Understanding Javascript Code using Machine Learning

Maria Jaramillo
Computer Science, NYUAD
maria.jaramillo@nyu.edu

Advised by: Yasir Zaki, Moumena Chaqfeh, Riyadh Baghdadi

ABSTRACT

This paper aims to test the accuracy of Github's *CodeXGlue* model, which predicts the purpose of code for its autocomplete functions. To do this, thousands of Stack Overflow questions and highest-voted answers were scrapped and used to train my own transformer model. Then, scripts from ten websites — separated into functions — were used to compare the labels assigned by each model. Results showed that Github's model can be augmented using other sources of data — in addition to code hosted on Github — to train the model.

KEYWORDS

web simplification, Javascript, NLP, machine programming, CodeBERT, transformers

Reference Format:
Maria Jaramillo. 2021. Understanding Javascript Code using Machine Learning. In *NYUAD Capstone Project 2 Reports, Fall 2021, Abu Dhabi, UAE.* 8 pages.

1 INTRODUCTION

The growing amount of publicly available code has motivated research into the use of deep learning to predict the functionality of code functions. Microsoft's and Github's *Copilot* is one of the first AI programmers that helps people write code faster by drawing context from written code and suggesting additions. I aim to replicate and identify areas of improvement in existing predictive deep learning models. Finally, by understanding the functionality of a website's inline scripts, I can also contribute to an existing tool that simplifies Javascript and HTML to increase website speed: JSCleaner.

This report is submitted to NYUAD's capstone repository in fulfillment of NYUAD's Computer Science major graduation requirements.

جامعة نيويورك أبوظبي
NYU ABU DHABI

Capstone Project 2, Fall 2021, Abu Dhabi, UAE

1.1 Objectives

1.1.1 Providing insight on existing models and datasets. *CodeXGlue* is a Code-to-Text model built by Microsoft and Github. In their research, they used an existing transformer model called *CodeBERT* and trained it with code hosted on Github and their respective docstrings. Since their training data consisted of complete, organized, and possibly code-reviewed functions, I wanted to investigate how their model would perform on a more convoluted dataset.

In an effort to train the *CodeBERT* model on code that would most likely be replicated by developers, Stack Overflow data was used. To provide evidence on the performance of Github's model and ours, I asked students to rate its predictions on inline scripts found in ten of the most visited websites. Given that Stack Overflow's code lacks the same completeness and organization as the Github dataset, analyzing the predictions of the existing Github model will provide interesting insight that has not been explored previously. Likewise, the Stack Overflow dataset can complement previous datasets and be used to strengthen future Code-to-Text predictive models.

1.1.2 Contributing to JSCleaner. The growing complexity of mobile web pages has motivated research on the simplification of both Javascript and HTML to increase website speed. JSCleaner promises to declutter Javascript without impacting a website's design or functionality by labeling web page elements as critical or non-critical and then removing the non-critical ones so the browser can load faster. There are particular website elements, like the social media section, that are non-critical to the user. By predicting the functionality of script tags within a website's index file, non-critical elements can be identified before rendering, further improving JSCleaner's accuracy and speed.

Previous work done as a contribution to JSAnalyzer has focused on predicting the functionality of a script based

on the visual output. In other words, students have used computer vision to come up with website labels that will determine whether a script is critical or not. However, I aimed to build a model that could predict results based solely on the code.

1.1.3 Code Similarity Research. In addition, my research complements Jahnae Miller's research, which aims to measure the similarity between the scripts of today's most popular websites [Mil20] . By identifying how much code is copied from Stack Overflow, I can cross-reference similar code snippets and automatically propagate the labels provided by my model. This, in turn, will facilitate JSCleaner's ability to predict whether a script is critical or not.

2 RELATED WORK

2.1 Code2Vec

This project began taking shape after reading about Code2Vec, a neural network for learning distributed representations of code. The model, originally built for Java, takes a snippet of code and predicts the function name. It achieves this by first representing the code as an AST (Abstract Syntax Tree), which produces syntactic paths that can capture patterns. To identify which extracted paths were the most important, Code2Vec uses a novel attention network architecture, a technique that has gained popularity in machine translation. The attention model is able to identify which paths in the AST are the most important. [Kar+17]

The model is also fed a collection of method name embeddings that will produce semantically similar vectors. For example, *bubbleSort* and *sort* can be likely labels for a snippet of code that is sorting. Rather than returning a single label for each snippet, the model returns a list of labels along with their probabilities.

Code2Vec's approach achieves its success through a fairly simple model combined with an evaluation dataset of 14M code snippets. This massive dataset is one of the reasons I did not choose to fine-tune the Code2Vec model to fit my needs. Not only was the model built for Java, but it also used a dataset much bigger than ours. Though there exist some extensions to Code2Vec that include Javascript models, I don't have an equivalent collection of name embeddings that fit my purpose. I cannot use their name embeddings because I am not interested in naming a function but rather producing a more descriptive output that can explain the purpose of the code. Finally, the extensions available for Javascript have a considerable lack of documentation and have not been updated in several years.

2.2 Novel Code Similarity

The Novel Code Similarity System (MISIM) uses a context-aware semantic structure to represent snippets of code and then assigns a similarity score through an algorithm that uses various neural networks. This paper is particularly interesting because it 1) provided a different approach to preprocessing code than using an AST, and 2) it achieved extremely high accuracy in scoring code similarity, the second goal of my research.

The model proposes a context-aware semantic structure, which is different from an AST because it is less dense and therefore reduces the chance of misleading code similarity systems into memorizing syntax. Instead, it uses a code's *meaning* rather than its syntax.

This novel representation is combined with a highly complex architecture of neural networks to produce results up to 43x more accurate than Code2Vec's similarity scoring applications. [Ye+20]

Unfortunately, I was unable to replicate their results because they did not publish their models or the tool to produce the context-aware semantic structures. [Ye+20]

Unfortunately, we are unable to replicate their results because they did not publish their models nor the tool to produce the context-aware semantic structures.

2.3 CodeSearchNet

To motivate researchers to further study the task of using natural language queries to produce code, Github and Microsoft produced a corpus of 2 million functions and their respective natural language descriptions (docstrings, comments). The dataset is a collection of popular repositories in Javascript, Java, Python, PHP, Ruby, and Go. For Javascript, they have a total of 157,988 labeled data points. [Hus+19] I found this dataset to be better than the method name embeddings in Code2Vec because docstrings are more likely to include the name of the website element that they are trying to render. In addition, the dataset used in Code2Vec was not as accessible as this one.

2.4 CodeXGlue

This paper, published by Microsoft researchers, is aimed at fostering machine learning research for program understanding and generation. It includes 14 datasets, along with documentation explaining how to use baseline models such as BERT, GPT, and Encoder-Decoder to achieve a number of different tasks like Clone Detection, Defect Detection, Code-Completion, Code Summarization, and Text-Code Generation. They hope that researchers can fine-tune the models and contribute to the advancement of the field of program understanding by reporting their results. [Lu+21]

Since I aim to produce labels for snippets of code, I found that their Code Summarization task, combined with the CodeSearchNet dataset would be most relevant. Though they don't include the pre-trained model, they outline how to train each individual transformer model using the provided dataset.

2.5 CodeBERT

Since the Code Summarization task from CodeXGlue was most successful with the CodeBERT model, I decided to research this transformer model as well. CodeBERT is a bimodal pre-trained model for programming and natural language (NL-PL). It was developed with a transformer-based neural network, which has recently proven to be more accurate than RNNs and LSTMs because it avoids recursion and instead processes sentences as a whole by learning relationships between words through multi-head attention and positional embeddings. [Fen+20]

3 METHODOLOGY

3.1 Dataset

The research uses three main sources of data: CodeSearchNet, Stack Overflow, and inline scripts from ten websites.

3.1.1 CodeSearchNet. This dataset was used by CodeXGlue to train a model that uses CodeBERT as a baseline. CodeSearchNet is a collection of datasets provided by Github and Microsoft's Deep Understanding Program. It consists of 2 million pairs (code, docstring) obtained from open source repositories on Github.

3.1.2 Stack Overflow. Using the Stackoverflow API, I fetched Javascript questions with specific words in its title. I selected the labels —menu, header, footer, social media, content, navbar, navicon — because they had been previously used in Computer Vision research for JSCleaner. The labels are common parts of a website that can help us understand what parts of the code are doing what. Some of these, like "social media", are usually non-critical to the user. Predicting what part of a website a function of Javascript is responsible for can help us predict if the function is critical or not. The alternative of scraping all the Javascript questions from Stack Overflow was explored, yet this would have taken months due to the request limit of the site.

After requesting Stack Overflow to allow us to query using the maximum rate — 10,000 requests per day — I fetched Javascript questions that had been answered and that had one of the above-mentioned words in the title. Since the *Search API* only returns the `question_id`, I used the *Question API* to fetch the highest-voted answer. Finally, using the `question_id` and the `answer_id`, I used *BeautifulSoup* to scrape the URL and extract the code tags inside the div containing the answer.

Though I requested Stack Overflow questions tagged as Javascript, some of the answers included code in other languages. Thus, I used regex to filter out code written in other languages. Then, I tokenized the Javascript functions using the library *Esprema* and filtered functions that produced less than 8 tokens. This resulted in 20,000 labeled data points.

The *CodeXGlue* Github instructs that in order to train the CodeBERT model, the inputs must be put into a JSONL file with the format:

```
repo: the owner/repo
path: the full path to the original file
func_name: the function or method name
original_string: the raw string before
    tokenization or parsing
language: the programming language
code/function: the part of the
    original_string that is code
code_tokens/function_tokens: tokenized
    version of code
docstring: the top-level comment or
    docstring, if it exists in the
    original string
docstring_tokens: tokenized version of
    docstring
```

Since my dataset does not contain a repo or a path, these fields were left empty. The fetched code was used in two different datasets. The first one is made of code, sentence pairs where the sentence specifies which of the labels is present in the title. For example, "Returns footer" would be a potential label to a function of code where the respective Stack Overflow question is "How to make a footer in React?" The second dataset uses the same snippet of code but paired with the actual question fetched from Stackoverflow.

```
Example:
Dataset 1: ["displayElement (() =>
    getElement();)", "Returns footer"]
Dataset 2: ["displayElement (() =>
    getElement();)", "How to make a
    footer in React?"]
```

The labels exemplified above were placed in the `func_name` and `docstring` fields in the input data.

3.1.3 Inline Scripts. I randomly selected 10 websites and used a Selenium automated web browser to extract the content between each script tag. To avoid scripts belonging to third-party libraries, I only extracted those without the src attribute in the HTML of the index file. Then, I used a library called *Esprima* to separate the code into individual functions

and used those as input in my models. The websites' separated scripts yielded 290 code snippets.

3.2 Training the Model

I trained two different models using the two sets of labeled Stack Overflow data —one set with labels in the format "Returns x" and another with the title of the Stack Overflow question as the label. Using the inline scripts as input, I wanted to compare the predictions of the CodeBERT model trained on Github data (code, docstring pairs) with the model trained with my Stack Overflow dataset. To compare the results of the two models with the CodeXGlue model, I created three different testing datasets as shown in Table 1.

Model	Data	Example Prediction
CodeXGlue	Inline Scripts	Returns lowest number
Stack Overflow Label	Inline Scripts	Returns menu
Stack Overflow Title	Inline Scripts	How can I create a menu with React?

Table 1: The three different models were tested using the inline script functions.

3.3 Rating the predictions

Each of the 290 functions was run through the 3 different models, yielding 870 different predictions. To assess the accuracy of the predictions and compare my models — "Stack Overflow label" and "Stack Overflow question" — versus the CodeXGlue model, I recruited 12 participants and asked them to rate the results on a scale of 1-5 (1 being "extremely inaccurate" and 5 being "extremely accurate"). To recruit these participants, I advertised the position to Junior and Senior computer science students with knowledge of Javascript. I then contacted those participants that met the criteria and invited them to log into a web application.

To collect participants' opinions on the predictions, I built a Web application using a React frontend, NodeJS backend, and Postgres database. The application used a login to authenticate the user and ensured that different groups of users accessed different questions.

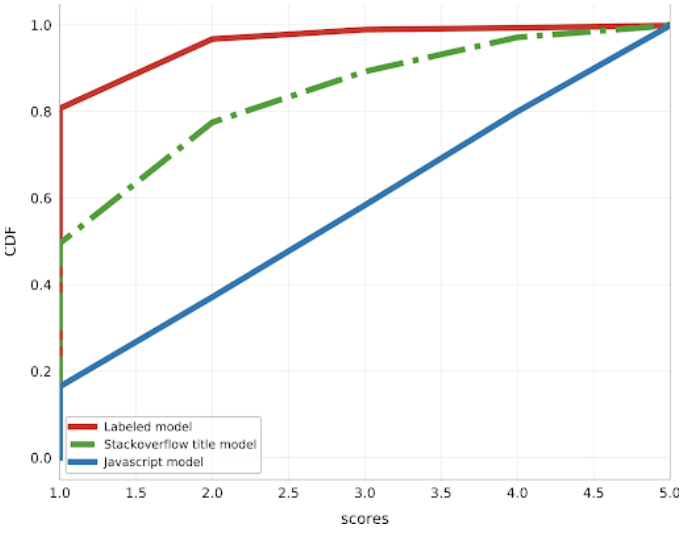
Given the time constraints of the research and the students' time, I aimed to have each prediction reviewed by 3 people, which means that I had to separate the 870 predictions into 4 groups. Thus, each student was assigned 217 labels to rate. Out of the 12 invited participants, 1 was not eligible because they were not an NYUAD student and 3 did not respond to the invitation. Unfortunately, out of the 9 students who responded, 1 responded to less than half of the assigned questions and another one responded to only 7 of the questions.

The lack of complete participation of the students led to an average of only 2 responses to each prediction, leading us to question whether the averaged scores for each response were trustworthy or not.

4 RESULTS

The cumulative distribution of each model's ratings can be seen in Figure 1. The average ratings of each model can be shown in Table 2. To my surprise, the CodeXGlue model averaged only a rating of 3.05. Though my best model averaged a rating of 1.8, it's important to take into account how my model had some serious disadvantages in terms of data: my dataset was 10% the size of CodeXGlue's; my code consisted of imperfect and oftentimes incomplete functions that were published on Stack Overflow, compared to the code-reviewed functions published on Github; and my labels were questions rather than descriptive docstrings.

Figure 1: Figure 1. CDF of ratings per model.



To identify potential differences between the two best models, I compared the lengths of the labels. Perhaps, since the Stack Overflow labels were question titles, they were much wordier and casual than docstrings, which tend to be straightforward. Thus, I started by calculating the average length — measured as the number of words — of the labels in both the *CodeXGlue* model and the *Title* model. I found that the *CodeXGlue* model had labels with an average of 4 words, whereas my model had an average of 9 words. To further

Model	Average Rating
CodeXGlue	3.05
Stack Overflow Label	1.2
Stack Overflow Title	1.8

Table 2: On a scale of 1-5 (1 being the worst and 5 being the best), the CodeXGlue model averaged 3.05, followed by the model trained on Stack Overflow question titles with a score of 1.8.

study the statistical significance of this finding, I ran a linear regression and found that there was a significant, negative relationship between the length of the label and the score, shown in Table 3.

Category	Effect Size	Significance
`length_label`	-0.160441	0.0

Table 3: Linear regression yields a coefficient of -0.160441 and a significance score of 0.0 for the relationship between the length of label and rating.

An example of this issue is shown in Figure 2 where the label predicted by *CodeXGlue* is much shorter and got rated much higher, despite the fact it made little sense.

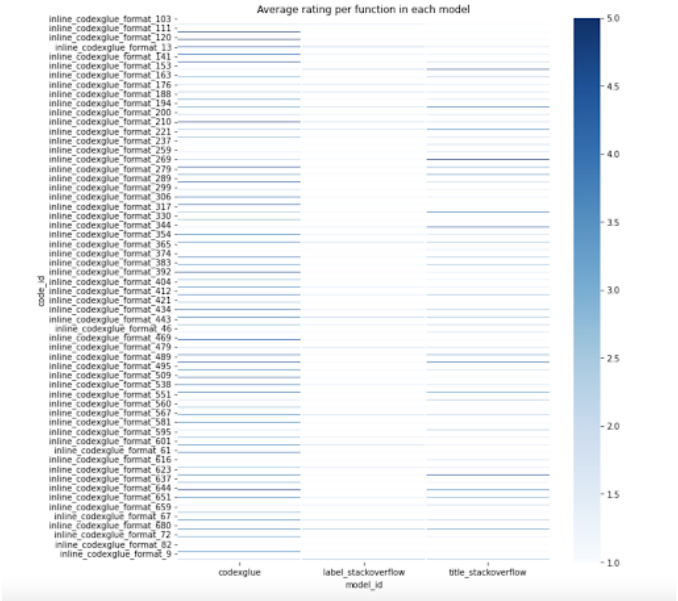
Figure 2: The table shows the ratings given to each of the labels, two per each.

	code	model_id	label	rating
76	el.classList.add('fade')	label_stackoverflow	Returns header and banner	1.0
157	el.classList.add('fade')	codexglue	Fade out the fade out	2.0
201	el.classList.add('fade')	title_stackoverflow	How to make a collapsible menu when scrolling ?	1.0
514	el.classList.add('fade')	label_stackoverflow	Returns header and banner	1.0
592	el.classList.add('fade')	codexglue	Fade out the fade out	3.0
636	el.classList.add('fade')	title_stackoverflow	How to make a collapsible menu when scrolling ?	1.0
2453	el.classList.add('fade')	label_stackoverflow	Returns header and banner	1.0
2531	el.classList.add('fade')	codexglue	Fade out the fade out	3.0
2575	el.classList.add('fade')	title_stackoverflow	How to make a collapsible menu when scrolling ?	1.0

This example also brings to my attention how the *CodeXGlue* prediction does a better job at *matching* words found in the function, regardless of whether they are needed to understand the purpose of the code. This is due to the fact that the Github model is composed of millions of functions, increasing the number of available terms for the model to use. Furthermore, the *CodeXGlue* dataset consisted of any Javascript repository, meaning that the code could have been written for a web service, a command-line program, desktop software, or desktop widgets. On the other hand, my dataset came from questions about web applications, further limiting the corpus of available vocabulary.

From Figure 3, I can make various conclusions about the different ratings given to each function. First, the abundance of light blue in the CodeXGlue column, shows that even though the average rating for the *CodeXGlue* model was 3, there were plenty of low ratings. Furthermore, there are instances where the average rating for the Title model was higher, indicating that a hybrid approach could maximize the results.

Figure 3: The heatmap shows large white areas where none of the models were able to perform accurately. In addition, it shows instances where the Title model outperformed the *CodeXGlue* model.

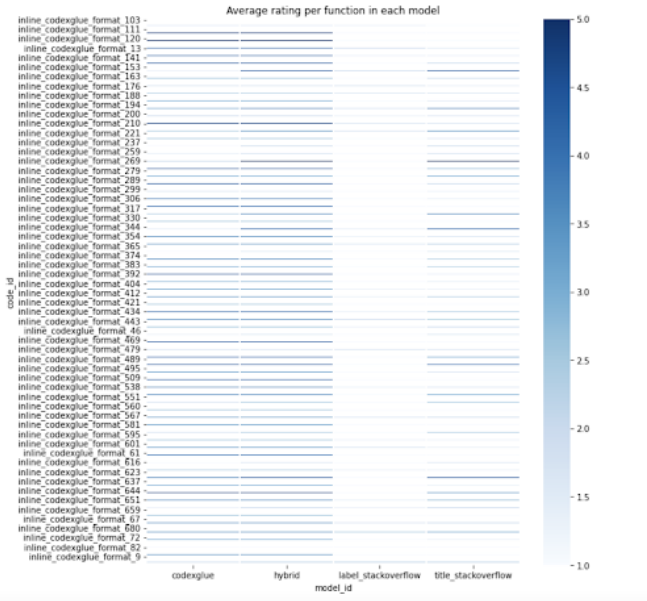


This indicates that there are clear limitations with both models. While *CodeXGlue* can predict a wider variety of words, the *Title* model may sometimes provide a better alternative. If I were to create a hybrid model using *CodeXGlue* and *Title*, it would have an average rating of 3.2. Table 4 shows the improved averages and Figure 4 shows the new heatmap.

Model	Average Rating
CodeXGlue	3.05
Stack Overflow Label	1.2
Stack Overflow Title	1.8
Hybrid	3.2

Table 4: Hybrid model yields an average of 3.2.

Figure 4: Though there are still considerable white spaces, the hybrid model provides an improved alternative.



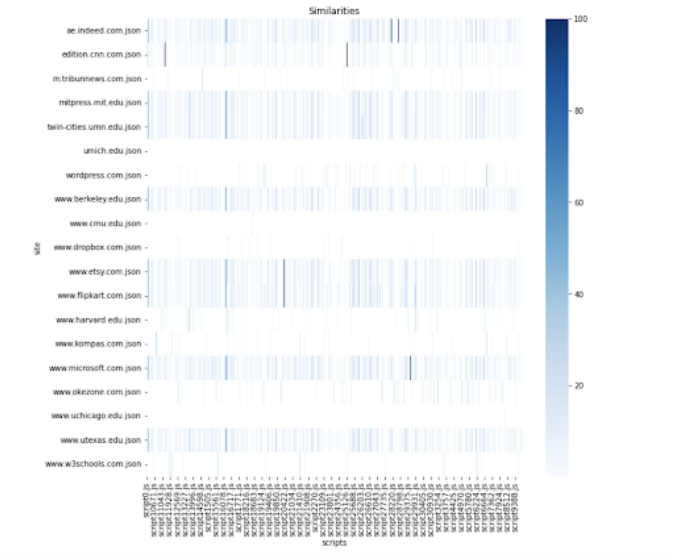
4.1 Measuring similarity

Using previous research [Mil20], I compared the similarities between two of my datasets: Stack Overflow and Inline Scripts. Initially, I ran my *Inline Scripts* dataset through the *Labels* model in the hopes that it could narrow down the number of comparisons each script needed to run. (i.e the Stack Overflow functions where the Label model predicted "Returns header" vs the Inline scripts functions that where the Label model predicted "Returns header). The results, however, could be grouped into one of the following: 1) Either the similarity score was negligible (only about 20% of the inline script matched the Stack Overflow function) 2) The score was higher, but it was mainly due to the fact that it would match extremely common lines of code, such as `for (int i = 0; i < n; i++)`.

Figure 5 shows the distribution of similarity scores between the 10 selected websites and the Stack Overflow scraped functions.

The fact that the similarity scores did not produce significant matches is largely due to two factors. First, the similarity algorithm being used is based on syntactic similarities, rather than semantic [Mil20]. This means that the algorithm will not catch instances where small details were changed (i.e variable names) but the code meaning remained the same. Secondly, many of the functions were extracted from a minified file,

Figure 5: The heatmap shows some columns where there is a consistent, dark blue present. This is because these scripts were composed of generic functions, like `var i;`



where most of the code was changed to make the website load faster. Nevertheless, this invites further exploration of this work. For instance, libraries can be used to unminify the files. In addition, semantic comparisons, though harder to achieve, could yield more meaningful results. Furthermore, perhaps the scripts from smaller, less popular websites are more likely to have code copied from Stack Overflow, given that more Junior developers would be responsible for publishing them and they may have undergone fewer code reviews and edits.

5 CHALLENGES

5.1 Javascript in the context of the web

One of the greatest challenges of labeling Javascript snippets as website elements is that website elements don't depend solely on Javascript. They come to life thanks to HTML and CSS, and are later given functionality with Javascript. Thus, if the aim is to label website elements to determine whether they are critical, training a model with Javascript can result in a lot of useful information being left out, having an impact on the accuracy of the model.

5.2 Javascript as a language

Another challenge of using Javascript is the use of anonymous and asynchronous functions, two of the most popular Javascript features. Including anonymous functions in training data can interfere with a model's ability to produce

similarly distributed vectors between functions, and asynchronous functions can hinder the accuracy of syntactic paths.

5.3 Website elements labeling challenges

Finally, in an attempt to increase the amount of training data, I tried to find additional terms I could look for in the question titles through the *Search API*. To do this, I crawled a significant amount of Javascript questions —without their answers, in order to prevent Stack Overflow from blocking my calls. I removed stopwords, verbs, and other irrelevant words from the new corpus of questions. However, when I managed to identify the most common words in the questions, I only found names of frameworks, which could not be used as labels.

5.4 Stack Overflow Data

One of the greatest challenges of this project was acquiring and making sense of the dataset. First, Stack Overflow has a rate limit of 10,000 requests per day when using the API, and only about 200 if you are scraping directly from their website (*i.e BeautifulSoup*). This meant that simply acquiring my dataset of around 20,000 code snippets took over a month, and attempting to scrape further questions was impossible given the time constraints I was working under. On the other hand, the Github dataset is composed of almost 8 times as many code snippets and was curated by Github researchers for years.

In addition, the data I managed to scrape was composed of question titles (output) and the code tags under the highest-voted accompanying answer (input). Thus, unlike the Github dataset, the input code snippets were not always complete functions, and the labels used to train were questions rather than descriptive docstrings.

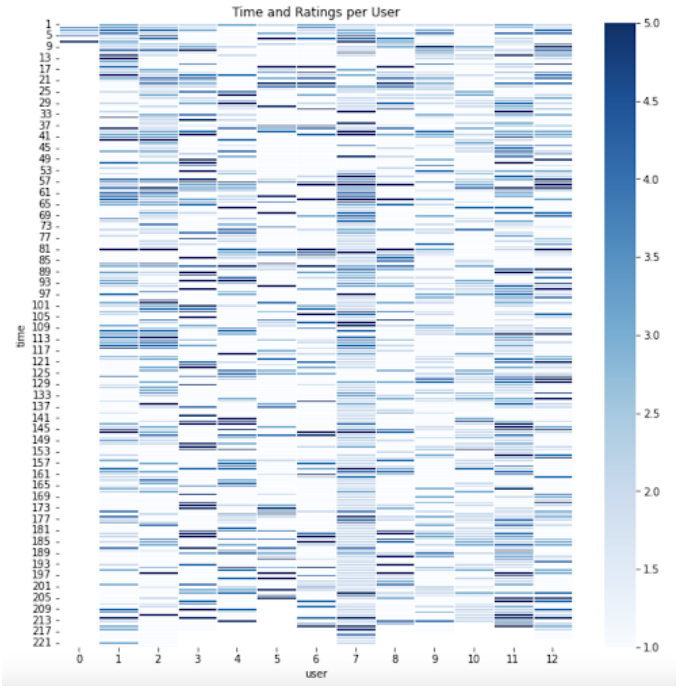
5.5 Data Collection

Due to time constraints, I was unable to test whether the students rating the labels had sufficient Javascript knowledge to understand the code snippets presented. In addition, each student's responses were not normalized to account for those individuals who tend to rate things higher, or lower. Given that each label only has 2 ratings, because of the difficulties getting participants, making conclusive observations is difficult (Figure 6 and 7).

6 DISCUSSION

In my research, I aimed to test current predictive models with the goal of replicating their strengths and facilitating the process of identifying non-critical elements in a website before rendering. To do this, I trained the CodeBERT model (what *CodeXGlue* used) using Stack Overflow's highest-voted

Figure 6: The heatmap shows how some students did not complete all the assigned questions, and how some tend to rate much higher than others. Though only 9 participants confirmed, 3 of them were willing to answer double the questions. Though this helped us get more ratings, it reduces the variety of interpretations, increasing the impact of unnormalized responses.

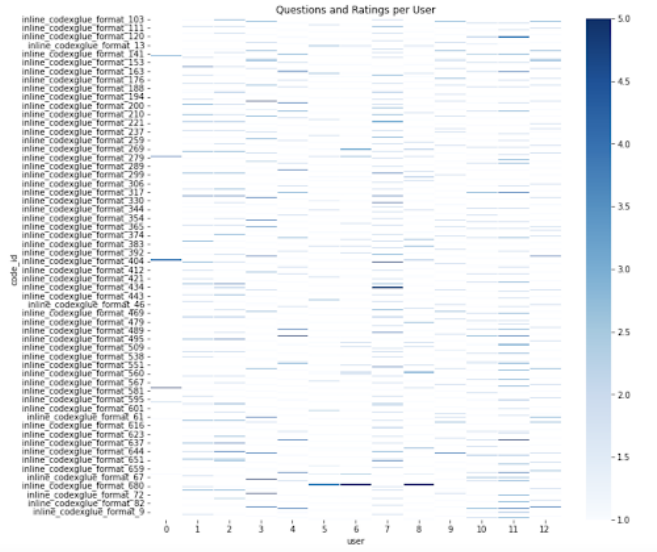


answer code snippets and their respective questions. I then used the inline script tags in ten website index files and separated them into functions. I ran the functions through the models and asked students to rate the predictions of each model.

From my results, I made a number of observations. First, the *CodeXGlue* model scored only an average rating of 3 out of 5, a surprisingly low result given the millions of code snippets it was trained on. The low score could be attributed to the fact that the testing input was imperfect. Since the testing data came from entire scripts separated using a Javascript library, some of them were very short (only one or two lines). The *Github* model, however, was trained using Github functions, which were most likely complete.

This observation encourages a hybrid approach that would benefit from the strengths of each model. Since my model was trained on incomplete functions as well — Stack Overflow answers — it can offer an advantage when faced with ambiguity. On the other hand, the *CodeXGlue* model offers a wider variety of words, while keeping the labels more concise

Figure 7: The heatmap shows how each label was only rated twice, and some of the ratings vary greatly. Someone may rate a label with a 4.0, while another person may give a 2.0.



and easy to understand, as shown in the negative correlation between label length and rating. Furthermore, in the future, when more research on semantic analysis of code has been done, both similarity research and code understanding can be re-explored. Through augmenting the *CodeXGlue* model, we can contribute to some of the most groundbreaking research in programming with natural language.

REFS

[Kar+17] David Kartchner et al. "Code2Vec: Embedding and Clustering Medical Diagnosis Data". In: *2017 IEEE International Conference on Healthcare Informatics (ICHI)*. 2017, pp. 386–390. DOI: 10.1109/ICHI.2017.94.

[Hus+19] Hamel Husain et al. "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search". In: *CoRR* abs/1909.09436 (2019). arXiv: 1909.09436. URL: http://arxiv.org/abs/1909.09436.

[Fen+20] Zhangyin Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP 2020, Online Event, 16-20 November 2020*. Ed. by Trevor Cohn, Yulan He, and Yang Liu. Association for Computational Linguistics, 2020, pp. 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139. URL: https://doi.org/10.18653/v1/2020.findings-emnlp.139.

[Mil20] Jahnae Miller. "An Empirical Comparison of Code Similarity Algorithms". In: *NYUAD Capstone Project 2 Reports*. 2020, p. 1.

[Ye+20] Fangke Ye et al. "MISIM: An End-to-End Neural Code Similarity System". In: vol. abs/2006.05265. 2020. arXiv: 2006.05265. URL: https://arxiv.org/abs/2006.05265.

[Lu+21] Shuai Lu et al. "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation". In: *CoRR* abs/2102.04664 (2021). arXiv: 2102.04664. URL: https://arxiv.org/abs/2102.04664.