Muhammad Haseeb 20100192
Aafaq Sabir 20100196

NLP: Project Literature Review

Research Topic: Using Natural Language Processing Principles for Categorizing Modern Web Pages JavaScript code

Abstract: Webpages of modern age are loaded with scripts. These scripts perform a variety of tasks from essential tasks such as data loading to non essentials ones such as animations or loading ads etc. These scripts claim system resources to run which causes the devices (specially low end devices) to become slow and deteriorate their performance of web surfing. We propose a solution that analyses the scripts in a webpage using using simple language model to classify scripts into 12 predefined categories so that we can remove nonessential ones to make the webpage ligther inorder to improve user experience of web surfing.

Introduction and Motivation: Determining the functionality of a program without actually running it can enable us to get the sense of the program as we read human readable text. This can bridge the gap between natural languages and code that can only be understood by computer after converting it to machine code. It has many practical applications like program classification, clone detection, code suggestions etc. It is an old and well researched problem with many solutions covering a wide variety of methods. Many solutions use ASTs (abstract syntax trees) of code to match it with similar programs because ASTs provide an abstract level view of a program and a seemingly different program with the similar functionality may have a similar AST. But this technique requires some knowledge of the program and its programming language. Which becomes a limitation. We propose a technique to categorize JavaScript code in web pages into subclasses so that we can remove non essential parts of the codebase of a website. We want to achieve this by treating code as natural language so that it can be trained on any program.

Research Questions:
RQ1: Does a piece of code have similar properties as natural language?
RQ2: Can the techniques of natural language work on code in the similar fashion?
RQ3: Since we want to classify JavaScript code on web pages, how well we can classify javascript code using NLP approach?

Following are some of the papers that we studied so get an idea of how language models are being implemented for code, for what applications they are useful and how

well every technique performs. The methodology that we are going to suggest is inspired by the work of these papers.

We used the following method to analyze (select or reject) the papers based on this process:

Step 1: If keywords are available, look at relevant keywords and choose if they are relevant.

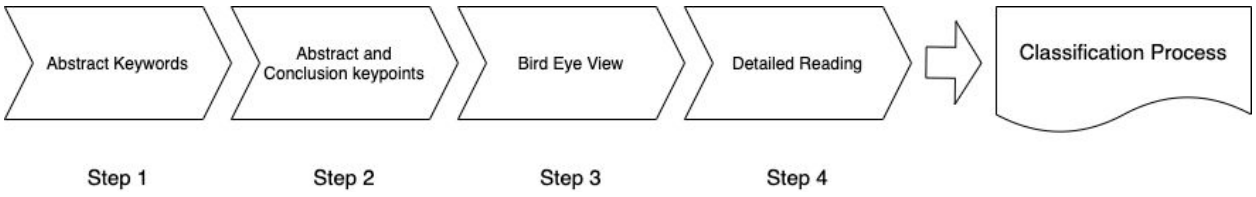
Step 2: Read the abstract and conclusion to understand what the goal of the paper was. If the goal was something other than using ML/NLP on JS, then we excluded that paper.

Step 3: Take a bird eye view of the paper and see the techniques used and validation methods.

Step 4: For the papers that are filtered out till this step are read in detail and then classified in terms of contributions, research facets and identifying gaps.

Step 5: After analyzing the papers in detail, we classified the papers into the broader categories of techniques that different papers used.

The following diagram illustrates the process of literature review:



Related papers that helped us analyzing different methods for code classification or unmasking its function:

1. **Source:** https://dl.acm.org/doi/10.5555/3045118.3045344
   Title: Bimodal modelling of source code and natural language

   They considered the problem of building probabilistic models that jointly model short natural language utterances and source code snippets. The aim was to bring together recent work on statistical modelling of source code and work on bimodal models of images and natural language. The resulting models are useful for a variety of tasks that involve natural language and source code. They demonstrated their performance on two retrieval tasks: retrieving source

code snippets given a natural language query, and retrieving natural language descriptions given a source code query (i.e., source code captioning). Experiments show there to be promise in this direction, and that modelling the structure of source code improves performance.

2. **Source:** https://dl.acm.org/doi/10.1145/3192366.3192412
   Title: A general path-based representation for predicting program properties

This paper presented a general path-based representation for learning from programs. Our representation is purely syntactic and extracted automatically. Its main idea is to represent a program using paths in its abstract syntax tree (AST). This allows a learning model to leverage the structured nature of code rather than treating it as a flat sequence of tokens.
They showed that this representation is general and can: (i) cover different prediction tasks, (ii) drive different learning algorithms (for both generative and discriminative models), and (iii) work across different programming languages. They also evaluated their approach on the tasks of predicting variable names, method names, and full types. They uses their representation to drive both CRF-based and word2vec-based learning, for programs of four languages: JavaScript, Java, Python and C#. Our evaluation shows that our approach obtains better results than task-specific handcrafted representations across different tasks and programming languages.

3. **Source:** http://proceedings.mlr.press/v48/bielik16.html
   Title: PHOG: Probabilistic Model for Code

The paper introduced a new generative model for code called probabilistic higher order grammar (PHOG). PHOG generalizes probabilistic context free grammars (PCFGs) by allowing conditioning of a production rule beyond the parent non-terminal, thus capturing rich contexts relevant to programs. Even though PHOG is more powerful than a PCFG, it can be learned from data just as efficiently. They also trained a PHOG model on a large JavaScript code corpus and showed that it is more precise than existing models, while similarly fast. As a result, PHOG can immediately benefit existing programming tools based on probabilistic models of code.

4. **Source:** https://dl.acm.org/doi/10.1145/2491411.2491458
   Title: A statistical semantic language model for source code

Recent research has successfully applied the statistical n-gram language model to show that source code exhibits a good level of repetition. The n-gram model is shown to have good predictability in supporting code suggestion and completion. However, the state-of-the-art n-gram approach to capture source code regularities/patterns is based only on the lexical information in a local context of the code units. To improve predictability, this paper introduces SLAMC, a novel statistical semantic language model for source code. It incorporates semantic information into code tokens and models the regularities/patterns of such semantic annotations, called sememes, rather than their lexemes. It combines the local context in semantic n-grams with the global technical concerns/functionality into an n-gram topic model, together with

pairwise associations of program elements. Based on SLAMC, they developed a new code suggestion method, which is empirically evaluated on several projects to have relatively 18-68% higher accuracy than the state-of-the-art approach.

From this paper, we got the idea of using sequences (n-grams) rather than simple keywords in our vocabulary. However, then we also needed to figure out the optimal number of n in n-grams which is explained in our methodology. After having simple keywords and sequences as well in our vocabulary, the size of vocabulary inflated too much (2600) so we devised some methods for reducing this feature-set which is also explained in our methodology.

5. **Source:** https://dl.acm.org/doi/10.1145/2983990.2984041
   Title: Probabilistic model for code with decision trees
   In this paper they introduce a new approach for learning precise and general probabilistic models of code based on decision tree learning. Their approach directly benefits an emerging class of statistical programming tools which leverage probabilistic models of code learned over large codebases (e.g., GitHub) to make predictions about new programs (e.g., code completion, repair, etc). The key idea is to phrase the problem of learning a probabilistic model of code as learning a decision tree in a domain specific language over abstract syntax trees (called TGen). This allows the prediction of a program element in a dynamically computed context. Further, the problem formulation enables them to easily instantiate known decision tree learning algorithms such as ID3, but also to obtain new variants they referred to as ID3+ and E13, not previously explored and ones that outperform ID3 in prediction accuracy. The suggested approach is general and can be used to learn a probabilistic model of any programming language. They also implemented our approach in a system called Deep3 and evaluated it for the challenging task of learning probabilistic models of JavaScript and Python. Our experimental results indicate that Deep3 predicts elements of JavaScript and Python code with precision above 82% and 69%, respectively.

6. **Source:** https://dl.acm.org/doi/10.1145/2775051.2677009
   Topic: Predicting Program Properties from "Big Code"
   This paper presents a new approach for predicting program properties from massive codebases (aka "Big Code"). Their approach first learns a probabilistic model from existing data and then uses this model to predict properties of new, unseen programs. The key idea of our work is to transform the input program into a representation which allows us to phrase the problem of inferring program properties as structured prediction in machine learning. This formulation enables to leverage powerful probabilistic graphical models such as conditional random fields (CRFs) in order to perform joint prediction of program properties. As an example of this approach, they built a scalable prediction engine called JSNice for solving two kinds of problems in the context of JavaScript: predicting (syntactic) names of identifiers and predicting (semantic) type annotations of variables. Experimentally, JSNice predicts correct names

for 63% of name identifiers and its type annotation predictions are correct in 81% of the cases.

7. **Source:** https://dl.acm.org/doi/abs/10.1145/2382196.2382274

   Title: You are what you include: large-scale evaluation of remote javascript inclusions

   JavaScript is used by web developers to enhance the interactivity of their sites, offload work to the users' browsers and improve their sites' responsiveness and user-friendliness, making web pages feel and behave like traditional desktop applications. An important feature of JavaScript, is the ability to combine multiple libraries from local and remote sources into the same page, under the same namespace. While this enables the creation of more advanced web applications, it also allows for a malicious JavaScript provider to steal data from other scripts and from the page itself. Today, when developers include remote JavaScript libraries, they trust that the remote providers will not abuse the power bestowed upon them. In this paper, they reported on a large-scale crawl of more than three million pages of the top 10,000 Alexa sites, and identify the trust relationships of these sites with their library providers. They showed the evolution of JavaScript inclusions over time and developed a set of metrics in order to assess the maintenance-quality of each JavaScript provider, showing that in some cases, top Internet sites trust remote providers that could be successfully compromised by determined attackers and subsequently serve malicious JavaScript. In this process, they identify four, previously unknown, types of vulnerabilities that attackers could use to attack popular web sites. Lastly, they reviewed some proposed ways of protecting a web application from malicious remote scripts and show that some of them may not be as effective as previously thought.

8. **Source:**https://www.usenix.org/legacy/events/sec11/tech/full_papers/Curtsinger.pdf

   Topic: ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection

   JavaScript malware-based attacks account for a large fraction of successful mass-scale exploitation happening today. Attackers like JavaScript-based attacks because they can be mounted against an unsuspecting user visiting a seemingly innocent web page. While several techniques for addressing these types of exploits have been proposed, in-browser adoption has been slow, in part because of the performance overhead these methods incur. In this paper, they propose ZOZZLE, a low-overhead solution for detecting and preventing JavaScript malware that is fast enough to be deployed in the browser. Our approach uses Bayesian classification of hierarchical features of the JavaScript abstract syntax tree to identify syntax elements that are highly predictive of malware. Our experimental evaluation shows that ZOZZLE

is able to detect JavaScript malware through mostly static code analysis effectively. ZOZZLE has an extremely low false positive rate of 0.0003%, which is less than one in a quarter million. Despite this high accuracy, the ZOZZLE classifier is fast, with a throughput of over one megabyte of JavaScript code per second.

9. **Source:** https://hovav.net/ucsd/dist/jspriv.pdf
   Topic: Fingerprinting Information in JavaScript Implementations
   In this paper, they identified two new avenues for browser fingerprinting. The new fingerprints arise from the browser's JavaScript execution characteristics, making them difficult to simulate or mitigate in practice. The first uses the innate performance signature of each browser's JavaScript engine, allowing the detection of browser version, operating system and microarchitecture, even when traditional forms of system identification (such as the user-agent header) are modified or hidden. The second subverts the whitelist mechanism of the popular NoScript Firefox extension, which selectively enables web pages' scripting privileges to increase privacy by allowing a site to determine if particular domains exist in a user's NoScript whitelist. They have experimentally verified the effectiveness of our system fingerprinting technique using a 1,015-person study on Amazon's Mechanical Turk platform.

10. **Source:** https://dl.acm.org/doi/abs/10.1145/1062455.1062491
    Title: Using structural context to recommend source code examples
    When coding to a framework, developers often become stuck, unsure of which class to subclass, which objects to instantiate and which methods to call. Example code that demonstrates the use of the framework can help developers make progress on their task. In this paper, they described an approach for locating relevant code in an example repository that is based on heuristically matching the structure of the code under development to the example code. Our tool improves on existing approaches in two ways. First, the structural context needed to query the repository is extracted automatically from the code, freeing the developer from learning a query language or from writing their code in a particular style. Second, the repository can be generated easily from existing applications. They demonstrated the utility of this approach by reporting on a case study involving two subjects completing four programming tasks within the Eclipse integrated development environment framework.

11. **Source:** https://ieeexplore.ieee.org/abstract/document/738528
    Title: Clone Detection Using Abstract Syntax Trees
    This paper presents simple and practical methods for detecting exact and near miss clones over arbitrary program fragments in program source code by using abstract syntax trees. Previous work also did not suggest practical means for removing detected

clones. Since our methods operate in terms of the program structure, clones could be removed by mechanical methods producing in-lined procedures or standard preprocessor macros.

12. **Source:** https://dl.acm.org/doi/abs/10.1145/1900008.1900143
    Title: Code template inference using language models

This paper investigates the use of a natural language processing technique that automatically detects project-specific code templates (i.e., frequently used code blocks), which can be made available to software developers within an integrated development environment. During software development, programmers often and in some cases unknowingly rewrite the same code block that represents some functionality. These frequently used code blocks can inform the existence and possible use of code templates. Many existing code editors support code templates, but programmers are expected to manually define these templates and subsequently add them as templates in the editor. Furthermore, the support of editors to provide templates based on the editing context is still limited. The use of n-gram language models within the context of software development is described and evaluated to overcome these restrictions. The technique can search for project-specific code templates and present these templates to the programmer based on the current editing context.

Then finally we came across this paper that gave us the right direction to work on:

Source:http://paginaspersonales.deusto.es/isantos/papers/2016/2016-TR-WebTrackingAnalysis.pdf
Title: Known and Unknown Generic Web Tracking Analyzer: A 1 Million Websites Study
Summary: Web tracking is a widespread technique on the Internet to gather user data. While tracking may not pursue user damage, it may violate user consent and privacy. In addition, some recent reports have linked web tracking with targeted malware campaigns. Recent research studied well-known and advanced tracking techniques. Despite the fact that these works improved the understanding of the current tracking landscape, they were not intended to generically detect and understand all types of web tracking techniques. In this paper, the authors present the first general large-scale analysis of different known and unknown web tracking scripts on the Internet to understand its current ecosystem and behavior. To this end, they implemented TRACKINGINSPECTOR the first automatic method to detect generic web tracking scripts. This technique automatically retrieves scripts of a website and, through code similarity and machine learning, detects modifications of known tracking scripts and discovers unknown web tracking script candidates. TRACKINGINSPECTOR analyzed the Alexa top visited 1,000,000 websites, computing the tracking prevalence and its

ecosystem, as well as the influence of hosting, website category, and website reputation. More than 90% websites performed some sort of tracking and more than 50% scripts were used for web tracking. Over 2,000,000 versions of known tracking scripts were discovered and they also examined the script renaming techniques they used to avoid blacklists. In addition, 5,500,000 completely unknown likely tracking scripts were found, including more than 700 new different potential device fingerprinting (canvas and font probing) unique scripts. Our system also automatically detected the fingerprinting behavior of a previously reported targeted *fingerprinting-driven malware* campaign in two different websites not previously documented.

We implemented this technique and found the accuracy around 60%. Then we analysed the literature review again and found these two papers so we tried these techniques as well in order to get better results. After analyzing these papers, we came across most promising methods of inferring code functionality.

Source: **https://arxiv.org/pdf/1803.09473.pdf**
Title: code2vec: Learning Distributed Representations of Code
Summary: The authors presented a neural model for representing snippets of code as continuous distributed vectors (“code embeddings”). The main idea is to represent a code snippet as a single fixed-length code vector, which can be used to predict semantic properties of the snippet. This is performed by decomposing code to a collection of paths in its abstract syntax tree, and learning the atomic representation of each path simultaneously with learning how to aggregate a set of them. They demonstrate the effectiveness of our approach by using it to predict a method’s name from the vector representation of its body. They also evaluate our approach by training a model on a dataset of 14M methods. They showed that code vectors trained on this dataset can predict method names from files that were completely unobserved during training. Furthermore, they show that our model learns useful method name vectors that capture semantic similarities, combinations, and analogies. Comparing previous techniques over the same data set, our approach obtains a relative improvement of over 75%, being the first to successfully predict method names based on a large, cross-project, corpus.

From the same authors of code2vec, another papers proposed another technique:
Source: **https://arxiv.org/pdf/1808.01400.pdf**
Title: CODE2SEQ: Generating sequences from structured representations of code
Summary: The ability to generate natural language sequences from source code snippets has a variety of applications such as code summarization, documentation, and retrieval. Sequence-to-sequence (seq2seq) models, adopted from neural machine

translation (NMT), have achieved state-of-the-art performance on these tasks by treating source code as a sequence of tokens. They presented CODE2SEQ: an alternative approach that leverages the syntactic structure of programming languages to better encode source code. Our model represents a code snippet as the set of compositional paths in its abstract syntax tree (AST) and uses attention to select the relevant paths while decoding. They also demonstrated the effectiveness of our approach for two tasks, two programming languages, and four datasets of up to 16M examples. Our model significantly outperforms previous models that were specifically designed for programming languages, as well as state-of-the-art NMT models.

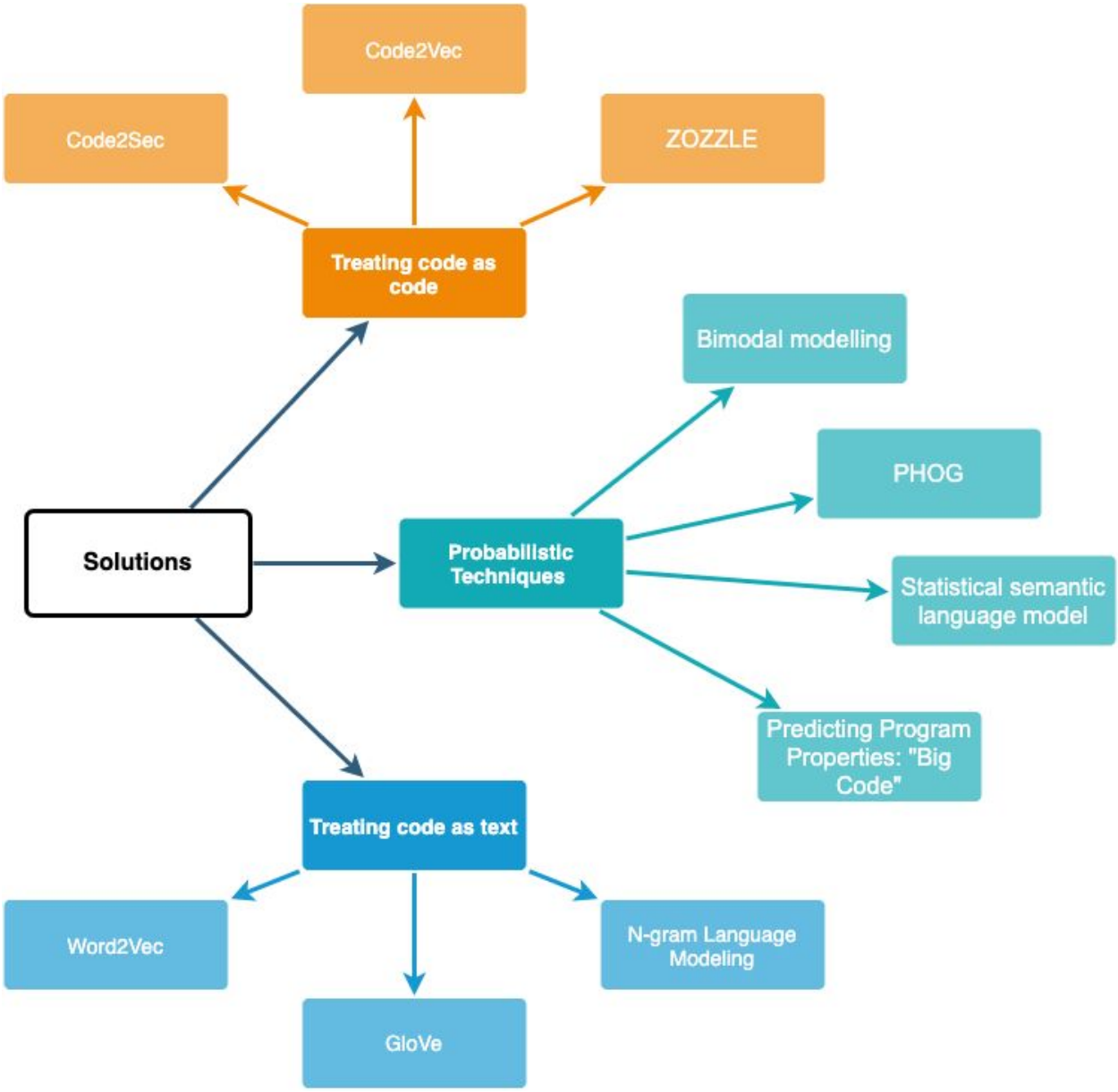
Now we map the paper of literature review to the RQs in the following table:

RQs	**Paper(s)**
1. Does a piece of code have similar properties as natural language?	1. **Source:** https://dl.acm.org/doi/10.5555/3045118.3045344 **Title:** Bimodal modelling of source code and natural language 2. **Source:** http://proceedings.mlr.press/v48/bielik16.html **Title:** PHOG: Probabilistic Model for Code 3. **Source:** https://dl.acm.org/doi/10.1145/2491411.2491458 **Title:** A statistical semantic language model for source code
2. Can the techniques of natural language work on code in a similar fashion?	1. **Source:** https://dl.acm.org/doi/10.1145/3192366.3192412 **Title:** A general path-based representation for predicting program properties

	2. **Source:** https://dl.acm.org/doi/10.1145/2983990.2984041 **Title:** Probabilistic model for code with decision trees 3. **Source:** https://dl.acm.org/doi/10.1145/2775051.2677009 **Topic:** Predicting Program Properties from "Big Code" 4. **Source:** https://dl.acm.org/doi/abs/10.1145/1062455.1062491 **Title:** Using structural context to recommend source code examples 5. **Source:** https://dl.acm.org/doi/abs/10.1145/1900008.1900143 **Title:** Code template inference using language models
3. Since we want to classify JavaScript code on web pages, how well we can classify javascript code using NLP approach?	1. **Source:** https://dl.acm.org/doi/abs/10.1145/2382196.2382274 **Title:** You are what you include: large-scale evaluation of remote javascript inclusions 2. **Source:** https://hovav.net/ucsd/dist/jspriv.pdf **Topic:** Fingerprinting Information in JavaScript Implementations

	3. **Source:** https://ieeexplore.ieee.org/abstract/document/738528 **Title:** Clone Detection Using Abstract Syntax Trees 4. **Source:**http://paginaspersonales.deusto.es/isantos/papers/2016/2016-TR-WebTrackingAnalysis.pdf **Title:** Known and Unknown Generic Web Tracking Analyzer: A 1 Million Websites Study 5. **Source:** **https://arxiv.org/pdf/1803.09473.pdf** **Title:** code2vec: Learning Distributed Representations of Code 6. **Source:** **https://arxiv.org/pdf/1808.01400.pdf** **Title:** CODE2SEQ: Generating sequences from structured representations of code

While analyzing the paper we shifted out attention to the methods used in every paper and that helped us to develop the methodology that we proposed. The following diagram illustrates the methods used by paper in literature review to solve the underlying problem:



Conclusion: The method that we finally decided to implement is a lighter version of Code2Vec which is a novel way of using word2vec for code developed by Facebook. It uses the principles of word2vec functionality but adapts them to work on programming languages. Before implementing this model, we have also implemented other models e.g. TF-IDF based classifiers so we are also going to present them in our final report. At the end, after modeling this code2vec, we are planning to use LSTM networks for finding out whether our accuracy will further improve or not. However, LSTM modeling will be contingent on the time we have after code2vec modeling.