

Fall 12-17-2021

The Impact of Programming Language's Type on Probabilistic Machine Learning Models

Sherif Elsaid
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects



Part of the [Artificial Intelligence and Robotics Commons](#), and the [Programming Languages and Compilers Commons](#)

Recommended Citation

Elsaid, Sherif, "The Impact of Programming Language's Type on Probabilistic Machine Learning Models" (2021). *Master's Projects*. 1050.
https://scholarworks.sjsu.edu/etd_projects/1050

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

The Impact of Programming Language's Type on Probabilistic Machine Learning
Models

A Project
Presented to
The Faculty of the Department of Computer Science
San Jose State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Sherif Elsaid
December 2021

© 2021

Sherif Elsaid

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

The Impact of Programming Language's Type on Probabilistic Machine Learning
Models

by

Sherif Elsaid

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2021

Katerina Potika Department of Computer Science

Mark Stamp Department of Computer Science

Thomas Austin Department of Computer Science

ABSTRACT

The Impact of Programming Language's Type on Probabilistic Machine Learning Models

by Sherif Elsaid

Software development is an expensive and difficult process. Mistakes can be easily made, and without extensive review process, those mistakes can make it to the production code and may have unintended disastrous consequences.

This is why various automated code review services have arisen in the recent years. From AWS's CodeGuro and Microsoft's Code Analysis to more integrated code assistants, like IntelliCode and auto completion tools. All of which are designed to help and assist the developers with their work and help catch overlooked bugs.

Thanks to recent advances in machine learning, these services have grown tremendously in sophistication to a point where they can catch bugs that often go unnoticed even with traditional code reviews.

This project investigates the use of code2vec [1], which is a probabilistic machine learning model on source code, in correctly labeling methods from different programming language families. We extend this model to work with more languages, train the created models, and compare the performance of static and dynamic languages.

As a by-product we create new datasets from the top stared open source GitHub projects in various languages. Different approaches for static and dynamic languages are applied, as well as some improvement techniques, like transfer learning. Finally, different parsers were used to see their effect on the model's performance.

ACKNOWLEDGMENTS

I would like to express my deepest appreciation and thanks to my project advisor Prof. Katerina Potika, whose help and support through out the length of the two semesters was vital for the successful completion of this project. Prof. Potika shard her invaluable knowledge and guidance with me as she always do with all of her students.

I would also like to thank the committee members, Prof. Thomas Austin and Prof. Mark Stamp for their help and support, as well as their valuable feedback and time.

Finally, I would like to thank my parents for their continuous support throughout my academic journey.

TABLE OF CONTENTS

CHAPTER

1	Introduction	1
1.1	Terminology	1
1.2	Big Code	2
1.3	Motivation	3
1.4	Main Challenges	4
1.5	Overview	5
1.6	Historical Background	5
1.7	Current Research Areas	6
1.8	Research Questions	7
1.9	Contributions	8
1.10	Chapters Overview	9
2	Related Work	10
2.1	Statically and Dynamically typed Languages	10
2.2	Deep Learning on Source Code	12
2.3	MLonCode Applications	14
2.4	code2vec Applications	15
3	Technical Background	17
3.1	Abstract Syntax Trees (AST)	17
3.1.1	AST Path	18
3.1.2	Path Context	19

3.2	PathMiner A Library for Mining of Path-Based Representations of Code	20
3.2.1	Astminer Tool	21
3.3	Code2Vec	24
3.3.1	Code Embeddings	24
3.3.2	Context Path Example	26
3.3.3	Code2vec Architecture	28
4	Datasets	29
4.1	Datasets Size	29
4.2	Data Preprocessing	30
4.3	Dataset Split	32
4.4	JavaScript Dataset	33
5	Experiments	34
5.1	Requirements	34
5.2	Hardware	35
5.3	Workflow Pipeline	35
5.4	Feature Extraction	36
5.4.1	Path contexts extraction	37
5.4.2	Preprocessing the path contexts files	38
5.5	Feature Extraction Results	38
5.5.1	Observations about the feature extraction results	39
5.5.2	Notes about JavaScript	41
5.6	Hyper-parameters	41
5.6.1	Batch Size	42

5.6.2	DropOut Rate	42
5.6.3	Number of Kwords Ratio	43
5.6.4	Context Paths Length and Width	43
5.6.5	Number of Epochs	45
5.6.6	Number of Contexts	46
5.6.7	Hyper-parameters Used	46
5.7	Model Evaluation	48
6	Results	49
6.1	Precision, Recall, F1-Score, and Accuracy	49
6.2	Results	50
6.2.1	Training Results	50
6.2.2	Testing Results	51
6.3	Transfer Learning	54
6.4	Different Parsers	56
6.4.1	GumTree	56
6.5	Python Large	58
6.5.1	Python Datasets Comparison	59
6.5.2	Baseline Comparison	60
7	Conclusion and Future Work	62
7.1	Summary	62
7.2	Conclusion	62
7.3	Future Work	63
	LIST OF REFERENCES	64

APPENDIX

..... 67

LIST OF TABLES

1	Source Code Datasets	32
2	Feature extraction for Training data	39
3	Feature extraction for Validation data	40
4	Feature extraction for Testing data	40
5	The results from training the models from scratch	51
6	The results from the testing dataset	52
7	The results from fine-tuning the models -Transfer learning from the Java 14m trained model-	54
8	Feature extraction for Python Dataset using GumTree Parser	57
9	The results from training the Python model from scratch using GumTree	57
10	The results from training the Python model from scratch using the python large dataset	59
11	Python Datasets	60

LIST OF FIGURES

1	AST example from PathMiner [2]	18
2	PathMiner Architecture Paper [2]	23
3	A code snippet and its predicted labels as computed by our model as demonstrated in [1].	24
4	Sample method [1]	26
5	Top 4 paths in the method below where the width of each colored path is proportional to the attention it was given (red 1 : 0.23, blue 2 : 0.14, green 3 : 0.09, orange 4 : 0.07) [1].	27
6	Top predicted results [1].	27
7	The path attention network architecture [1]	28
8	Feature Extraction and Model Training Pipeline/Workflow	36
9	Batch Size Hyper-Parameter Comparison	42
10	DropOut ratio Hyper-parameter Comparison	43
11	Number of Kwords Hyper-parameter Comparison	44
12	Context Path Length and Width Hyper-parameter Comparison	44
13	Number of Epochs Hyper-parameter Comparison	45
14	Number of Path Contexts Hyper-parameter Comparison	46
15	Training Results for Java, Python, and JavaScript	52
16	Testing Results for Java, Python, and JavaScript	53
17	Transfer Learning Results for Python and JavaScript	55
18	Training and Testing Results for Python large	59
19	Python Training Results Comparison	60

20	Python and Java Baseline Comparison	61
----	---	----

CHAPTER 1

Introduction

Due to their nature, graph structures tend to hold more information, and can provide extra contextual meaning about the data they represent. Moreover, computer programs are the perfect medium to be represented as graphs, since all applications in all languages can be represented as abstract syntax trees (AST).

This allows a program being represented as a graph to hold both syntactical and in some languages also semantic meaning. This rich data structure is perfect as an application for machine learning (ML). Consequently, by applying machine learning techniques, usually borrowed from the Natural Language Processing (NLP) area, on the source code, we can unlock new areas of research and explore novel possibilities that were unavailable before.

1.1 Terminology

Throughout this work, the following acronyms are used.

- **ML**: Machine Learning
- **AI**: Artificial Intelligence
- **SAAS**: Software As a Service
- **IDE**: Integrated Development Environment
- **SMT**: Statistical Machine Translation
- **GNN**: Graph Neural Networks

- **GGNN**: Gated Graph Neural Networks
- **AST**: Abstract Syntax Tree
- **LSTM**: Long Short Term Memory
- **RNN**: Recurrent Neural Network
- **NLP**: Natural Language Processing

1.2 Big Code

Big Code is a new term that refers to the large amount of source code available on source control repositories, like GitHub and BitBucket. Both private code bases within the confinement of a company and open source code that is accessible by anyone offer billions of lines of code. Code that could be used by data scientists and ML experts to gain insights into the inner workings of a particular codebase, help in the development process, catch potential errors and improve the overall quality of these projects. The abundance of these patterns of code allow researchers to combine machine learning, programming languages, and software engineering to open up new areas of research that were not feasible before.

For decades, research in software engineering has been dominated by formal, or logico-deductive approaches, and these approaches are characterised by its rigorous mathematical terms. Algorithms and various techniques were developed to provide tools for program verification, bug detection, and refactoring [3].

But thanks to the emergence of Big Code, a new domain of research named MLonCode (short for Machine Learning on Source Code) have been created that aim to take advantage of the already existing codebases that were written by industry

professionals. Most of that have already undergone an extensive review process to ensure high quality and reliability. This new “Data driven” approach to developing software tools greatly reduces the cost of developing quality software and reduces the time for testing it.

1.3 Motivation

Machine learning has seen a resurgence in popularity and usage in the past decade after a long period of inactivity or what researchers called the “ Artificial Intelligence (AI) Winter”, But thanks to breakthroughs in graphics processing unit (GPU) architectures and the resurgence of deep learning (DL) techniques. Both academia and industry have once again turned their attention to artificial intelligence and machine learning as a novel and efficient method to solve a variety of problems across multiple domains.

From healthcare, medicine and pharmaceutical, to production lines, security, and online shopping and streaming services, ML models have become an integral part of everyone’s daily lives.

As such it comes as no surprise that the software engineering (SE) industry and computer science (CS) academics have started to research ways to use ML to solve some of their problems as well. This is mainly done by producing ML and DL models that work on the source code itself. Therefore, they manage to provide solutions to a plethora of issues and problems that normally could only be fixed using manual reviewing from experienced developers.

Representative problems are detecting software bugs [4] (both syntactical and logical ones), code duplication and code similarity, malware detection, methods and variables naming, API discovery and retrieval, code summarization and even code

generation. All of these problems are now solved as applications thanks to the advances in ML techniques and the innovative ways they are applied to source code.

Although, ML on source code is relatively speaking a new field of study, the current applications and produced results are nothing short of staggering. Thankfully, it opens the doors to a whole new way for both experienced and novice software engineers to interact with code and their programs.

1.4 Main Challenges

There are many common software challenges that would benefit greatly from ML on source code, and probabilistic machine learning models. The first challenge is method naming [5], which is often considered one of the hardest tasks in computer science, along with cache invalidation.

The second is duplicate code detection, which have applications from detecting plagiarism in academic settings [6], to reducing code smells in code repositories, like GitHub and BitBucket [7]. The third application is detecting dangerous source code, both malicious malware [8], and benign software vulnerabilities [9].

The final challenge is detecting logical bugs that often go unnoticed during static analysis, and would cause run-time crashes and major issues unless caught by the developer. The most widely occurring logical errors are out-of-bound array access [4], wrong return value, and incorrect assertions (often due to hasty coding or copy and paste mistakes) [10].

1.5 Overview

This project starts by covering the basics on ML on source code, and the recent state of the art on the subject. Then the focus is shifted to examples of using DL models on source code.

In particular we focus on the performance of code2vec, which is one of the most recent and advanced state of the art probabilistic machine learning models, for source code prediction in various languages.

The research studies how the choice of programming language and its type can affect both the performance of the model as well as the performance of its applications (such as method naming).

1.6 Historical Background

Static code analysis tools have existed for a very long time, and they are a staple development tool for all software engineers, since they help developers to code faster, cleaner, and with fewer bugs. Almost every major Integrated development environment (IDE) will have either a built-in or a plugin code analysis tool.

However, the majority of these tools run static analysis of the code base, meaning that they cannot detect the type of errors that are logical in nature. Some examples are incorrect method naming, off-by-one errors, copy and paste mistakes, duplicate code, incorrect assertions, and various other types of code smells that traditionally require a human eye to be detected and corrected.

Furthermore, and not surprisingly, code analysis tools tend to perform better for statically and strong typed languages. While all compiled languages can benefit from code analysis before execution, it is a fact that dynamic languages need code

analysis even more, since bugs in them can easily go unnoticed and make its way to production code, causing issues that require spending extra time and resources to get it fixed.

All the previously mentioned problems -both for dynamic and static languages- and many other challenges such as code summarization [11], code generation [12], [13], API discovery and retrieval [14] become tractable by applying ML techniques directly on source code.

1.7 Current Research Areas

Machine Learning on source code or more formally probabilistic models of code work by drawing statistical conclusions gained from analyzing or ‘Learning’ from millions of lines of code that share similar patterns.

As with natural languages, programming languages are also a method of communication. In this case between the developer and the machine. Thus, they have strict rules, grammar, but compared to natural languages, they have a much smaller set of vocabulary that could be structured in a finite number of ways.

Probabilistic code models could be divided into two main areas: generation and representation. Code generation models are often used to auto generate simple sequences of code, while code representation models are used to name methods, predict types, or detect bugs.

The N-gram model is one example of a sequence based code generation model that is popular for its simplicity. Although DL architectures, like Recurrent Neural Networks (RNN), and Long short-term memory (LSTMs) have started to replace N-gram models in the same task due to their superior performance [3].

On the other hand, Semantic Models that view source code as a graph, which lends itself well to source code representation, since all programs can be viewed as graphs through their ASTs. These models however are not as good at generating code snippets compared to their sequence based model counterparts.

Code Representation is where semantic models often shine, they work by assigning a conditional probability distribution of a code property (like a type or name), and they can be broadly classified into two main types.

The first type is structured prediction, which is a generalization of standard classification problems to multiple output variables. This can be used to build dependency networks among variables within the code snippet [15].

The second type is Distributed representation of code (the main focus of this work), which is a technique that is often used in the NLP domain. There is ongoing research on learning the distributed representation of code for different usages, such as predicting names for methods and variables, code summarization, and bug detection.

Allamanis et al. [16] used code vectors to detect logical errors in code that often go undetected by traditional static analyzers. And more advanced models such as code2vec [1] take these tasks even further and could be used to detect code similarity, predict names, detect malware and code bugs.

It is worth noting that both structured prediction and distributed representation are not mutually exclusive, but can be and often are used together in developing representational code models.

1.8 Research Questions

This research project aims to answer the following research questions:

- How does the code2vec model perform on different programming languages, in particular dynamic vs static ones.
- Does the choice of the parser used to build the ASTs and extract the context paths affect the performance of the model itself?
- How much can the usage of transfer learning on a well performing model from static languages help improve the performance of other much harder to train dynamic languages.

1.9 Contributions

The main contributions of this project can be summarized as follows:

- Collating and preparing multiple comprehensive datasets from the top 1000 open source projects on GitHub. For the purpose of experimentation on source code in different languages (C#, Python, JavaScript, and C)
- Evaluating the performance of the code2vec model on weak and strong typed dynamic languages and comparing the results with the baseline of static strong typed languages; by utilizing the ASTminer tool to extend code2vec to work on Python and JavaScript.
- Studying the effect of transfer learning techniques on the code2vec model when applied on static and dynamic languages.
- Studying the effect of using different parsers for feature extraction on the performance of code2vec.

1.10 Chapters Overview

We present a study on the performance of code2vec model on different programming languages, and different sized datasets. Along with discussing some possible applications that benefit from probabilistic machine learning models, like code2vec.

Chapter 2 covers some of the related research in the topic, how it pertains to different languages and possible practical applications for it. Chapter 3 goes into great detail explaining the necessary technical background required for this research. It explains what an AST is, how source code gets parsed to extract its features. And finally how the code2vec model works and a brief overview of its architecture.

In Chapter 4, the datasets used for this research is presented and all the necessary clean up steps are shown. Chapters 5 and 6 pertain to the actual experimental design, the hyperparameters used, the results produced, and the implications of the results. Finally, Chapter 7 covers the conclusion for this paper and an evaluation of the produced results, along with the future work that was not completed in this research.

CHAPTER 2

Related Work

2.1 Statically and Dynamically typed Languages

Most programming languages can be divided into two main categories depending on their data type system, languages could be either statically typed like Java or C# or dynamically typed, and among the dynamic typed languages, they could be strongly typed such as Python or weak typed such as JavaScript. Languages such as C and Objective C are considered statically weak typed languages but are not considered in this research.

Note that there are some languages such as Lambda calculus which are not typed at all, but they are not the focus of this study.

Static typing means that the types are checked at compile time before the program's execution while dynamic typing means that the types are interpreted at runtime during the execution of the program. Strongly typed languages do not allow type coercion (i.e, changing a variable type from one kind to another) while weakly typed languages like JavaScript allows it. Thus a variable could hold a value of a literal "1" string at some point then be changed to a numerical "1" value later on. Such conversion will throw a type error in strong languages such as Java or Python.

Even though statically typed languages are always more verbose than their dynamic counterparts. This extra information provides many benefits, run time type errors are prevented in static languages, and the compiler for static languages utilizes the type information to provide extra optimization under the hood for faster execution. And while languages like JavaScript or Python are more flexible due to their

typing system. And allow the developers to write “less code”. This lack of extra information often limits the capabilities of static code analyzers and IDE code complete features.

And this disparity extends itself as well when performing machine learning operations on source code. For languages like Java or C#, the type information and the more verbose syntax provides extra data for the machine learning models to learn from. Since the parsed syntax trees simply have more information due to the richer syntax, not to mention that even for equivalent projects, static languages tend to be much larger in size than their dynamic counterparts, which limits the size of potential datasets that are collected from dynamic languages.

Furthermore, in a study by Baishakhi et al. on the effect of programming languages on the code quality and number of reported bugs of the top open source projects on GitHub. They found a small but significant result which implies that static typing is better than dynamic typing, and strong typing is better than weak typing [17]. Interestingly the same study found that functional languages had higher quality and lower number of reported bugfixes than procedural languages, but functional languages are not part of this study.

All of the previous points leads to the obvious conclusion, that the majority of the research on probabilistic machine learning models on source code is often carried on on statically typed languages such as Java, C++ or C#. As such this research aims to study the differences in performance and applications when applying the current state of art probabilistic model for machine learning on source code in static and dynamic languages. And explores various methods to improve the performance of ML models of dynamically typed languages to be on par with their statically typed counterparts.

2.2 Deep Learning on Source Code

In the paper `code2vec: Learning Distributed representations of code` by Alon et al. [1]. The main idea of their research is to “to represent a code snippet as a single fixed-length code vector, which can be later used to predict semantic properties of the snippet” in the same way other NLP techniques like `word2vec` [18] works, i.e to provide semantic labels for code snippets like methods. This was used to predict method names using the contents of the method’s body where a good descriptive name gives the developer an insight to what the method can do. The model was trained on a dataset containing more than 14 million methods and consisting of code snippets like methods and their corresponding labels, i.e the method name.

The authors leveraged the structured, and syntactical nature along with the logical code flow of programming languages to produce better vector embeddings from the paths in a program’s AST for better code representation. (similar to word embedding in NLP applications). To achieve this goal they utilized a neural attention network architecture using soft attention that gives different weights to the different possible code paths (syntactic paths) in each code snippet. Some important Design decisions were made that differentiated `code2vec` from other similar probabilistic ML models.

- Model considers syntactical only context, making it language agnostic
- Each method or code snippet will have its own unordered bag of path-contexts
- The model utilizes a simple architecture that uses a large corpus of data

Another major paper in the realm of representational distributed code is learning to represent programs with graphs by Allamanis et al [16] . This one takes a different approach to the same problem, instead of depending entirely on syntactical analysis with no regards to the programming language semantics. The authors used a gated graph neural network to represent both the syntactic and semantic structure of the code.

The authors focused their applications on the tasks of variable naming and variable misuse. The main contribution of their research was the incorporation of data flow and type hierarchies information in the analysis phase of the source code. This was done by “encoding the programs or code snippets as graphs in which edges represent syntactic relationships (e.g. “token before/after”) as well as semantic relationships (e.g. “variable last used/written here”, “formal parameter for argument is called stream”, etc.)”. This design decision however traded the language neutrality for performance since the model requires a compilable static language for its input.

The paper Summarizing Source Code using a Neural Attention Model by Iyer et al [19] is another example of using NLP techniques on source code to automate a task that most developers have to do manually. The authors used LSTM (Long Short Term Memory) networks with attention to produce a model (called CODE-NN) that can produce complete sentences to describe C# code snippets and SQL queries. Their dataset was scrapped and collected from Stack Overflow. They tested their model on two main areas, code summarization and code retrieval. The model managed to outperform the previous state of the art on both tasks.

2.3 MLoNCode Applications

Big Code ML models or probabilistic code models are not just for theoretical researchers, they are actively being used in both academia and industry for many applications. Code auto-completion, code recommendation, and automatic code review systems have already been utilized within the software engineering industry and are available for usage as a SAAS service by all of the major cloud providers. Almost all major IDEs nowadays have some form of machine learning assisted code completion (such as Visual Studio’s Intellisense), and some even offer suggestions for coding conventions (such as PyCharm’s Python pip 8 suggestions).

Bug detection is another major application of probabilistic code models, Runtime errors like off-by-one and Null-Reference exceptions and other code defects are impossible to detect by static analyzers and can often go unnoticed during the code review development phase till they reach production.

Therefore, it comes as no surprise that statistical machine translation (SMT) models have began to replace rule based models in code translations tasks such as transcoding an application in one language like Java to a different one like C#, thus saving the developer teams hundreds of hours of work. While other models can even go beyond detecting bugs and can do code cleanup for not so obvious errors like copy and paste mistakes and incorrect API usages as shown by the work of Allamanis et al. who managed to achieve a 58.6% accuracy in the ask of smart paste where a specific snippet of code is intelligently adapted to the surrounding existing source code [20].

Other models use an auto-encoder architecture to detect code similarity and duplication like in the work by White et al. [21] while others like Iyer et al. [19] utilized the neural attention architecture to summarize code and to auto generate

documentation.

The opposite direction is also possible with some models like the seq2sql model which able to generate valid SQL statements from natural language sentences [22]. The list of possible applications goes on and on, from API discovery and Retrieval to malware detection and even program synthesis [3].

2.4 code2vec Applications

Code2vec as a probabilistic model have plenty of applications, and although the authors Alon et el in their paper only explored the task of method naming, they mentioned that the model could be used for many more tasks, in the paper “Learning Off-By-One Mistakes: An Empirical Study” by Hendrig et el [23]. The authors used the code2vec model to detect runtime bugs that can not be detected by static analysis tools in Java projects, in particular they explored the infamous off-by-one mistake that developers often make by forgetting to set the correct boundary for an iterator (i.e. using ‘<’ or ‘>’ instead of ‘<=’ or ‘>=’ or vice versa) thus going over the array or list limit and potentially causing a runtime exception.

By selecting methods that are candidate for the off-by-one errors (containing for loops), the authors introduced errors in those methods by mutating the body of the method and adding or removing the ‘=’ operator, all mutated methods was labeled defective while the original methods was labeled non-defective. By doing that they effectively changed the predictor on code2vec’s original model to be a binary classifier instead of giving a probability distribution to potential method names, thus they were able to predict whether a method contained an off-by-one error or not. They managed to achieve a precision of 80% and a recall of 77% on the Java large dataset (roughly the top 10000 open source projects from GitHub provided by Alon

et al in their original research) , but suffered a performance drop (43% precision and 23% recall) on real world code bases.

Another popular task that code2vec model is suitable for is detecting code similarity or clone detection, this could be achieved based on the program’s functionality or the program’s source code content. In the paper titled ‘Code Clone Detection using Code2Vec’ by Anupriya Prasad [24], the author utilized the code2vec model for the purpose of detecting functionally similar methods in Java source code. The dataset used is the top starred Java projects from GitHub as prepared by [1]. The results were evaluated using the precision and recall metrics by utilizing the ‘Measure Precision’, ‘BigCloneBench’ [25], and ‘InspectorClone’ [26] open source tools.

Code2vec was also the basis of the research by Compton et al. who investigated the effect of obfuscating the variable names from the extracted methods during the training phase of the code2vec model [8]. By removing the model’s reliance on variable names for prediction and thus forcing it to focus solely on the code structure and patterns. They tested the new model on Java source code on a variety of tasks from method naming (where the obfuscating model saw a performance degradation) to duplicate code detection and malware classification.

CHAPTER 3

Technical Background

3.1 Abstract Syntax Trees (AST)

An Abstract Syntax Tree or AST is a tree structure representation of a program's code, more specifically the syntactic structure of the code. Thus an AST does not hold any semantic meaning regarding the source code but conveys its structure. ASTs do not include some information such as formatting and whitespaces. A node in the AST represents a syntactic unit of the program's source code such as a variable, an operator or an operation. The children of the node represent the lower-level units associated with this current node.

Mathematically an abstract syntax tree can be defined as

"An Abstract Syntax Tree (AST) for a snippet of source code can be defined as a tuple in the form of $\langle N, T, X, s, \delta, \phi \rangle$ [1], where:

- N and T are the sets of non-terminal and terminal nodes in the tree respectively
- X is the set of values
- $s \in N$ is the root node of the tree
- $\delta : N \rightarrow (N \cup T)$ is a function mapping a non-terminal node to a list that contains all of its children
- $\phi : T \rightarrow X$ is a function mapping a terminal node (i.e. leaf node) to some associated value

An example of a generated AST for a simple squaring function can be shown in Figure 1 [2]:

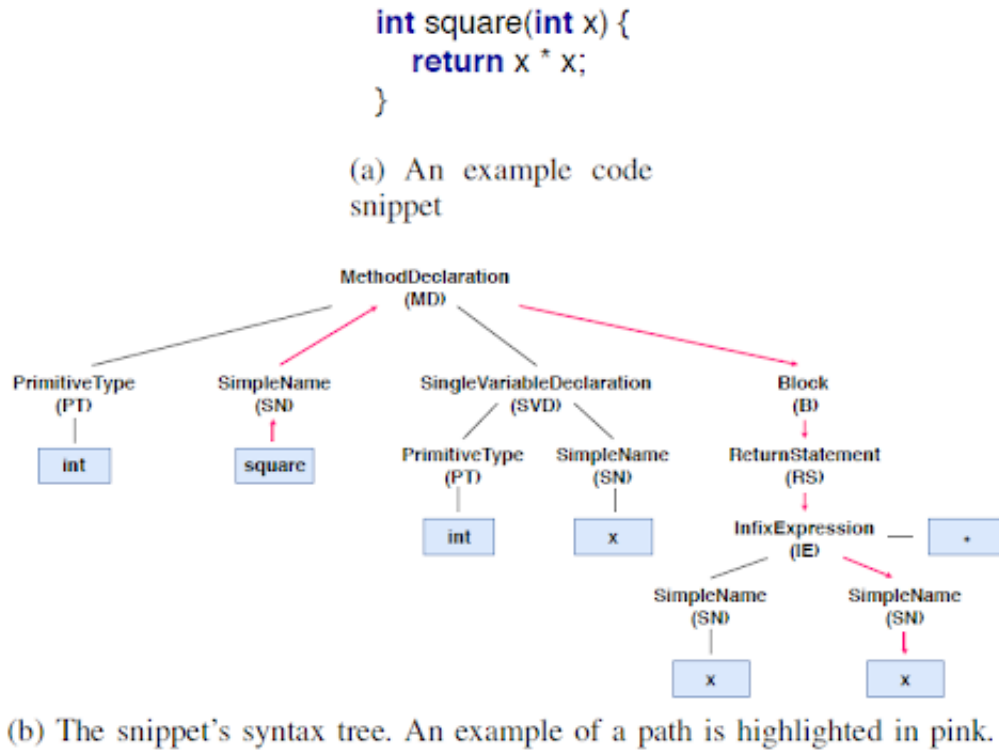


Figure 1: AST example from PathMiner [2]

3.1.1 AST Path

An AST path 'p' is a sequence of connected vertices or nodes in the syntax tree, representing a path from one vertex (often a leaf) to another leaf. Leaves in an AST represent a code token like a type or a variable name. Thus an AST path is a way to represent a logical flow or operation within the constructed code snippet. Officially a path is denoted by a sequence of its start and end nodes and the traversal direction between them.

This was mathematically defined by Alon et al. [1] as

"An AST-path of some length k is a sequence of the form $n_1d_1 \dots n_kd_kn_{k+1}$ ", where:

- n_1 and $n_{k+1} \in T$ are the starting and ending terminal nodes in 'p' denoted as $start(p)$ and $end(p)$ respectively
- $n_i \in N$ are non-terminal nodes, for $i \in [2 \dots k]$
- $d_i \in \uparrow, \downarrow$, for $i \in [1 \dots k]$ are movement directions, traversing the AST in an upward or downward direction to create the path between the two terminal nodes.

Therefore, if $d_i = \uparrow$, then $n_i \in \delta(n_{i+1})$, and if $d_i = \downarrow$, then $n_{i+1} \in \delta(n_i)$. Using an example from Figure 1, the AST path highlighted in red in that tree is (SN \uparrow MD \downarrow B \downarrow RS \downarrow IE \downarrow SN) [2].

3.1.2 Path Context

Having defined an AST, and an AST-path, a path-context can now be defined as a triplet $\langle xs, p, xt \rangle$ that consists of the value associated with start vertex (a leaf node), the value associated with the end vertex (another leaf node), and the path between the two of them, where:

- $xs = \phi(start(p))$
- $xt = \phi(end(p))$

So, again by using the highlighted red line example in Figure 1, the path-context shown in red can be denoted as (square, SN \uparrow MD \downarrow B \downarrow RS \downarrow IE \downarrow SN, x) [2].

From the simple example in Figure 1, It's clear that a single simple function will have more than one path-context, and that can increase exponentially with the increased complexity of the source code. It's also clear that not all path-contexts are important or contribute the same weight.

So to get a fair representation of what the function or code snippet does, we need a collection of the most important path-context extracted from each code snippet. And since different programming languages have different syntaxes and grammar rules. Creating an AST and extracting path-contexts for different languages is a difficult and often repetitive task. Thus the need for an external tool that encapsulates this complexity.

3.2 PathMiner A Library for Mining of Path-Based Representations of Code

Building the dataset for code2vec or similar representational ML models involves transforming the source code files into a format digestible by the machine learning algorithm. This format which often take the form of a Json or csv file(s) includes the important parts of the code snippets that we want the model to focus on, and their associated labels.

In the case of code2vec, each code snippet is transformed into a collection of path contexts extracted from the program's AST. In the current research, there are two main methods to build the dataset from the original source code files. Either by using or extending the custom mining/Extractor library provided by the authors of code2vec, or by using the external library 'PathMiner' and its provided tools to add extra languages.

PathMiner (sometime referred to as ASTminer based on the implementation of

the paper with the same name [2]) is a novel approach to represent programs (source code) in a format suitable for machine learning algorithms by extracting the path-based representation -path-contexts- from code.

It was developed by Kovalenko et al from the JetBrains Research group to facilitate the experimentation on source code by machine learning researchers by separating the often difficult and technically challenging part of building the graph based dataset into a separate tool, making it easier for non-developers and newcomers to this research area to focus their time and effort on designing the algorithms used to process code for machine learning models.

The library currently supports the extractions of Path-based representations of files/methods, as well as their Raw ASTs. This approach utilizes the program’s AST to represent the code snippet as a collection of paths derived from its syntax tree. This representation captures the structure of the code along with its semantic meaning.

It is a two step process involving parsing the code into its AST representation, then extracting the relevant paths from its syntax tree. This process is technically challenging and often gets in the way of machine learning researchers who rather focus on developing the ML model rather than the technical details of building the dataset.

3.2.1 Astminer Tool

The Astminer tool is a fast and flexible open source library for mining path-based representations of code based on the pathminer paper. Path-based representation of code is one of latest ways to represent source code in a way suitable for ML algorithms to operate on, Other methods such “vector of tokens” and “traversal sequence of the syntax tree” have been used by researchers before, but do not offer the same flexibility and wide use of applications.

Currently Astminer supports mining of code in Java, Python, and JavaScript, and is designed to be easily extensible to support other programming languages. This is achieved by providing a convenient extension point for parsers generated by ANTLR.

ANTLR an acronym for "Another Tool for Language Recognition" is a powerful parser generator that uses LL(*) for reading, processing, executing, or translating structured text or binary file [27]. Antlr takes as input a context-free grammar file for a specific language, and it can then generate a parser and lexer for that language that can build and traverse the language's AST. It is currently at version 4.

The integration of ANTLR within the Astminer tool provides easy extending capabilities for additional languages within the same mining framework. An overview of PathMiner extraction workflow and its main components can be shown in Figure 2:

At its core, Astminer/pathminer has two main stages, A parsing step to build an AST for a particular language -either using ANTLR or using other parsers-. And an Extraction step to extract the path from the generated AST based on predefined filters such as the width and length of the paths between the leaves and then storing the paths in a configurable format after encoding it numerically for memory efficiency purposes. Finally PathMiner is implemented in Kotlin but have Python wrappers for easier integration with other machine learning workflows.

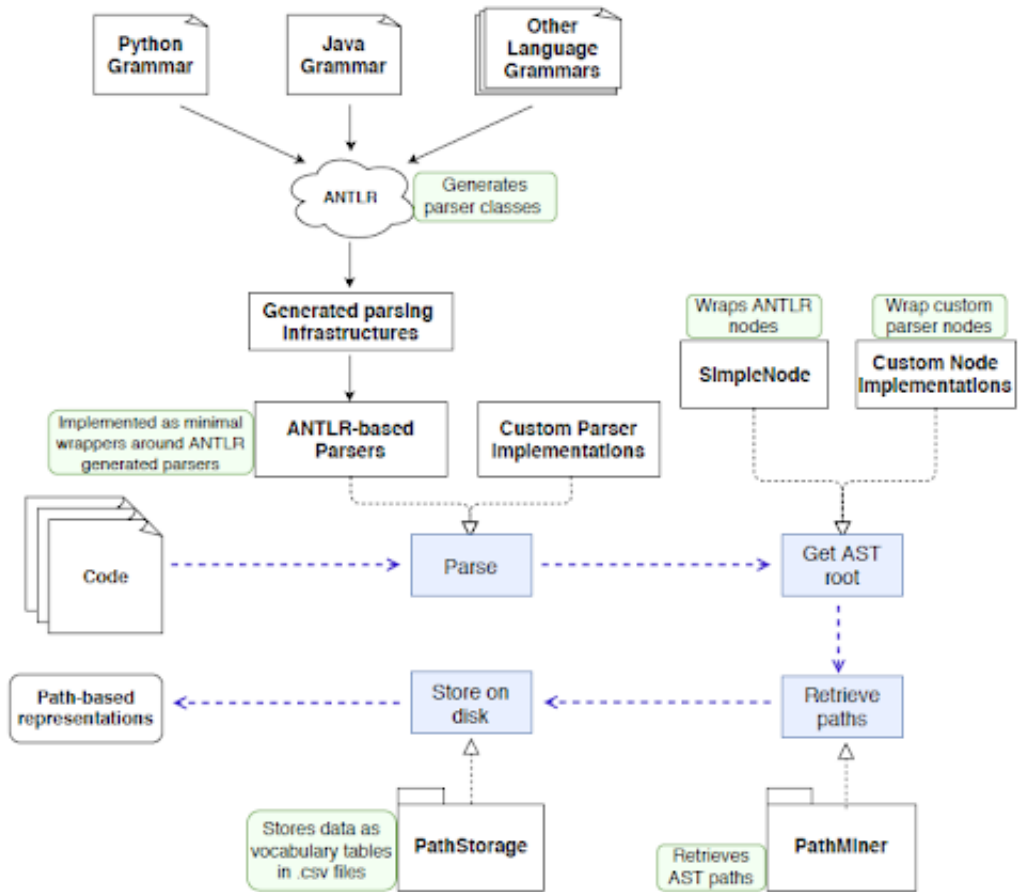


Figure 2: PathMiner Architecture Paper [2]

3.3 Code2Vec

Code2Vec is a deep learning model created by Alon et al. based on the bag of paths method in order to generate continuous distributed vectors (also known as code embeddings) from source code files. The model itself is language agnostic but the authors currently implemented the feature extraction and preprocessing for Java and C# languages and only tested the model on Java datasets.

In the original paper [1], Alon et al. experimented with the task of predicting method names, but since the code embeddings are saved during the extraction process, the model could be extended to perform other tasks such detecting code similarity, doing binary classification, and other similar tasks. This can be achieved by modifying the extractor to spit out different labels rather than the method name.

An example of code2vec usage on the task of method name prediction can be seen in Figure 3.



Figure 3: A code snippet and its predicted labels as computed by our model as demonstrated in [1].

3.3.1 Code Embeddings

The main idea behind code embeddings is that by mapping code snippets to a distributed collection of vector representations, similar code snippets (semantically speaking) will have similar code vectors.

The authors presented a neural network model that can learn those code embed-

ding and assigns weights to them, thus making a corresponding relation between a method's (or any code snippet) label and it's most significant distributed vectors.

Code2Vec leverages the structured nature of source code to reduce the learning effort, as opposed to other vector representation models in traditional natural language processing tasks that have to re-learn the entire vocabulary of the language. But thanks to formal languages strict syntax, code2vec can learn to generate the distributed vectors from the code's AST directly. It uses the syntax paths (code embeddings) derived from the code's AST to capture the most common code patterns.

This helps lower the training effort since the model does not need to learn unnecessary information within the entire's code text, while still being general enough that it is not tightly coupled to a single problem domain (i.e. only predicting a method name and nothing else).

By representing each code snippet (such as a method) as a collection or a bag of paths extracted from its body, and assigning a weight to each path in that bag using a soft attention mechanism architecture[28], the neural network can then aggregates the top weighted paths (i.e. most important paths in the code snippet) into a single weighted average vector along with its associated label.

This ensure that the resulting aggregated vector contains all of the relevant information about the code without being a strict representation of it. Thus allowing the model to predict labels for methods that share similar code vectors (i.e. the same relevant information) even if they are not strictly similar.

3.3.2 Context Path Example

An example of how code2vec works on a sample method can be seen in Figure 4 which shows a simple java method that checks for the existence of an element within an a given list. The highlighted paths are the most imports logical flows or patterns in this method.

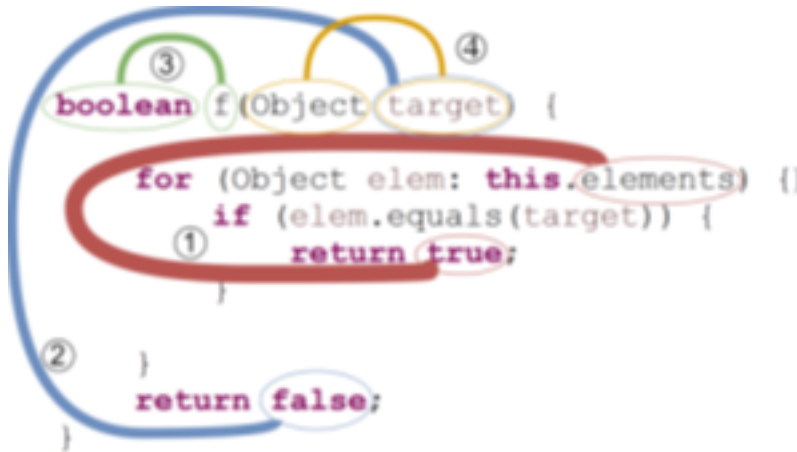


Figure 4: Sample method [1]

Figure 5 shows the AST representation of the method in Figure 4. The code2vec attention neural network assigns weights to the selected context paths within the tree. Where the widest highlighted path corresponds to the most important logical flow within that method, and is given the largest weight.

Finally, Figure 6 shows the final top predicted labels for the method, and the accuracy score for each label. Here the model correctly assigned the 'contains' label to the method's name.

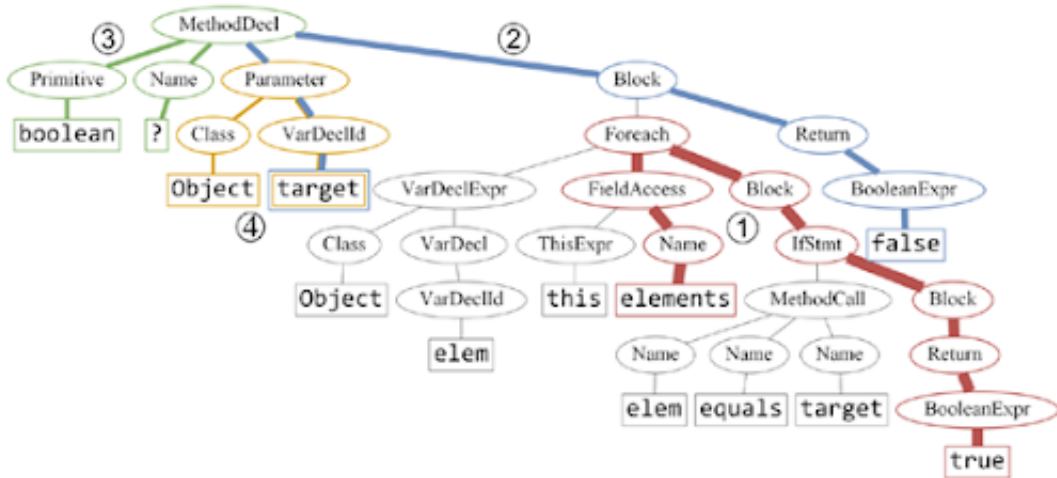


Figure 5: Top 4 paths in the method below where the width of each colored path is proportional to the attention it was given (red 1 : 0.23, blue 2 : 0.14, green 3 : 0.09, orange 4 : 0.07) [1].

Predictions:



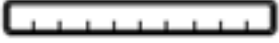


contains		90.93%
matches		3.54%
canHandle		1.15%
equals		0.87%
containsExact		0.77%

Figure 6: Top predicted results [1].

3.3.3 Code2vec Architecture

The code2vec model has a simple but novel architecture, utilizing an attention neural network and fully connected layer that learns to combine the path context vectors and assigns different weight to each of them.

The aggregated vector is calculated by weighting each code vector by a factor of its dot product with another global attention vector. Both of the individual vectors and the aggregated vector are trained and learned at the same time using back-propagation.

The final aggregated code vector is what is used to for label prediction. The architecture of the path attention network can be seen in Figure 7:

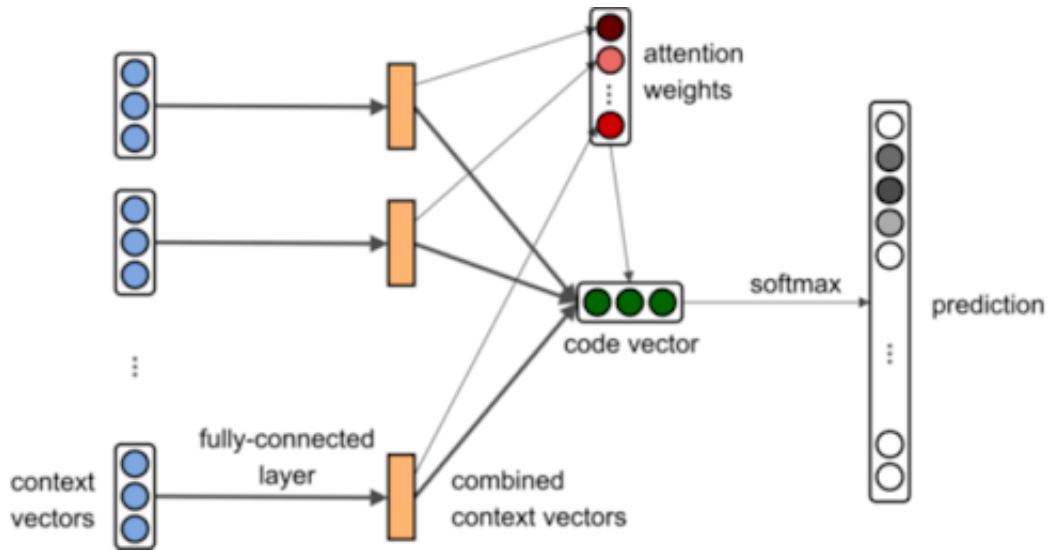


Figure 7: The path attention network architecture [1]

CHAPTER 4

Datasets

4.1 Datasets Size

For the purpose of this research, 4 different datasets with different sizes from 4 languages were collected and prepared (C, C#, Python, and JavaScript). Although eventually only 2 of them were utilized in this research with other two being available for future research.

Java was chosen as a static strongly typed language and acts as the baseline for the rest of the comparisons since the original code2vec experiments by Alon et al. were performed on Java. For Dynamic languages, Python was picked as a strong typed dynamic language while JavaScript was picked as an example for weakly typed dynamic languages.

For the Java and Python experiments, a medium and large sized datasets was used. It consisted from the top 1000 and 10000 open source projects on GitHub respectively. While for JavaScript, a similar medium sized dataset was curated but eventually a smaller sized dataset consisting of the top 100 rated projects was used. Hyperparameter tuning was done using a small Python dataset of around 100 projects.

Note that Alon et al. [1] prepared 3 Java datasets for the purpose of their research. A large dataset containing top starred 9500 Java projects from GitHub (with a 9000/200/300 split for train/val/test). A medium dataset consisting of the top starred 1000 Java projects on GitHub (with a 800/100/100 split for train/val/test). And Finally a smaller dataset of only 11 Java projects.

All the datasets for this research was collected and built from the top starred

open source projects on GitHub. The working assumption is that the top rated and starred projects on GitHub are of high enough quality to ensure best practices, good naming conventions for methods and variables and a high software quality, since they are active and continuously maintained.

As such a Python script was created to pull and clone the top 1000 starred projects for Python and JavaScript. The Java dataset was provided by Alon et al. [1] also from the top rated open source projects on GitHub. Due to the difference in languages used, the top projects from each language usually vary greatly in size and oftentimes quality. However by selecting a relatively speaking large sample size, all of these individual differences should even out and provide a common starting point for the training of the subsequent models.

4.2 Data Preprocessing

Considerable preprocessing steps was needed to be done before the source code files could be used or fed into the ASTminer extractor. Since when dealing with huge amounts of unfiltered source code, Data cleaning and preprocessing becomes a vital step for producing good results. Therefore, another Python script was created that handled the cleaning up process, to perform the following tasks:

- The first step in cleaning up was to remove all the .git associations and any unrelated files as well as any files that do not have the explicit extension for the target languages such as Json files, XMLs, projects builds, etc, in order to keep only the source code (i.e. ‘.js’ for JavaScript, ‘.Java’ for Java, and ‘.py’ for Python)
- All projects files were flattened into one level, such that for every project, all

of its files will reside in the same directory. This was achieved by a script that moved all of the files nested inside the project's directory hierarchy to the top. By doing this, the extractor would have an easier and faster time in processing the source code files, and prevent possible memory issues during the processing of very long files paths.

- To prevent names clashes that could arise from the previous step, For dynamic languages such as Python and JavaScript, all source code files were renamed to a randomly generated UUID. UUID or Uiversal Unique Identifier is a 128 bit random object that gets displayed as a 36 character alphanumeric string that guarantees uniqueness. This is done to preserve as much as possible of the original source code files since for these languages a large size of the business logic could reside inside the main entry point file (Ex: 'main.js' for JS and 'main.py' for Python), and since the objective is collect Paths information (code embedding) from as many methods as possible, files with duplicate names could collide within the same project
- For static languages, this is not an issue since the entry point files that could clash ('program.cs' for C and 'main.java' for java) often does not include important methods of its own nor should they have any business logic that should be mined. Thus any duplicate files were simply removed.
- For JavaScript, any file larger than 1 MB was removed from the dataset, this is to prevent heap memory issues during the extraction process, since very large files (often the case with JS minimized files) would cause the extractor to run out of memory.

This cleaning-up process is essential to trim down the size of the cloned repos

and only keep the relevant source code files. It also greatly reduces the size of the final dataset by around 97.5%. More often entire projects are trimmed and removed from the final trimmed dataset. For example, in the Python large dataset, the original cloned projects of around 7300 repositories contained 1268519 different files with a size of 186.7 GB. But after the cleanup process, the resulting files are only 433438 with a final size of around 3.6 GB and with more than 40 projects completely removed.

4.3 Dataset Split

All medium and small sized datasets were divided into train, validation and test sets with a split of roughly 80%, 15% and 5% respectively, while the large datasets had a split of roughly 90%, 5%, 5% for the train, val, and test sets.

An overview of the datasets and their split that was used in this research after the preprocessing phase was completed can be seen in Table 1

Table 1: Source Code Datasets

	Java	Python	JavaScript
Small Dataset Size	506.7 MB	118.7 MB	643.3 MB
No of Projects	11	24	N/A
No of Files	96552	22016	10204
Source	Code2Vec	GitHub	GitHub
Medium Dataset Size	2.5 GB	1.5 GB	1 GB
No of Projects	995	902	330
No of Files	466667	194903	127331
Source	Code2Vec	GitHub	GitHub
Large Dataset Size	12.2 GB	2.7 GB	2.8 GB
No of Projects	9556	7188	814
No of Files	3186464	366016	308067
Source	Code2Vec	GitHub & SRILAB	N/A

4.4 JavaScript Dataset

Both Java and Python experiments were performed on similar sized datasets with respect to the number of projects (considered to be medium sized datasets for this sort of experiments). However due to issues with with extracting the context-paths from JavaScript files causing memory issues, additional preprocessing was needed to successfully generate the JavaScript input data to eliminate larger sized files. Thus the final size of the JavaScript dataset is only one third the size of original prepared dataset.

CHAPTER 5

Experiments

5.1 Requirements

AstMiner is implemented in Java and Kotlin and has the following requirements:

- The Java 8 JDK
- Gradle for handling build dependencies
- Optionally Docker for handling parser dependencies

Code2vec is implemented in Python and has two model implementations, the default implementation in pure TensorFlow which is used for all experiments in this research, and a more limited TensorFlow Keras implementation. Either of them have the following requirements:

- The Java JDK for using the provided Java Extractor for building Java datasets
- DotNet Core for using the provided C# Extractor for building C# datasets
- Python 3
- TensorFlow 2.0.0
- CUDA 10.0 for GPU support
- Optionally Docker for handling parser dependencies

5.2 Hardware

All Dataset cleanup, and feature extraction was completed on an Azure NC6 Standard instance, with the following specifications:

- Ubuntu 18 OS
- 6 vCPUs
- 56 GB Memory Ram
- 1x K80 GPU

All the model training was completed on SJSU's CoS HPC cluster nodes with following specifications:

- Centos 7.9.2009 OS
- Xeon E5-2680 v4 2.4GHz (broadwell) (14 core/CPU)
- NVIDIA Tesla K40

5.3 Workflow Pipeline

The workflow for training a model consists of parsing the ASTs from the source code files for a specific language and extracting all the context paths along with their associated labels. This step is handled by the Astminer library. After which the generated pathcontexts.c2s files undergo an extra preprocessing step by the code2vec preprocessing script to shuffle the training data, and pad the context paths to be the same length.

After the specified number of context paths to keep for each method along with the desired number of words, paths and target words to keep in the vocabulary are

supplied. The preprocessing script will generate a histogram of the possible target names from the training data. The histogram will be later used to build the model’s vocabulary file which contains all of the possible labels that it can predict. The output of the preprocessing step is are the ‘c2v’ files that are then fed to train the model. The entire workflow can demonstrated in figure 8:

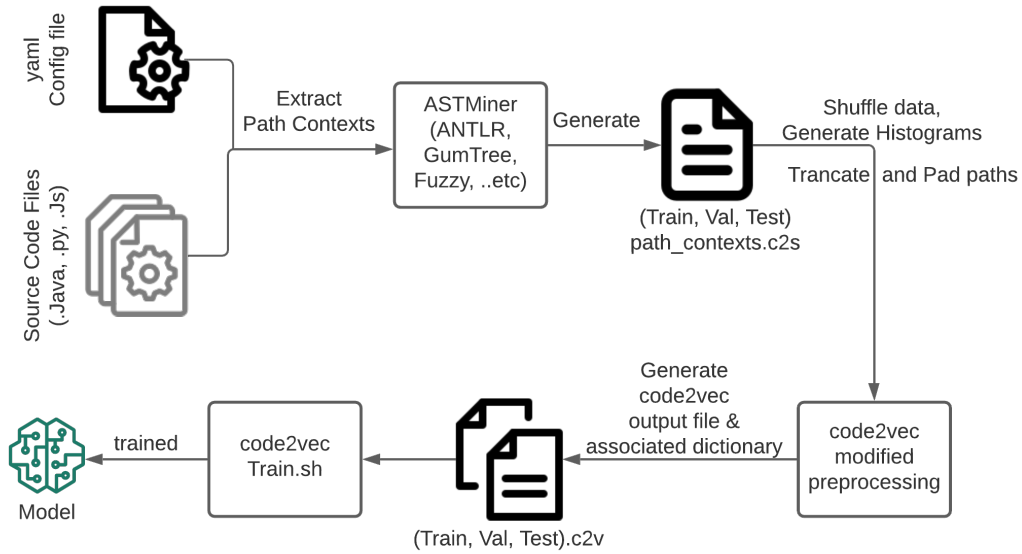


Figure 8: Feature Extraction and Model Training Pipeline/Workflow

5.4 Feature Extraction

The main objective of this research is to compare the performance of the code2vec model on different language types, to see how much of a performance drop will occur on dynamic languages versus their static counterpart, and finally what can be done to improve that performance.

As such the first experiment is to evaluate the different target languages based on their performance as depicted in the original code2vec research in the task of predicting method names. Intuitively static languages should perform better due to

their strong typed nature, whereas dynamic languages do not have this advantage and thus their code embedding carries less information.

More over the top starred Java projects tend to be of higher quality and have better naming conventions for their methods and variables as compared to more beginner friendly languages like JavaScript or Python.

To produce an accurate and objectively comparable results, all the training data was mined using the same parser (Antlr) and extractor (AstMiner) before being processed by the code2vec preprocessor script. The code2vec model requires a language agnostic file (extension c2v) that contains all the path context and their associated labels. Producing that file for external languages requires a two step process:

5.4.1 Path contexts extraction

First step is the feature extraction of path contexts from the source code files, this involves parsing all the code in a given file, splitting the methods inside, extracting the paths from each method and associating each one with its label. Different parsers (Ex: ANTLR, GumTree, ..etc), and different filters can be applied in this step (like ignoring constructors, empty methods, or methods larger than a specified size).

The ASTminer tool that was used for this step requires the input data to be split into three directories (train, test, and val), and outputs three path contexts files (train, test, and val c2s files) where each line in those files represent a label followed by a sequence of space-separated triples.

Each triple contains comma-separated IDs of the start token/vertex, path, and end token/vertex. The tool also outputs extra csv files that contain the mapping between the numerical IDs in the path context files and their corresponding node

types and tokens.

5.4.2 Preprocessing the path contexts files

The second step is to transform the previous output into a format that the code2vec model accepts, this is done by passing the path context files to a modified version of code2vec’s preprocessing script.

The script will pad or truncate the data in the files to a specified number of fields, The output of this step is the language agnostic c2v file that the model uses for training, along with the dictionary file that contains all vocabulary for the dataset (i.e. all possible labels used for prediction).

5.5 Feature Extraction Results

Tables 2, 3, and 4 show the results of preprocessing the train, Val, and Test datasets using ASTminer respectively. The dataset size and the number of projects and files per dataset are shown for each languages, along with the most important settings that were configured for mining the ASTs.

The Max and Average number of contexts refer to the specified amount of path contexts to be extracted from each method’s AST, and the actual amount that was extracted (i.e. a simple short function may not have enough contexts to match the max number of contexts specified in the setting).

Along with the parser type, ASTminer support filters for the AST itself, from the maximum size of the tree (i.e. the number of nodes/tokens in the AST), to how deep should each context path be considered, and how far apart each two leafs would be considered. So a max path of length 8 and width 2 will restrict the extraction of

contexts paths to any path that only goes up to 8 levels down in the abstract syntax tree, with no more than 2 leafs apart at the bottom of the tree.

The total examples refer to the complete number of methods that were extracted successfully from the source code files, and hence will be used for training, validation, or testing. Note that all of these settings are configurable through YAML files that gets passed as argument to the ASTminer tool.

Table 2: Feature extraction for Training data

Language	Java	Python	JavaScript
Dataset size	1.9 GB	1.2 GB	523.5 MB
No of Projects	670	727	N/A
No of Files	347735	155264	7790
Extraction Time	21:56	58:27	39:36
Max No Contexts	300	300	300
Avg No Contexts	191.182	254.480	242.76
Total Examples	2627188	1377753	55726
Max Path Length	8	8	8
Max Path Width	2	2	2
Parser	antlr	antlr	antlr
Max Tree Size (no of nodes)	500	500	500

5.5.1 Observations about the feature extraction results

The results from Tables 2, 3, and 4 show a positive correlation between the size of the dataset and the extraction time, where the 4x larger sized train sets took on average 4x times to complete compared to the val sets. We can also see that dynamic languages took more time than the static ones, with Python feature extraction taking 3x the time compared to Java even though the Java source code files were roughly double the size of the Python ones. This shows that even by using the same parser on the same hardware, building the ASTs and extracting the code embeddings is much

Table 3: Feature extraction for Validation data

Language	Java	Python	JavaScript
Dataset size	389.9 MB	257.3 MB	90.6 MB
No of Projects	185	145	N/A
No of Files	67083	29032	1315
Extraction Time	05:01	12:29	N/A
Max No Contexts	300	300	300
Avg No Contexts	195.351	253.053	231.9
Total Examples	607707	334865	1837
Max Path Length	8	8	8
Max Path Width	2	2	2
Parser	antlr	antlr	antlr
Max Tree Size (# nodes)	500	500	500

Table 4: Feature extraction for Testing data

Language	Java	Python	JavaScript
Dataset size	306.5 MB	77.2 MB	29.2 MB
No of Projects	140	30	N/A
No of Files	50850	9051	288
Extraction Time	03:38	3:46	N/A
Max No Contexts	300	300	300
Avg No Contexts	200.156	265.939	228
Total Examples	415156	84847	207
Max Path Length	8	8	8
Max Path Width	2	2	2
Parser	antlr	antlr	antlr
Max Tree Size (# nodes)	500	500	500

easier for static languages like Java than it is for dynamic languages like Python or JavaScript.

5.5.2 Notes about JavaScript

The JavaScript dataset proved to be the hardest to extract from and transform into the appropriate format, often times failing to complete the extraction process, and running out of heap memory space.

Thus additional preprocessing was done on all JS source code files, where all the files in every project were renamed to random UUIDs, then grouped together in one directory and later split into individually UUID named directories each containing exactly 50 files per directory. Any file larger than 1 MB was discarded and a smaller set was used for the ASTminer extraction to avoid the heap memory issues. This resulted in reducing the memory issues, and helped to better isolate and identify the problematic files.

5.6 Hyper-parameters

Code2vec's hyper-parameters can be adjusted by modifying the 'config.py' file. All three models for Java, Python, and JavaScript were trained using similar hyper-parameters in order to for the performance comparison to remain fair.

To find out the optimal hyper-parameters, multiple experiments were conducted using a small sized Python dataset, by keeping all other code2vec hyper-parameters and ASTminer settings constant and changing only one parameter at a time. Then performing the feature extraction and model training on the two variations, and comparing the results by measuring precision, recall, F1-score, and the highest predicted label accuracy. we identified the best hyper-parameters to be used for training the larger datasets and the other programming languages.

5.6.1 Batch Size

The batch size refers to the number of samples that will be propagated through the neural network. By comparing a batch size of 512 against 1024, the later proved to be marginally better as can be seen in Figure 9. Note that the authors of code2vec

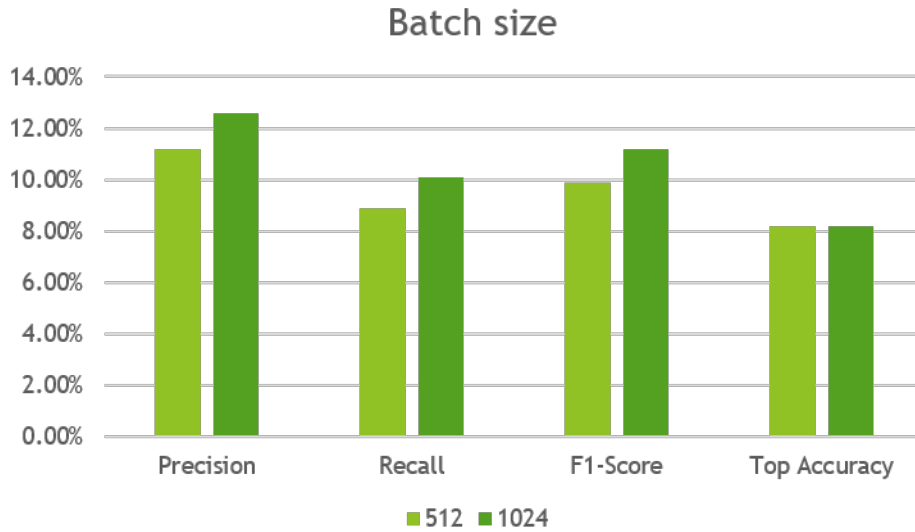


Figure 9: Batch Size Hyper-Parameter Comparison

used a batch size of 1024 in their experiments.

5.6.2 DropOut Rate

Dropout is a regularization technique that is used to prevent over-fitting the model. The dropout rate is probability of training a given node within the neural network layer. A ratio of 0.75 implies that 25% of the neurons will be turned off, and the remaining 75% will be trained. By comparing a dropout rate of 0.5 against 0.75, the later proved to be marginally better as can be seen in Figure 10. Note that the authors of code2vec used a dropout rate of 0.75 in their experiments.

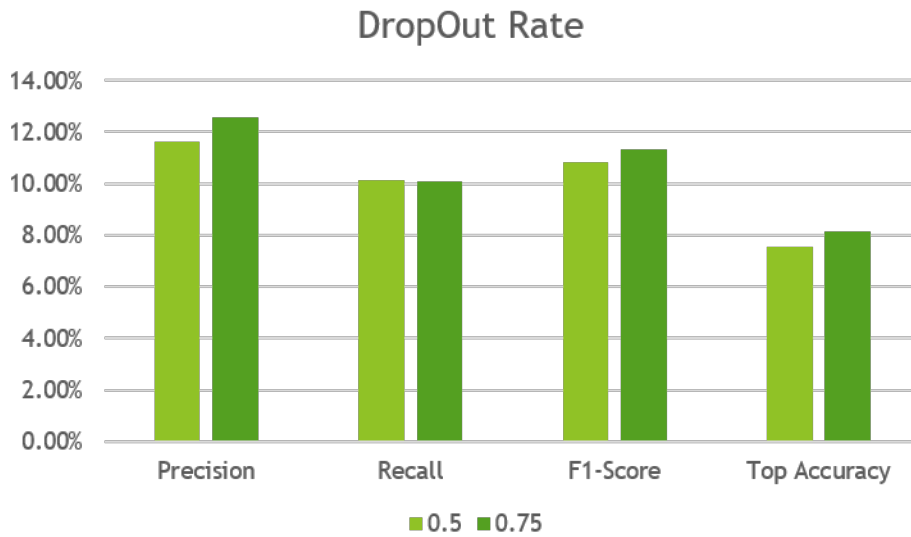


Figure 10: DropOut ratio Hyper-parameter Comparison

5.6.3 Number of Keywords Ratio

The number of keywords refer to the number of possible predicted labels that the model will take into account when scoring a sample method, each will be assigned a probabilistic accuracy score that refers to the model's confidence in the predicted label. By comparing a keyword count of 5 against 10, the later proved to be marginally better as can be seen in Figure 11. Note that the authors of code2vec used a keyword count of 10 in their experiments.

5.6.4 Context Paths Length and Width

The context path length refer to the maximum depth of an AST path within the code's abstract syntax tree, while the context path width refer to the maximum distance between any two terminal nodes within the code's abstract syntax tree. More complex and larger methods usually have contain deep ASTs with much more leaves, and overall more context paths. By comparing a combination of context path length

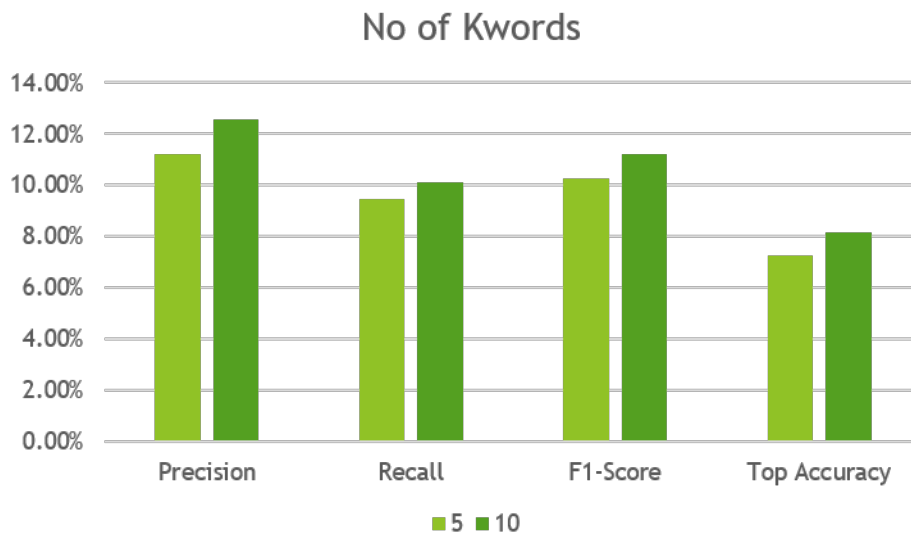


Figure 11: Number of Kwords Hyper-parameter Comparison

and width of 8 and 2 respectively against a 12 and 6 length and width, the former proved to be marginally better as can be seen in Figure 12. Note that the authors



Figure 12: Context Path Length and Width Hyper-parameter Comparison

of code2vec used values of 8 and 2 for paths length and width respectively in their experiments.

5.6.5 Number of Epochs

The number of epochs refer to how many forward and backward passes the dataset goes through by the neural network during the training phase. The model usually keeps on improving with each subsequent pass until it converges, then any further training will not improve the model's performance after that. By comparing the number of epochs of 50 against 200, the former proved to be much faster to train as one might expect, while also producing slightly better results as can be seen in Figure 13. By investigating the output log during the model training, it is

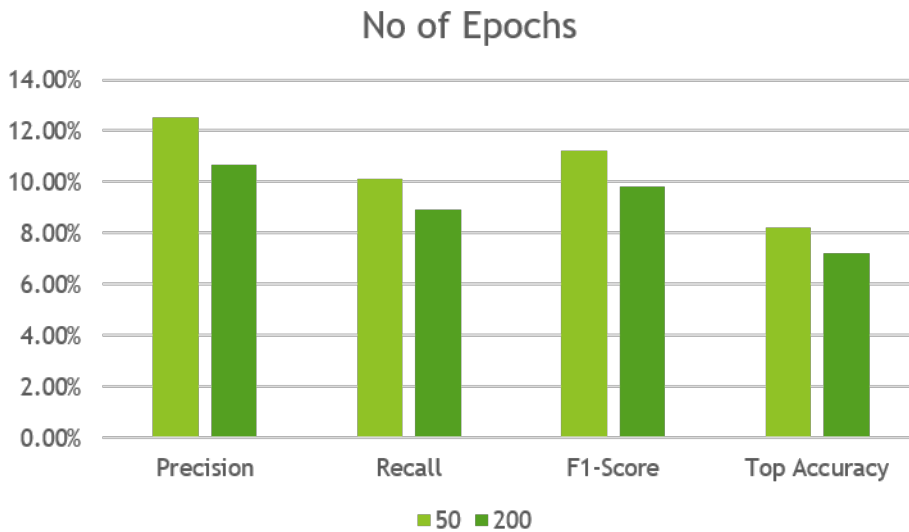


Figure 13: Number of Epochs Hyper-parameter Comparison

noticed that the model tend to reach its peak performance in the early epochs, with further training not resulting in any added benefits, but rather some performance degradation.

5.6.6 Number of Contexts

The number of contexts refer to the number of path contexts that were extracted from a sample method. By comparing a context count of 50, 100, and 400, the results were too close to draw any definitive conclusions as can be seen in Figure 14. Note

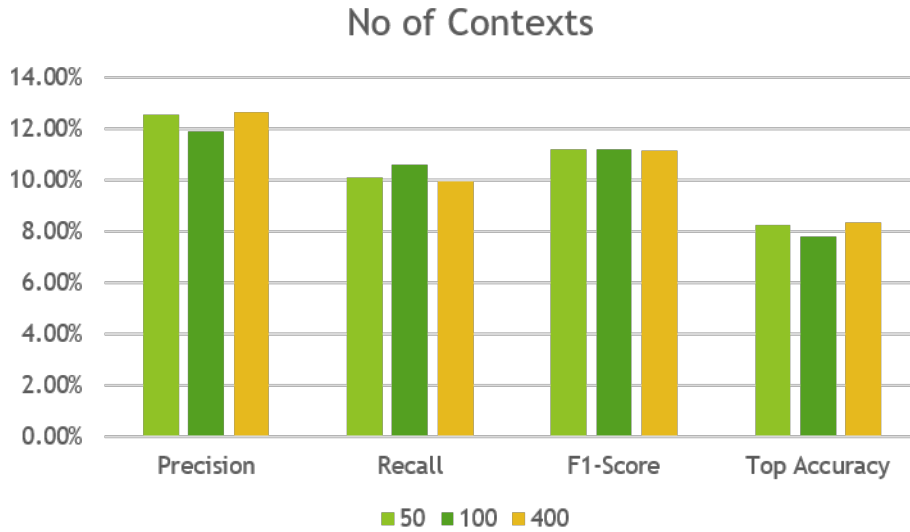


Figure 14: Number of Path Contexts Hyper-parameter Comparison

that the authors of code2vec used a value of 200 contexts in their experiments, and mentioned that adding additional contexts will yield very small performance gains, which conforms the results obtained from Figure 14.

5.6.7 Hyper-parameters Used

After comparing the results from the hyper-parameter tuning experiments, the following parameters were used for training the Java, Python, and JavaScript models:

- Max number of epochs for training the model: 30
- Training and Testing batch size(s): 1024

- Framework used: TensorFlow
- Starting number of words considered for each prediction: 10
- Default embedding size: 128
- Drop out rate: 0.75
- Max number of contexts: 300
- Max path vocabulary size: 911417
- Max target vocabulary size: 261245
- Max token vocabulary size: 1301136
- Max trained versions to keep: 3

Some Key decisions in Hyper-parameter tuning:

- The max number of the latest trained model versions to keep was reduced from the default 10 to 3 to save on disk space.
- The max number of extracted contexts was increased from the default 200 to 300 to make up for the smaller sized datasets.
- The max number of epochs was increased from the default 20 to 30 to make up for the smaller sized datasets.
- The vanilla tensor flow back-end was used instead of the Keras implementation, since it had all the latest features and fixes.
- The rest of the default hyper-parameters were used as is, since those settings produced the highest results in the original research by Alon et al.

- The model performs early stopping based on the epoch version that maximizes the F1-Score on the validation dataset

5.7 Model Evaluation

After the training step is completed, a model can be released so it can be easily used for inference. A released model does not have any of its training parameters anymore, Thus it can not be trained further or used for transfer learning with other datasets. The advantage of releasing a model is to reduce the model's size by up to 3 times.

Along with the trained model, all of the vocabularies from the training phase (i.e. all the possible prediction labels) are saved to a 'dictionaries.bin' file. This file is used in tandem with released model to run inference on the test data.

There are two main methods to evaluate a code2vec model, either by passing the model the test data file that was extracted in the feature extraction phase, and running batch evaluation on it and viewing the results in an output log file. Or by manually evaluating a single method from a single input file and examining the model's predictions on the output console. Both methods are explored in chapter 6.

CHAPTER 6

Results

6.1 Precision, Recall, F1-Score, and Accuracy

All results are measured in term of precision, recall, F1 score, and accuracy.

Some definitions before we present our results:

- Precision also known as positive predictive value, is a measure of the correctly predicted instances relative to the set of all positively predicted instances. A low precision implies that the model incorrectly predicted a lot of the test data (causing a large number of false positives).
- Recall also known as sensitivity, is a measure of the correctly predicted instances relative to the set of all instances, a low recall implies that the model did not classify a lot of correct test data (causing a large number of false negatives)
- F1 score also known as F-score, is a measurement that combines both the precision and recall results of a model. It's defined as the harmonic mean of the models precision and recall. Unlike the accuracy measurement, F-score takes into account any imbalance that might have been introduced due to an unevenly distributed dataset.
- Accuracy is the most intuitive measurement, it is the ratio of the correctly predicted labels relative to the all predictions that the model made. However in an unbalanced dataset, a high accuracy could be a sign of the model over-fitting and will often fail on unseen test data.

Within code2vec's evaluation metrics, accuracy refer to the assigned value for

each label after scoring a method, where as the most probable label as predicted by the model will be assigned the highest accuracy score. The number of possible labels considered by the model depends on the number of kwords used as a hyper-parameter.

The total number of potential labels that are available for the model during the training phase will depend on the size of the vocabulary extracted from the dataset. Note that the code2vec model can not predict out of vocabulary labels, i.e. it can not assign a label for a method that it did not see during its training phase.

6.2 Results

The model considers the top 10 possible labels during prediction (as configured in the hyper-parameters), and assigns an accuracy value to each prediction. In this paper, we look at the highest accuracy prediction which corresponds to the most likely label/name for the test method (Top accuracy result from all 10 possible values).

6.2.1 Training Results

The results from training the Java, Python, and JavaScript after the training is complete and the model converged are shown in Table 5 and Figure 15:

The results in Table 5 shows a clear performance difference among the 3 models, with the Java model coming out ahead followed by Python model, and the JavaScript one coming last. The difference in performance -although expected- can be attributed to both the size of the datasets and the nature of the programming languages themselves.

Even though both Java and Python datasets were collected from the top 1000

Table 5: The results from training the models from scratch

Language	Java	Python	JavaScript
Training Time	23:01:57	22H:47M:2S	1H:31M:4S
Number of epochs	11	13	29
Precision	37.8	30.4%	18.3%
Recall	26.9	19.3%	19.3%
F1 score	31.5	23.6%	18.8%
Top accuracy score	30.7	24.2%	20.4%

projects of GitHub, the Java dataset is still roughly 1.7x the size of the Python one with 2.4x the number of files. This translated to almost double (1.9x to be exact) the number of training examples (i.e. extracted methods) used in the Java model compared to its Python equivalent.

Thus despite the Python model using more contexts per method (1.3x) than the Java one during training, the models appears to benefit more from a larger corpus of data rather than smaller but 'denser' one.

The JavaScript results on the other hand are not surprising considering the much smaller dataset size that were used for both training and validation, as evidenced by the much smaller training time.

6.2.2 Testing Results

The results from testing the Java, Python, and JavaScript using the test datasets on a released model, are shown in Table 6 and Figure!16:

As shown from Tables 5 and 6, For the Java model, the testing scores are in-line with validation ones, and the results are as expected.

The same can not be said for the Python and JavaScript ones. The Python model

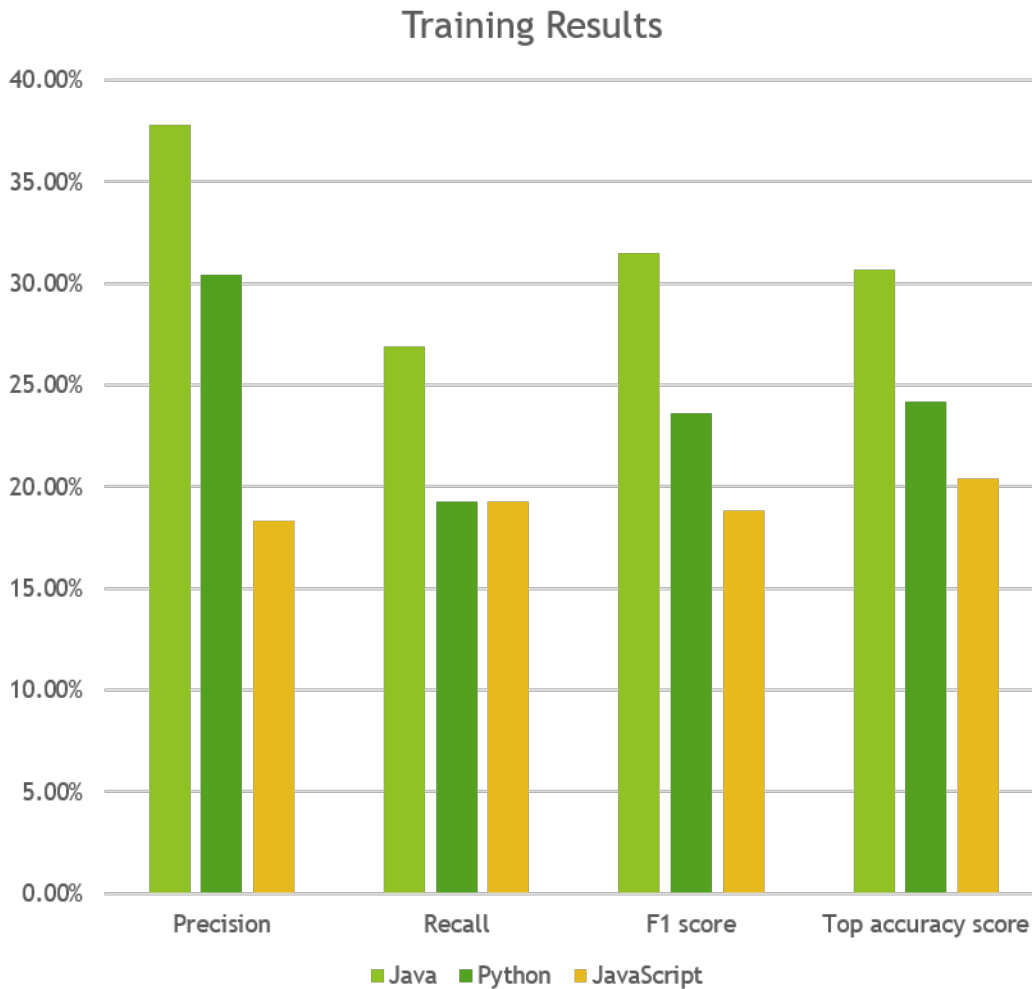


Figure 15: Training Results for Java, Python, and JavaScript

Table 6: The results from the testing dataset

Language	Java	Python	JavaScript
Testing Time	07M:14S	01M:28S	0M:6S
Precision	39.4	22.9%	23.5%
Recall	27.3	13.7%	25.7%
F1 score	32.2	17.2%	24.5%
Lowest accuracy score	23.2	11.9%	23.2%
Top accuracy score	31.3	16.7%	25.1%

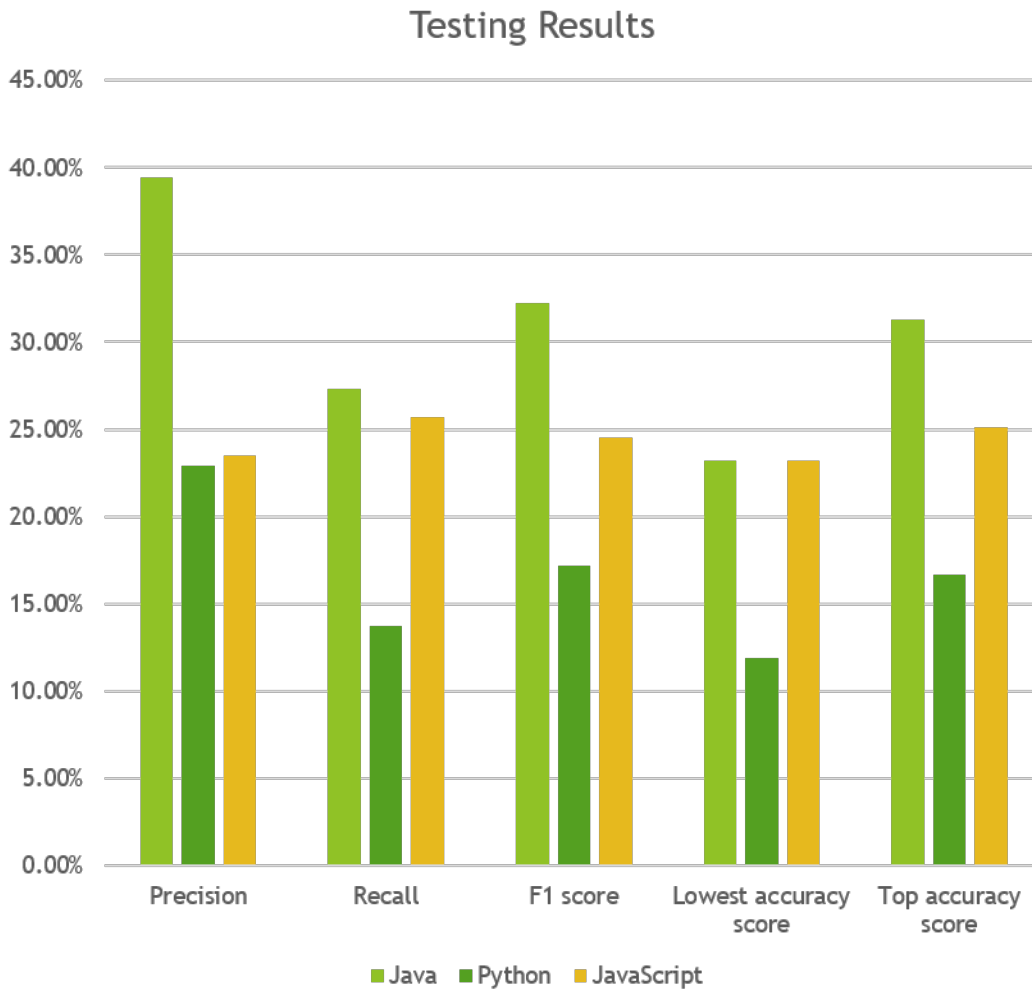


Figure 16: Testing Results for Java, Python, and JavaScript

suffered a 5% to 7.5% loss in all metrics. While on the other hand the JavaScript model gained a 4% to 7.5% increase in all metrics. Such results could be attributed to the size of the datasets, with JavaScript being much smaller and as such not a true representative of a wide test data. Or more likely, it enforces the hypothesis that the dynamic nature of these languages (utilizing type inference instead of all the variables being declared with types as with Java code) makes it hard for the model to generalize among unseen source code. This coupled with fewer syntax and less verbose methods are the reasons behind the difference in results between Java, Python, and JavaScript

in the testing scores.

6.3 Transfer Learning

In an attempt to improve upon the results obtained from the experiment, but without requiring a much larger dataset, transfer learning techniques were used. By using a Java model that was trained on a much larger dataset and fine-tuning it using the same datasets from the previous experiments, i.e. The Python and JavaScript pre-processed datasets.

For this experiment, A non-striped (i.e. not in a release state and thus can be trained further) trained model was used. That model was originally trained by Alon et al [1] on a large Java dataset of 14 million examples from 10000 projects roughly 10 times the size of datasets used in this research. By loading this model and continuously training it on the datasets from other languages using the same hyper-parameters from the previous experiment, the results can be seen in Table 7 and Figure 17.

Table 7: The results from fine-tuning the models -Transfer learning from the Java 14m trained model-

Language	Python	JavaScript
Training Time	15H:50M:43S	0H:29M:16S
Number of epochs	8	9
Precision	24.6%	15.8%
Recall	13.6%	10.6%
F1 score	17.5%	12.7%
Top accuracy score	19.6%	14.8%

It's interesting to see that both the Python and JavaScript models have lower scores across all metrics than their trained from scratch counterparts. This shows

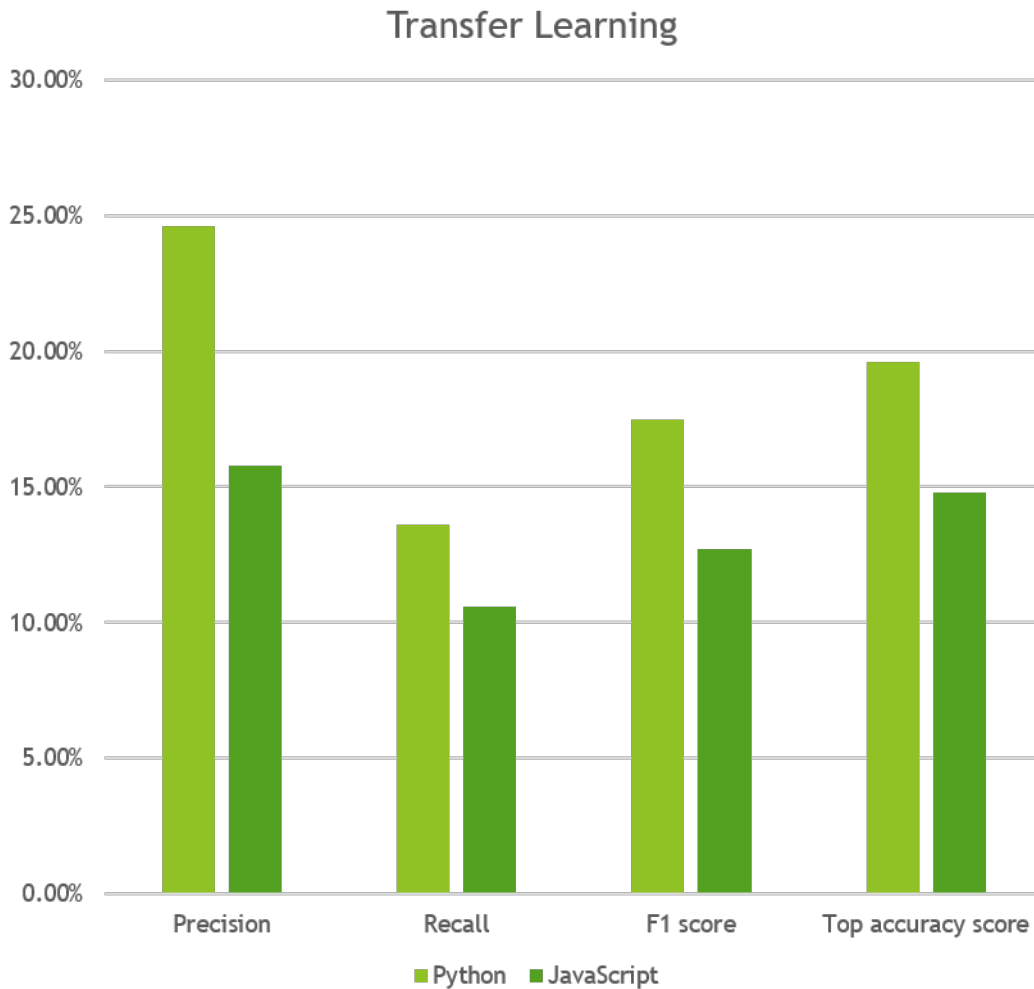


Figure 17: Transfer Learning Results for Python and JavaScript

that even though the code2vec architecture itself is language agnostic, the model did not gain any advantages from the previously trained Java model.

And that the lower scores that the fine-tuned models achieved can be attributed directly to the training gained from the Python/JavaScript dataset and not from the transfer learning process itself, i.e. the F1 score achieved for the hyper-tuned Python model is 17.5% after 8 epochs, which is in line with the trained from scratch Python model after the same number of epochs, since the F1 score after 13 epochs was 23.6%.

This leads us to conclude that the technical and syntax differences between the 2 static and dynamic pairs of languages (Java and Python, Java and JavaScript) are too great for the model to generalize across languages. It would be interesting to see if transfer learning would work better among the similar types of programming languages (ex: Java and C # or Python and JavaScript).

6.4 Different Parsers

The ANTLR parser was picked in this research as the parser of choice for feature extraction in the ASTminer library since it has support for all target languages (Java, Python, JavaScript) so the comparison would be the same, and provides an easy extendable options. However other parsers could be used with ASTminer to produce similar results. In this experiment, The parser GumTree was used on the Python dataset to see of the choice of parser during feature extraction would affect the final performance of the code2vec model.

6.4.1 GumTree

GumTree is a tool that deals with source code, and can compute the differences between source code files. It is also able to convert code snippets into language agnostic tree formats. For the purpose of using it with ASTminer, it can be used as a back-end parser for Java and Python files.

The result from the feature extraction phase can be seen in Table 8:

It is clear from table 8, that the features extracted from ANTLR and GumTree are identical, with ANTLR being faster by a minute or two, which is statistically insignificant.

Table 8: Feature extraction for Python Dataset using GumTree Parser

Language	Train	Val	Test
Dataset size	1.2 GB	257.3 MB	77.2 MB
No of Projects	727	145	30
No of Files	155264	29032	9051
Extraction Time	1:03:00	13:40	04:00
Max No Contexts	300	300	300
Avg No Contexts	254.48	253.053	265.939
Total Examples	1377753	334865	84847
Max Path Length	8	8	8
Max Path Width	2	2	2
Parser	gumtree	gumtree	gumtree
Max Tree Size (no of nodes)	500	500	500

After training the model from scratch again using the same hyper-parameters from before. But this time using the features extracted using GumTree, the results in Table 9 were obtained:

Table 9: The results from training the Python model from scratch using GumTree

Language	Python
Training Time	22H:58M:23S
Number of epochs	13
Precision	28.5%
Recall	18.1%
F1 score	22.2%
Top accuracy score	23.8%

Again, it comes at no surprise that the results obtained from this 'GumTree extracted' model is very similar (with a small statistically insignificant differences) to the ones obtained using the 'ANTLR extracted' one. Which aligns with how the code2vec model operate being language agnostic.

Since the model does not care about how the features were extracted from the source code files. and since both parsers manage to transform the source code files into their AST representation and extract the same number of path contexts (i.e. embeddings), it follows that the results would be identical. With personal preference and ease of use being the only differentiator.

6.5 Python Large

In a final attempt to improve the results of the Python model, the larger Python dataset was used. This dataset was collected from a combination of GitHub and from the 150k Python dataset available from SRILab [29]. This was done to increase the size of dataset by as much as possible. The dataset was split into 90%, 5%, 5% train, val, and test respectively. This followed the same split that was performed on the Java large dataset by Alon et al. [1].

The same hyper-parameters from the python medium experiment were used, but the maximum number of contexts was increased to 400. The feature extraction for the entire dataset took almost 2 hours and 45 minutes, with an average extracted context count of 347.5 and total vocabulary size of 261,245 tokens. Overall, the model trained around 3,166,778 training samples over a period of 2 days with the model’s performance peaking in the $8_T h$ epoch. Both the training and testing results can be seen in Table 10 and Figure 18.

The results obtained from the Python large dataset was much better than the previous experiments, which proves that for this type of probabilistic models like code2vec, dataset size have the most important factor in determining the model’s performance. Even more interesting is the fact that, in this experiment, the testing results were inline with training ones, suffering no performance degradation, with a

Table 10: The results from training the Python model from scratch using the python large dataset

	Training Results	Testing Results
Precision	44.9%	43.4%
Recall	30.5%	30.5%
F1 score	36.2%	35.8%
Lowest accuracy score	30%	29.5%
Top accuracy score	38.2%	37.5%

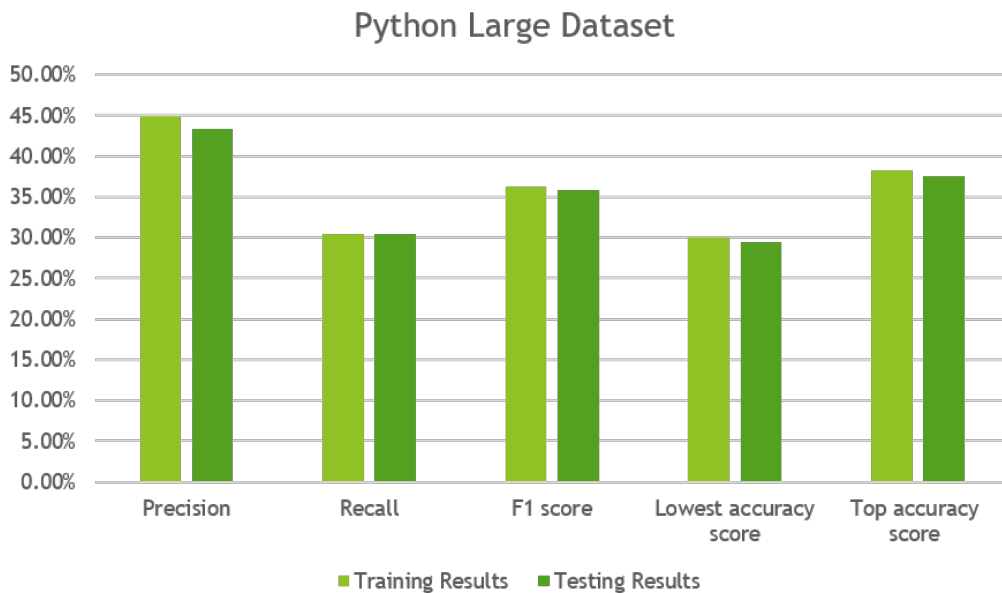


Figure 18: Training and Testing Results for Python large

very small statistically insignificant difference between the two results. This could be due to the larger and more diverse sized combined dataset, or the difference in the dataset split itself.

6.5.1 Python Datasets Comparison

A quick comparison at the different sized python datasets and their respective performance after training each one from scratch can be seen in Table 11 and Figure 19

Table 11: Python Datasets

	Small	Medium	Large
Dataset Size	118.7 MB	1.5 GB	2.7 GB
No of Projects	24	902	7188
No of Files	22016	194903	366016

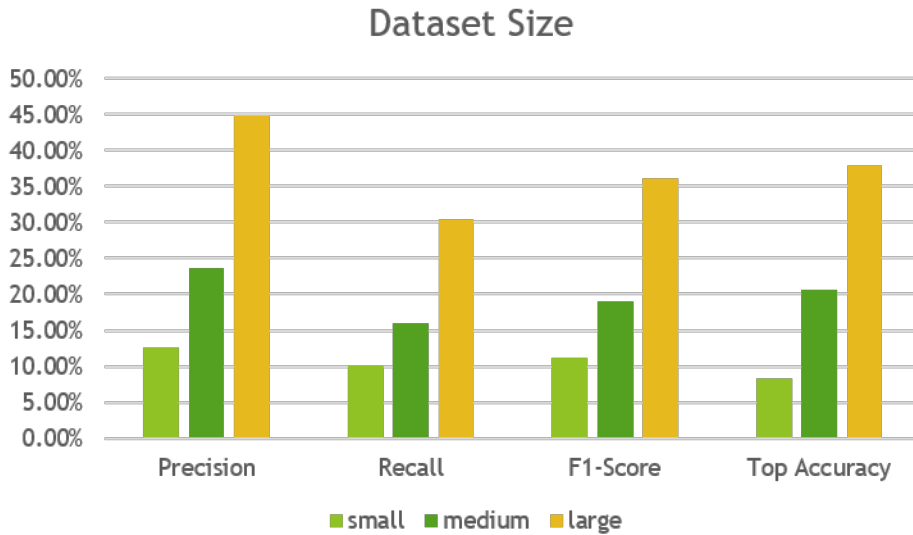


Figure 19: Python Training Results Comparison

The results in Table 11, along with the ones obtained from the hyper-parameter tuning experiments, confirms the importance of dataset size in training code2vec and other similar models. No other factor yielded as much difference in performance as dataset size, which is inline with how deep learning models work in general.

6.5.2 Baseline Comparison

Finally, we compare our best results for dynamic languages that was obtained from training the Python large dataset, with a little over 3 million methods, against the static languages baseline Java model that was trained by Alon et al. on the Java

large dataset, containing 12 million methods [1]. The results can be seen in Figure 20.

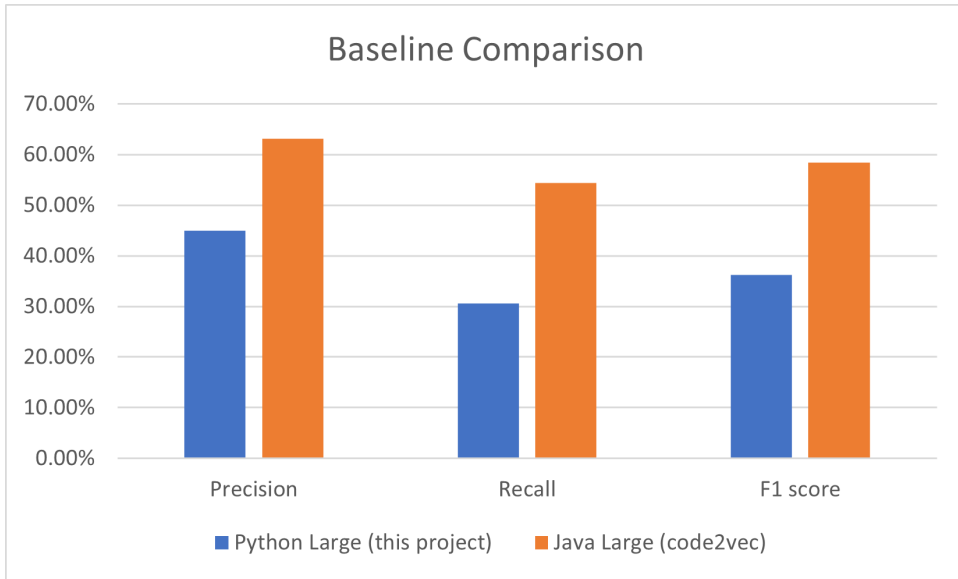


Figure 20: Python and Java Baseline Comparison

The Python model while still lagging behind its Java equivalent in all metrics, managed to reach around 70% of the performance scores of the Java one. Implies that while there is still a gap in performance between static and dynamic languages in this particular machine learning tasks, it is possible with further experimentation to bridge that difference, and raise the dynamic language's performance to the level of its static counterpart.

CHAPTER 7

Conclusion and Future Work

7.1 Summary

This project aims to provide a starting point for studying the differences in performance among static and dynamic programming languages when it comes to probabilistic machine learning models that act on source code.

It briefly explains the motivation and history behind machine learning on source code, some of the current state of the art research on the subject. And a deep dive into `code2vec`, which is one of the most innovative deep learning models for MLoNCode tasks.

The projects experimented with extending `code2vec` to work with the Python and JavaScript programming languages using the `ASTminer` library. It compared the training results from the two aforementioned languages against the standard Java one, and experimented with techniques like transfer learning and using different parsers in order to improve their performance.

The experiments showed the difficulty of dealing with dynamic languages in machine learning tasks on source code, with the best dynamic language model reaching 70% of the performance of its static counterpart. And provided a baseline for future work to improve their performance and explore their potential applications.

7.2 Conclusion

In conclusion, machine learning on source code is without a doubt an exiting new domain for ML researchers, it has picked up a lot of interest in recent years as

evidenced by the increasing number of published research papers on the subject.

However, the majority of this interest seem directed towards static languages like Java or C#. By completing this project, the authors hope to shine a new light on the other type of programming languages, and showcase how dynamic languages like Python or JavaScript would too benefit from these ML applications.

7.3 Future Work

- Since the final results for the dynamic languages are still less than their static counterparts. A good start for future work, would be to improve the performance with further experimentation.
- Due to preprocessing issues, the final size of the JavaScript used for training the model was much smaller than initially expected, as such for future work, and after solving these issues, it would be interesting to see how a much larger JavaScript dataset would affect the performance of the trained model.
- It would also be interesting to see how functional languages would perform. Given their nature, functional languages hold all of the application's logic within functions only, thus models trained on functional languages like Haskell or Clojure could in theory perform better than their Object oriented counterparts like Java.
- Further experimentation to explore tasks such as detecting malware in JavaScript source code files, or detecting run-time errors in Python scripts could be done and compared with similar tasks that are often performed on the Java model.

LIST OF REFERENCES

- [1] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: Learning distributed representations of code,” 2018.
- [2] V. Kovalenko, E. Bogomolov, T. Bryksin, and A. Bacchelli, “Pathminer: a library for mining of path-based representations of code,” in *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 2019, pp. 13–17.
- [3] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” 2018.
- [4] J. A. Briem, J. Smit, H. Sellik, and P. Rapoport, “Using distributed representation of code for bug detection,” 2019.
- [5] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Suggesting accurate method and class names,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 38–49. [Online]. Available: <https://doi.org/10.1145/2786805.2786849>
- [6] U. Bandara and G. Wijayarathna, “A machine learning based tool for source code plagiarism detection,” *International Journal of Machine Learning and Computing*, vol. 1, pp. 337–343, 01 2011.
- [7] M. O. F. Rokon, P. Yan, R. Islam, and M. Faloutsos, “Repo2vec: A comprehensive embedding approach for determining repository similarity,” 2021.
- [8] R. Compton, E. Frank, P. Patros, and A. Koay, “Embedding java classes with code2vec: Improvements from variable obfuscation,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 243–253. [Online]. Available: <https://doi.org/10.1145/3379597.3387445>
- [9] Z. Bilgin, M. A. Ersoy, E. U. Soykan, E. Tomur, P. Çomak, and L. Karaçay, “Vulnerability prediction from source code using machine learning,” *IEEE Access*, vol. 8, pp. 150 672–150 684, 2020.
- [10] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, “Self-supervised bug detection and repair,” 2021.

- [11] P. Fernandes, M. Allamanis, and M. Brockschmidt, “Structured neural summarization,” 2021.
- [12] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, “Generative code modeling with graphs,” 2019.
- [13] U. Alon, R. Sadaka, O. Levy, and E. Yahav, “Structural language models of code,” in *International Conference on Machine Learning*. PMLR, 2020, pp. 245–256.
- [14] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Code-searchnet challenge: Evaluating the state of semantic code search,” 2020.
- [15] V. Raychev, M. Vechev, and A. Krause, “Predicting program properties from “big code”,” in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 111–124. [Online]. Available: <https://doi.org/10.1145/2676726.2677009>
- [16] M. Allamanis, M. Brockschmidt, and M. Khademi, “Learning to represent programs with graphs,” 2018.
- [17] B. Ray, D. Posnett, V. Filkov, and P. Devanbu, “A large scale study of programming languages and code quality in github,” ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 155–165. [Online]. Available: <https://doi.org/10.1145/2635868.2635922>
- [18] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” 2013.
- [19] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “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)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. [Online]. Available: <https://www.aclweb.org/anthology/P16-1195>
- [20] M. Allamanis and M. Brockschmidt, “Smartpaste: Learning to adapt source code,” 2017.
- [21] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 87–98. [Online]. Available: <https://doi.org/10.1145/2970276.2970326>

- [22] V. Zhong, C. Xiong, and R. Socher, “Seq2sql: Generating structured queries from natural language using reinforcement learning,” 2017.
- [23] H. Sellik, O. van Paridon, G. Gousios, and M. Aniche, “Learning off-by-one mistakes: An empirical study,” 2021.
- [24] A. Prasad and C. V. Lopes, “Code clone detection using code2vec,” UC Irvine Electronic Theses and Dissertations, 2020.
- [25] J. Svajlenko and C. K. Roy, “Bigcloneeval: A clone detection tool evaluation framework with bigclonebench,” in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 596–600.
- [26] V. Saini, F. Farmahinifarahani, Y. Lu, D. Yang, P. Martins, H. Sajjani, P. Baldi, and C. V. Lopes, “Towards automating precision studies of clone detectors,” ser. ICSE '19. IEEE Press, 2019, p. 49–59. [Online]. Available: <https://doi.org/10.1109/ICSE.2019.00023>
- [27] T. Parr, “the definitive antlr 4 reference. [frisco, tx]: The pragmatic programmers,” 2014.
- [28] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhutdinov, R. Zemel, and Y. Bengio, “Show, attend and tell: Neural image caption generation with visual attention,” 2016.
- [29] V. Raychev, P. Bielik, and M. Vechev, “Probabilistic model for code with decision trees,” *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 731–747, 2016.

APPENDIX