UNIVERSITY OF CALIFORNIA,
IRVINE

Generating natural-language descriptions of code with the GPT language model

THESIS

submitted in partial satisfaction of the requirements
for the degree of

BACHELOR OF SCIENCE

in Computer Science

by

Brian Minh-Tuan Chu

Thesis Advisor:
Richard Futrell

2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Page

ABSTRACT OF THE THESIS

Generating natural-language descriptions of code with the GPT language model

By

Brian Minh-Tuan Chu

Bachelor of Science in Computer Science

University of California, Irvine, 2022

Richard Futrell, Advisor

Much work has been done to evaluate how well the GPT model is able to translate English language text to code. In this paper we explore in the opposite direction, and evaluate GPT's ability to comprehend code and generate English descriptions. Using the Mostly Basic Python Problems dataset published by Google, we take the code snippets from each problem and apply various obfuscations before generating prompts to give to the model, in order to see if there is a correlation between prompt length and quality of the final completion (as measured by distance metrics between the generated text and the original problem description). Overall, we find that there is no such correlation, which indicates that we may have to use some other metric or quality of the prompt to predict the overall performance of the model.

Chapter 1

Introduction

1.1 Background

Large language models are commonly used to help computers perform natural language processing tasks; in particular, the BERT model is the primary component powering the Google search engine [1], and the GPT family is used in products such as Github Copilot, which is used to take incomplete samples of code and generate new code that accomplishes the task at hand. These models are large in the sense that they are often implemented as deep neural networks with billions of parameters, and trained on large language corpora comprised of millions of documents. For example, researchers training the GPT-3 model utilized the Common Crawl dataset, consisting of the contents of publically available websites on the internet, as well as Wikipedia and other collections of literature adding up to a total of almost 500 billion English-language tokens.

In the case of the GPT-Codex model, researchers took the base GPT-3 model which is trained on natural language data, and extended it further by fine tuning on text taken from open-source code repositories on Github and elsewhere on the internet. As a result, Codex is also good at generating completions when prompted with snippets of code in programming languages like Python. As such, its skills at completing code in conjunction with its pre-

[1] https://www.blog.google/products/search/search-language-understanding-bert/

existing ability to process English language text makes Codex a useful tool for research into the heavily studied field of translating English language text into computer code. After the beta release of the Codex model, many projects and papers were published that demonstrated its effectiveness at the task of taking English text describing a goal such as formatting a website or implementing a neural network architecture and subsequently generating HTML code that accurately describes the requested layout, or Python Tensorflow code that follows the high-level specification given in the input.

1.2 Motivation

It is evident that the ability to take English descriptions and generate code from them is very useful in many cases, since it reduces the cognitive overhead of translating from high-level specifications to the low-level implementation details when actually writing code, speeding up the process of developing programs. However, it is also useful to go into the opposite direction and convert code snippets back into English text. From the perspective of code documentation, much work has been done to automate the process of automatically generating comments in code that accurately describe what task the code is doing. Common heuristics include parsing function names and method calls to get a general description of the code. However, there are flaws to current approaches to this problem. If the method names are not reflective of the actual task (i.e. if a function is named `addTwoNumbers()` but the function actually subtracts the numbers) then the generated documentation is inaccurate and is therefore useless at its purpose. Having intelligent methods of understanding code well enough to document is therefore a useful task.

It is also of scientific interest to evaluate how well the GPT model understands its linguistic inputs; English language prompts result in completions that are strikingly realistic, but the question still remains as to how much the model is capable of understanding its inputs versus simply generating statistically likely outputs. By probing the model with how

well it comprehends code we can get a better idea of its ability to comprehend its inputs, especially given that programming languages have a set structure with no ambiguity as to what it means, as opposed to natural languages that are more ambiguous.

1.3 Outline

The GPT model is specifically designed to perform text completion tasks where it is prompted with a text snippet as its input, and outputs text that best completes the prompt. As a result, one of its strengths is zero to few-shot learning: Given a small number of samples, the model is able to extrapolate from its input and generate high quality completions without having to specifically train on a larger dataset of inputs. Consequently, we aim to exploit this capability to evaluate how well GPT describes code.

Chapter 2

Method

2.1 Model

We aim to evaluate the strengths of the GPT family of models on describing code. As such, our main focus is on evaluating the GPT-Codex model in particular, which is based on the GPT-3 model trained specifically on text, but has been fine-tuned and trained to also work well at predicting code snippets. In order to access the model and to generate completions, we use the OpenAI API to pass prompts to the model and to retrieve its output.

2.2 Dataset

Our goal is to evaluate how well the GPT model can describe code, which requires datapoints comprised of a code snippet to be described, as well as a code description that serves as the base truth with which we compare the completion generated by GPT.

2.2.1 MBPP

We use the "Mostly Basic Python Problems" dataset published by Google, consisting of roughly 1,000 Python problems. Each problem consists of a English-language description

of the task to be solved, and a solution written in Python. Because the problems are crowdsourced from various people, the authors of this dataset also provide a sanitized subset, where the problems and solutions are assessed by hand to ensure quality. For example, problem statements that are considered to be too vague, or Python solutions that use non-standard programming idioms like taking a list and its size as parameters when just the list would suffice, are filtered out from this subset of problems.

Since the sanitized subset of snippets is idiomatically correct and because the problem statements are clear to understand, we expect that those traits make it better for the purpose of generating prompts, since the code is a straightforward implementation of the problem statement with no pitfalls that make it harder to comprehend, and so the model should be able to understand what the task at hand is.

2.2.2 OEIS

Before we settled on using the MBPP dataset, we originally had planned on using the Online Encyclopedia of Integer Sequences as our source of code snippets and natural language descriptions. This was because the encyclopedia lists descriptions of many different numerical sequences and cites verifiably correct code snippets from research papers that generate said sequences. By scraping the OEIS website for entries that had Python code snippets in particular, we were able to collect almost 7,000 sequences.

However, there were some flaws with this approach since the sequences are from mathematics papers and so require prerequisite knowledge in number theory. Descriptions of sequences might refer to high-level concepts that aren't necessarily common knowledge, or sequences might be named after a specific person, in which case the description isn't very useful to a math layperson. Also, due to the interconnected nature of math, there are many ways to implement the same sequence [1], so the description may describe one way of evalu-

[1]For example, the nth element in row k of Pascal's triangle can be seen as either the sum of the two numbers immediately above it in the above row of the triangle, or simply $\binom{n}{k}$, which certainly affects the code that would generate the sequence.

Figure 2.1: Examples of code obfuscations

Original	Obfuscated
`map(f, xs)`	`[f(x) for x in xs]`
`filter(f, xs)`	`[x for x in xs if f(x)]`

`for x in xs:`
`  do something with x`
	`sequence_iterator = iter(xs)`
	`  while True:`
	`  try:`
	`  x = next(sequence_iterator)`
	`  do something with x`
	`  except StopIteration:`
	`  break`

`y = {x for x in xs}`	`y = set()`
	`for x in xs:`
	`  y.push(x)`

ating the sequences and the code might express another. As such, this makes it harder to evaluate the completions from the GPT model to see how closely it matches with the original description. We considered ways to get around this, like crowdsourcing mathematicians to help evaluate completions, but it was ultimately too impractical to accomplish.

2.3 Obfuscation

Because GPT is trained on a very large corpus of text from the internet, there is a chance that the model has memorized some of its inputs, so when given a code snippet it might just repeat the description that the code snippet was associated with. To make sure that the model is actually comprehending its input and not repeating previous training data, we perform some transformations on the Python code using the Python AST library. For example, given a call to the filter or map function with an iterable as input, we can rewrite that as a for loop that iterates over the iterable. In addition, Python has some syntactic features that we can rewrite as well: list/set comprehensions can be unrolled into loops. By applying each obfuscation, we can get new snippets of code that accomplishes the same task as the original code, but rewritten in such a way that it is not likely to have been seen before.

In other words, we are only changing the syntactic structure. If the model truly understands the code, it should still be able to express the semantic intent.

Listing 2.1: Implementation of obfuscation

```
class ReplaceReduce(ast.NodeTransformer):
    '''
    y = reduce(f, xs, initial?) ->

    reduce_iter = iter(xs)
    y = next(reduce_iter) OR y = initial IF initial exists
    for iter_element in reduce_iter:
        y = f(y, iter_element)
    '''

    def __init__(self):
        self.iter_index = 0

    def visit_Assign(self, node):
        match node.value:
            case ast.Call(func=ast.Name(id='reduce')) as reduce_func:
                self.iter_index = self.iter_index + 1
                target = node.targets[0] # assume single assignment,
                    not like a = b = 1
                func = reduce_func.args[0]
                it = reduce_func.args[1]
                iter_assign = ast.Assign(targets=[ast.Name(id='
                    reduce_iter'+str(self.iter_index), ctx=ast.Store())
                    ],
                                         value=ast.Call(func=ast.Name(
```

```
                                id='iter', ctx=ast.Load()),
                                    args=[it],
                                        keywords=[])
                                        )
    initial_assign = ast.Assign(targets=[target],
                                value=ast.Call(func=ast.
                                    Name(id='next', ctx=ast.
                                    Load()),
                                            args=[ast.
                                                Name(id='
                                                reduce_iter
                                                '+str(
                                                self.
                                                iter_index
                                                ), ctx=
                                                ast.Load
                                                ())],
                                            keywords=[])
                                                 if len(
                                                reduce_func
                                                .args) ==
                                                 2 else
                                                reduce_func
                                                .args[2])
    loop = ast.For(target=ast.Name(id='iter_element'+str(
        self.iter_index), ctx=ast.Store()),
                    iter=ast.Name(id='reduce_iter'+str(self.
                        iter_index), ctx=ast.Load()),
                    body=[ast.Assign(targets=[target],
```

```
                value=ast.Call(func=
                    func,
                                args=[
                                    target
                                    , ast
                                    .Name
                                    (id='
                                    iter_element
                                    '+str
                                    (self
                                    .
                                    iter_index
                                    ),
                                    ctx=
                                    ast.
                                    Load
                                    ())],
                                keywords
                                    =[]))
                                    ],
                        orelse=[])
        return [iter_assign, initial_assign, loop]
    case _:
        return node
```

2.4 Prompt Engineering

Our main line of inquiry lies in the prompts given to GPT: our overall goal is to evaluate how well the model describes code. However, the quality of the model's output is dependent on the input to a certain degree. So, what we need to find out is which prompts are best to get high quality descriptions, and what makes them different from other prompts?

The GPT model works by taking in a prompt and returning an output that best completes its input. However, the model is also powerful enough that it not only completes the text, but the format of the text as well. We can take advantage of this to make GPT generate text that describes programming-language inputs by prompting the GPT model as follows:

```
<code snippet A>
The above code is <code description A>.

<code snippet B>
The above code is <code description B>.

<code to be described>
The above code is
```

where we take two problems to prime GPT, and take a third code snippet for GPT to describe in its completion. By using a prompt template like this, we can then generate multiple prompts in a consistent manner where the main variable is the choice of code snippets for the model to emulate.

2.5 Evaluation

To consistently evaluate prompts for GPT, we set aside an arbitrary problem from the sanitized subset for GPT to complete. Using the rest of the problems, we then generate

prompts by selecting pairs at random. Then, we evaluate how well GPT can predict the completion for the prompt by taking the description and the completion provided by GPT and calculating various text distance metrics between the two texts. Specifically, we tokenize the texts, and calculate the Jaccard similarity, cosine similarity, and BLEU score between the target description and the completion.

There are many qualitative ways to say how a prompt differs from another; for example, you could state that one prompt is more complex than another since the task statement is more involved. However, since these are highly subjective, it is harder to draw meaningful conclusions on such bases. Instead, we calculate quantitative metrics based on the length of the prompt, since that seems to be the main quantity that differs between prompts. At first glance, this is also a useful heuristic for qualities like complexity since shorter prompts are generally simpler and longer prompts are more complex.

In order to evaluate how much the length of the prompt affects the overall quality of the generated text, we calculate the Spearman correlations between the length of the text and each metric; as we see in the results, the data seems to cluster around certain values with a lot of outliers and so a linear regression would not be appropriate in this case.

Chapter 3

Results

3.1 Selected Data (Non-obfuscated)

Figure 3.1: length of prompt versus BLEU score

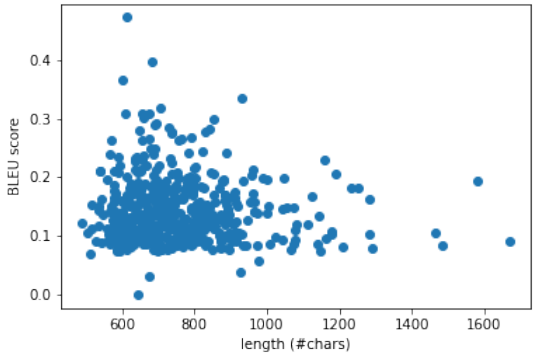


Figure 3.2: length of prompt versus Jaccard score

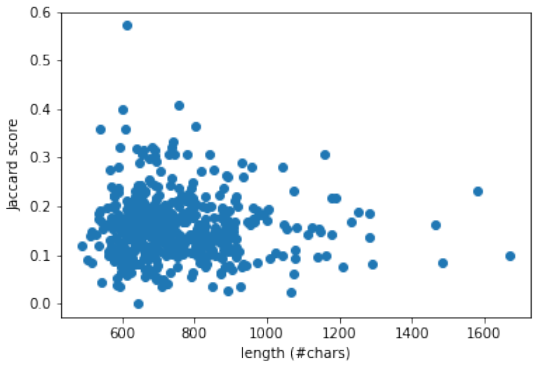


Figure 3.3: length of prompt versus cosine score

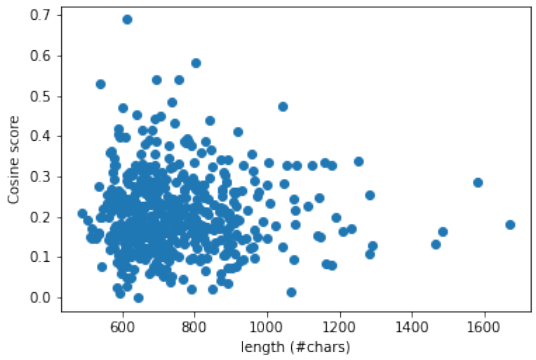
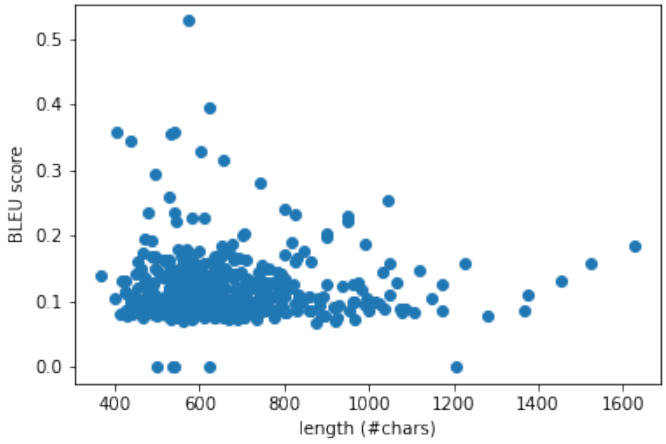
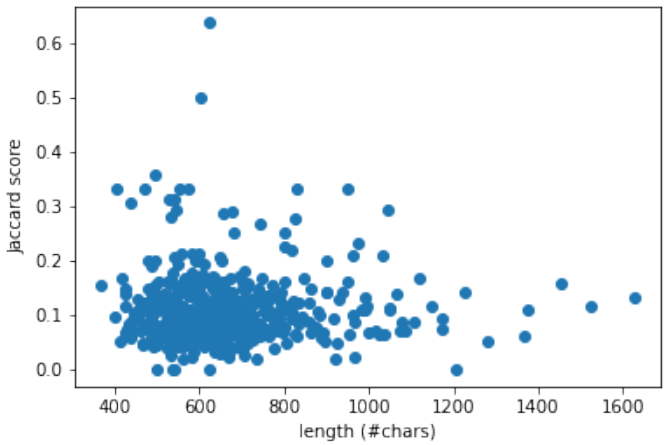


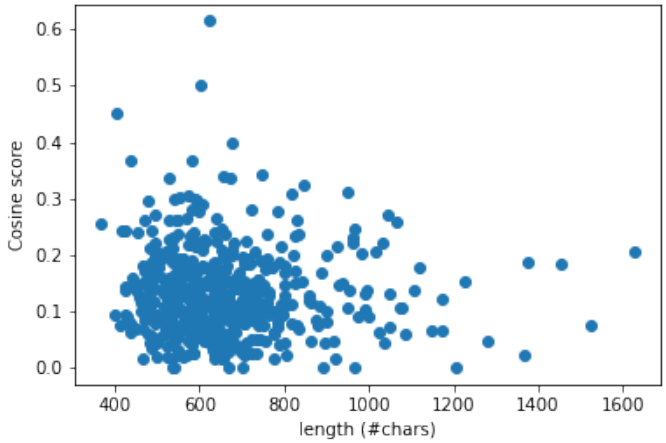
Figure 3.4: Spearman correlation rho and p-values

	Rho	P-value
BLEU	-0.0358	0.4250
Jaccard	-0.0405	0.3660
Cosine	-0.0091	0.8399

Figure 3.5: Prompt: Write a python function to count minimum number of swaps required to convert one binary number represented as a string to another.

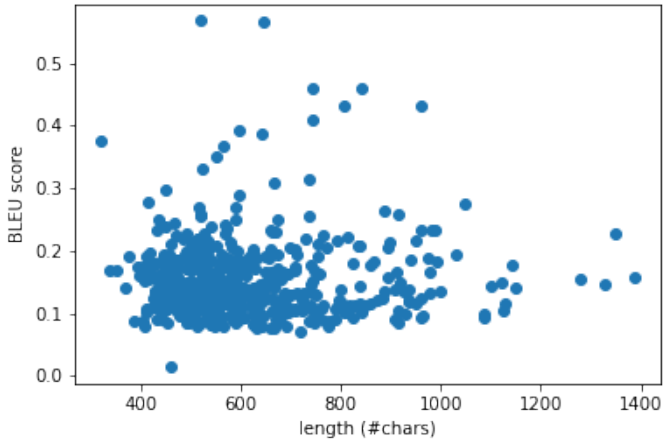


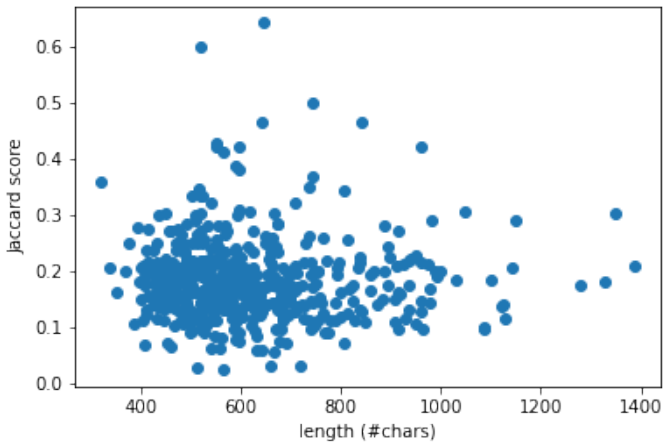


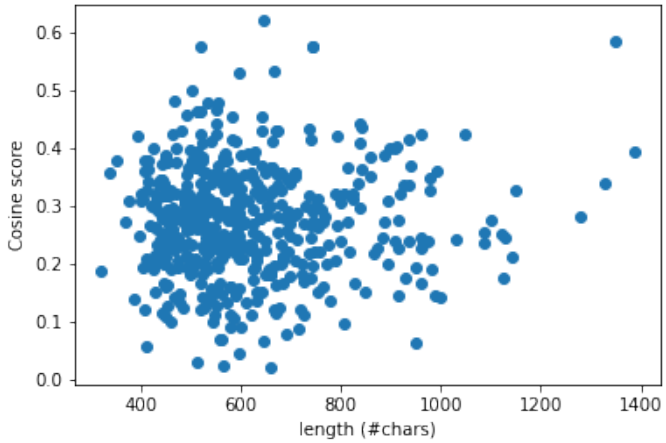


	Rho	P-value
BLEU	-0.0068	0.8801
Jaccard	-0.0511	0.2537
Cosine	-0.0445	0.3203

Figure 3.6: Prompt: Write a function to sort a dictionary by value.







	Rho	P-value
BLEU	-0.0506	0.2591
Jaccard	-0.0801	0.0735
Cosine	-0.0146	0.7448

Figure 3.7: Prompt: write a function to find the third side of a right angled triangle.

3.2 Commentary

In the above graphs we can visually see that there is little to no correlation between the length of the input prompt and the BLEU/Jaccard/cosine similarity metrics. Calculating the Spearman correlation further confirms this, as we get rho values that are very close to zero and p-values higher than the standard cutoff of 0.05. For the purposes of identifying an ideal prompt for GPT to describe code, we conclude that the length of the prompt is probably not related to the overall quality of the output and that another, as yet unknown metric is better suited for this purpose.

One thing to note with the BLEU scores is that, barring a few outliers, the data seems to bottom out at a score of just under 0.1. This is because of the structure of the prompt and the way in which GPT generates completions. Since the code problem descriptions are phrased imperatively (e.g. "write a function to do X"), GPT generates outputs that also follow that sentence structure. As such, because the first part is the same across most of the outputs, even if the rest of the generation is completely different, the BLEU score tends to stay above that floor.

(a) Prompt

```
def centered_hexagonal_number(n): return 3 * n * (n - 1) + 1
The above code is a function to find nth centered hexagonal number.

def find_star_num(n): return (6 * n * (n - 1) + 1)
The above code is a function to find the n'th star number.

def min_Swaps(str1,str2) :
    count = 0
    for i in range(len(str1)) :
        if str1[i] != str2[i] :
            count += 1
            if count % 2 == 0 : return (count // 2)
            else : return ("Not Possible")

The above code
```

(b) Completion

```
shows an interesting way to find minimum number of swaps whose result will be
one of the star numbers.

def nth_star_number(n) :
    if n < 1 : return 0
    dec_part = n % 1
    lower_val = floor(n)
    upper_val = ceil(n) # if n is an integral number if dec_part == 0 :
```

Figure 3.8: Sample completion for task: Write a python function to count minimum number of swaps required to convert one binary number represented as a string to another.

Chapter 4

Discussion

4.1 Results

Previously, we mentioned that length can sometimes be a good indicator of complexity when it comes to designing prompts for GPT to complete. More complex problem descriptions often need more words to express, and so result in longer prompts. However, we can see in our results that length does not have a strong correlation with the quality of the final completion. This indicates that one of our assumptions is possibly wrong or that it is entirely off course: length might not have any connection to the complexity of the prompt at all, or if it does, then the complexity has no bearing on the quality of the prompt. In either case, we would have to find some other quantity associated with each prompt that has a correlation with the quality of the generated output.

Besides the prompt, we could also consider the effect that the obfuscations have on GPT. Certain obfuscations are idiomatic and are commonly used in real-world code like that which GPT was trained on. For example, transforming list comprehensions to loops and back are commonplace. However, converting calls to the map/filter/reduce functions to calls to the `next()` function on an iterator are more complicated and definitely non-idiomatic. Even human programmers would avoid using such a construct and so the code would definitely be

much rarer in training data.

4.2 Future Directions

As such, we can address these issues in a future iteration of this work. Because the main difference we see in the prompts is generally how complex they can be, we would have to find another way to quantify complexity besides length. Another way to predict complexity could be how rare each word is in context, since simple tasks are more common and so would have more common words, whereas something more complex could have words that don't appear as frequently, hence seem to be more complex. By calculating some sort of rarity score, we could then have another metric to correlate with the quality of the final generation.

For the obfuscation portion, we can generate more obfuscations of the code; list comprehensions and other functional programming constructs are not the entirety of Python, and there are potentially more areas in the syntax we can explore to convert into something that is equally as unfamiliar to GPT from the training data, but that is qualitatively more idiomatic and easier to comprehend.