

Deep code comment generation with Neural Network

Alok Kumar Sharma
Sridhar Chimalakonda
Venigalla Akhila Sri Manasa
cs20m001@iittp.ac.in
ch@iittp.ac.in
cs19d504@iittp.ac.in

ABSTRACT

Software development depends on good source code documentation to understand existing source code to perform various tasks such as fixing bugs and implementing new features. Manual documentation by developers is often missing or outdated. Past research has suggested automatic code summarization tools to remedy this problem. While several works on code summarization have been done to generate code summaries but very little work exists on using other information sources. The novelty of our work is that we conduct an empirical study on code summarization using deep learning techniques.

1. INTRODUCTION

During software development and maintenance developers spend around 59% of their time on program comprehension activities previous studies have shown that good comments are important to program comprehension since developers can understand the meaning of the piece of code by reading the natural language description of the comments, unfortunately due to the tight project schedule and other reasons code

comments are often misunderstood. Automatic generation of good comments can not only save developers time in writing comments but also help in source code understanding.

Many approaches have been proposed to generate comments for methods in past years. Although there are many such techniques which is used in past but there are two main limitation first, they fail to extract accurate keyword using the identified similar code snippets when identifiers and methods are poorly named Secondly, they rely on whether similar code snippet can be retrieved and how similar the snippets are.

Paper Organization The remainder of this paper is organized as follows.

Section 2 Describe the design decisions taken to create the dataset. How the dataset is collected.

Section 3 Explains the Research Questions, and explains each research question.

Section 4 Explains the results of each research question.

Section 5 Explains the limitations about the study and about technical challenges related to experiments, dataset creation and modeling.

Section 6 Related research papers similar to your study.

Section 7 Suggestions on how empirical study can be extended.

2. DATASET:

Data Scraping:

Download all the repositories which have a Python file and then check all the directories for the Python file with extension “.Py” in the repository folder which was downloaded .

Once able to get the Python file. We read those files line by line to find a method. Once we encounter it then we save the method and the associated comment of that method.

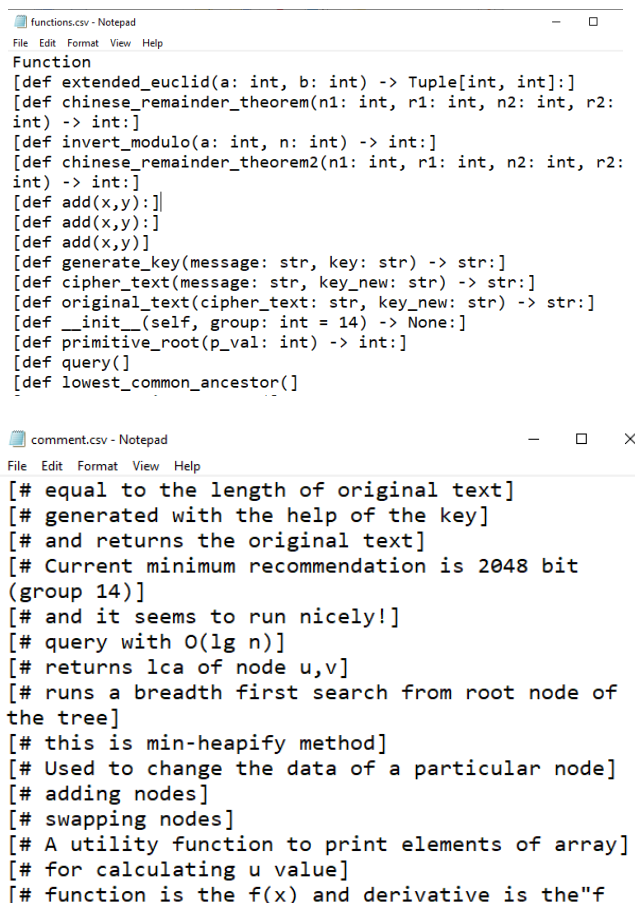
Now we are able to generate a CSV file in the format as:

```
[method]:[ comment], [method]:[ comment], [method]:[ comment]
```

In dictionary format.

Preprocessing:

Now we need to do some preprocessing before we convert into vectors. We need to read comments and methods separately as we are using supervised learning where X is given and we need to predict Y. So X in our model are methods and Y are comments. So to separate X&Y in our file we manually divided our file into two halves. the first half is converted into a function.CSV file. and the other half is converted into Comments.CSV. Now as we have X&Y in two separate files we need to go for vectorization for our data so that we can use some ML method for processing our data.



The image shows two Notepad windows. The top window, titled 'functions.csv - Notepad', contains Python code defining various cryptographic functions. The bottom window, titled 'comment.csv - Notepad', contains a list of comments explaining the code in the top window.

```
functions.csv - Notepad
File Edit Format View Help
Function
[def extended_euclid(a: int, b: int) -> Tuple[int, int]:]
[def chinese_remainder_theorem(n1: int, r1: int, n2: int, r2:
int) -> int:]
[def invert_modulo(a: int, n: int) -> int:]
[def chinese_remainder_theorem2(n1: int, r1: int, n2: int, r2:
int) -> int:]
[def add(x,y):]
[def add(x,y):]
[def add(x,y)]
[def generate_key(message: str, key: str) -> str:]
[def cipher_text(message: str, key_new: str) -> str:]
[def original_text(cipher_text: str, key_new: str) -> str:]
[def __init__(self, group: int = 14) -> None:]
[def primitive_root(p_val: int) -> int:]
[def query()]
[def lowest_common_ancestor()]

comment.csv - Notepad
File Edit Format View Help
[# equal to the length of original text]
[# generated with the help of the key]
[# and returns the original text]
[# Current minimum recommendation is 2048 bit
(group 14)]
[# and it seems to run nicely!]
[# query with O(lg n)]
[# returns lca of node u,v]
[# runs a breadth first search from root node of
the tree]
[# this is min-heapify method]
[# Used to change the data of a particular node]
[# adding nodes]
[# swapping nodes]
[# A utility function to print elements of array]
[# for calculating u value]
[# function is the f(x) and derivative is the"f
```

Cleaning the data using nltk:

The Natural Language Toolkit (**NLTK**) is a platform used for building Python programs that work with human language data for applying in statistical natural language processing. It contains text processing libraries for tokenization, parsing, classification, stemming, tagging and semantic reasoning. We have used NLTK for Cleaning our data.

Clean text often means a list of words or tokens that we can work with in our machine learning models. This means converting the raw text into a list of words and saving it again. There are different processes involved in it.

Firstly, we splitted the comments. A very simple way to do this would be to split the document by white space, including " ", new lines, tabs and more.

```
single_tokenized_lowered = list(map(str.lower,word_tokenize(data2)))
print(single_tokenized_lowered)
```

```
['comment', '[', '#', 'extended', 'euclid', ']', '[', '#', 'uses', 'extendedeuclid', 'to', 'find', 'inverses', ']', '[', '#', 'this', 'function', 'find', 'the', 'inverses', 'of', 'a', 'i.e.', ',', 'a^', '(', '-1', ')', ']', '[', '#', 'same', 'a', 'above', 'using', 'invertingmodulo', ']', '[', '#', 'add', 'function', ']', '[', '#', 'add', 'function', ']', '[', '#', 'add', 'function', ']', '[', '#', 'equal', 'to', 'the', 'length', 'of', 'original', 'text', ']', '[', '#', 'generated', 'with', 'the', 'help', 'of', 'the', 'key', ']', '[', '#', 'and', 'returns', 'the', 'original', 'text', ']', '[', '#', 'current', 'minimum', 'recommendation', 'is', '2048', 'bit', '(', 'group', '14', ')', ']', '[', '#', 'and', 'it', 'seems', 'to', 'run', 'nicely', '!', ']', '[', '#', 'query', 'with', 'o', '(', 'lg', 'n', ')', ']', '[', '#', 'returns', 'lca', 'of', 'node', 'u', ',', 'v', ']', '[', '#', 'runs', 'a', 'breadth', 'first', 'search', 'from', 'root', 'node', 'of', 'the', 'tree', ']', '[', '#', 'this', 'is', 'min-
```

Secondly, We have filtered out all tokens that we are not interested in, such as all standalone punctuation like !"#%&'()*+,-./:;<=>?@[\\]^_`{|}~. Then we filtered out stop words. Stop words are those words that do not contribute to the deeper meaning of the phrase. They are the most common words such as: “the”, “a”, and “is”.

```
# output after removing punctuation and stop words
print([word for word in single_tokenized_lowered if word not in stopwords_en_withpunct])
```

```
['comment', 'extended', 'euclid', 'uses', 'extendedeuclid', 'find', 'inverses', 'function', 'find', 'inverses', 'i.e.', 'a^', '-1', 'using', 'invertingmodulo', 'add', 'function', 'add', 'function', 'add', 'function', 'equal', 'length', 'original', 'text', 'generated', 'help', 'key', 'returns', 'original', 'text', 'current', 'minimum', 'recommendation', '2048', 'bit', 'group', '14', 'seems', 'run', 'nicely', 'query', 'lg', 'n', 'returns', 'lca', 'node', 'u', 'v', 'runs', 'breadth', 'first', 'search', 'root', 'node', 'tree', 'min-heapify', 'method', 'used', 'change', 'data', 'particular', 'node', 'adding', 'nodes', 'swapping', 'nodes', 'utility', 'function', 'print', 'elements', 'array', 'calculating', 'u', 'value', 'function', 'f', 'x', 'derivative', 'f', 'x', 'test', 'convert_to_negative', 'test', 'change_contrast', 'canny.gen_gaussian_kernel', 'canny.py', 'filter_gaussian_filter.py', 'returns', 'f', 'n', 'f', 'n-1', 'print', 'order', 'matrix', 'ai', 'matrix', 'random', 'population', 'created', 'time', 'evaluate', 'select', 'crossover', 'mutate', 'new', 'population', 'finding', 'articulation', 'points', 'undirected', 'graph', 'function', 'search', 'path', 'find', 'isolated', 'node', 'graph', 'vertices', 'set', 'vertices', 'set', 'printing', 'graph', 'vertices', 'adding', 'edge', 'two', 'vertices', 'based', 'min', 'heap', 'sample', 'depth', 'first', 'search', 'used', 'max_flow', 'calculate', 'flow', 'reaches', 'sink', 'handles', 'repetition', 'handles', 'input', 'exist', 'destination', 'meant', 'default', 'value', '-1', 'count', 'random', '10', '10000', 'handles', 'repetition', 'handles', 'input',
```

Thirdly, Some comments and functions may benefit from stemming in order to both reduce the vocabulary and to focus on the sense or sentiment of a document rather than deeper meaning. And after the stemming we have done word tokenization.

```
# output after using Stemmer, Lemmatizer and word tokenization.
stop_words=set(stopwords.words('english'))
tokenized=sent_tokenize(data2)
for i in tokenized:
    wordslist=nlk.word_tokenize(i)
    workslist=[w for w in wordslist if not w in stop_words]
    tagged = nltk.pos_tag(workslist)
    print(tagged)
S , VBZ , ( 2648 , CD ) , ( 010 , NN ) , ( ( , ( , ( group , NN ) , ( 14 , CD ) , ( ) , ( ) , ( ) , ( FW ) , ( [ , JJ ) ,
( '# , #' ) , ( 'and' , CC ) , ( 'it' , PRP ) , ( 'seems' , VBZ ) , ( 'to' , TO ) , ( 'run' , VB ) , ( 'nicely' , RB ) , ( '!' , . ) ]
[ ( ']' , JJ ) , ( ( ']' , NNP ) , ( '# , #' ) , ( 'query' , NN ) , ( 'with' , IN ) , ( 'o' , NNP ) , ( ( ' ( ' , 'lg' , JJ ) , ( 'n' , NN ) ,
( ' ) , ' ) ) , ( ( ']' , VBZ ) , ( ( ']' , JJ ) , ( '# , #' ) , ( 'returns' , NNS ) , ( 'lca' , NN ) , ( 'of' , IN ) , ( 'node' , JJ ) , ( 'u' , N
N ) , ( ( ' , ' , ' ) , ( 'v' , FW ) , ( ( ']' , NNP ) , ( ( ']' , NNP ) , ( '# , #' ) , ( 'runs' , VBZ ) , ( 'a' , DT ) , ( 'breadth' , NN ) , ( 'firs
t' , RB ) , ( 'search' , NN ) , ( 'from' , IN ) , ( 'root' , JJ ) , ( 'node' , NN ) , ( 'of' , IN ) , ( 'the' , DT ) , ( 'tree' , NN ) ,
( ']' , NNP ) , ( ( ']' , NNP ) , ( '# , #' ) , ( 'this' , DT ) , ( 'is' , VBZ ) , ( 'min-heapify' , JJ ) , ( 'method' , NN ) , ( ( ']' , NNP ) ,
( ( ']' , NNP ) , ( '# , #' ) , ( 'Used' , VBN ) , ( 'to' , TO ) , ( 'change' , VB ) , ( 'the' , DT ) , ( 'data' , NN ) , ( 'of' , IN ) , ( 'a' ,
DT ) , ( 'particular' , JJ ) , ( 'node' , NN ) , ( ( ']' , NNP ) , ( ( ']' , NNP ) , ( '# , #' ) , ( 'adding' , VBG ) , ( 'nodes' , NNS ) ,
( ']' , NNP ) , ( ( ']' , NNP ) , ( '# , #' ) , ( 'swapping' , VBG ) , ( 'nodes' , NNS ) , ( ( ']' , NNP ) , ( ( ']' , NNP ) , ( '# , #' ) , ( 'A' ,
NNP ) , ( 'utility' , NN ) , ( 'function' , NN ) , ( 'to' , TO ) , ( 'print' , VB ) , ( 'elements' , NNS ) , ( 'of' , IN ) , ( 'array' , N
N ) , ( ( ']' , NNP ) , ( ( ']' , NNP ) , ( '# , #' ) , ( 'for' , IN ) , ( 'calculating' , VBG ) , ( 'u' , JJ ) , ( 'value' , NN ) , ( ( ']' , NN
P ) , ( ( ']' , NNP ) , ( '# , #' ) , ( 'function' , NN ) , ( 'is' , VBZ ) , ( 'the' , DT ) , ( 'f' , NN ) , ( ( ' ( ' , 'x' , NNP ) , ( ' ) ,
' ) ) , ( 'and' , CC ) , ( 'derivative' , JJ ) , ( 'is' , VBZ ) , ( 'the' , DT ) , ( 'f' , NN ) , ( ( ' ( ' , 'x' , NNP ) , ( ' ) ,
( ' ) , ' ) ) , ( ' ' ' ' , ' ' ' ' ) , ( ( ']' , $ ) , ( ( ']' , NNP ) , ( '# , #' ) , ( 'Test' , NNP ) , ( ( ' : ' , ' : ' ) , ( 'convert_to_negative' , NN ) ,
( ' ) , ' ) ) , ( ' ' ' ' , ' ' ' ' ) , ( ( ']' , $ ) , ( ( ']' , NNP ) , ( '# , #' ) , ( 'Test' , NNP ) , ( ( ' : ' , ' : ' ) , ( 'convert_to_negative' , NN ) ,
```

This process helps us to extract the meaning of the words associated with a comment or method which will help us to get more accuracy when we work on ML models.

3. EXPERIMENTS IN THE EMPIRICAL STUDY

In this section, we evaluate different approaches by measuring their accuracy on generating python ‘methods’ and ‘comments’. Specifically, we mainly focus on the following research questions:

RQ1: What is the impact of source code and comments with different lengths, on the performance of Deep Neural Network?

RQ2: what are the different challenges while vectorization of source code and comments .

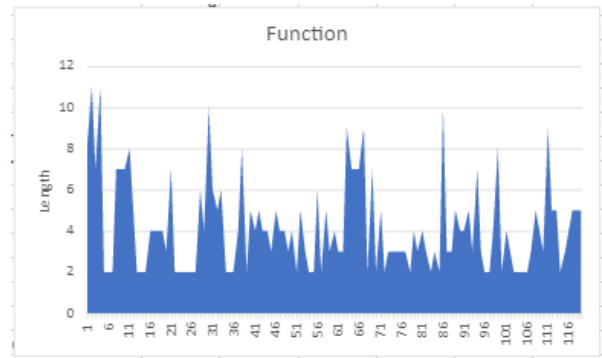
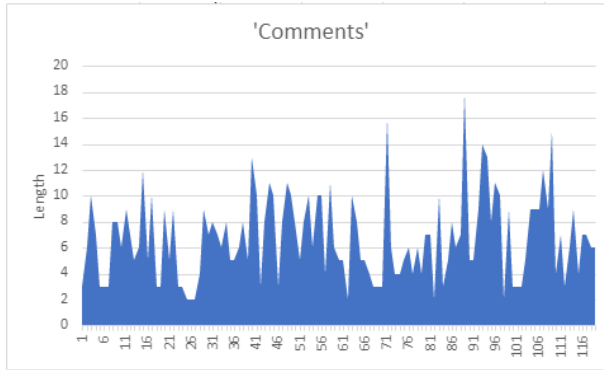
RQ3: How effective is the Deep Neural Network?

4. RESULTS

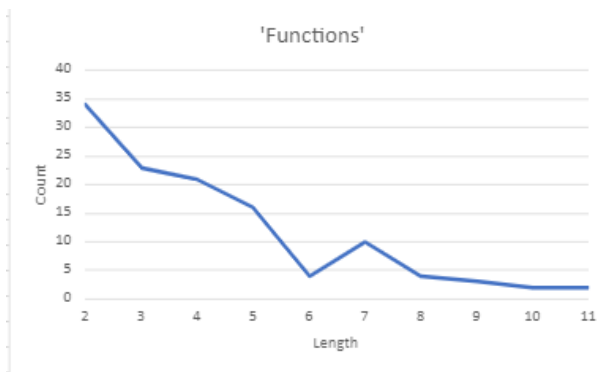
One advantage of Deep Neural Network is generating comments directly by learning source code instead of synthesizing comments from keywords for searching similar code snippets comments. Synthesizing comments from keywords usually uses some manually crafted templates. The procedure of template definition is time-consuming and the quality of keywords depends on the quality of a given Python method. They fail to extract accurate keywords when the identifiers and methods are poorly named. The Information Retrieval metrics based approaches usually search the similar code snippets and take their comments as the final results. They rely on whether similar code snippets can be retrieved and how similar the snippets are. Deep Neural Networks builds language models for code and natural language descriptions. The language models are able to handle the uncertainty in the correspondence between code and text. Deep Neural Networks learn common patterns from a large-scale source code and the encoder itself is a language model which remembers the likelihood of different Python methods.

RQ1: What is the impact of source code and comments with different lengths on the performance of Deep Neural Network?

We further analyze the prediction accuracy for Python methods and comments of different lengths. Figures which are attached below for Comments and Function Length. If we process the comment and the function without preprocessing then there will be a dip in the accuracy level. We can see in the below figure that comment length varies between 0 to 18 words and function varies between 0 to 11 word.



We can see that 'Comments' which are greater than 14 words are just 4 comments. If we can anyhow reduce the number of words in the comments which is greater than 14. Then that will help.



The technique which is used is the Bag-of-Words Model. If the length of the comments or function increases then that will result in an increase in the number of features. And all the features are not that important to train the model. Therefore if we can decrease the length of the sentence which will help to build the great model. So, therefore if we do some preprocessing and reduce the length of the sentence and remove some of the words which do not add much importance to the sentence.

RQ2: what are the different challenges while vectorization of source code and comments .

There are many challenges while vectorizing the source code and comments. The main challenges are, there are multiple ways in which we can vectorize our source code and

comments. Each vectorizer will give us different performance once we build a model so it is very difficult to pick one of the vectorizers which will give the optimal performance. It also depends on the data set which we provide to the vectorizer.

The main challenge is that we can't come to any conclusion about the vectorizer until the model has been built and the performance is checked. So therefore I used two different kinds of vectorizer.

```
In [14]: from sklearn.feature_extraction.text import TfidfVectorizer
         from sklearn.feature_extraction.text import HashingVectorizer
         import numpy as np
         import matplotlib.pyplot as plt
         import pandas as pd
```

- TfidfVectorizer

Using Tfidf Vectorizer there were lots of challenges as the output of the vector was a sparse matrix.

(0, 87)	1.0
(1, 207)	0.3252270097096923
(1, 106)	0.8549558797748033
(1, 73)	0.3964112889296219
(1, 60)	0.0783321505062868
(2, 166)	0.2691607187128395
(2, 134)	0.2691607187128395
(2, 165)	0.2691607187128395
(2, 133)	0.2691607187128395
(2, 46)	0.2923949401840794
(2, 106)	0.7882746414771074
(2, 60)	0.05777818417739952
(3, 107)	0.5230278632998198
(3, 106)	0.8460261340342816
(3, 60)	0.10335199438343137
(4, 47)	0.2923949401840794
(4, 166)	0.2691607187128395
(4, 134)	0.2691607187128395
(4, 165)	0.2691607187128395
(4, 133)	0.2691607187128395
(4, 106)	0.7882746414771074
(4, 60)	0.05777818417739952

The matrix has to be changed to a dense matrix and then further changed to an numpy array to do the further processing. Building the model using this vectorizer, faced a lot of problems at the end when the model was being built we did not get any accuracy level which means that for the data set which had been given to the vectorizer it did not work. Probably the vectorizer was not suitable for the data set which was fed in this vectorizer so therefore we have to move to a different kind of vectorizer which is a hashing vectorize.

- HashingVectorizer

Using Hashing Vectorizer is an efficient way of mapping terms to features.

The main advantage of using the hashing vectorizer is even if the of features are very high in number we can reduce the number of features using this vectorizer which makes it less expensive to process but with an increased risk of collision.

`HashingVectorizer(n_features=n)` # where n can be any number of features.

Now using this we are able to change the 250 features to 1 feature.

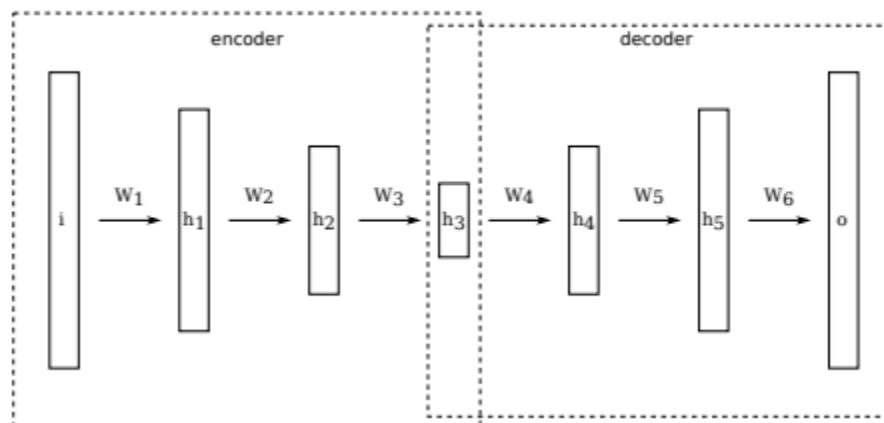
```
In [35]: # List of text documents
text = file2
# create the transform
vectorizer = HashingVectorizer(n_features=1)
# encode document
vector = vectorizer.transform(text)
# summarize encoded vector
print(vector.shape)
y=vector.toarray()

(120, 1)
```

At the end when we trained the model at the very initial state it gave 23% accuracy level which means that for the data set which had been given to the vectorizer it worked but not to the optimal level. Probably we have to work with a Neural Network to increase the accuracy.

RQ3: How effective is the Deep Neural Network?

Deep Neural Networks help us to mold our model as per our choice. Once we build the model we manipulate the number of hidden layers which give us different results. The structure of the neural network model is shown in the figure below



A deep autoencoder with hidden layers h1 to h5 and weight matrices W1 to W6. The goal of the training is to restore the output “O” resemble input “I” as closely as possible. After the training, an encoder part of the autoencoder can be used to generate a low-dimensional representation h3 of input data i.

The main advantage of using Neural Networks is that we can give any number of input features for the processing. And with the help of one-hot encoding it is possible to get the better result. Gradient descent is used for model training.

```
In [42]: def get_predictions(A2):
          return np.argmax(A2, 0)

def get_accuracy(predictions, Y):
    #print(predictions, Y)
    return np.sum(predictions == Y) / Y.size

def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    #print(W1.shape)
    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 10 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            print(get_accuracy(predictions, Y))
    return W1, b1, W2, b2
```

```
In [43]: W1, b1, W2, b2 = gradient_descent(X_train, Y_train, 0.05, 500)
          print(W1, b1, W2, b2)
```

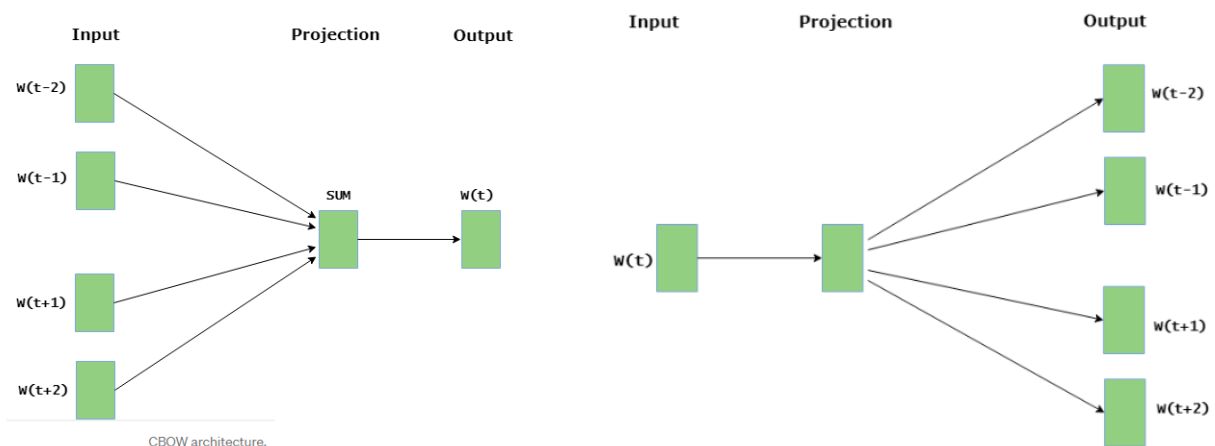
```
Iteration: 0
0.0
Iteration: 10
23.0
Iteration: 20
23.0
Iteration: 30
23.0
Iteration: 40
```

As gradient descent is computational efficient, it produces a stable error gradient and a stable convergence. Some disadvantages are the stable error gradient can sometimes result in a state of convergence that isn't the best the model can achieve. As the model is built from scratch **without** the use of tensorflow or keras. Therefore we are not able to get good results. In the future we can improve the result using tensorflow or keras to build the Neural Network.

5. LIMITATIONS OF THE STUDY

Translation between source code and natural language is challenging due to the structure of source code. One simple way to model source code is to just view it as plain

text. However, in such a way, the structure information will be omitted, which will cause inaccuracies in the generated comments. we could have achieved better results compared to what we have achieved. As we used the automatic vectorizer for converting the source code and functions to vector which is the main limitation as we have used automatic vectorizer we can't do much about it's output which we have received we cannot manipulate it to overcome this difficulty or to overcome these challenges we have to come up with some design where we can convert the comment and functions in a vector format by using some high end algorithm we can use a kind of neural network to convert the comment into vector where we can use the English grammar rules to extract the information and then convert it into vector.



If we can convert the text to a vector which can give a good prediction level we would be able to achieve a great result so we have to work in the area of converting text to vector.

Once. We can increase the quality of vectors then we can build a great model.

Using Neural Networks and Graph we can come up with Graph Neural Networks (GNNs) which can also help to improve the result.

There are many good models which can be referred to to improve the result. Only some of the ideas have been discussed which can help to improve the existing model.

Related research papers similar to your study.

6. RELATED WORK

One of the related works which we come across is Hybrid-DeepCom. Hybrid-DeepCom leverages both the source code and its AST structure to generate the code comment. These ASTs are converted to specially formatted sequences using a new structure-based traversal (SBT) method. SBT can express the structural information and keep the representation lossless at the same time. To reduce the out-of-vocabulary tokens, we split the identifier according to the camel casing naming convention. Hybrid-DeepCom outperforms the state of the art approaches and achieves better results on both machine translation metrics and information retrieval metrics.

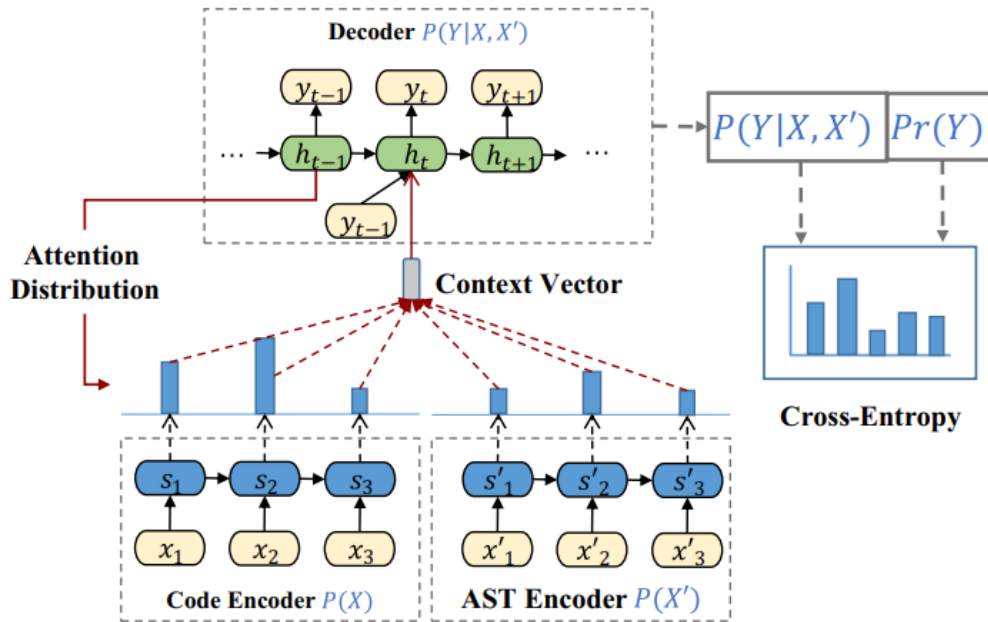
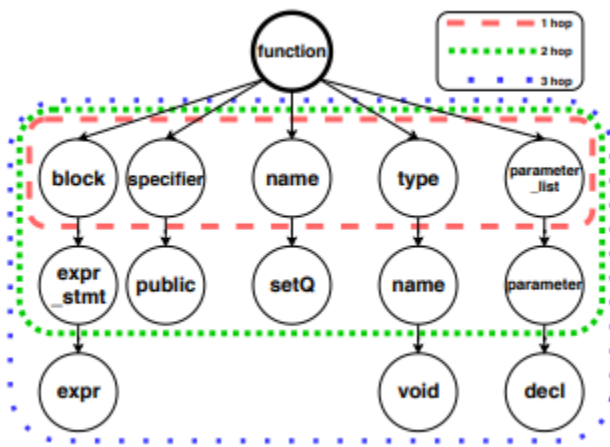


Fig. 3 The detailed Hybrid-DeepCom model

The other related work which we come across is Convolutional Graph Neural Networks. Convolutional Graph Neural Networks (ConvGNNs) can be used because they are well suited for this task, and this is what we can use to improve our model. In my research we have found that ConvGNNs were developed after Recurrent Graph Neural Networks and were designed with the same idea of message passing between nodes. They have also been shown to encode spatial information better than RecGNNs and are able to be stacked, improving the ability to propagate information across nodes. ConvGNNs take graph data and learn representations of nodes based on the initial node vector and its neighbors in the graph. The process of combining the information from neighboring nodes is called “aggregation.” By aggregating information from

neighboring nodes a model can learn representations based on arbitrary relationships. These relationships could be the hidden structures of a sentence, the parts of speech, dependency parsing trees, or the sequence of tokens. ConvGNNs have been used for similar tasks before, such as in graph2seq for semantic parsing and natural question generation. ConvGNNs also allow nodes to get information from other nodes that are further than just a single edge or “hop” away. In the figure below we show an example partial AST and what 1, 2, and 3 hops look like for the token ‘function’.



Each time a hop is performed, the node gets information from its neighboring nodes. So, on the first hop the token ‘function’ aggregates information from the nodes ‘block’, ‘specifier’, ‘name’, ‘type’, and ‘parameter_list’. In the next hop that occurs, the ‘function’ node will still only combine information from its neighbors, but now each of those nodes will have aggregate information from their children. For example, the node ‘block’ will contain information from the ‘expr_stmt’ node. Then when the ‘function’ node aggregates the ‘block’ node, it has information from both ‘block’ and ‘expr_stmt’. There are many related works which are similar to our work; only a few works have been listed.

7. CONCLUSION AND FUTURE WORK

In his paper formulates code summarization tasks as a machine translation problem which translates source code written in a programming language to comments in natural language. We point out two challenges while learning the source code, vectorization of comment and function. And also training the Neural Network with the right number of

hidden layers. We propose a Neural Network to train our model where there are many challenges to convert data sets into vectors. Once the data is converted we have to come up with the right number of features. As if we increase the number of features then it may not train the model properly and if there are less features then there is a problem of collision.

In future work, we plan to improve the effectiveness of our proposed approach by introducing Hybrid-DeepCom, a variant of attention-based Seq2Seq model, to generate comments for python methods.

Applying machine learning to static code analysis is an exciting area of research, allowing the large amount of free source code available today to help solve tasks like plagiarism and malware detection. However, as further methods are researched for extracting information from large corpora of source code in the future it is imperative that they can generalise effectively and are robust at inference-time to surface-level changes in the code. Training on any large-scale crowd-sourced data should be approached carefully as biases in the data can be amplified in the models, causing them to be fooled easily and generalise poorly.

8. ARTIFACTS

All the related files and data set are uploaded in the git repository. Please visit the web page for more information.

[Code Comment Generation](https://github.com/alok275/Code_Comment_Generation.git)

https://github.com/alok275/Code_Comment_Generation.git