Code2Cap: Automated Code Captioning

Fabrice Harel-Canada
fabricehc@g.ucla.edu
UID:705221880
University of California
Los Angeles

Vishwa Karia
vishwakaria2@ucla.edu
UID:705223110
University of California
Los Angeles

Arghya Mukherjee
arghya@cs.ucla.edu
UID:905225938
University of California
Los Angeles

Zeel Doshi
zeeldoshi@cs.ucla.edu
UID:905220399
University of California
Los Angeles

Vaishnavi Pendse
vaishnavipendse@cs.ucla.edu
UID:005226683
University of California
Los Angeles

Victor Chang
ippouzutsu@g.ucla.edu
UID:805219141
University of California
Los Angeles

ABSTRACT

The ability to automatically summarize or caption target sequences has an extensive history within Natural Language Processing (NLP) research. However, such applications to computer programming languages within the Software Engineering field are less well studied. We approach the code summarization problem by re-framing it into a translation task. Doing so allows us to employ a mixture of techniques developed in the neural machine translation (NMT) sub-field to produce concise and accurate natural language outputs for a given source code snippet as input. We call our code-to-caption approach Code2Cap and demonstrate its effectiveness on two benchmark data sets written in Java and Python. Our code, data, and trained models are available at `https://github.com/vishwakaria/code-summarization`.

KEYWORDS

Software Engineering, NLP, NMT, Code Summarization, Transformer, AST

ACM Reference Format:
Fabrice Harel-Canada, Vishwa Karia, Arghya Mukherjee, Zeel Doshi, Vaishnavi Pendse, and Victor Chang. 2019. Code2Cap: Automated Code Captioning.

1 INTRODUCTION

Automated code captioning is the process of leveraging natural language processing to generate a brief text-based description for an input snippet of source code. Software developers regularly perform this task while familiarizing themselves with new code, performing code reviews, settling on a function name or generating documentation for their own code. In a broader sense, any task that involves understanding code necessarily entails an antecedent mental captioning of its purpose. We view the automation of this first step as an important time-saving contribution to the development process.

Previous work in this area can be categorized into two main paradigms: models that account for the differences between natural languages and programming languages and those that do not. Most sequence-to-sequence (seq2seq) models, adopted from neural machine translation (NMT), frame the code captioning task as a machine translation problem and treat source code like any other natural language - as a sequence of tokens. Such models were not specifically designed for handling programming languages, but have nonetheless been able to achieve state-of-the-art results. In contrast, approaches that leverage the syntactic structure of programming languages in the pre-processing and encoding portions of their pipelines to speed up training and improve the quality of the captions.

We present an alternative approach, Code2Cap that synthesizes elements of both paradigms to generate code captions that are both concise and reasonably accurate. We demonstrate the effectiveness of our approach on a dataset of the statically typed Java programming language [14] as well as a dataset of the dynamically typed Python programming language [2].

2 MOTIVATION

In general, there are at least four primary applications that we expect will benefit from code captioning - code review, documentation, method name suggestion, and code familiarization. Of these we elaborate on the particular challenges related to code review as a motivating example for our work.

Code review is a common practice in software engineering, wherein the code written by one member in the team is inspected for quality and correctness by one or more of his colleagues [17] [16]. However, studies suggest that this

CS 230 project, June 2019,
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

is often a time-consuming process, as the colleagues performing the review have to first determine on their own what each function in the code is intended for [18]. At the time of code review, developers rely on in-line comments, self-explanatory variable names as well as formal documentation to understand the working of different pieces of code [15]. However, in practice it has been observed that the comprehensiveness, clarity and hence, the usefulness of both in-line comments and variable names are heavily dependent on the developer who wrote the code. For this reason, it is not always practical to rely on them completely. Formal documentation that is manually written by the team of developers is much more reliable in comparison to comments and variable names. They often provide a more detailed explanation of what a piece of code intends to do. But, even these documents may not be entirely reliable. They need to be manually updated each time the code is changed or a new version of a software is developed [4] [13]. Consequently, there is a possibility of inconsistencies between the documentation and the code version. From this, we infer the need for a real-time high-level summary of code snippets that can overcome the aforementioned benefits of comments and documentation. Real-time generation ensures that the descriptions correspond exactly to the current code version.

Hence, we see strong motivation in building an intelligent model that can automate the task of generating a high-level understanding of code snippets. Such a model would be helpful not only to code reviewers but also for novice developers attempting to understand a legacy code on their organization. With this regard, in this paper, we propose a system that leverages natural language processing using ASTs and transformers to achieve the task of automated code summarization. While we do not lay out any concrete plans for tools that would implement our approach, We note that such a tool could instrumental in software maintenance tasks like those aforementioned.

3 LITERATURE SURVEY

Code summarization is an ongoing and active research topic within the fields of software engineering and natural language processing. Recent breakthroughs in the study of artificial neural network design have inspired many to devote their time and efforts towards this particular research application. In this section, we discuss some of the relevant prior art and benchmarks.

As alluded to earlier, all previous work in this area can be broadly categorized into two main paradigms: models that account for the differences between natural languages and programming languages, and those that do not. The latter generally leverage the structural semantics of the code, e.g. Control Flow Graphs (CFGs), Abstract Syntax Trees (ASTs), etc.

First, we discuss those approaches that take into account the syntactic structure of programming languages. Iyer *et al.* [11] propose a model called CODE-NN that makes use of Long Short Term Memory (LSTM) networks [7] with an attention-based mechanism to achieve the code summarization task. Both LSTM and attention model ensure that the syntax of the code is not ignored during both training and testing. Additionally, Fowkes *et al.* [6] parses the AST generated from the source code, and introduces the concept of "code-folding"- removing the less informative regions while maintaining the constraint that no information is lost. On the high-level, the underlying idea behind this work is the same as extracting a meaningful, informative summary from a document, and that might explain intuitively why the same idea applies to code summarization.

The model DeepCom proposed by Hu *et al.* [9] attempts to generate code comments automatically. The mechanism used to achieve this is a combination of that proposed in [11] and [6]. More specifically, it first generates ASTs to effectively capture the structures and semantics of the input code. Then, it uses an encoder-decoder based LSTM with attention mechanism to generate natural language sequences. On the other hand, instead of using ASTs, Li *et al.* [8] represents them as vector embeddings. This embedding is achieved through the use of RNN as an encoder. Consequently, learning is carried out as usual using the LSTM-based approach.

Lastly, the most similar approach to ours was that of Alon *et al.* [1] in their work known as Code2Seq. In this approach, the input code snippets are similarly first encoded into ASTs to leverage their syntactic structure. However, they employ standard bi-directional LSTM for the purposes of encoding the ASTs into their latent representation vectors. What makes Code2Seq state-of-the-art is the fact that the encoding process is followed by generating sub-token embeddings that help capture the compositional nature of the AST's terminal's values. Code2Seq model is known to outperform the benchmark NMT models. To sum up, this category of approaches that leverage the syntactic structure of programming languages in the pre-processing and encoding portions of their pipelines are known to speed up training and improve the quality of the captions.

On the contrary, most sequence-to-sequence (Seq2Seq) models, adopted from neural machine translation (NMT), frame the code captioning task as a machine translation problem and treat source code like any other natural language - as a sequence of tokens. Such models were not specifically designed for handling programming languages, but have nonetheless been able to achieve state-of-the-art results until recently. The works by Sutskever *et al.* [10], Cho *et al.* [3] and Vaswani *et al.* [19] belong to this paradigm to yield state-of-the-art performances.

In this paper, we present an alternative approach that synthesizes elements of both paradigms to generate code summaries. The goal of this research is to check the effectiveness of such a hybrid approach in comparison to their independent counterparts. We demonstrate the effectiveness of our approach on a dataset of the statically typed Java programming language as well as a dataset of the dynamically typed Python programming language.

4 APPROACH

There have been various approaches for code summarization in the past, some have leveraged ASTs in their input and many others have chosen to feed in the function bodies verbatim. However, a comparative study of the influence of syntactic structure in the inputs, in the form of ASTs, has not been conducted yet. Hence, in addition to evaluating the results of code summarization for statically vs. dynamically typed languages, we also evaluate the results when the input consists of syntactic components vs. when it does not for both the languages.

While using syntactic structures in the input, our first step in accounting for the particularities of captioning code snippets is to represent them as an AST. The root and leaves of an AST usually refer to user-defined values that represent identifiers and names from the code while the internal nodes represent some operator of the language like for loops, if statements, and var declarations. The first and last node of an AST path are terminals whose values are tokenized as part of preprocessing. Furthermore, we recognize that many user-defined nodes employ stylized naming conventions wherein a small number of natural language words are concatenated to name a code variable or method. Therefore, we perform an additional ‘sub-tokenization’ step on all such nodes to increase the degree of correspondence with natural language. For example, the single token `createNewFile` would become three separate tokens `create`, `new`, and `file`.

The next step is to embed the tokenized ASTs into a fixed-length vector which is suitably representative of the input data. For our study, we elected to use a transformer model [8] because of its strong performance on standard machine translation tasks. To our knowledge, we are the first to generate and incorporate this model for the code summarization task on any programming language. The transformer’s encoding layers provide an opportunity to learn a deep representation of the AST paths. During decoding, the transformer attends over a different weighted average of the path embeddings to produce each output token, much like NMT models attend over token representations in the input sentence.

In order to evaluate the influence of ASTs on the performance of the model, we fed two versions of the input to our model. In the first version, we used the method bodies and declarations concatenated together. For the second one, we replaced the method bodies with ASTs generated for them. The raw code and ASTs produced contain a lot of information in terms of the syntactic structure, so we decided to evaluate their performance both with and without punctuation, leading to a total of 4 preprocessing variations. Moreover, all 4 variations were tested for both languages, Java and Python. Thus, we tested 8 different variations of preprocessing in total.

5 MODEL ARCHITECTURE

Our project aims to leverage the syntactical structure of programming languages to generate more precise and accurate descriptions. An effective way to do so is making use of the Abstract Syntax Tree (AST) representations before training the model. However, we want to check the effectiveness of ASTs when it comes to model performance. Hence, each model combination that we attempt will be tried out both with as well as without AST encoding. Consequently, we report our inference about the efficacy of ASTs after observing the results thus obtained.

We have used the Transformer model [19] as the main architecture for our work. The standard encoder-decoder architecture has served to perform well for many machine translation tasks, hence the transformer architecture was built on top of it. The success of transformers in most NLP tasks has proven the fact that a simple architecture composed of attention mechanisms alone, can outperform the complex CNN and RNN architectures.

The basic encoder-decoder framework [3] converts the entire input sequence into a fixed-length vector, called the ‘context vector’. This context vector is then fed to the decoder model. It uses the information ‘encoded’ in the context vector to predict the output sequence of words. The main drawback of this model is the use of a fixed-length representation of the input data. Generally, the input statements fed to a network are of varying lengths. Encoding the longer statements into a smaller representation can lead to loss of information, causing our model to perform sub-optimally on longer inputs.

This shortcoming was addressed by the attention mechanism [12] which uses variable length encoding for sentences, achieving better results for especially those sentences which are longer than the ones in our training corpus. As the name suggests, attention tries to align the input and output, capturing information about which words in the input are relevant to which ones in the output. Thus, for each output word, a different context vector is produced.

A simple combination of encoder-decoder and attention is the Transformer model shown in Figure 1. The encoder consists of 6 identical layers, each consisting of a multi-head self attention mechanism followed by a fully-connected feed forward network. Self-attention associates different words in

the input with each other, to encode dependencies between them. 'Multi-headed attention' associates each layer with a query and a set of key-value pairs, so that a word can focus on multiple words in the input while computing self-attention. The size of the input to each layer is 512, with the input to the first layer being the word embeddings. The decoder arrangement is similar, consisting of 6 layers stacked one upon the other. However, each of these layers also have an attention layer, allowing the decoder to focus on relevant words in the input.

Since the model does not use convolutions and recurrences, a different mechanism was needed to account for the word orderings. Hence, the model leverages positional encodings which help to determine the distance between different words in the sequence. On a high level, a vector, composed of sine and cosine frequencies, is added to each word embedding. Learning this vector during training provides sufficient information regarding the relative positions of words.

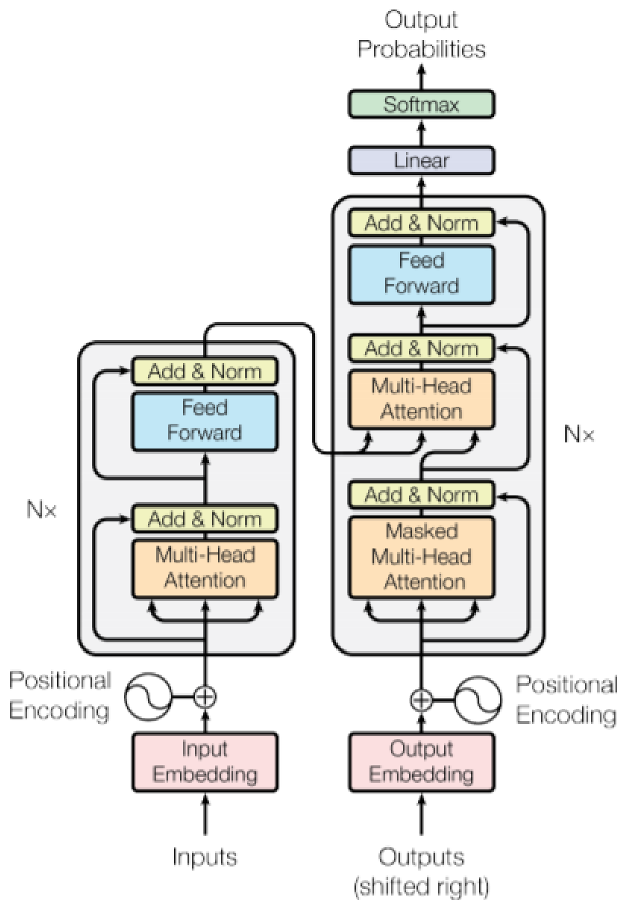

Figure 1: The Transformer-model Architecture as presented in [19]

6 DATASETS

Java

The Java dataset was obtained from [14]. In order to facilitate comparison with others, we used the same train, validation, and test splits provided by the dataset authors. The training data has 1.9 million observations, the validation data has 120k observations, and the testing data consists of approximately 125k observations.

Python

The Python source code [2], obtained from GitHub repositories, was preprocessed by removing the comments, semantically irrelevant spaces and new lines. For the initial preprocessing, we follow the similar steps as in [2]. However for AST generation, there were some conflicts due to the fact that the source code is written in Python 2.7, and the `ast()` module expects Python 3.x. Due to these conflicts in some parts of the dataset, we have used a smaller version of it which contains 72,000 methods. Out of this, 68,000 were used for training, 2,000 for validation and 2,000 for testing. An example of an extracted method is shown in Figure 2.

7 EVALUATION

In this exploration of the subject, we focus on a particular type of captioning task known as source code summarization, which aims at outputting a single, concise, and grammatically correct English sentence for a given source code input. Another captioning task in this domain includes method name prediction, which we view as a special, more narrow kind of source code summarization where the output is generally limited to five (5) output tokens (English words). This other task we leave to future work.

The nature of summarizations and assessments of their quality rely on the semantics being well captured and conveyed in the summary. This makes it a task best suited for qualitative analysis and so we personally review several examples that deviate from the true label, but nonetheless capture the semantic meaning of the source code input.

We also evaluate the results obtained from different models using the following quantitative metrics:

BLEU Score

Bilingual Evaluation Understudy(BLEU) is a score for comparing a candidate translation of text to one or more reference translations. It can be used to evaluate the text generated for a suite of natural language processing tasks to a reference sentence. It is computed by counting matching n-grams in the candidate translation to n-grams in the reference text where unigram would be each token and a bigram comparison would be each word pair. The comparison made does not consider the ordering of words. The counting of

```
def _interceptdot (w, X, y):

  """
  Computes  y*np . dot (X, w).
  It takes into consideration whether  the  intercept  should  be  fit.
  Parameters
  ----------
  w : ndarray ,  ndarray ,  shape  ( n_features , )  or  ( n_features +  1 , )
    Coefficient  vector .
  [ . . . ]

  """

  c =  0
  if w.size  == X. shape [ 1 ]  +  1:
    c = w[-1]
    w = w[:-1]
    z =  safe_sparse_dot (X, w)  + c
    yz = y*z
  return w,  c ,  yz
```

(a) An example Python method

```
def _interceptdot(w, X,  y):
```

(b) Extracted declaration

```
DCSP c = 0.0 DCNL DCSP  if  (w.size == (X.shape [1] + 1) ): DCNL
DCSP  DCSP c = w[(-1)] DCNL DCSP  DCSP w = w[:(-1)] DCNL DCSP z
=(safe_sparse_dot(X, w) + c) DCNL DCSP yz = (y*z) DCNL DCSP
return w,  c ,  yz
```

(c) Extracted function body after pre-processing

```
Computes  y*np . dot (X, w) . DCNL It  takes  into  consideration
if  the  intercept  should be  fit  or  not. DCNL P arameters DCNL
w :  ndarray ,  shape  ( n_features , )  or (n_features  +  1 , )
DCNL Coefficient  vector . DCNL  [ . . . ]
```

(d) Extracted function docstring

```
"Module(body=[FunctionDef(name='_interceptdot', args=arguments(args=[arg(arg='w', annotation=None),
arg(arg='X', annotation=None), arg(arg='y', annotation=None)], vararg=None, kwonlyargs=[],
kw_defaults=[], kwarg=None, defaults=[]), body=[Expr(value=Str(s='\\n Computes y*np . dot (X, w).\\n
It takes into consideration whether the intercept should be fit.\\n Parameters\\n ----------\\n w :
ndarray , ndarray , shape ( n_features , ) or ( n_features + 1 , )\\n Coefficient vector .\\n [ . . .
]\\n\\n ')), Assign(targets=[Name(id='c', ctx=Store())], value=Num(n=0)), If(test=Compare(
left=Attribute(value=Name(id='w', ctx=Load()), attr='size', ctx=Load()), ops=[Eq()],
comparators=[BinOp(left=Subscript(value=Attribute(value=Name(id='X', ctx=Load()), attr='shape',
ctx=Load()), slice=Index(value=Num(n=1)), ctx=Load()), op=Add(), right=Num(n=1))]), body=[Assign(
targets=[Name(id='c', ctx=Store())], value=Subscript(value=Name(id='w', ctx=Load()), slice=Index(
value=UnaryOp(op=USub(), operand=Num(n=1))), ctx=Load())), Assign(targets=[Name(id='w',
ctx=Store())], value=Subscript(value=Name(id='w', ctx=Load()), slice=Slice(lower=None, upper=UnaryOp(
op=USub(), operand=Num(n=1)), step=None), ctx=Load())), Assign(targets=[Name(id='z', ctx=Store())],
value=BinOp(left=Call(func=Name(id='safe_sparse_dot', ctx=Load()), args=[Name(id='X', ctx=Load()),
Name(id='w', ctx=Load())], keywords=[]), op=Add(), right=Name(id='c', ctx=Load()))), Assign(
targets=[Name(id='yz', ctx=Store())], value=BinOp(left=Name(id='y', ctx=Load()), op=Mult(), right=Name
(id='z', ctx=Load()))), Return(value=Tuple(elts=[Name(id='w', ctx=Load()), Name(id='c', ctx=Load()),
Name(id='yz', ctx=Load())], ctx=Load()))], orelse=[])], decorator_list=[], returns=None)])"
```

(e) Extracted AST represented as a list

Figure 2: A Python function with its extracted declaration, body, docstring and AST

matching n-grams is modified to make sure that it takes the occurrence of the words in the reference text into account and does not reward a candidate translation that generates an abundance of reasonable words.

ROUGE

Recall-Oriented Understudy for Gisting Evaluation (ROUGE) is essentially used for evaluating automatic summarization of texts as well as machine translation. It works by comparing an automatically produced summary or translation against a set of reference summaries.

ROUGE-1. : It measures the overlap of unigrams between the predicted summary and true summary.

ROUGE-2. : It measures the overlap of bigrams between the predicted and actual summaries.

ROUGE-L. : It measures the overlap of the longest common sequence between the predicted and actual summaries. An advantage of using the longest common sub-sequence is that it does not require consecutive matches but in-sequence matches that reflect sentence level word order. Since it automatically includes longest in-sequence common n-grams, a predefined n-gram length is not needed.

BLEU and ROUGE are not fully representative of the quality of a code summarization. They are simply metrics that compare a predicted summary to a select few human-defined summaries which do not account for the many valid and possibly better alternatives to summarize the code. Thus, it could be the case where our model finds an alternative summarization that is actually more suited than the one provided by the dataset. However, still we use BLEU and ROUGE as they are concrete quantitative metrics for the task given that we want our model to generate code comments similar to comments written by humans.

8 RESULTS AND DISCUSSIONS

As can be seen from Tables 1 and 2, the pre-processing technique that consists solely of code tokenization achieved the best performance for all the 4 evaluation metrics across both Java and Python. However, we are reluctant to conclude that it is therefore the best pre-processing technique. As observed by Uri *et al.* [1], the process of encoding raw source code into ASTs is strongly associated with the performance of their Code2Seq model and we expected a similar effect in our efforts. We theorize that the curious result of ASTs harming performance in our case while strengthening performance in their case is attributable to the precise manner in which ASTs were produced. In their case, only $k = 3$ branches of the AST were sampled and *only* the contents of the AST nodes were used as input. We can think of their pre-processing as *slim AST*.

On the other hand, we kept all branches of the AST in addition to keeping the metadata tags added by the `ast` conversion library, which describe the contents of each node. In practice, this meant that our input was considerably more verbose than the *slim ASTs*. This increased verbosity is theorized to have two distinct impacts that can explain the degradation of performance. The first is that the tags acted as distracting noise, competing with the actual code for attention within with translation process and diffusing focus from relevant tokens. The second explanation is that the metadata tags were repetitive and common across nearly all code inputs, which served to homogenize the training corpus and make it more difficult to detect useful patterns in the data.

One other observation we make is that AST pre-processing does indeed improve BLEU scores with respect to unmodified (raw) versions of the source code. We theorize that this is

Table 1: Java Evaluation Results

Model	BLEU	ROUGE-1	ROUGE-2	ROUGE-L	No. of methods evaluated
Raw*	10.38	10.87	11.83	9.55	54,360
Cleaned AST*	7.44	9.93	3.45	8.51	16,990
AST*	11.46	11.42	14.49	10.13	49,600
Tokenized	16.47	32.25	14.14	28.55	105,832

Table 2: Python Evaluation Results

Model	BLEU	ROUGE-1	ROUGE-2	ROUGE-L	No. of methods evaluated
Raw*	6.66	13.08	5.24	11.16	992
Cleaned AST*	5.05	6.61	1.75	5.23	1,110
AST*	10.46	14.86	12.07	12.30	1,260
Tokenized	14.46	22.18	12.52	19.09	2,000

because the ASTs provide the structural information of the code to the model "for free", whereas the model must learn how to interpret code syntax constructs like brackets and parentheses from scratch.

One final note related to quantitative metrics is that higher BLEU scores have been observed in other work, like that of Uri *et al.* [1] achieving a score of 23.04. Unfortunately, due to the lack of standardization in code captioning benchmarks, we cannot determine that our datasets are of similar difficulty or quality even if they both pertain to the same language. Differences in constituent code extraction methods, projects selection criteria, and data set splitting decisions all undermine attempts to draw direct comparisons with work not done on the exact same datasets. Due to this, we focus primarily on comparison with the benchmarks provided by the original dataset creators.

In the subsections below, we briefly report on the quantitative metrics and highlight some example summaries that ccharacterize the concision and accuracy of our model even when generated summary differs from the ground truth label. In all cases, the ground truths were taken to be the documentation string extracted from the function. Since these were produced by the original author of the function, we believe that they will represent an accurate summary of the intended behavior.

Please note that the evaluations marked with * indicate that, in the midst of making predictions with these models, the cloud GPUs / CPUs crashed. Hence a subset of what was targeted was referred to without penalizing the remaining unevaluated examples. Only the tokenized set managed to process all given predictions.

Java

The best results were obtained for the model that was trained on tokenized methods. The BLEU score for this model is 16.47 and the ROUGE-1, ROUGE-2 and ROUGE-L scores are 32.25, 14.14 and 28.55 respectively. The average benchmark BLEU score from the original authors for this dataset was 17.41 [14]. This represents a net shortfall of 0.94 BLEU.

(1) **Example 1**
Ground truth summary: *get members by name*
Predicted summary: *get the list of member options*

(2) **Example 2**
Ground truth summary: *gets the maximum value for this ability score*
Predicted summary: *returns the maximum value*

Python

The best results were obtained for the model that was trained on tokenized methods. The BLEU score for this model is 14.46 and the ROUGE-1, ROUGE-2 and ROUGE-L scores are 22.18, 12.52 and 19.09 respectively. The benchmark BLEU score from the original authors for this dataset was 13.84 [14]. [2]. This represents a net gain of 0.62 BLEU.

An example of the predicted and the true summaries for this model are shown below:

(1) **Example 1**
Ground truth summary: *get the email addresses collected between startdate and enddate.*
Predicted summary: *return a dictionary of email address.*

(2) **Example 2**
Ground truth summary: *Requires that the user is logged in.*
Predicted summary: *a view that is login protected.*

Though the predicted and the corresponding true summaries for the two examples are not an exact match, it shows that the model has been able to get a general idea of the what the methods do. We believe that the model has the potential to perform better using a larger data set.

The quantitative performance of the models does not look particularly strong at the first glance. However, upon manual inspection of the output generated, we assess the captions to be reasonably accurate most of the time, showing a good degree of correspondence between the target description and our interpretation of the function bodies directly.

9 THREATS TO VALIDITY

External Validity

Although we have used a large dataset containing a wide variety of function types, these datasets have not been extensively explored to ensure that they are comprehensive in terms of the types of functions that they include. For example, some functions may throw exceptions, some may attempt to connect to a remote server or try to access I/O. It is quite possible that our dataset may not cover all kinds of codes. Consequently, if a new function type that is not originally present in the dataset is given as an input to these models, the performance may differ.

Internal Validity

While the Java dataset [14] consists of more than 2,000,000 examples, the Python dataset [2] contains only 72,000 instances. This disparity in data set size threatens the validity of a reasonable comparison between model performance on Java and Python functions. Because of this, we cannot conclude that the model works better with statically typed programming languages like Java in comparison to their dynamically typed counterparts like Python. However, theoretically, due to static typing, the model is, indeed, expected to perform better for Java.

Another internal threat to validity is the possibility of data leakage within our Python data set. Code functions are organized at a macro-level into projects and each project contains functions that are more likely to be similar to one another than functions from other projects. Therefore, it's important to actually group code by project and prevent them from spanning more than one split - either train, validation, or test subsets. The Java data set explicitly structured its data splits to contain constituent projects cleanly while our Python data set did not provide any such guarantee. Therefore, all else being equal, the Python data set theoretically has an unfair advantage over the Java data set.

Construct Validity

It is possible that the doc strings extracted from the functions is not a reliable or suitable caption of the function body to use as a ground truth. Instead, it may reflect instructions on how to use the function or some other miscellaneous information that strictly speaking is not summarization-oriented. We did not perform a broad analysis of the target sequences to conclusively determine that this evaluation construct was valid, but from the dozens of functions reviewed during our quantitative analysis, no doc string stood out as being anything but a faithful caption of the original function.

10 FUTURE WORK

One major goal of this effort that unfortunately did not materialize in time was the incorporation of BERT [5] to obtain contextual embeddings of the code before feeding it to the transformer model. To our knowledge, no other research has produced a contextualized word embedding for a programming language and this would have been the most valuable delta provided by this research. Ultimately, the newness of BERT and our particular application idea contributed to a lack of online resources to guide this branch of development work. However, in consideration of the challenges unique to source code we anticipate one particular challenge and therefore also list it as future work areas.

This other challenge is the large size of programming language vocabularies, which are required for recognizing valid input tokens and for probability assignment during softmax in the final layers of the transformer. Most programming languages offer a significantly reduced set of functional keywords and should therefore be easier to work with than natural languages. Unfortunately, the ability to freely name function, variable, and other identifiers makes the vocabularies both incredibly volatile in form and volume. Add to that the prevalence of different naming conventions (camelCase, underscore_usage, etc) across different programming languages and the task of maintaining a reasonably vocabulary quickly becomes unmanageable. For context, our Java data set contained about 273k unique tokens, several times larger

than an English vocabulary. Processing such a massive data set would require incredibly powerful computation resources unavailable save for a few privileged institutions. To address this problem, future work must extend to the investigation of new approaches to tokenize and subtokenize, as well as means of abstracting away user-defined identifiers into a smaller subset of manageable types that somehow preserve the meaning of the code.

11 CONCLUSION

Various approaches for summarizing source code have been explored in the past, however, a systematic study for understanding the influence of syntactic elements like ASTs on the results has not been conducted. In this project, we evaluated the effect of using 4 different kinds of source code inputs - raw, tokenized, AST, and AST-clean - on our summarization model, Code2Cap. The evaluation was carried out on both Python as well as Java data sets. This helped us assess how the same model performs on two completely different languages. More specifically, we analyzed how the same model can have different performances for a statically-typed programming language and it's dynamically-typed counterpart. Based on the BLEU and ROUGE scores obtained, we were able to conclude that the use of ASTs can actually deteriorate model performance. While best results did not make use of AST encodings, we expect that the rationale behind leveraging the internal structure of programming languages will play an increasingly important role in providing models with more easily interpretable inputs. We also observed that the results were in general better for Java as compared to Python. However, this can also be attributed to the fact that the Java data set was significantly larger than its Python counterpart.

To the best of our knowledge, we are also the first to use a transformer model for the task of code summarization. Transformers are better suited for encoding long-term dependencies in inputs as they use self-attention. Since codes have more long-distance dependencies compared to simple text inputs, transformers were expected to perform well. The results support our claim as we obtained competitive results for some input combinations in spite of the limited resources.

ACKNOWLEDGMENTS

We would like to thank Dr. Miryung Kim for her continuous support and guidance throughout the execution of this project. We also appreciate our institute, University of California Los Angeles and the Computer Science Department for providing us access to the required computing resources. Additionally, we would also like to extend our gratitude to Uri Alon, a co-author of Code2Seq [1] for agreeing to give additional insight about his work and directing us to the OpenNMT implementation of the transformer model that is employed in this paper.

REFERENCES

[1] Uri Alon, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[2] Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275* (2017).

[3] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[4] Nicolas Anquetil de Souza, Sergio Cozzetti B. and KÃąthia M. de Oliveira. 2005. A study of the documentation essential to software maintenance. In *23rd annual international conference on Design of communication: documenting designing for pervasive information.* ACM, 68–75.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018).

[6] Jaroslav Fowkes, Pankajan Chanthirasegaran, Razvan Ranca, Miltiadis Allamanis, Mirella Lapata, and Charles Sutton. 2016. TASSAL: Autofolding for source code summarization. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C).* IEEE, 649–652.

[7] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[8] Ge Li Xin Xia David Lo Shuai Lu Hu, Xing and Zhi Jin. 2018. Summarizing source code with transferred api knowledge. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence* (2018).

[9] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension.* ACM, 200–210.

[10] Quoc V Le Ilya Sutskever, Oriol Vinyals. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems.* ACM, 3104–3112.

[11] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers),* Vol. 1. 2073–2083.

[12] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering.* IEEE Press, 135–146.

[13] Mira Kajko-Mattsson. 2005. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering* (2005).

[14] McMillan Collin LeClair, Alexander. 2019. Recommendations for Datasets for Source Code Summarization. In *2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL'19), Short Research Paper Track.*

[15] Paul W. McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *22nd International Conference on Program Comprehension.* ACM, 279–290.

[16] Yasutaka Kamei Bram Adams McIntosh, Shane and Ahmed E. Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering, Volume 21, Issue 5*

(2016).

[17] Emma Söderberg Luke Church Michal Sipko Sadowski, Caitlin and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *International Conference on Software Engineering: Software Engineering in Practice*. ACM, 181–190.

[18] Dang Y. Xie T. Zhang D. Tao, Y. and S. Kim. 2012. How do software engineers understand code changes?: an exploratory study in industry. In *SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 51.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.