. The transformed data is then parsed through two different tokenizers which are the Word2Vec tokenizer (Section 2.1) and the CodeBERT tokenizer. The output of the Word2Vec tokenizer is fed into two different NLP Deep Learning models (Simple RNN in Section 2.2 and LSTM in Section 2.3) which then predicts the class of the given input (coherent or non-coherent). Similarly the output of the CodeBERT tokenizer is fed into the CodeBERT Large Language Model (LLM) (see Section 2.4) which then predicts the class of the given input as well.

In this architecture, only the CodeBERT model makes use of transfer learning, and its weights are already pretrained. The Simple RNN and the LSTM models were trained from scratch.

4.4 Data Conversion

The Gensim word2vec-google-news-300 corpus[1] is utilized to encode data for Simple RNN and LSTM. In particular, each code-comment pair is tokenized, whereby a token is represented using a vector of 300 entries. Since the length of sentences varies, each entry of the list of vectors is constrained to a maximum length of 45 vectors. Taking a look at the histogram plot of the sentence length of each dataset as shown in fig, it can be observed that most of the sentences have a length that falls between 40 – 70 words, which was the reason for this threshold chosen for the maximum length of vectors for each entry.

4.5 Classification Engine

We address the aforementioned issues by focusing on the encoding method used on the text accounts for the internal meaning of words, and the classification algorithms used to account for the sequential word order. This is done by using a combination of the Gensim word2vec encoding method [19], and a Simple RNN [14], or an LSTM model [11]. To validate the efficiency, we also use CodeBERT together with tokenization to encode an entire sentence, paying attention to both the internal meaning and sequential word order.

Simple RNN and LSTM are specified with the same configuration, i.e., both having a single hidden layer with 100 nodes, and an output layer with dense connections. The categorical cross entropy is used to compute losses, the Adam optimizer is for defining the learning rate, with a batch size of 50, and both models were trained using back propagation. The models were trained using two Tesla T4 GPUs. The Adam optimizer was applied in the training of CodeBERT. We split the entire dataset following the 0.8:0.2 ratio, i.e., 80% and 20% of the data are used for training and testing, respectively. In Figure 2.3, we depict the proposed CO3D architecture. To facilitate the presentation, we name three configurations as follows:

- $\mathbf{C}_1$: Simple RNN + Gensim word2vec;
- $\mathbf{C}_2$: LSTM + Gensim word2vec;
- $\mathbf{C}_3$: Tokenization + CodeBERT.

[1] https://huggingface.co/fse/word2vec-google-news-300

Chapter 5

Evaluation

This chapter introduces the evaluation conducted to study the proposed solution CO3D using an existing dataset, and compare it with two baselines. Section 5.1 introduces three research questions to investigate different configurations, as well as training corpus. The datasets and baselines are described in Section 5.2. We explain in detail the configurations and evaluation metrics in Section 5.3 and Section 5.4, respectively. Afterward, in Section 5.5, we report and analyze the experimental results by answering the three research questions. Section 5.6 provides concrete information on the libraries used in the implementation. Section 5.7 discusses probable implications. Finally, Section 5.8 lists the probable threats to the validity of our findings.

5.1 Research Questions

This section introduces the research objectives, and the evaluation conducted using an existing dataset and two baselines. The CO3D proposed solution comprises of three configurations which are as follows: $\mathbf{C}_1$: Simple RNN + Gensim word2vec; $\mathbf{C}_2$: LSTM + Gensim word2vec; $\mathbf{C}_3$: Tokenization + CodeBERT.

We study CO3D with the following research questions:

- **RQ**$_1$: *Is the prediction obtained by* C_1 *and* C_2 *comparable to that by* C_3*?* C_1, C_2 are the two configurations taking into account word meaning and order when learning code comment pairs, while C_3 is built on top of a pre-trained model. We study if lightweight models also yield a comparable accuracy with respect to more complex ones.

- **RQ**$_2$: *How does* CO3D *compare with the baselines?* By using the same datasets and experimental settings, we compare CO3D with two state-of-the-art approaches [6, 26] in the detection of code comment coherence that work on top of SVM and BERT.

- **RQ**$_3$: *Does the type of corpus used for encoding a textual dataset play any role in improving the model performance?* C_3 is a similar model to the pre-trained BERT model with one major difference being the corpora on which both models are built. The corpus used for C_3 is more suitable for code related text, while the corpus for the BERT is built on regular natural language. Comparing the result from these two models could give us some form of insight into this research question.

5.2 Dataset and Baselines

We use the dataset curated by Corazza *et al.* [6], containing code-comment pairs and their coherence status from the Benchmark, CoffeeMaker, JFreeChart060, JFreeChart071 and JHotDraw741 methods [5], which possess 2,881, 47, 461, 588, and 1,785 code-comment pairs, respectively.

Concerning the baselines, we opt for the work by Corazza *et al.* [5], which employed Bag of Word (BoW) to encode input data, and SVM as the classification engine. Moreover, we consider a more recent tool developed by Steiner and Zhang [26] as another baseline, built of top of BERT [7]. The encoding for BERT is very similar to that of the Gensim corpus. The input data is parsed through the BERT-based tokenizer, where each word is converted into its corresponding vector, then the entire sequence of vectors is encoded as a sentence. To aim for a fair comparison, we run both baselines using their original implementation on the same dataset.

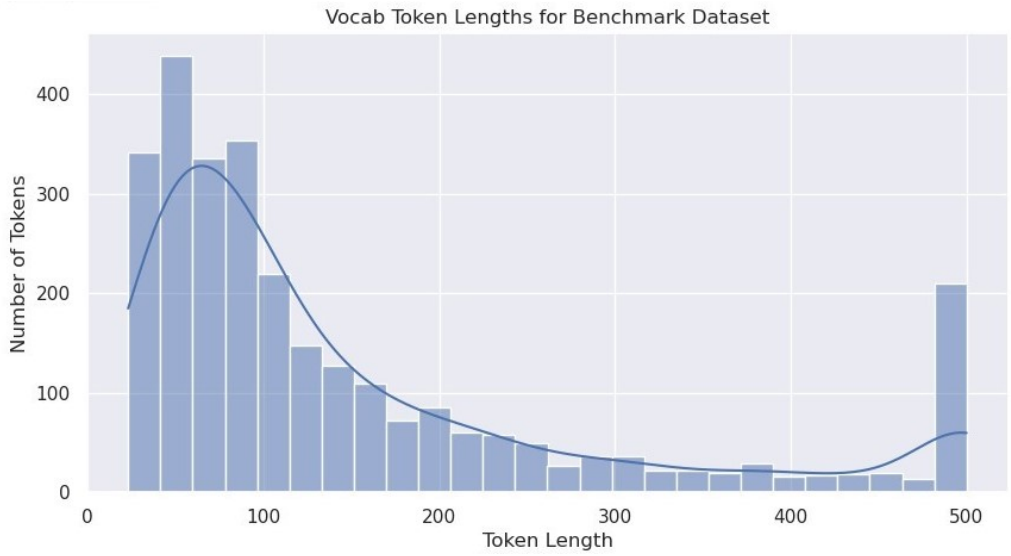

FIGURE 5.1: Frequency of token length of Benchmark Dataset.

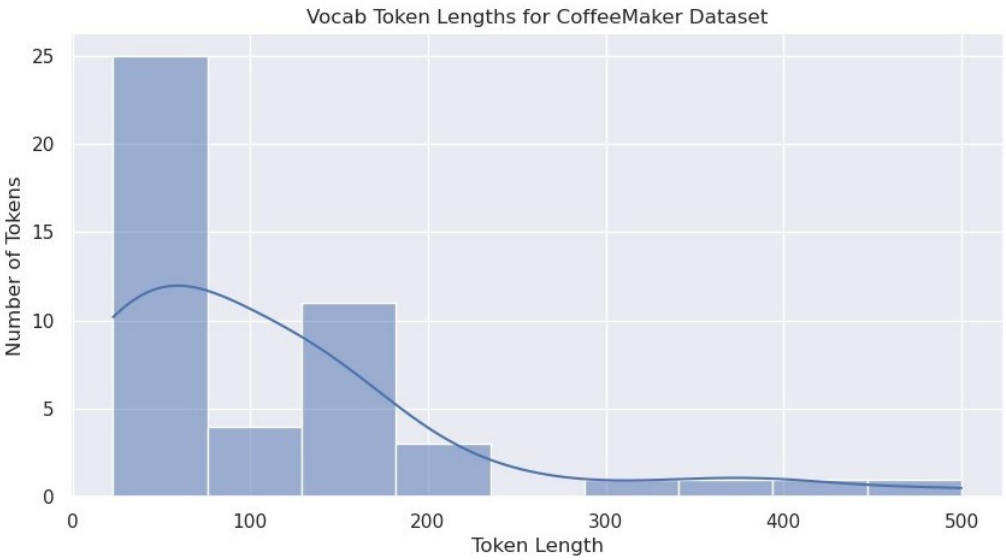

FIGURE 5.2: Frequency of token length of CoffeeMaker Dataset.

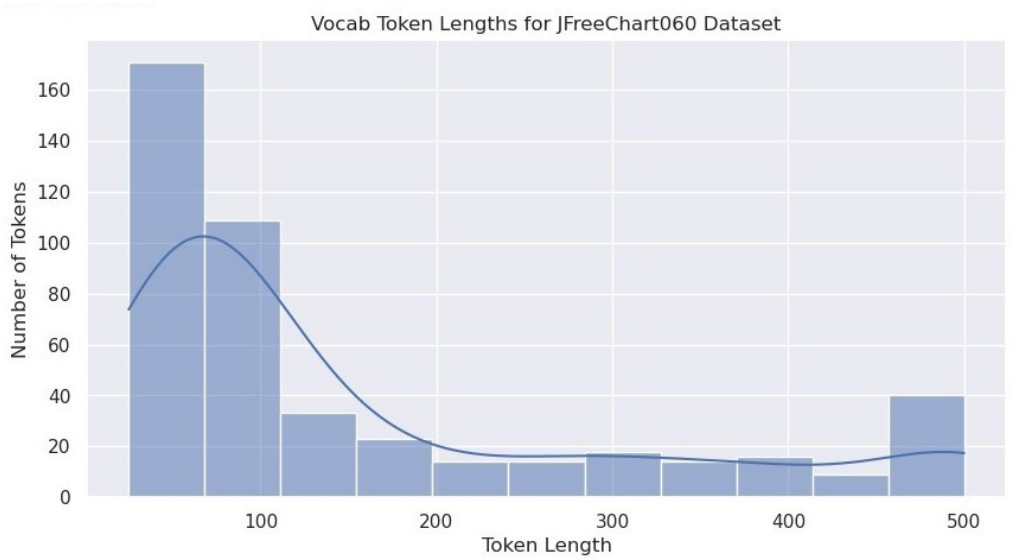

FIGURE 5.3: Frequency of token length of JFreeChart060 Dataset.

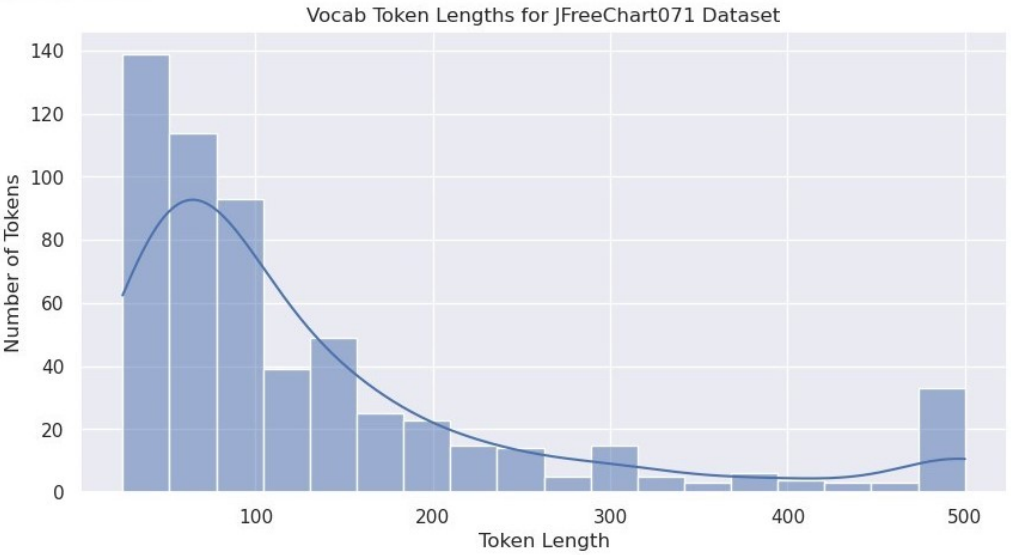

FIGURE 5.4: Frequency of token length of JFreeChart071 Dataset.

5.3 Configurations

We propose CO3D–a **Co**de **Co**mment **Co**herence **D**etector on top of 3 independent configurations:

- $\mathbf{C}_1$: a combination of Gensim word2vec and a simple recurrent neural network.
- $\mathbf{C}_2$: a combination of Gensim word2vec and a long short-term memory neural network (LSTM).
- $\mathbf{C}_3$: tokenization and CodeBERT.

While C_1 and C_2 are conceived to capture the internal meaning, as well as sequential ordering of words, C_3 is built on top of a pre-trained model. We came

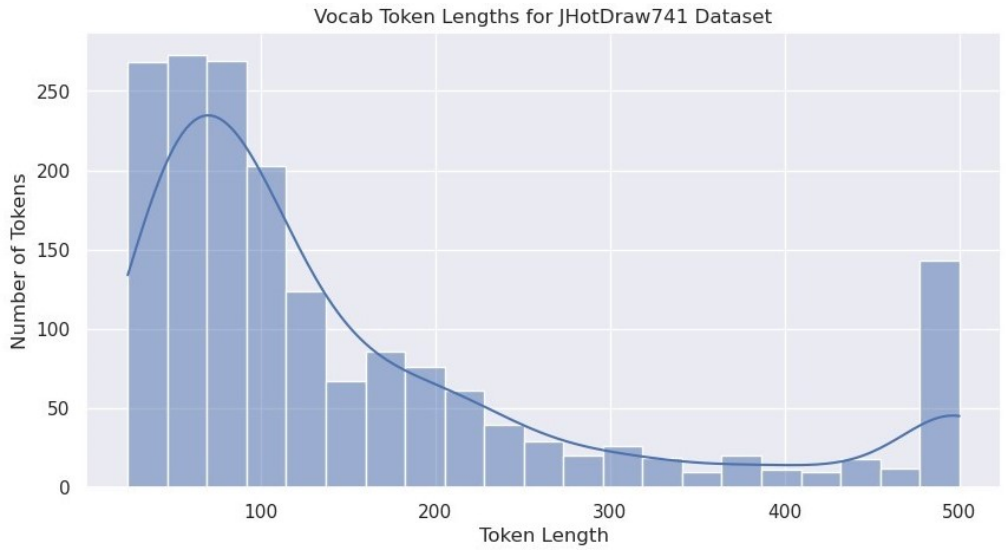

FIGURE 5.5: Frequency of token length of JHotDraw741 Dataset.

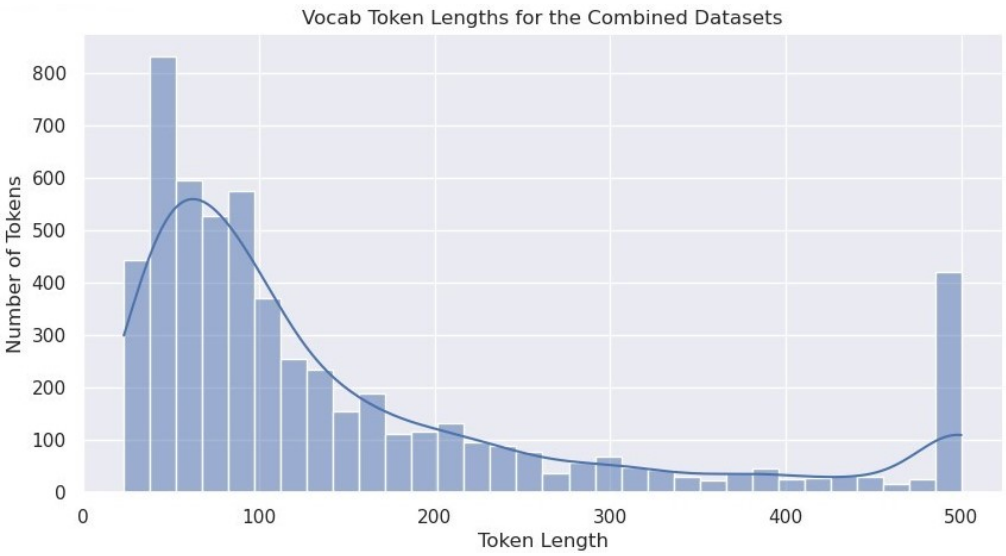

FIGURE 5.6: Frequency of token length of Combined Dataset.

across interesting results, i.e., despite being simple architectures, C_1 and C_2 bring an encouraging performance, compared to using BERT and CodeBERT–two more complicated infrastructures. It is important to highlight that the proposed approach does not necessitate a pre-training phase, distinguishing it from the considered baselines. Despite its straightforward nature, CO3D outperforms two well-founded baselines, providing evidence for the need to explore the application of these technologies in addressing the challenge of code and comment coherence.[1]

5.4 Evaluation Metrics

In this research work, four metrics were used to evaluate the performance of each model on unseen data. These four evaluation metrics are Accuracy, Precision, Recall, and F_1 as they have been widely used in evaluating machine learning systems. To

[1] https://github.com/dub-em/Co3D-Codes

first define the metrics, it is important to explain what "positives" and "negatives" mean to these concepts, as they form the basis of each of these metrics. Positives in a binary classification refers to the class being identified by the model, while negative refers to the class that falls outside the category being found. Hence, in this particular instance (this research work), positive refers to the "coherent" category, while negative refers to the "not-coherent" category. These two fundamental concepts make up the four terms that constitute the four evaluation metrics considered in this research namely; true positive (TN), false positive (FP), true negative (TN) and false negative (FN). Just as their names imply, true positive refers to positive predictions by the model that are actually positive according to the labeled data, false positive refers to positive predictions that are not actually positive, true negative refers to negative predictions that are actually negative, and false negative refers to negative predictions that are not actually negative.

- **Accuracy**: Accuracy of a model is the total number of accurate predictions (both positive and negative category) divided by the total number of predictions made.

  $$\frac{TP + TN}{TP + TN + FP + FN}$$

  This presents an absolute way to look at the model's performance.

- **Precision**: Precision however, is the total number of accurate positive predictions divided by the total number of positive predictions made.

  $$\frac{TP}{TP + FP}$$

  A somewhat intuitive way to look at this is that precision shows an inward view of models performance, which is how the model performed based on its own effort in predicting the positive category.

- **Recall**: Recall is the total number of positive predictions divided by the total number of actual positive categories.

  $$\frac{TP}{TP + FN}$$

  In a way, this can be seen as a counterpart to precision. It gives an outward look at the model's performance, which is how the model performed compared to the actual target in terms of predicting the positive category.

- **F_1 Score**: The F_1 score is just a combination of the precision and recall, and is obtained by calculating the harmonic mean of these two metrics. Due to the balanced nature of the f1 score compared to the precision and recall metrics, it can present a more generalized and dependable view of the performance of the model with respect to what is expected of the model in terms of predicting the positive category.

  $$2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

5.5 Experimental Results

We study the prediction performance of CO3D by answering the three aforementioned research questions.

TABLE 5.1: Average Accuracy.

	CO3D			Baselines	
	C_1	C_2	C_3	SVM	BERT
Benchmark	0.790	0.800	**0.847**	0.826	0.813
CoffeeMaker	**1.000**	0.958	0.917	0.778	0.875
JFreeChart060	0.916	**0.921**	0.875	0.876	0.883
JFreeChart071	0.909	**0.910**	0.878	0.875	0.871
JHotDraw741	0.752	0.761	0.787	**0.805**	0.769
All	0.924	0.928	**0.929**	0.928	0.879

TABLE 5.2: Average Recall.

	CO3D			Baselines	
	C_1	C_2	C_3	SVM	BERT
Benchmark	0.848	0.862	0.846	**0.885**	0.823
CoffeeMaker	**1.000**	0.937	**1.000**	0.952	0.833
JFreeChart060	0.970	0.975	0.954	0.944	**1.000**
JFreeChart071	0.972	0.986	**1.000**	0.957	**1.000**
JHotDraw741	0.745	0.749	0.785	0.762	**0.836**
All	**0.968**	0.963	0.928	0.954	0.851

TABLE 5.3: Average Precision.

	CO3D			Baselines	
	C_1	C_2	C_3	SVM	BERT
Benchmark	0.813	0.818	**0.836**	0.830	0.798
CoffeeMaker	**1.000**	**1.000**	0.833	0.724	0.944
JFreeChart060	0.940	**0.941**	0.908	0.917	0.883
JFreeChart071	**0.931**	0.921	0.878	0.908	0.878
JHotDraw741	0.704	0.724	**0.787**	0.774	0.689
All	0.932	**0.941**	0.920	0.935	0.901

TABLE 5.4: Average F_1 score.

	CO3D			Baselines	
	C_1	C_2	C_3	SVM	BERT
Benchmark	0.830	0.839	**0.836**	0.857	0.807
CoffeeMaker	**1.000**	0.966	0.905	0.814	0.867
JFreeChart060	0.955	**0.958**	0.928	0.930	0.938
JFreeChart071	0.951	**0.952**	0.933	0.932	0.933
JHotDraw741	0.724	0.731	**0.785**	0.768	0.742
All	0.950	**0.952**	0.922	0.944	0.868

5.5.1 RQ$_1$: *Is the prediction obtained by C_1 and C_2 comparable to that by C_3?*

By comparing the accuracy scores by C_1 and C_2 with C_3 in the top of Table 5.1, we can see that using word2vec together with Simple RNN and LSTM brings a comparable performance in relation to using CodeBERT. In particular, by C_1 and C_2, CO3D obtains the best accuracy by CoffeeMaker, JFreeChart060, and JFreeChart071; While by C_3, it gets the maximum accuracy with Benchmark and All. With respect to the Precision scores, we witness a similar trend: using simple architectures built on top of word2vec combined with Simple RNN and LSTM allows CO3D to obtain a similar, and somehow better performance compared to employing CodeBERT. Especially, CO3D yields a maximum Precision of 1.000 for both C_1 and C_2 by the CoffeeMaker category in Table 5.2. By the Recall scores, we also see that C_1 and C_2 bring a better performance compared to C_3. This is further confirmed when we consider the F_1 scores as the final evaluation metric, i.e., C_1 and C_2 account for four best scores among six categories, while C_3 does this by the remaining two categories.

Answer to RQ$_1$. Using two simple architectures, i.e., word2vec combined with Simple RNN and LSTM, we obtain a satisfying prediction performance, compared to using CodeBERT–a more complex pre-trained model.

5.5.2 RQ$_2$: *How does* CO3D *compare with the baselines?*

In this research question, we evaluate CO3D by its three configurations, i.e., C_1, C_2, and C_3, against the two baselines including Corazza *et al.* [6] built of top of SVM, and Steiner and Zhang [26] with the BERT classification engine. Following Tables 5.1 - 5.4, it is evident that CO3D outperforms the SVM baseline model by all the four considered metrics. In fact, SVM only gets the maximum scores by Accuracy and Precision with the JHotDraw741 and Benchmark categories, respectively. Apart from that, CO3D obtains superior scores by almost all the remaining rows. Among others, C_2 brings more highest scores for CO3D by different data categories.

Similarly, compared to BERT, CO3D also yields better evaluation scores by almost all the rows in Tables 5.1 - 5.4 In particular, among the total 24 rows, BERT gets maximum Recall scores by three rows, i.e., for JFreeChart060, JFreeChart071, and JHotDraw741. Meanwhile, CO3D yields maximum scores by 19 over the 21 remaining rows across all the three configurations, i.e., C_1, C_2, and C_3.

Answer to RQ$_2$. On the considered dataset, with the three internal configurations, CO3D outperforms both baselines with respect to Accuracy, Precision, Recall, and F_1 score.

5.5.3 RQ$_3$: *Does the type of corpus used for encoding a textual dataset play any role in improving the model performance?*

Comparing the results obtained by the C_3 model to that obtained by the BERT baseline model, we can observe that the C_3 model outperforms the BERT model in all the evaluation metrics considered in the research work. Considering that both models are very similar with a major difference being their corpus, we can make the claim that the difference in corpus does play a significant role in improving a model performance. Since the subject area deals heavily with code related data, more elements in the dataset will be better represented and the internal meaning will be

better captured, which will in turn lead to better pattern recognition and eventual performance by the model.

Answer to RQ$_3$. After considering the result of the C_3 model and the BERT model, the significant difference in performance suggests that there the corpus does play a significant role in determining model performance.

Table 5.1 - 5.8 reports the evaluation metrics where the best values are printed in bold. After training the models on each of the datasets, and then training them on the combined datasets (the rows with "**All**"), we obtained the scores which are then averaged out. We analyze the results by answering the research questions (see Section 5.1) as follows.

5.5.4 Average Performance

Tables 5.1 - 5.4 show the average performance of each model across different combinations of hyper-parameters.

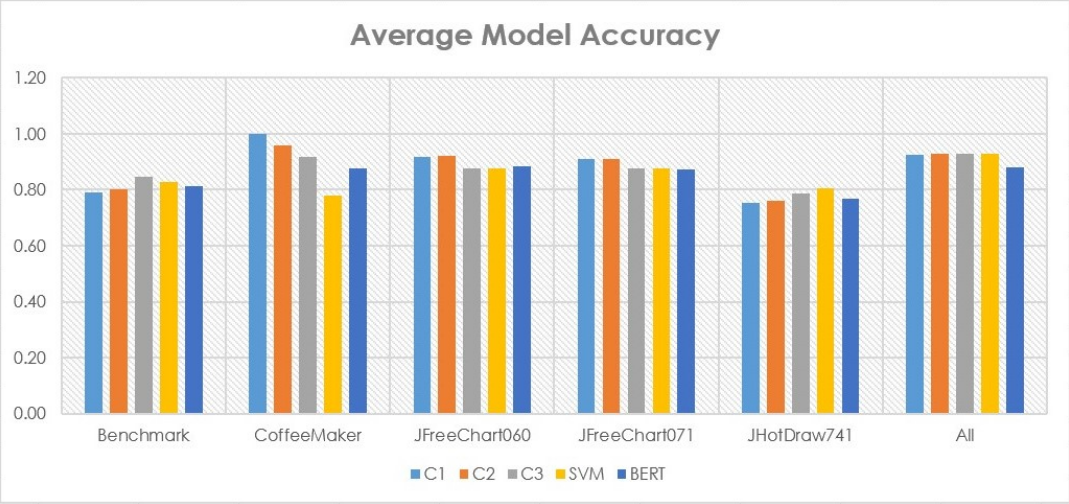

FIGURE 5.7: Average Model Acuracy

5.5.5 Best Performance

Tables 5.5 - 5.8 show the best performance of each model across different combinations of hyper-parameters.

TABLE 5.5: Best Accuracy.

	CO3D			Baselines	
	C_1	C_2	C_3	SVM	BERT
Benchmark	0.804	0.821	**0.860**	0.832	0.831
CoffeeMaker	**1.000**	**1.000**	0.917	0.833	0.917
JFreeChart060	0.924	**0.946**	0.878	0.887	0.883
JFreeChart071	**0.932**	0.924	0.878	0.878	0.878
JHotDraw741	0.773	0.790	0.790	**0.832**	0.769
All	0.936	**0.943**	0.932	0.936	0.879

TABLE 5.6: Best Recall.

	Co3D			Baselines	
	C_1	C_2	C_3	SVM	BERT
Benchmark	0.877	0.903	**0.926**	0.895	0.878
CoffeeMaker	**1.000**	**1.000**	**1.000**	0.952	**1.000**
JFreeChart060	0.988	**1.000**	0.954	0.944	**1.000**
JFreeChart071	**1.000**	**1.000**	**1.000**	0.958	**1.000**
JHotDraw741	0.761	**0.929**	0.789	0.780	0.836
All	0.977	**0.982**	0.953	0.962	0.936

TABLE 5.7: Best Precision.

	Co3D			Baselines	
	C_1	C_2	C_3	SVM	BERT
Benchmark	0.828	0.827	**0.852**	0.838	0.841
CoffeeMaker	**1.000**	**1.000**	0.833	0.857	1.000
JFreeChart060	0.953	**0.953**	0.908	0.951	0.883
JFreeChart071	**0.938**	0.929	0.878	0.919	0.878
JHotDraw741	0.731	0.794	**0.842**	0.810	0.689
All	0.939	**0.955**	0.927	0.948	0.932

TABLE 5.8: Best F_1 score.

	Co3D			Baselines	
	C_1	C_2	C_3	SVM	BERT
Benchmark	0.844	0.858	**0.866**	0.861	0.853
CoffeeMaker	**1.000**	**1.000**	0.909	0.857	0.933
JFreeChart060	0.960	**0.971**	0.928	0.937	0.938
JFreeChart071	0.964	**0.960**	0.933	0.933	0.933
JHotDraw741	0.743	0.761	**0.815**	0.795	0.742
All	0.958	**0.962**	0.936	0.952	0.897

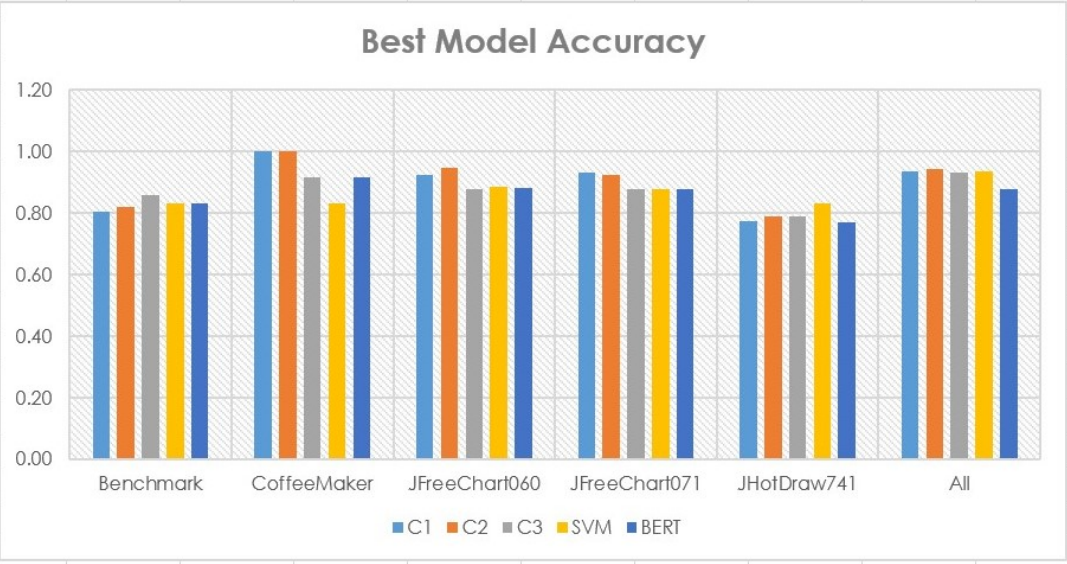

FIGURE 5.8: Best Model Accuracy

5.5.6 Text Embedding Time

The tables below show the conversion time (in minutes) for each of the text embedding methods used by each of the classification engines.

TABLE 5.9: Text Embedding Time (in minutes).

	Co3D		Baselines	
	Word2Vec	CodeBERT	BERT	TF-IDF
Benchmark	50	**0.017**	0.167	0.133
CoffeeMaker	0.80	**0.000**	0.001	0.001
JFreeChart060	7	**0.005**	0.017	0.006
JFreeChart071	9	**0.006**	0.031	0.007
JHotDraw741	32	**0.020**	0.100	0.060
All	98	**0.067**	0.333	0.250

TABLE 5.10: Dataset Size.

	Number of Rows
Benchmark	2,881
CoffeeMaker	47
JFreeChart060	461
JFreeChart071	588
JHotDraw741	1,785
All	5,762

5.5.7 Conversion Time Evaluation

The numbers shown in Table 5.9 shed some light on the real-time applicability of the methods explored in this research paper. These recorded times show the duration it takes for the embedding method used by each of the classification engines to convert the raw text into vector format before training. From the record, we can observe that the tokenizer used by CodeBERT has the fastest conversion time amongst all the tokenizers. The TF-IDF embedding schema used by the SVM baseline was the second fastest, closely followed by the BERT Tokenizer. The Word2Vec embedding method turned out to have the worst conversion time, and far outweighs the conversion of the rest of the methods explored (even when combined). This could be dependent on the nature and size of the embedding model used in the Word2Vec method, as well as the fact that each individual word in each code-comment pair is searched and converted separately as opposed to the sentence embedding used in the likes of BERT and CodeBERT. The TD-IDF embedding method is also relatively simple compared to the Word2Vec method.

Considering these record, as well as the model performance gotten from the various classification engines explored in this research, the CodeBERT can be considered the best model in terms of both performance and execution time, despite the fact that the SimpleRNN and LSTM were able to match the CodeBERT method in performance. If these two methods were to be employed in a real-time situation, some modification might be needed to mitigate this significantly longer conversion time. Some of these methods could involve the integration of distributed processing or the use of an entirely different embedding method along with the Simple RNN and LSTM method. These possible improvements are discussed more in-depth in Chapter 6.

5.6 Model Interpretation

Considering the further integration of AI into much of our daily activities and online interactions, there is a rising need to be able to explain these models for ethical and legal reasons, as well as for general acceptability. The public is more likely to accept these integrations when they can look below the hood to have an idea of what principles drive these models [29]. It can also serve as a means to confirm the underlying theories that lead up to selection of these models and how they are expected to work.

For this purpose, the *transformers_interpret* library was used to parse a sample code-comment block along with the trained CodeBERT model and the importance of each token towards the final predicted class of the sample was visualized.

```
In [16]:
sample_str1 = list(codes6['Code and Comment'][14])[0]
sample_str1

Out[16]:
'/**    * returns the inventory of the coffee maker    *  inventory    */ 2   public synch
ronized string checkinventory() {     return inventory.tostring();'
```

FIGURE 5.9: A selected sample code-comment pair used for the sake of model interpretation

```
In [17]:
word_attributions1 = cls_explainer(sample_str1)
#word_attributions

cls_explainer.predicted_class_name

# The numerical word attributions can be viewed visually using the visualize method.
cls_explainer.visualize()
```

Legend: Negative Neutral Positive

True Label	Predicted Label	Attribution Label	Attribution Score	Word Importance
0	LABEL_0 (0.99)	LABEL_0	0.95	[CLS] / * * * returns the inventory of the coffee maker * inventory * / 2 public synchronized string check ##in ##vent ##ory ( ) { return inventory . to ##st ##ring ( ) ; [SEP]

FIGURE 5.10: A visualization of the importance of each word in the final predicted class.

To give a high-level explanation of the visualization in Figure 5.10, first we start by explaining the significance of the colour coding used by the library to render each text in the sample code-comment pair. Similar to how colour codes are used in a statistical correlation heat-map, the green colour indicates positive direction and the red colour indicates a negative direction, while the intensity of the colour used to render a particular token shows the level of impact that particular token has in the positive or negative direction. This can also be likened to the hypothesis check conducted on coefficients of a given regression model to obtain their p-value and conclude whether they can be considered statistically significant or not. Simply put, this can be seen as the positive or negative relationship between the input and the output, and the importance of this relationship.

Taking the above explanation into consideration and observing the results of the visualization, we can see the words like *return*, *the*, and *maker* having the most positive relationship, while the word *inventory* has the most negative relationship, but

all come together to aid the model in deciding the given category for this particular sample. Also as expected of the intent behind the methods used in this research work (accounting for semantic meaning and sequence order), we can observe how similar words appearing later down the line helped in correctly classifying this sample. We can see this in how the word *return* reoccurs, having positive relations both times. This same effect occurs with the word *inventory*. When this input was being passed through the model, the model was encouraged to pay more attention to these words, due to their recurrence in both the comment and the code. This also makes sense when judging this from a programmer's point of view, as these set of words actually help show that the code-comment pair is coherent. Regardless of the fact the the middle part of the sample containing "*public synchronized string*" is still very important in the functionality of the code, the words prioritized by the model give the sample code-comment pair some context in terms of checking what the comment stated the code will do, and what the code is eventually doing.

5.7 Implications

The outcomes of our empirical evaluation reveal an interesting finding: Even with simple machine learning (ML) architectures, we can obtain a satisfying performance compared to that by pre-trained models, i.e., BERT and CodeBERT. This is *perversely counter-intuitive*, as we expect that these models–being trained on large corpora of source code and text–should have been the best classifier, surprisingly they are not. This is important in practice as training on complex pre-trained models requires time and computational resources, and thus using lightweight models while still preserving a comparable accuracy is highly beneficial. Apart from technical consequences, this has an obvious social impact, as we can avoid spending too much energy (e.g., electricity) to train complex deep learning models, instead we just employ a simple, yet efficient and effective ML technique.

One of the main differences between BERT and CodeBERT is the corpora for encoding and training. The corpus for CodeBERT is more suitable for input text involving code and programming languages [8], implying that some tokenized words might have a better representation after being encoded, which will in turn lead to better pattern recognition during training or fine tuning. Through the experiments, we see that although BERT and CodeBERT have been successful in various tasks [1, 4, 12], they are not a *silver bullet*, i.e., a solution to every problem. Altogether, this calls for fundamentally new directions of future research, where we need to empirically select a model that fits a specific problem, rather than picking a model for any probable Software Engineering issue, just because the model is either '*advanced*' or '*deep*.'

5.8 Threats to validity

- **Internal validity:** The threat is related to both data quantity [10] and quality, and this can be seen from the overall model performance on the JHotDraw741 dataset, which is with the lowest Kappa index, i.e., below 0.70 [5] across all the experimented models.

  Another notable threat is the amount of information discarded during the text encoding section of the research. Each code-comment pair was truncated or padded to the first 50 vectors corresponding to the first 50 word/symbols from

each code-comment pair (for the Simple RNN and LSTM pairs). This can be a loss of information considering that the longest code-comment pair had a word/symbol length above 500. We anticipate that with a more distributed and parallel approach to this encoding process, the models might produce better predictions than they already did.

- **External validity:** This concerns how generalized our proposed approach is. Only the Java programming language was considered for the evaluation. Therefore, we do not know if the results obtained are specific to Java, or if it will be applicable to other languages. As we have already seen from this paper, the corpus (which can be language specific) does seem to play a role in model performance, hence can also be considered a threat to the work. Lastly, a more generic threat to this paper is the size of the dataset, i.e., more data is supposed to bring better training and inference [10].

Chapter 6

Conclusion and Future Work

In this paper, we performed an investigation into the contributing factor of the NLP implementation in code-comment coherence prediction. The paper sought to answer three crucial research questions stated in the methodology section, the most important of which was whether the internal meaning of words and the sequential order of words in text played any role in the model performance for code-comment coherence check. Three models were trained (all paying attention to these two factors) and some included transfer learning where large pre-trained models were fine-tuned on the available dataset, and the results were compared to some baseline models from the original paper addressing this subject matter [6]. We conclude that even simple architectures are both effective and efficient compared to complex pre-trained models. All results obtained from the experiment points to the confirmation of this suspicion, that these two factors did indeed play a role in improving model performance. Additionally it was observed that size of model didn't seem to matter after these two factors have been successfully accounted for. Finally it was also observed that the nature of the corpus used for the encoding process also seemed to play a significant role in predicting code-comment coherence. This research aims at improving the collaboration amongst developers and overall improving the quality of software related practices and products used globally.

A lot of the further studies that can be conducted to further improve the methods used in paper comes from accounting for the inadequacies of the methods used to predict code comment coherence. We discuss some of these possible improvements in the following sections.

6.1 FastText Embedding

This consideration is due to the existing limitation of the Word2Vec Embedding method used in combination with the Simple RNN and the LSTM methods. Although this method proves to be able to capture more semantic meaning in individual words than other methods seen in many of the works in the related works, it is still very much limited to the exact words in whichever corpus it was trained. To simplify this, if a particular word wasn't included in the initial training of the Word2Vec method, it doesn't bring up any result when searched. Therefore the code which utilizes the embedding model was written in such a way that it skips words that aren't present in the model's corpus to avoid break in continuity. However, the downside to this is that some of these words not found in the embedding model's corpus could hold important information that could positively impact the classification engine's performance.

This is where the FastText embedding method comes into play. The FastText embedding model is trained not only on the word in its given corpus but also their respective n-grams. For a given word "*Town*", its bi-gram or n-gram (with n = 2)

would comprise of "*To*", "*ow*" and "*wn*". Simply put, for a chosen n-gram size of n equal to a chosen number, each word is split using this n-size and each sub-word is then included in the training as well [2]. This method helps in accounting for words not explicitly included in the original training. When the embedding model comes across a word not explicitly included in its original corpus, it splits this word by n-gram and finds the vector representation of its n-gram instead and combines these vectors to get the final representation of the word. This representation might not capture accurately the semantic of this new foreign word as it would other words included in its original corpus and could also result in false similarities amongst words (foreign or not). However, the word isn't omitted entirely, and if this word is significant in classifying a given input and happens to be used in many other inputs, this results in not losing its information entirely by omission.

While this is good for capturing extra information from the input dataset, this embedding method takes a considerably longer time to train due to all the extra sub-words now being considered due to n-gram inclusion. Hence, this all comes down to a matter of trade and balance.

6.2 Bi-directional LSTM

This possible improvement, although focused on the LSTM model, can also be applied to the Simple RNN model. The LSTM model was focused on due to the fact that it outperformed the Simple RNN model given the metric considered in this research work. The LSTM methods as explained in Section 2.3 has an advantage on the Simple RNN models due to the fact that it uses what is referred to as the forget, input and output gate to filter the information carried across each time step to decide what to retain and what to discard. This produces a long term and short term memory effect, with the long term memory applied on relevant information and the short term memory applied on irrelevant information.

While this long and short term memory effect is a significant improvement in terms of information retention in recurrent neural networks, it is still limited to one directional attention. The traditional recurrent neural networks are only capable of paying attention to the past, in the sense that to understand the context of a word at a certain time step in the sentence, the model only considers the words that came before it. However, in many cases the word in a future time step could influence the meaning of a word at the current time step. This phenomenon isn't accounted for by the traditional versions of recurrent neural networks, and this is where the Bi-directional recurrent models (LSTM) introduced by Graves and Schmidhube [9].

The Bi-directional LSTM model as earlier explained processes the input data in both directions, with one layer processing the forward sequence of words in the sentence, and another layer processing the backward sequence of the words in the sentence. This captures both relevant future and past relationships amongst the words in the sentence, which could lead to a much richer representation of information in the input data. Quite similar to how the FastText extends the capability of the Word2Vec Embedding method while still being bound by some of the limitations of the Word2Vec method, the Bi-directional LSTM is also bound by the memory limitations of the regular LSTM even though it does extend the capability of the LSTM model in terms of context representation. Considering the fact that it is built to represent twice the direction represented by the regular LSTM it is only natural that the training time and model complexity will be more, which will subsequently reflect

on the cost of training. Just as in the case of the FastText vs Word2Vec, this would also be a case of trade and balance.

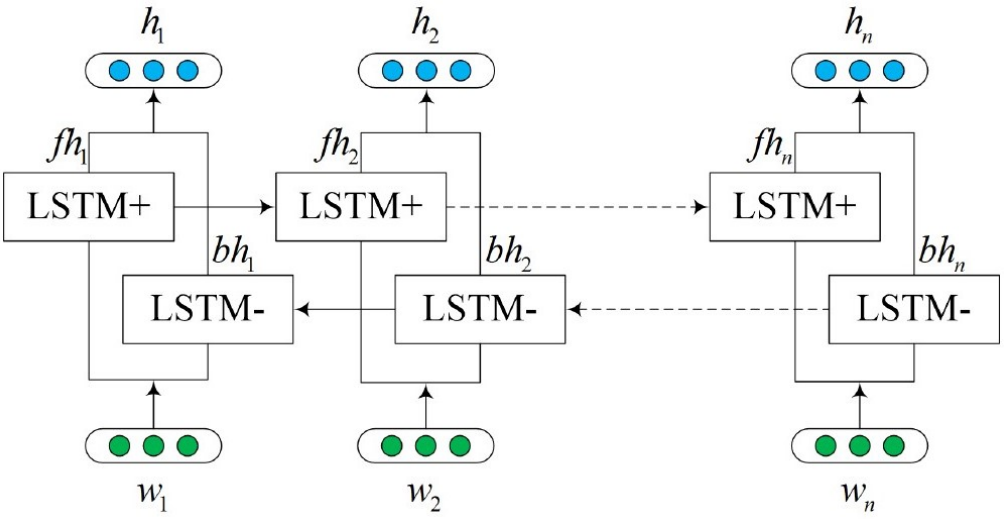

FIGURE 6.1: Architecture of a basic Bi-LSTM [30].

6.3 Faster Word Embedding Options

As much as the performance of the models explored in this research outperforms the baselines, one area which these models fall short in is the data preparation phase, which involves the conversion of the raw input text into their vector representations. The conversion time required by the Word2Vec model far outweighs the time required by the CodeBERT, BERT and the TF-IDF encoding methods. This is disadvantageous for the SimpleRNN and LSTM model when combined with the Word2Vec embedding methods if applied in a time-sensitive scenario. In order to solve this or at least improve this condition, two possible solution can be explored;

- **Distributed Processing**: Considering the nature of the embedding process, each word in a given raw text is parsed separately in a sequential order and the separate vector representation of each of these words are then reconstituted (with possible padding or truncation in order to maintain uniform input length) to form the final vector representation of one given code-comment input pair. This is a process that can be executed in a distributed manner using methods like map-reduce. In this way, multiple nodes (computer units) parse through the text rather than one single unit which could significantly reduce the transformation time in a real and time sensitive setting.

- **Exploring other Embedding Methods**: In this research, it was observed that the embedding methods used for the input to the CodeBERT model was much faster in parsing through the entire input dataset than the method used for input to the Simple RNN and the LSTM models. A possible consideration could be using the embedding method of the CodeBERT models (or any other equally fast and quality embedding method) along with the Simple RNN and LSTM methods. However, it is worth noting that the embedding method for the CodeBERT differs from the Word2Vec in the sense that, the former encodes both the semantic and sequential order of the words in the text together, while

the Word2Vec only focuses on the semantic information in each word in a given text, while the Simple RNN and LSTM models focus on the sequential order of the words. Another important note is that due to the fact that the Word2Vec as used in this research focuses on one word at a time, it tends to represent a lot more information with regards to each word than the CodeBERT embedding method, which could also have an effect on the accuracy of the Simple RNN and LSTM models if changed.

Bibliography

[1] Himanshu Batra et al. "BERT-Based Sentiment Analysis: A Software Engineering Perspective". In: *Database and Expert Systems Applications - 32nd International Conference, DEXA 2021, Virtual Event, September 27-30, 2021, Proceedings, Part I*. 2021, pp. 138–148. DOI: 10.1007/978-3-030-86472-9_13. URL: https://doi.org/10.1007/978-3-030-86472-9_13.

[2] Piotr Bojanowski et al. "Enriching Word Vectors with Subword Information". In: *Transactions of the Association for Computational Linguistics* 5 (July 2016). DOI: 10.1162/tacl_a_00051.

[3] Alfonso Cimasa et al. "Word Embeddings for Comment Coherence". In: *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2019, pp. 244–251. DOI: 10.1109/SEAA.2019.00046.

[4] Giuseppe Colavito, Filippo Lanubile, and Nicole Novielli. "Issue Report Classification Using Pre-trained Language Models". In: *2022 IEEE/ACM 1st International Workshop on Natural Language-Based Software Engineering (NLBSE)*. 2022, pp. 29–32. DOI: 10.1145/3528588.3528659.

[5] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. "A New Dataset for Source Code Comment Coherence". In: *Proceedings of Third Italian Conference on Computational Linguistics (CLiC-it 2016) & Fifth Evaluation Campaign of Natural Language Processing and Speech Tools for Italian. Final Workshop (EVALITA 2016), Napoli, Italy, December 5-7, 2016*. Ed. by Pierpaolo Basile et al. Vol. 1749. CEUR Workshop Proceedings. CEUR-WS.org, 2016. URL: https://ceur-ws.org/Vol-1749/paper17.pdf.

[6] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. "Coherence of comments and method implementations: a dataset and an empirical investigation". In: *Softw. Qual. J.* 26.2 (2018), pp. 751–777. DOI: 10.1007/s11219-016-9347-1. URL: https://doi.org/10.1007/s11219-016-9347-1.

[7] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. Ed. by Jill Burstein, Christy Doran, and Thamar Solorio. Association for Computational Linguistics, 2019, pp. 4171–4186. DOI: 10.18653/v1/n19-1423. URL: https://doi.org/10.18653/v1/n19-1423.

[8] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL].

[9] A. Graves and J. Schmidhuber. "Framewise phoneme classification with bidirectional LSTM networks". In: vol. 4. Jan. 2005, 2047 –2052 vol. 4. ISBN: 0-7803-9048-2. DOI: 10.1109/IJCNN.2005.1556215.

[10] Alon Halevy, Peter Norvig, and Fernando Pereira. "The Unreasonable Effectiveness of Data". In: *IEEE Intelligent Systems* 24.2 (2009), pp. 8–12. DOI: 10.1109/MIS.2009.36.

[11] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[12] Md. Jahidul Islam, Ratri Datta, and Anindya Iqbal. "Actual rating calculation of the zoom cloud meetings app using user reviews on google play store with sentiment annotation of BERT and hybridization of RNN and LSTM". In: *Expert Systems with Applications* 223 (2023), p. 119919. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2023.119919. URL: https://www.sciencedirect.com/science/article/pii/S0957417423004207.

[13] Maite Jaca-Madariaga et al. "Sentiment Analysis Model Using Word2vec, Bi-LSTM and Attention Mechanism". In: Mar. 2023. DOI: 10.1007/978-3-031-27915-7_43.

[14] Michael I. Jordan. "Chapter 25 - Serial Order: A Parallel Distributed Processing Approach". In: *Neural-Network Models of Cognition*. Ed. by John W. Donahoe and Vivian Packard Dorsel. Vol. 121. Advances in Psychology. North-Holland, 1997, pp. 471–495. DOI: https://doi.org/10.1016/S0166-4115(97)80111-2. URL: https://www.sciencedirect.com/science/article/pii/S0166411597801112.

[15] Bofang Li et al. "Scaling Word2Vec on Big Corpus". In: *Data Science and Engineering* 4 (June 2019). DOI: 10.1007/s41019-019-0096-6.

[16] Everton da S. Maldonado and Emad Shihab. "Detecting and quantifying different types of self-admitted technical Debt". In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. 2015, pp. 9–15. DOI: 10.1109/MTD.2015.7332619.

[17] Abhishek Mallik and Sanjay Kumar. "Word2Vec and LSTM based deep learning technique for context-free fake news detection". In: *Multimedia Tools and Applications* (May 2023). DOI: 10.1007/s11042-023-15364-3.

[18] Rafael Meneses Santos et al. "Long Term-short Memory Neural Networks and Word2vec for Self-admitted Technical Debt Detection". In: May 2020, pp. 157–165.

[19] Tomás Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2013. URL: http://arxiv.org/abs/1301.3781.

[20] Putra Muhammad, Retno Kusumaningrum, and Adi Wibowo. "Sentiment Analysis Using Word2vec And Long Short-Term Memory (LSTM) For Indonesian Hotel Reviews". In: *Procedia Computer Science* 179 (Jan. 2021), pp. 728–735. DOI: 10.1016/j.procs.2021.01.061.

[21] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015.

[22] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. "Classifying code comments in Java software systems". In: *Empir. Softw. Eng.* 24.3 (2019), pp. 1499–1537. DOI: 10.1007/s10664-019-09694-w. URL: https://doi.org/10.1007/s10664-019-09694-w.

[23] Anthony Peruma et al. "Refactoring Debt: Myth or Reality? An Exploratory Study on the Relationship Between Technical Debt and Refactoring". In: *19th IEEE/ACM International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022*. ACM, 2022, pp. 127–131. URL: https://ieeexplore.ieee.org/document/9796264.

[24] Fazle Rabbi and Md. Saeed Siddik. "Detecting Code Comment Inconsistency using Siamese Recurrent Network". In: *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 2020, pp. 371–375. DOI: 10.1145/3387904.3389286. URL: https://doi.org/10.1145/3387904.3389286.

[25] Fazle Rabbi et al. "An Ensemble Approach to Detect Code Comment Inconsistencies using Topic Modeling". In: *The 32nd International Conference on Software Engineering and Knowledge Engineering, SEKE 2020, KSIR Virtual Conference Center, USA, July 9-19, 2020*. Ed. by Raúl García-Castro. KSI Research Inc., 2020, pp. 392–395. DOI: 10.18293/SEKE2020-062. URL: https://doi.org/10.18293/SEKE2020-062.

[26] Theo Steiner and Rui Zhang. "Code Comment Inconsistency Detection with BERT and Longformer". In: *CoRR* abs/2207.14444 (2022). DOI: 10.48550/arXiv.2207.14444. arXiv: 2207.14444. URL: https://doi.org/10.48550/arXiv.2207.14444.

[27] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[28] Lili Wei, Yepang Liu, and Shing-Chi Cheung. "Taming Android fragmentation: characterizing and detecting compatibility issues for Android apps". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. Ed. by David Lo, Sven Apel, and Sarfraz Khurshid. ACM, 2016, pp. 226–237. DOI: 10.1145/2970276.2970312. URL: https://doi.org/10.1145/2970276.2970312.

[29] Sandareka Wickramanayake, Wynne Hsu, and Mong Lee. "FLEX: Faithful Linguistic Explanations for Neural Net based Model Decisions". In: Feb. 2019.

[30] Jun Xie et al. "Self-Attention-Based BiLSTM Model for Short Text Fine-grained Sentiment Classification". In: *IEEE Access* 7 (Dec. 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2957510.