

2023

## Increasing Code Completion Accuracy in Pythia Models for Non-Standard Python Libraries

David Buksbaum

Follow this and additional works at: [https://nsuworks.nova.edu/gscis\\_etd](https://nsuworks.nova.edu/gscis_etd)



Part of the [Computer Sciences Commons](#)

### Share Feedback About This Item

---

This Dissertation is brought to you by the College of Computing and Engineering at NSUWorks. It has been accepted for inclusion in CCE Theses and Dissertations by an authorized administrator of NSUWorks. For more information, please contact [nsuworks@nova.edu](mailto:nsuworks@nova.edu).

Increasing Code Completion Accuracy in Pythia Models  
for Non-Standard Python Libraries

by


David T. Buksbaum

A dissertation submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy  
in  
Computer Science


College of Computing and Engineering  
Nova Southeastern University

2023


We hereby certify that this dissertation, submitted by David Buksbaum conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

  
\_\_\_\_\_  
Francisco J. Mitropoulos, Ph.D.  
Chairperson of Dissertation Committee

11/29/23  
Date


  
\_\_\_\_\_  
Michael Laszlo, Ph.D.  
Dissertation Committee Member

11/29/23  
Date

  
\_\_\_\_\_  
Sumitra Mukherjee, Ph.D.  
Dissertation Committee Member

11/29/23  
Date

Approved:

  
\_\_\_\_\_  
Meline Kevorkian, Ed.D.  
Dean, College of Computing and Engineering

11/29/23  
Date

College of Computing and Engineering  
Nova Southeastern University

2023

An Abstract of a Dissertation Submitted to Nova Southeastern University  
In Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Increasing Code Completion Accuracy in Pythia Models  
for Non-Standard Python Libraries

by

David T. Buksbaum

November 29, 2023

Contemporary software development with modern programming languages leverages Integrated Development Environments, smart text editors, and similar tooling with code completion capabilities to increase the efficiency of software developers. Recent code completion research has shown that the combination of natural language processing with recurrent neural networks configured with long short-term memory can improve the accuracy of code completion predictions over prior models. It is well known that the accuracy of predictive systems based on training data is correlated to the quality and the quantity of the training data. This dissertation demonstrates that by expanding the training data set to include more references to specific Python third-party modules, the quality of the predictions increase for those specific Python third-party modules without degrading the quality of predictions of the originally represented modules.

## Acknowledgements

While an individual receives a doctorate at the end of a significant effort, it is not the individual alone who invested in that effort. So many made it possible for me to accomplish this journey. I could never thank them all, so thank you to anyone I missed.

My committee chair and advisor Dr. Francisco Mitropoulos has been exceptionally patient with me as the universe opted to provide me with an endless stream of non-dissertation distractions along my journey. His pointed questions at just the right time were invaluable. I must also thank the rest of the committee members, Dr. Michael Laszlo and Dr. Sumitra Mukherjee. Their questions and feedback were essential to helping me rethink what sometimes seemed like an endless quagmire of complexity.

The professors for all my courses and NSU have been truly invaluable in guiding my thinking from a pre-doctorate perspective to one required to complete this task.

There are those I have worked with that were critical to my ability to complete this dissertation. Specifically, I must thank Robert Palatnick for both his understanding in allowing me the time away from work to pursue this effort and his encouragement to do so. Alexandros Deliyannis listened to me prattle on endlessly about my approach and pushed me to explain it better which led to my understanding it better. There are many others I worked with that need thanks and to all of them – thank you.

To my dearest friend Shaheen for providing just the right mixture of sarcasm, humor, and support. Pushing me when I needed it, reminding me to step away when I needed to, questioning my sanity multiple times, and always being there. Cheers Mate!

My children, Jared, Scott, and Rachel have shared my time with this effort for too many years, and they kept the grumbling about it to a minimum. Now they are entering the stage where I can support their similar journeys. My wife Hope has given the most. Her understanding and support never wavered despite compromising for years to let me pursue this dream of mine. While she believes I misunderstood the goal of a mid-life crisis, she always encouraged me to complete it. I thank and love them all dearly.

And last, but certainly not least, my parents Murray and Nina. They knew this would happen long before I even dreamt of it. Buying my first computer at 10 years old; driving me to numerous hobbyist meetups; paying unforeseen phone bills while I wandered the world of Usenet and BBSes; never once complaining despite no understanding of what I was doing. They provided endless encouragement to pursue my dreams, to grow and go further, even when I had become complacent. All that I have accomplished is because of their belief in me and the start they gave me, and I am eternally grateful.

This is for you dad.

D.T.B.

## Table of Contents

**Abstract** iii

**Acknowledgements** iv

**List of Tables** viii

**List of Figures** x

### Chapters

**1. Introduction** 1

Background 1

Problem Statement 6

Dissertation Goal 9

Research Questions 11

Relevance and Significance 11

Barriers and Issues 14

Assumptions, Limitations, and Delimitations 15

Summary 16

**2. Review of the Literature** 17

Introduction 17

Intelligent Assistants 19

Code Completion 22

Optimistic Code Completion 22

Intelligent Code Completion 26

Intelligent Code Completion with Bayesian Networks 30

Improving Best Matching Neighbor for Dynamically Typed Languages 34

Long Short-Term Memory Neural Networks 38

Measuring Code Completion 42

Prefix Scoring with Change Replay 42

Mean Reciprocal Rank 44

Top-k Accuracy 44

Recall 45

Precision 45

Replicating Pythia	46
<b>3. Methodology</b>	<b>47</b>
Introduction	47
Data Collection	48
GitHub Data Collector	49
Preprocessing	50
Dataset Generator	50
Code Processing	51
Dependency Evaluator	53
Encoding Code Snippets	53
Offline Model Training	54
Pythia Deep Learning Model Training	54
Hyperparameter Tuning	55
Evaluation	56
Key Questions	56
Evaluation Metrics	57
Differences from Pythia	58
Preprocessing	58
Model Training	58
Model Quantization	58
Serving Recommendations	59
Conclusion	59
<b>4. Results</b>	<b>60</b>
Introduction	60
Changes from Planned Methodology	61
Baseline Model	61
Expansion by 100 Repositories	65
markdown Library	65
tqdm Library	67
yaml Library	69
Results Analysis	70
Expansion by 270 Repositories	71
markdown Library	72
tqdm Library	74
yaml Library	76
Results Analysis	78
Impact Beyond Target Module	78
<b>5. Conclusions, Implications, Recommendations, and Summary</b>	<b>84</b>
Conclusions	84
Implications	88
Recommendations	88
Summary	89

**Appendices 90**

**A. Hyperparameters 91**

**B. Experiment Result Data 93**

**References 101**



## List of Tables

Table 1 – Top 10 Commands Executed by the Most Developers	17
Table 2 – Top 10 Commands Executed Across all 41 Developers	18
Table 3 – Scores for the untyped algorithms of all projects (Robbes & Lanza, 2008)	25
Table 4 – PyReco Results for Python Standard Libraries (D’Souza et al., 2016)	36
Table 5 – PyReco Results for Python Third-Party Libraries (D’Souza et al., 2016)	37
Table 6 – Accuracy of Pythia and Markov Chain (Svyatkovskiy et al., 2019)	39
Table 7 – Top-5 Accuracy of Different RNN Models for Pythia (Svyatkovskiy et al., 2019)	40
Table 8 – Comparison of Accuracy and MRR of Python and Four Models (Svyatkovskiy et al., 2019)	41
Table 9 - Comparison of Pythia and Baseline Model for K5 Accuracy	63
Table 10 - Comparison of Pythia and Baseline Model for MRR	64
Table 11 - Markdown Library Results using Baseline Model	66
Table 12 - Markdown Library Results using Baseline+100 Model.	66
Table 13 - Markdown Library Improvement Results of Baseline v Baseline+100	67
Table 14- TQDM Library Results using Baseline Model	67
Table 15 – TQDM Library Results using Baseline+100 Model.	68
Table 16 – TQDM Library Improvement Results of Baseline v Baseline+100	68
Table 17 - Yaml Library Results using Baseline Model	69
Table 18 - Yaml Library Results using Baseline+100 Model.	70
Table 19 – Markdown Library Results for Baseline, Baseline+100, and Baseline+270 Models	72

Table 20 – Markdown Library Change % for Baseline, Baseline+100, and Baseline+270

Models 74

Table 21 - TQDM Library Results for Baseline, Baseline+100, and Baseline+270 Models

75

Table 22 – Yaml Library Change % for Baseline, Baseline+100, and Baseline+270

Models 76

Table 23 - Yaml Library Results for Baseline, Baseline+100, and Baseline+270 Models

77

Table 24 – Yaml Library Change % for Baseline, Baseline+100, and Baseline+270

Models 77

Table 25 - Overall Raw Counts and MRR for Baseline and 10 Derivative Models 80

Table 26 - Overall Percentages for Baseline and 10 Derivative Models 81

## List of Figures

- Figure 1 – JetBrains IntelliJ Lexical Code Completion 3
- Figure 2 – Visual Studio Code Python Lexical Code Completion 3
- Figure 3 – PyCharm 2020.2 Creating the Python Index 4
- Figure 4 – Python Socket Code Completion in Visual Studio Code Part 1 4
- Figure 5 – Python Socket Code Completion in Visual Studio Code Part 2 5
- Figure 6 – Pythia Preprocessing Phase 10
- Figure 7 – Filtered Code Completion in Eclipse (Robbes & Lanza, 2008) 23
- Figure 8 – Performance of EcCCS, FreqCCS, ArCCS, and BMNCCS (Bruch et al., 2009)  
29
- Figure 9 – Conditional Probabilities in a Based Bayesian Network (Proksch et al., 2015)  
31
- Figure 10 – Structural Representation of the Bayesian Network used in PBN (Proksch et al., 2015) 31
- Figure 11 – Quality and Size for Different Distances Compared to BMN (Proksch et al., 2015) 33
- Figure 12 – Effect of Increasing Number of Object Usages for SWT Button (Proksch et al., 2015) 33
- Figure 13— Workflow from source code to testing. 48
- Figure 14— Pythia Neural Network Architecture 55
- Figure 15— Pythia Hyperparameters (Svyatkovskiy et al., 2019) 56
- Figure 16 - Distribution of Method Calls 83

Figure 17 - Markdown Accuracy for Baseline, Baseline+100, and Baseline+270 85

Figure 18 - Missing Markdown Predictions for Baseline, Baseline+100, and

Baseline+270 85

Figure 19 - tqdm Accuracy for Baseline, Baseline+100, and Baseline+270 86

Figure 20 - Missing tqdm Predictions for Baseline, Baseline+100, and Baseline+270 86

Figure 21 - yaml Accuracy for Baseline, Baseline+100, and Baseline+270 87

Figure 22 - Missing yaml Predictions for Baseline, Baseline+100, and Baseline+270 87

Figure 23 - Gensim word2vec Hyperparameters 91

Figure 24 - LSTM Hyperparameters 92

Figure 25 - Baseline Data 94

Figure 26 - Markdown-100 Variant 95

Figure 27 - Markdown-270 Variant 96

Figure 28 - tqdm-100 Variant 97

Figure 29 - tqdm-270 Variant 98

Figure 30 - yaml-100 Variant 99

Figure 31 - yaml-270 Variant 100

# Chapter 1

## Introduction

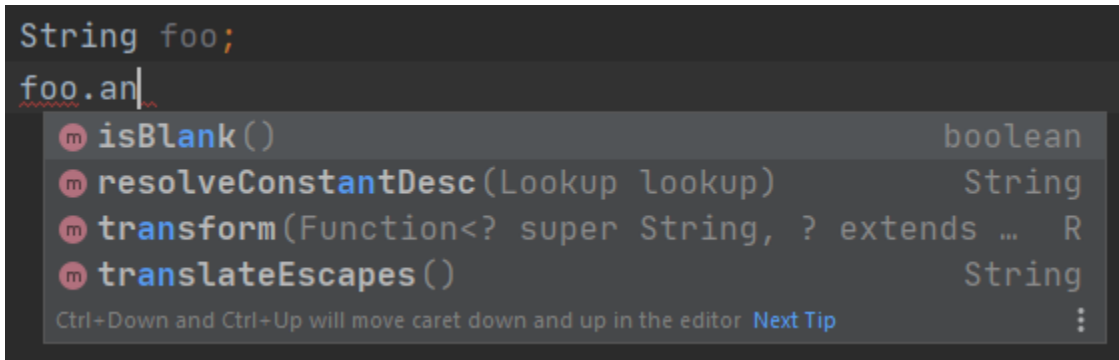
### Background

The software development process has centered around the Integrated Development Environment (IDE) as the primary tool for developers (Gail C Murphy, 2019). These IDEs analyze the content and structure of the code in projects to form a contextual awareness about the developer's intent, and to provide capabilities to minimize the work of the developer. One of the most used capabilities provided by an IDE is code completion (G.C. Murphy et al., 2006). The ability for the IDE to suggest the method to call based on the context of the developer's location in the code has become a ubiquitous feature that is expected, with Murphy stating code completion to be as popular as Cut and Paste.

The reasoning behind why code completion is so popular in IDEs is debated, with Stylos & Clarke (2007) claiming it is because of API exploration and discovery, and Mărășoiu, Church, & Blackwell (2015) stating it is due to the improvement to the speed and accuracy of the developer. The root cause of why it is so popular may not be fully determined, but the result of broad demand amongst developers for code completion is accepted and supported by prior research.

Code completion provides the developer with a set of options available at a specific context point in the code, such as a method for a specific type. With smaller types, this is a highly effective mechanism of providing rapid results, usually sorted alphabetically, that relies on the developer's knowledge of the type to choose the correct method. However, as development languages have grown to support more complex constructs, the types available in those languages have grown in the number of methods that they make available. These lists of methods can exceed several hundred, making it unwieldy for a developer to find the right method, or even know which the right method is to select. Mărășoiu, Church, & Blackwell (2015) defines two strategies for reducing the list of possible methods shown to the developer: lexical and semantic.

The lexical strategy relies on pattern matching to restrict the suggestions provided to the developer. Typically, this is done by pattern matching from the start of the method, and reducing the list of available methods as the developer types more characters. Some IDEs improve upon this by matching the exact character sequence against anywhere in the method name, such as JetBrains IntelliJ in Figure 1. Other IDEs look for the letters to appear anywhere without requiring the sequence to match, such as Visual Studio Code's Python matcher shown in Figure 2.


 A screenshot of the JetBrains IntelliJ IDE showing a code completion popup. The code in the background is:
 

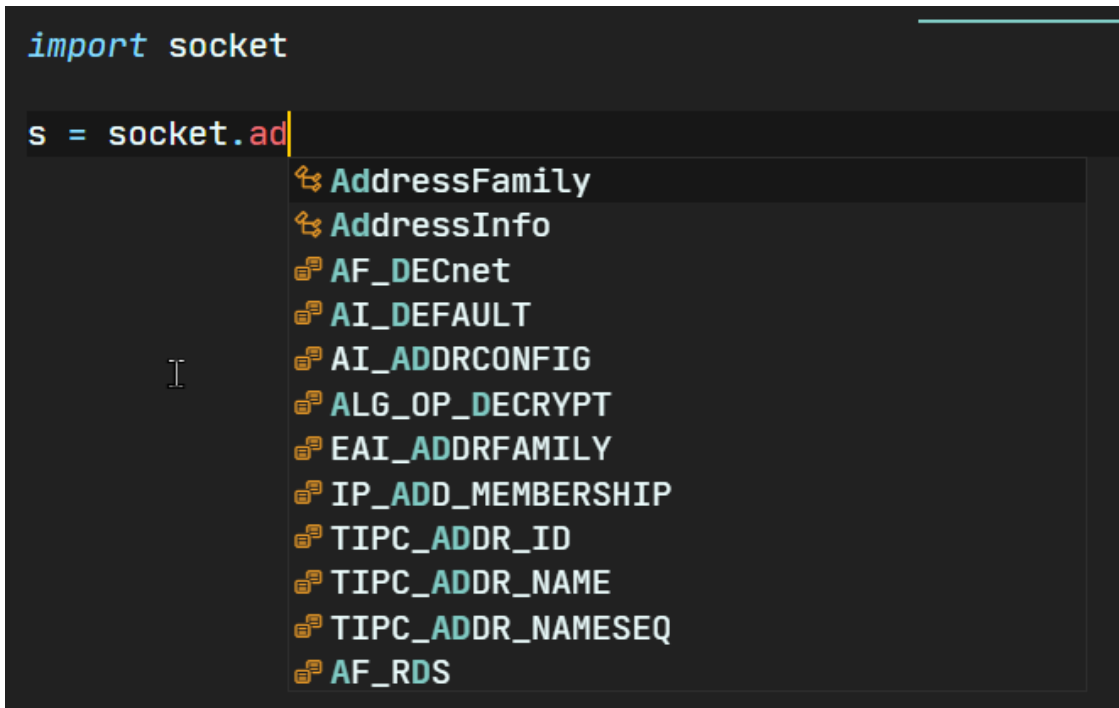
```
String foo;
foo.an|
```

 The popup menu lists the following methods:
 

- `isBlank()` with return type `boolean`
- `resolveConstantDesc(Lookup lookup)` with return type `String`
- `transform(Function<? super String, ? extends ... R`
- `translateEscapes()` with return type `String`

 At the bottom of the popup, there is a tip: "Ctrl+Down and Ctrl+Up will move caret down and up in the editor" and a "Next Tip" link.

Figure 1 – JetBrains IntelliJ Lexical Code Completion


 A screenshot of the Visual Studio Code IDE showing a code completion popup for Python. The code in the background is:
 

```
import socket

s = socket.ad|
```

 The popup menu lists the following attributes:
 

- `AddressFamily`
- `AddressInfo`
- `AF_DECnet`
- `AI_DEFAULT`
- `AI_ADDRCONFIG`
- `ALG_OP_DECRYPT`
- `EAI_ADDRFAMILY`
- `IP_ADD_MEMBERSHIP`
- `TIPC_ADDR_ID`
- `TIPC_ADDR_NAME`
- `TIPC_ADDR_NAMESEQ`
- `AF_RDS`

Figure 2 – Visual Studio Code Python Lexical Code Completion

These efforts require the IDE to know the context of the developer. For both cases, it is the type and an understanding of all the possible options available for that context. This is typically solved with a local index; such is the case of PyCharm 2020.2. The local index is created by the IDE when a new version of the language runtime is detected, as

seen in Figure 3. However, the suggestions by PyCharm 2020.2's code completion is similar to IntelliJ's in only using the exact sequence.

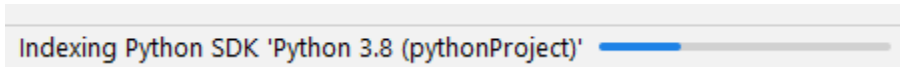


Figure 3 – PyCharm 2020.2 Creating the Python Index

Semantic code completion expands the definition of the context beyond that of just the type. This model attempts to constrain the possible results based on the valid options available for the type at the place in the code where the type is used. Consider that a Python socket should not be bindable before it is created. In Figure 4, we can see that the code completion for a socket has five starred elements. These represent the most likely candidates for the developer to use. Following that is the alphabetical list of options. The most likely option is socket, which is used to create a new socket.

```
1  import socket
2
3  s = socket.|
4  ↩
```

- ★ socket
- ★ create\_connection
- ★ inet\_aton
- ★ fromfd
- ★ gethostname
- AF\_AAL5
- AF\_ALG
- AF\_APPLETALK
- AF\_ASH
- AF\_ATMPVC
- AF\_ATMSVC
- AF\_AX25

Figure 4 – Python Socket Code Completion in Visual Studio Code Part 1



If the code for creating a socket is already written, the context changes but the type remains the same. Figure 5 shows the code completion for the same type but after creating the socket. In this context, we have a created socket type, so the most likely next step is to bind it to an address.

```
1 import socket
2
3 s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4 s.
```

- ★ bind
- ★ setsockopt
- ★ settimeout
- ★ connect
- ★ sendto
- accept
- close
- connect\_ex
- detach
- dup
- family
- fileno

Figure 5 – Python Socket Code Completion in Visual Studio Code Part 2

Figures 4 and 5 show the difference in semantic code completion in Visual Studio Code, while Figure 3 shows lexical code completion in Visual Studio Code. Why does the same editor show both? Because semantic code completion falls back to lexical code completion. These are not mutually exclusive options, but a means of presenting the developers the ability to receive suggestions about what is the most likely option for their context, but still allow them to quickly reach the methods they know they want to use when they want to use it.

In all the above examples, the code completion is demonstrating a few distinct systems working together to provide the results. Ignoring the user interface elements, the core components is a system in the editor to create a best guess about the language type the developer is referring to and other contextual information. This contextual information could include the line of code, and depending on the model, several prior lines of code. The editor then asks the code completion system to provide suggestions based on that contextual information. The code completion then returns the list of suggestions back to the editor. This interaction can happen all within the editor, or the editor can call out to a language service, such as Microsoft's Python Language Server (*Microsoft/Python-Language-Server*, 2018/2020).

The Pythia code completion system implements the semantic model using a novel combination of natural language processing using word2vec and recurrent neural networks to improve the accuracy of suggestions provided (Svyatkovskiy et al., 2019).

## Problem Statement

Python, like most programming languages, has a set of standard libraries considered part of the core software development kit (SDK). In the case of Python version 3.9, the standard library contains the built-in functions, constants, types, and exceptions; and over two hundred modules exposing a wide range of functionality (*The Python Standard Library — Python 3.9.0 Documentation*, n.d.). These libraries will be referred to as the Python Standard Libraries. Building on top of the Python standard libraries is over 267,000 projects, as of late 2020, shared through the Python Package Index (PyPi) for consumption in Python applications (*Search Results · PyPI*, n.d.). A filtered list of those

projects to those that have a Development Status of Production / Stable yielded over 10,000 libraries. Those libraries, and others not tracked on PyPi, but are not part of the Python Standard Libraries are referred to as Third-Party Libraries.

Python modules must meet specific criteria and undergo a strict onboarding process to be included in the standard library set. The result is that the included modules are more highly specialized, targeting specific functionality with close affinity instead of being broad libraries with diverse capabilities (*19. Adding to the Stdlib — Python Developer's Guide*, n.d.).

Third-party libraries lack the same disciplined approach of the standard library. This leads the Third-party libraries to vary greatly in size and complexity. The boto project is the Python SDK for accessing Amazon Web Services (AWS), and has just shy of 50 different functional capabilities in one module (*Boto · PyPI*, n.d.). Another example is NumPy, which is a Python module for scientific computing (*Numpy · PyPI*, n.d.). The NumPy project has over 334,000 lines of code in a single module (*The NumPy Open Source Project on Open Hub*, n.d.), which is not the typical small module included in the Python standard library.

Applications written in Python have utilized libraries beyond just the standard libraries. D'Souza called out this distinction between standard and third-party libraries in his analysis of 20,000 GitHub Python projects which were scanned against 11 standard and 9 third-party libraries (D'Souza et al., 2016). D'Souza's paper introduces a system called PyReco to provide code completion recommendations using a nearest neighbor classified on the usage patterns located in the ASTs of the parsed Python projects. Their analysis distinguished between the Python standard libraries and the third-party libraries,

pre-determining the important libraries to use for benchmarking against prior code completion strategies. Pythia does not make this distinction, and instead just uses the ten most frequent libraries used in their smaller dataset.

In Pythia's results, only four were third-party libraries. The **NumPy** library was second only to the **os** standard library in frequency. Despite its frequent occurrences in the Pythia dataset and having significantly more methods than the eight following libraries, it still had a significantly lower accuracy than any of the standard libraries. Of the ten libraries measured by Pythia, the 6 standard libraries were significantly more accurate than the 4 third-party libraries. This lack of improvement to the accuracy of third-party libraries created an imbalance that favors the accuracy of the Python standard libraries and created an inefficiency for the developer.

Pythia's accuracy for the Third-Party Libraries ranged from 16% to 38% worse than the Python Standard Libraries (Svyatkovskiy et al., 2019). The data set used was the top 2,700 non-forked Python projects on GitHub, with the ranking of top based on the number stars a project had at the time of the search. From this dataset, Svyatkovskiy, Zhao, Fu, & Sundaresan (2019) identified the 10 most frequent Python modules used in the data set, and accuracy changes for each module.

The standard Python libraries, as part of the standard distribution, are more often used across the widest variety of applications. Developers are more familiar with these libraries, relying on code completion to present options that they are most likely to already know. The goal of the developer is to get to the method they know they want to use, and lexical code completion will be the most efficient in these use cases. However, the third-party libraries, with their significantly larger surface areas, and less frequent

usage, benefit significantly more from the accurate contextual predications provided by semantic code completion. Pythia presented suggestions that are less accurate, while the developers expected a higher level of accuracy. Since the developers expected accuracy, the result of inaccurate code completion suggestions is a mistaken belief that the developer code is wrong (Mărășoiu et al., 2015). This has led to wasted time and overall developer inefficiency.

### Dissertation Goal

The goal was to increase the accuracy of suggestions for third-party Python libraries in the Pythia recommender by dynamically modifying the input dataset during the preprocessing process.

The original Pythia preprocessing phase passed all the Python files in its dataset to an AST parser, which extracted contextual information, and built the matrices passed into word2vec as seen in figure 6. However, there is a step prior to preprocessing in which Pythia located the 2,700 target repositories in GitHub, cloned the repositories, and located all the Python source files in the cloned repositories. This is the data collection phase.

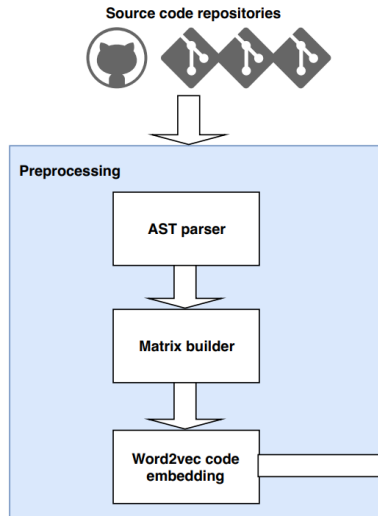


Figure 6 – Pythia Preprocessing Phase

The data collection phase needed to be enhanced to collect a larger universe of data than the original 2,700 repositories. All Python projects were scanned to inventory the modules included in every source file. This information was used during the preprocessing phase to include more repositories until the ratio of a specific third-party library was within a variably defined tolerance of a specified standard library.

The intended result of increasing the training dataset to include more third-party libraries is to allow the model training process to evaluate more occurrences of third-party module usage in a wider range of use cases. This will enable Pythia’s model to raise the accuracy level of code completion suggestions for third-party libraries to be more in line with the accuracy of the Python standard libraries. Closing the prediction accuracy gap between Python standard and third-party libraries resulted in more consistent efficiency for the developer using the code completion prediction system.

## Research Questions

The Pythia model leveraged 2,700 non-forked Python open-source projects from GitHub ranked by number of stars as the training dataset (Svyatkovskiy et al., 2019). There was no attempt to adjust the quantity of third-party libraries included in that dataset.

The primary hypothesis addressed by this research was that if the number of non-forked Python open-source projects from GitHub used as input data increased to include more projects that reference third-party libraries, the resulting accuracy for those third-party libraries will also increase.

A key question resolved is the appropriate strategy to use to select new projects to increase the projects in the input data set. It was not sufficient to just add more projects by moving from 2,700 projects to a higher number. Those new projects may not use the same third-party libraries and may introduce new third-party libraries. A blind increase in the quantity of projects resulted in widening the number of the third-party libraries at a lower accuracy. The goal was to identify projects that will increase the accuracy of the third-party libraries in the original data set to show a correlation between the selection of input data and the quality of the accuracy output by the model represented by Pythia.

## Relevance and Significance

The modern software developer depends on code completion to increase their efficiency and effectiveness with programming languages. Research by Mărășoiu, Church, and Blackwell (2015); Stylos and Clarke (2007); Murphy, Kersten, and Findlater

(2006) have demonstrated the integral nature of code completion in the developer experience. The research differed on why code completion is important. Stylos and Clarke (2007) state that API exploration and discovery is the significant usage model. Mărășoiu, Church, and Blackwell (2015) suggests that the speed and accuracy of development was the most significant use case. Murphy, Kersten, and Findlater (2006) survey found that code completion is as popular as Cut and Paste. Whichever use case is considered, they all agree on the broad demand and importance of code completion as a necessary capability for developers.

Code completion suggests a set of possible methods relevant to a specific type at a contextual point in the source code. Some types have hundreds of methods, or enough that scrolling through an alphabetical list becomes more time consuming than alternative approaches to discovering the correct method. Mărășoiu, Church, and Blackwell (2015) identified two common strategies for reducing the set of suggestions presented to the developer: lexical and semantic. The lexical strategy uses pattern matching to reduce the suggestions provided based on filtering the possible methods using a partial match of the characters typed by the developer. Common approaches for the partial match are matching the consecutive characters either anchored at the start or anywhere in the method; or matching all the characters regardless of position in the method name. As the developer types more characters, the list of suggestions is reduced through continued matching. The semantic strategy uses the grammar of the programming language to reduce the suggestions provided to the developers. Leveraging the grammar allows the system to restrict the list of candidate methods to those allowed at the contextual point in the code that triggered the search. Filtering private methods from a candidate result set is



a simple example, but this can be extended to include more contextual attributes. For example, an uninstantiated object could filter all but static or class methods.

Both the lexical and semantic models address filtering the set of possible suggestions provided to the developer, but neither addresses the ordering of the methods suggested. Further, they each have some limitations. The lexical strategy requires the developer to already know something about the method they want to call since the pattern match is based on the characters in the method name (Proksch et al., 2015). The semantic strategy requires the developer to understand the correctness of the suggested methods. Consider the example of a developer using a socket type, and the code completion suggests both open and close methods in an alphabetical list. The developer is forced to choose without any effective guidance.

The combination of filtering and ranking of the suggestion list is called Intelligent Code Completion (ICC) (Proksch et al., 2015). ICC leverages more contextual information about the type at the point of the suggestion request, and in some cases, historical usage from other similar scenarios, to provide a filtered list of methods with all or a subset of the list ranked in a suggested order of relevance to the specific context.

Python's lack of strong typing prevents the use of a typed variable to derive the context. D'Souza, Yang, and Lopes (2016) identified this challenge in their paper introducing their Python code completion system, PyReco. Their approach is to parse a large source dataset of Python projects, convert to ASTs, and analyze assignments to determine the most likely type.

## Barriers and Issues

GitHub is not a static representation of projects, or their attributes. The source code in the projects in GitHub have likely changed, as has the star ranking used to select the top entries. Using the exact same 2,700 projects used by Pythia is not possible without the specific list of projects they used, and the Git SHA commit code for each of projects they used. The model selected by Pythia can be reproduced, but it will not result in the same dataset. This may result in a skewing of the initial baseline results.

Adding new projects to the input dataset will increase the number of sources using the Python Standard Libraries. This may result in a change in accuracy for the Python Standard Libraries and make probable that the increase in third-party library accuracy will result in a gap between the standard library and third-party library remaining.

It is possible that third-party libraries that are significant in the 2,700 projects in the initial dataset may not be significant in other projects. This would impact the ability to identify other candidate projects to include in the input dataset to raise the accuracy of the third-party libraries.

The training of the models used by Pythia is compute intensive and time consuming (Svyatkovskiy et al., 2019). Svyatkovskiy, Zhao, Fu, & Sundaresan (2019) describes several strategies employed to reduce the computation time, including reducing the number of AST nodes used, pruning lowest used word2vec vectors, and parallel batch processing using GPUs. The increase over 2,700 projects will have a direct impact on the required computation time of the softmax. This may result in a potential upper limit on the number of projects that can be included in the input dataset.

### Assumptions, Limitations, and Delimitations

A key assumption is that using 2,700 top-starred non-forked open-source projects on GitHub queried at any time will result in comparable results generated by the Pythia research at the time of their query. This means that having the exact same 2,700 projects with exact same source code will not be required, only having a comparable dataset generated in the same manner.

Another assumption is that the model used in Pythia can function using more projects in the input dataset. While the time to generate the results is expected to increase, the overall model and selection of hyperparameters is expected to function with the increased dataset.

There are three delimitations between this research as the Pythia project. First, this research will not replicate the integration of the resulting data into the text editor as done in Pythia research. This integration is not related to the accuracy of the data and is only a means of demonstrating its utilization. The results can be demonstrated through the test cases. Second, this project will not implement model quantization implemented by Pythia. This was done to reduce the size of the dataset prior to being sent to the client systems. This quantization reduced the size of the dataset and reduced the predictive accuracy of the model. However, the results shared by Pythia are all pre-quantization, and post-quantization results were not shared, just summarized as reducing the top-5 accuracy by 3% (Svyatkovskiy et al., 2019). Third, the research will focus on the accuracy results, and not define specific performance characteristics. While some systems report

suggestion list performance, not all do, and there is not enough consistency between the models to try and replicate specific parameters.

### Summary

This research seeks to improve the accuracy of Python code completion by building on the prior research of Svyatkovskiy, Zhao, Fu, and Sundaresan (2019) with Pythia. The Pythia model showed using natural language processing combined with recurrent neural networks using long short-term memory can improve accuracy over current statistical models. Pythia's accuracy is tied to the frequency of programming patterns it learns during the training cycle (He et al., 2021). This research will increase the exposure of third-party libraries to the Pythia training model to result in higher accuracy for those third-party libraries.

## Chapter 2

### Review of the Literature

#### Introduction

Gail C. Murphy, Mik Kersten, and Leah Findlater (2006) monitored a number of developers to determine how they used Eclipse. Their results showed that editing commands are used the most, but it also showed that Eclipse's Content Assist (code completion) is used as much as common editing commands, which they specifically called. The tables showing the top 10 commands executed by the most developers and the top 10 commands as percentage of use are reproduced below as Table 1 and Table 2 respectively.

<b>Command</b>	<b>No. of Users</b>
Delete	41
Save	41
Paste	41
Content Assist	41
Copy	41
Undo	41
Cut	40
Refresh	40
Show View	40

Table 1 – Top 10 Commands Executed by the Most Developers

Command	Use (%)
Delete	14.3
Save	11.3
Next Word	7.3
Paste	6.8
Content Assist	6.7
Previous Word	5.9
Copy	4.6
Select Previous Word	3.4
Step (debug)	3.2

Table 2 – Top 10 Commands Executed Across all 41 Developers

Stylos and Clarke’s (2007) research looked at the usability of two different object construction models: the default constructor (“create-set-call”) and required constructor.

Their research worked with three groups of developers:

- Systemic Developers – Professional C or C++ developers with 5 or more years of experience
- Pragmatic Developers – Professional C# developers with 2 or more years of experience
- Opportunistic Developers – Professional Visual Basic developers with 2 or more years of experience

With all three groups of developers, they were provided a series of tasks involving the design, implementation, and reading of code without the support of code support tooling, such as code completion. One of the observations in the study was that all participants used code-completion as a “primary means of exploration,” even while debugging.

Mărășoiu, Church, and Blackwell (2015) analyzed the usage of code completion amongst six software developers with diverse levels of experience in the Dart language and most having no exposure to the libraries used in the experiment. The results confirmed the findings of Murphy, Kersten, and Findlater (2006) regarding the extensive

use of code completion amongst the developers. However, the findings challenged Stylos and Clarke's (2007) conclusion that code completion was used primarily for API exploration. The results suggested that the code completion was to increase the efficiency of writing code by filtering the suggestion list to find the appropriate option to accept. Further, the results showed that a significant number of code completion suggestions were not accepted by the developers.

In Mărășoiu, Church, and Blackwell's (2015) research, they observed that only 40.1% of code completions results in an accepted suggestion. When they analyzed 10,000 code completion suggestions from developers experienced with Dart and the used APIs inside Google, that result only rose to 44.2%. They concluded that the quality of code completion suggestions is still a significant challenge to be addressed.

Mărășoiu, Church, and Blackwell's (2015) also observed that when code completion did not offer useful suggestions, the developers considered that as indicative of errors elsewhere in their code. The result was a correlation between ineffective code completion suggestions and developer inefficiencies.

### Intelligent Assistants

The complexity and size of computer source code is an ever-increasing problem. The first version of Unix was 4,768 lines of assembly code in 13 files totaling 146.41kb (*GitHub - Dspinellis/Unix-History-Repo at Research-VI-Snapshot-Development*, n.d.). Linux in early 2020 was 27.8 million lines of code (*Linux in 2020: 27.8 Million Lines of Code in the Kernel, 1.3 Million in Systemd - Linux.Com*, n.d.). Despite that growth in size

and complexity, some challenges have remained the same. Terry Winograd (1973) postulated that software could be developed to assist the developer in tackling this complexity. He identified four key areas of helping: Error Checking, Question Answering, Trivia, and Debugging. His definition of Trivia outlines the concept of code completion. He states “Often, a programmer really doesn’t want to bother knowing the answer to a question. If he has a variable named ITEM which is to be added to LIST, he must worry about whether ITEM is the item itself, or a singleton list containing the item, and whether the list is ordered, or does not contain duplicate items, etc.” What he is proposing is that some developer assistant could be aware of the context of the source code that the developer is working on. He continues “Rather than asking for all this information, he [the developer] would rather say to his moderately stupid assistant, ‘Write the appropriate call to add ITEM to LIST.’” This defines the concept behind modern code generation tooling, but it also covers the basis of code completion. In short, the developer’s assistant should know what the thing is the developer is working with, the context or state it is in, and what the developers wants to or should do next with it. Winograd’s work was a thought experiment proposing a future state for developers that might be possible.

Fifteen years later, Gail E. Kaiser and Peter H. Feiler (1988) developed Project Marvel based on Winograd’s model of an Intelligent Assistant for developers. Kaiser and Feiler’s (1988) approach to implementing Winograd’s vision was to develop a system based on the concepts of insight and opportunistic processing. Insight is defined as being aware of the developer’s environment, or context, and being able to leverage that knowledge to share information or proposed actions the development should know about. Opportunistic



processing is the developer's environment using insight to take certain actions automatically, thus freeing the developer to work on their source code more efficiently. Common examples of this behavior are the background compilation, error checking, and automatic linting performed in modern IDEs.

Kaiser and Feiler's (1988) insight concept directly address the concept of code completion. Their approach was to consider the developer's environment as objects, in the object-oriented context. If everything is an object, and all possible objects can be identified, then a database of all possible information, could be constructed for use during the development cycle. This included relationships between objects, which allowed for identification of context.

This model of static awareness predicts code using a combination of identifying what the developer's context is and doing a lookup against the object database to determine the results of a prediction. As new types are added to the program, their information, including interaction relationships, would be added to the database.

Project Marvel was significantly more than just code completion. It was an entire environment that had a defined model for the process the programmer must follow. The database of objects and their relationships is like the run-time type identification available in many modern object-oriented languages. The implementation approach of Project Marvel is no longer appropriate, but the conceptual approach of using rich type information to make the developers environment more intelligent is the foundation of modern IDEs.

## Code Completion

### Optimistic Code Completion

Robbes and Lanza (2008) define code completion as taking an “input token to be completed and a context used to access all necessary information in the system, and outputs an ordered sequence of possible completions”. In their paper, they propose an improvement over the code completion systems provided in Eclipse, Visualworks, and Squeak. Those three systems provide a filtered code completion system based on a specific data type and optional input from the user. The filtered model starts with the universe of all possible results based on a provided datatype and filters the suggestion list based on the characters typed by the user and presents the suggestions alphabetically. Figure 7 shows Eclipse providing filtered code completion of 11 results out of 22 sorted alphabetically, with shorter parameter lists given priority.

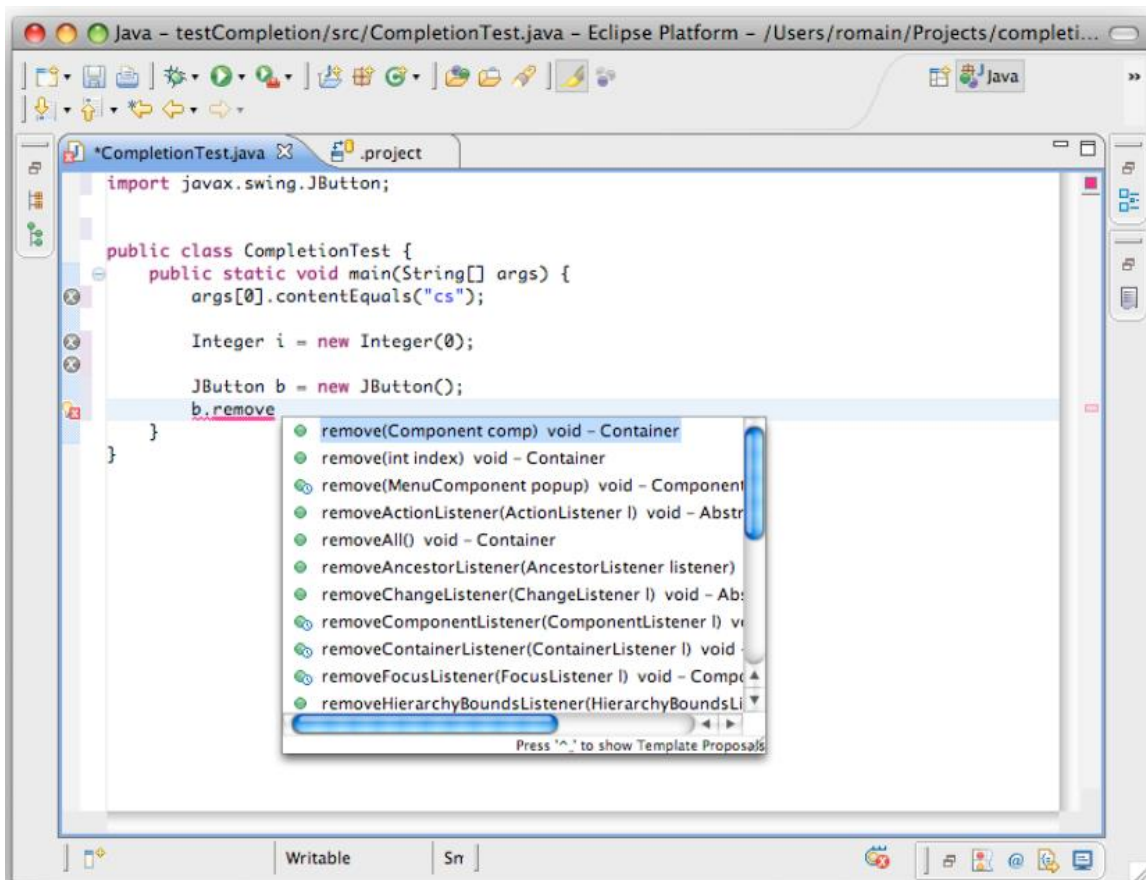


Figure 7 – Filtered Code Completion in Eclipse (Robbes & Lanza, 2008)

Robbes and Lanza (2008) propose that the three code completion systems all suffer from the same overall problem of making it difficult to find the single right suggestion because it is hidden amongst many incorrect suggestions due to the alphabetical sort. They classify this model of code completion as “pessimistic” due to its assumption of returning a large data set. They propose an “optimistic” model that would expect a shorter list of suggestions. The goal of an optimistic model would result in a smaller suggestion list that would not need to be alphabetized. They defined the following three assumptions to qualify a result set as optimistic:

- The result set is small. They used 3 as the limit in their paper.

- The match being sought must have a high probability of being in the smaller result set.
- The programmer typed prefix used for filtering must be short.

Robbes and Lanza (2008) only compare the accuracy score of method calls predictions, and no other constructs such as types, keywords, and variables. They take this approach because method calls are a significantly larger percentage of the predicted elements within a program (Robbes & Lanza, 2008). Their paper divides code completion strategies into two broad categories of typed and untyped. Robbes and Lanza (2008) use the Squeak IDE with Smalltalk as the language evaluated because it is untyped, and they reason that untyped languages require more improvement in code completion strategies over typed languages. However, they do include some typed code completion algorithms in the evaluation using a type inference engine built into Squeak.

Robbes and Lanza (2008) evaluated a total of 8 different strategies. The two pessimistic models served as a typed and untyped baseline. The six optimistic strategies were based on information gathered from the code changes captured in the AST, and were based around the following code history hypotheses:

- Structure – Local methods are called more often than distant methods.
- Names – Recently changed method names.
- Bodies – Recently changed method bodies.
- Inserted – Recently inserted code.
- Sessions – Per-Session vocabulary defined as terms, such as class names, methods, and variables, introduced in the past hour.

- Typed Per-Session vocabulary.

Robbes and Lanza (2008) show the results of untyped code completion algorithms compared against each other for 7 projects; their SpyWare monitoring project and 6 student projects. Table 3 below shows the results scored from least to most accurate using a 0 to 100 scale.

Project	SpyWare	Student1	Student2	Student3	Student4	Student5	Student6
Baseline	12.15	11.17	10.72	15.26	14.35	14.69	14.86
Structure	34.15	23.31	26.92	37.37	31.79	36.46	37.72
Names	36.57	30.11	34.69	41.32	29.84	39.80	39.68
Inserted	62.66	75.46	75.87	71.25	69.03	68.79	59.95
Bodies	70.14	82.37	80.94	77.93	79.03	77.76	67.46
Sessions	71.67	79.23	78.95	70.92	77.19	79.56	66.79

Table 3 – Scores for the untyped algorithms of all projects (Robbes & Lanza, 2008)

The results from Robbes and Lanza (2008) show a significant improvement above the baseline, however, it is based on information within an application. They do not identify the difference in accuracy between developer created methods, third-party library methods, or standard languages methods. Their code completion algorithms are all based on the actions of the developer, which implies that accuracy of a type imported into a project and not used before, cannot have the same accuracy results. Further, in their models, only the sessions model is not overtly biased towards developer created code. The sessions model is biased towards the methods used within the last hour, and that model should favor developers that remain in the same context. However, the selection of

one hour is not explained, and there is no data concerning the number of times a developer switches contexts within a session to help qualify the results.

### Intelligent Code Completion

Bruch, Monperrus, and Mezini (2009) defined the term Intelligent Code Completion as a system that learns by analyzing prior source code. They stated that the problem of too many incorrect suggestions being returned during code completion is still an unsolved problem and impairs developer productivity. They provide the example of the Java Standard Widget Toolkit (SWT) Text class having more than 160 callable methods, including all the methods in Java's Object type, with some of those methods never being called on the Text object. For example, the wait method on Java's Object is a method that was never called in their scan of the Eclipse codebase. From that same scan, they identified that only 5 methods out of over 160 are ever called on Text in the Eclipse code base. That leaves the developer with over 155 incorrect choices to filter out of the proposed suggestions during code completion.

Bruch, Monperrus, and Mezini (2009) identify the key criterion about methods, beyond frequency of use, is the context that it is called within. They provide the example of the configuration code that only happens in a Dialog.create() and the code for reading the input data in the Dialog.close(). This is information that can be used to filter the proposed suggestions based on the context of the code completion suggestion request.

Their paper proposes that intelligent code completion system must be capable of the following two behaviors:

1. Filter code completion suggestions from the list that are not relevant to the current context.
2. Rank the relevance of every proposed code completion suggestion.

The combination of these two behaviors will reduce the number of code completion suggestions that the user will receive and order those suggestions so that more relevant suggestions are earlier in the list.

Bruch, Monperrus, and Mezini (2009) developed and measured the accuracy of three strategies to analyze existing source code. Their test was limited to the accuracy of calling the Java SWT library in over 27,000 test cases. Similar to Robbes and Lanza (2008), they limited the resulting options presented to the developer. However, instead of the fixed number of entries used by Robbes and Lanza (2008), they chose a 30% confidence level as a threshold for filtering the results.

#### Frequency Based Code Completion

Bruch, Monperrus, and Mezini's (2009) first strategy is a frequency-based solution. This model assumes that the more frequently a developer calls a method for a specific type, the more likely it is going to be called again for the same type. The relevance rank is determined by ordering the methods called for a type by the absolute number of times that method is called.

#### Association Rule Based Code Completion

Bruch, Monperrus, and Mezini's (2009) second strategy is to look for association rules based on patterns in the code. They provide an example of mapping the typical behavior of calling setter methods inside a constructor, which would lead to a rule of

Object creation implies setter methods (Bruch et al., 2009). Similarly, a second rule could be when `Dialog.close()` is called, it will be followed by `getText()` method calls.

### Best Matching Neighbors Code Completion

Bruch, Monperrus, and Mezini's (2009) third strategy is a novel application of the k-nearest-neighbor (kNN) algorithm, and this is the primary contribution of their paper. They define the Best Matching Neighbors (BMN) algorithm as a kNN algorithm modified to the context of code completion. Their modifications are mapping the context of the variable to a vector; design of a novel distance measure; selection mechanism of nearest neighbors; and mapping nearest snippets to method recommendations (Bruch et al., 2009).

### Results

Bruch, Monperrus, and Mezini (2009) use precision, recall, and the F1-measure to evaluate their algorithms. Recall is the percentage of relevant methods returned, with 100% meaning all methods for a type were in the suggestion set. Precision is the percentage of methods actually needed by the developer. The F1-measure used, equally weights recall and precision. The F1-measure was used as the primary metric for tuning the experiments, with the goal of maximizing the F1-measure. Figure 8 below shows the results of comparing the Eclipse Code Completion System (EcCCS), Frequency Code Completion System (FreqCCS), Association Rule Code Completion System (ArCCS), and Best Matching Neighbors Code Completion System (BMNCCS) strategies.



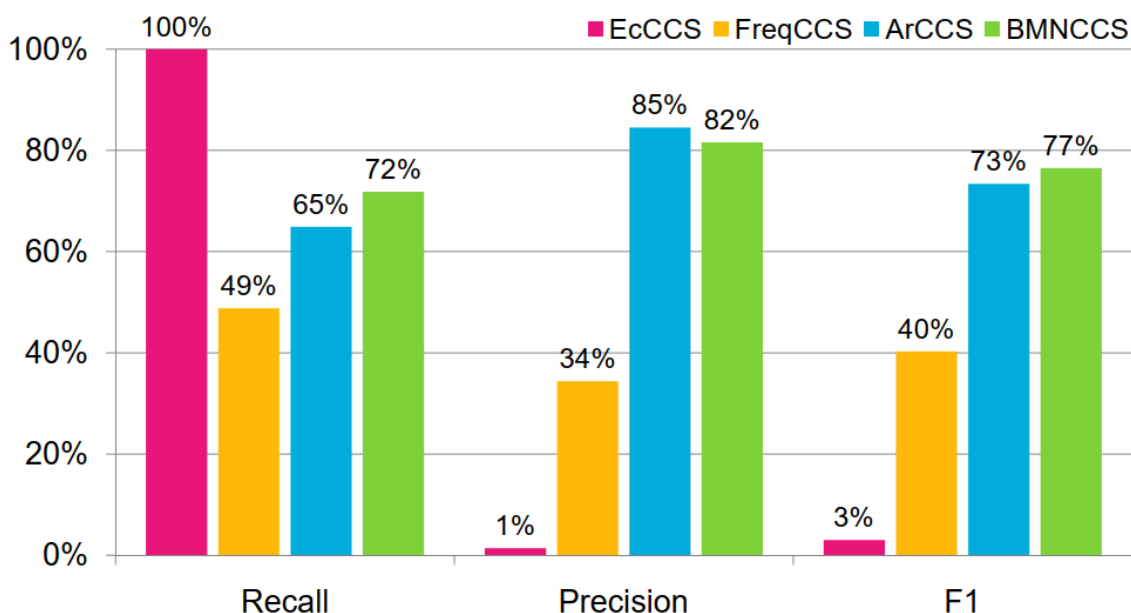


Figure 8 – Performance of EcCCS, FreqCCS, ArCCS, and BMNCCS (Bruch et al., 2009)

Bruch, Monperrus, and Mezini (2009) conclude that they were able to demonstrate a significant improvement using their BMN strategy, but that there is room for improvement in generalizing the solution for standard libraries. This highlights one of the challenges Bruch, Monperrus, and Mezini's (2009) work, in that it was evaluated within a narrow scope for a single subset of a larger framework. Their decision to tune their strategies based on the F1-measure for just SWT, does not indicate that these results will apply when multiple frameworks and libraries are included in the dataset. Despite that, their results do show that analysis of existing source code can improve the prediction results of code completion. Further, it is evidence that mapping source code to vectors for leveraging vector-based algorithms for prediction is a viable option. However, their paper does not provide significant details, or show metrics related to the issue they identified about the context of where code completion happens.

## Intelligent Code Completion with Bayesian Networks

Proksch, Lerch, and Mezini (2015) developed an approach using Bayesian networks that directly builds upon the Best Matching Neighbors (BMN) strategy of Bruch, Monperrus, and Mezini (2009). Proksch, Lerch, and Mezini (2015) acknowledge the advancement of the BMN strategy but identify a couple of specific areas for improvement. First, context is not sufficiently addressed in the results generation and in the prediction accuracy analysis. Second, runtime impacts of the model were not considered, specifically the speed of the results and the size of the required dataset.

Proksch, Lerch, and Mezini (2015) define three aspects used in comparing their approach with the BMN strategy: prediction quality, prediction speed, and model sizes. Their approach replaces the use of BMN with a Pattern-based Bayesian Network (PBN). This key difference allows the merging of different patterns, and to map the patterns to probabilities instead of just Boolean values. Further, the PBN is clustered, allowing for a model that can be tuned to balance prediction quality and size of the resulting model.

Proksch, Lerch, and Mezini (2015) provide details on how they analyze existing source code. Their approach is to capture *object usages*. They define object usages as a representation of any method that is called, and the context in which the object usage was observed (Proksch et al., 2015). This context information was implied by Bruch, Monperrus, and Mezini (2009), but its collection and usage were not explained, and is one of the key elements that Proksch, Lerch, and Mezini (2015) use to improve upon the BMN strategy. However, while Proksch, Lerch, and Mezini (2015) claim to extract as much reusable context information as possible, they do limit its usefulness by focusing on frameworks over libraries. Their rationale is that frameworks provide more structural

information with well-defined extension points, and that the typical usage model of frameworks is through extension with classes and interfaces defined in the framework. Their paper states that they expect the result of using frameworks to yield more contextual information that will provide more specific proposals.

The PBN approach seeks to provide a probabilistic result to the question of how likely a specific method call will happen within a given context. Figure 9 shows the conditional probabilities of PBN object usage, and figure 10 shows the structure model of the Pattern Based Bayesian Network.

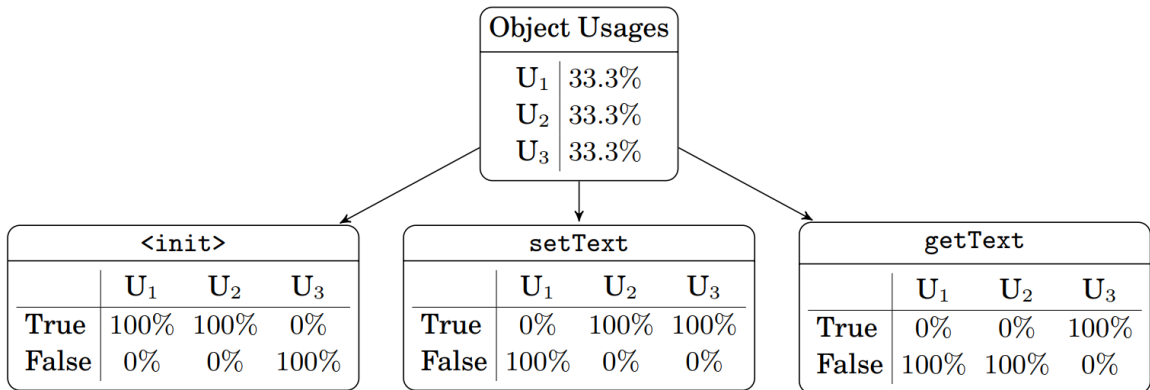


Figure 9 – Conditional Probabilities in a Based Bayesian Network (Proksch et al., 2015)

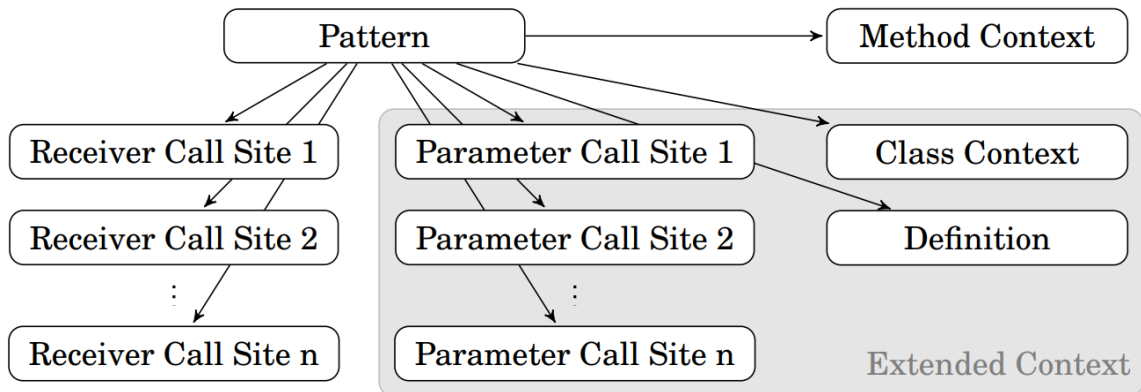


Figure 10 – Structural Representation of the Bayesian Network used in PBN (Proksch et al., 2015)

The PBN model clusters by selecting a random object usage to be the center of the cluster from the set of all usages for a given type. All other object usages below a distance threshold are placed into the same cluster and removed from the available set of object usages. This process repeats until all object usages are assigned to a cluster. Each cluster becomes the pattern node state in the Bayesian network. The probability is calculated as the number of object usages in the cluster divided by the total number of object usages (Proksch et al., 2015). Distance is calculated using cosine similarity, and the distance threshold is the parameter used to tune the trade-off of quality and model size. The lower the distance threshold, the higher quality, but larger the model size. Thresholds range from 0.0 to 1.0 and indicated in the result data as  $PBN_d$ , thus a distance of 0.10 is denoted as  $PBN_{10}$ , and the distance of 1.00, is denoted as  $PBN_{100}$ .

The results for PBN show that it can significantly reduce the model size while retaining comparable results to BMN, as seen in figure 11. This graph presents PBN and BMN both using definition contextual information (the  $+D$  marker). Other contextual information and combinations were tried but were all lower quality than just definition information. Figure 12 shows the effect of increasing the number of object usages for one type (SWT Button) on model size and inference speed over different distances.

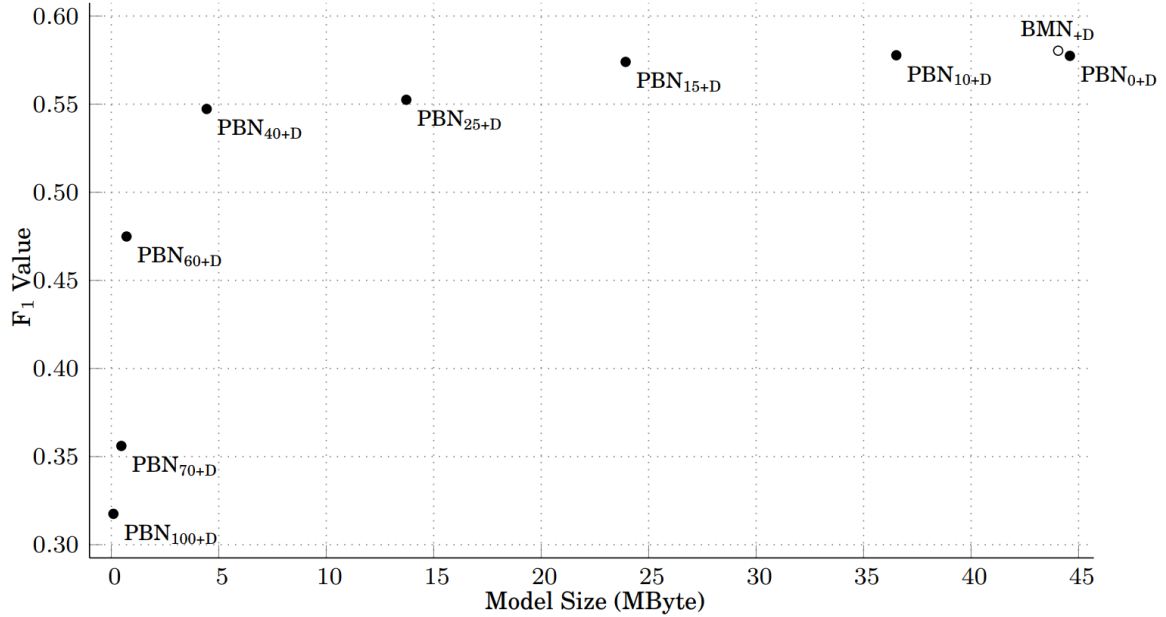


Figure 11 – Quality and Size for Different Distances Compared to BMN (Proksch et al., 2015)

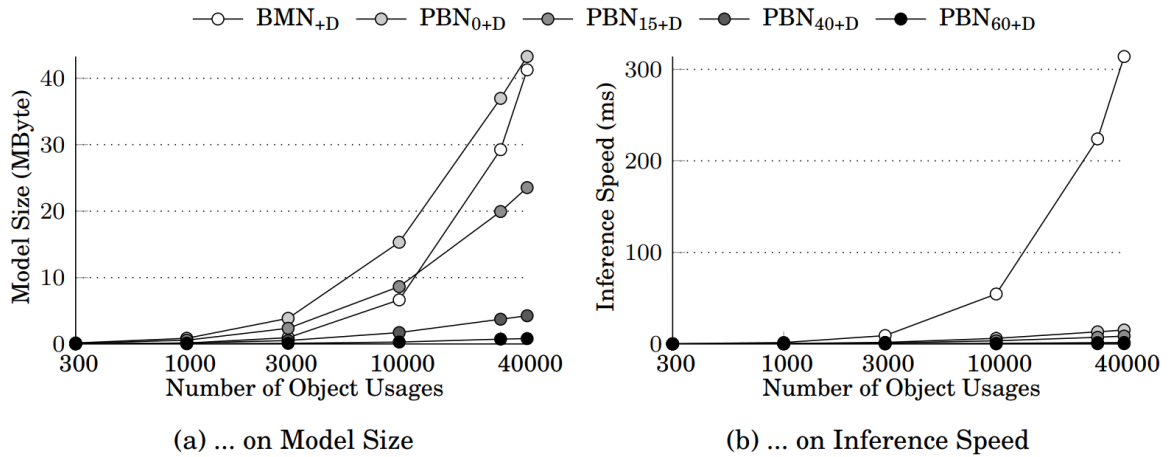


Figure 12 – Effect of Increasing Number of Object Usages for SWT Button (Proksch et al., 2015)

PBN can achieve similar quality as BMN with better performance and smaller data sizes. However, the authors state the same concerns with their results identified earlier in their paper with the BMN strategy. Specifically, focusing on just the SWT framework may mean this model will not be generally applicable to other frameworks. This should be considered an applicable limitation when considering libraries, standard or third-party, since they were deliberately out of scope for this analysis.

## Improving Best Matching Neighbor for Dynamically Typed Languages

D'Souza, Yang, and Lopes (2016) also build upon the Best Matching Neighbors (BMN) strategy of Bruch, Monperrus, and Mezini (2009) to demonstrate the applicability to dynamically typed languages, specifically testing with Python. D'Souza, Yang, and Lopes (2016) chose to use the BMN model over the Pattern Based Bayesian Networks (PBN) presented by Proksch, Lerch, and Mezini (2015) because their experiments demonstrated that a Vector Space Model outperformed the Naïve Bayes, Bayesian Network, and Tree Augment Naïve Bayes classifiers.

D'Souza, Yang, and Lopes (2016) identified that the prior research for code completion focused primarily on statically typed languages, usually Java. They developed a code completion model named PyReco to demonstrate how to apply prior research to the dynamically typed language Python. Their approach uses a significantly larger set of input source repositories for model training. They chose approximately 20,000 Python GitHub projects with the most stars. From these source files, they extracted the Python library and module information, object assignments, method calls, attributes, and object termination through forward parsing of the AST generated from the Python Standard Libraries AST parser. Further, contextual information, such as conditionals and loops, are captured in the resulting graph structure that they use to model the analyzed source code.

The output of the static code analysis is then transformed into vectors used for the training objects, and queried based on the method-call frequency (D'Souza et al., 2016). Manhattan distance is used in calculating the nearest neighbors based on the method-call frequency. Recommendations are then created by traversing the methods invoked in decreasing order of frequency. A notable change from the approach of Bruch, Monperrus,

and Mezini's (2009) BMN model is that D'Souza, Yang, and Lopes (2016) retain the original frequency values instead of the Boolean values distilled from the frequency values. This retains the knowledge of which methods are called more frequently than others for use in the recommendations.

D'Souza, Yang, and Lopes (2016) are not able to compare their approach to the approach of Bruch, Monperrus, and Mezini (2009) or Proksch, Lerch, and Mezini (2015) because they are evaluating the code completion accuracy of different programming languages. To demonstrate improvement, the paper compares its results against JEDI (Halter, 2012/2020), a popular open source Python code completion engine. The paper asserts that JEDI can be used in automated code completion tests that provide the ability to gather quantitative results at scale. However, they also performed manual comparisons against the JetBrains PyCharm IDE (*PyCharm: The Python IDE for Professional Developers by JetBrains*, n.d.), which they assert has a more powerful code completion engine than JEDI (D'Souza et al., 2016).

The experiments were evaluated for a total of 20 Python libraries, 11 standard libraries and 9 third-party libraries. The selection of those 20 libraries were based on the D'Souza, Yang, and Lopes's (2016) determination of popularity within the Python community and frequency of library occurrence within the dataset of source repositories.

D'Souza, Yang, and Lopes (2016) evaluated their results using Mean Reciprocal Rank (MRR) and Recall. They chose MRR because the goal was to identify how high the result was in the resulting suggestion list, and not how long the resulting suggestion list is. The Precision measurement would penalize longer lists, even if the result was the first choice. Further, since the developer will only choose one item from the list, the MRR is

equivalent to the Mean Average Precision (MAP) measurement. Table 4 below shows the results of PyReco and JEDI for the 10 Python standard libraries. Table 5 below shows the result of PyReco and JEDI for the 9 Python third-party libraries. In all but one case, the PyReco system outperforms JEDI in both MRR and Recall. The one exception is argparse. The authors propose that with argparse, and with the low values for mock, it is due to unique nature of how these libraries work, and the lack of training data based on normal method call stacks and object assignments. The results of the manual comparison to PyCharm just capture the rank of relevant result, and PyReco improved over PyCharm for 16 out of the 20 libraries tested. Further, for the mock, ctypes, and google libraries, PyCharm failed to provide any recommendations.

<b>Library</b>	<b>PyReco-MRR</b>	<b>JEDI-MRR</b>	<b>PyReco-Recall</b>	<b>JEDI-Recall</b>
os	0.592	0.037	0.943	0.356
re	0.727	0.196	0.967	0.853
ctypes	0.369	0.146	0.565	0.161
logging	0.425	0.080	0.730	0.615
datetime	0.485	0.040	0.845	0.429
time	0.516	0.0068	0.951	0.068
json	0.632	0.0137	0.950	0.068
collections	0.418	0.161	0.776	0.665
struct	0.646	0.237	0.927	0.843
subprocess	0.560	0.260	0.925	0.741
argparse	0.306	0.424	0.422	0.518

Table 4 – PyReco Results for Python Standard Libraries (D’Souza et al., 2016)



Library	PyReco-MRR	JEDI-MRR	PyReco-Recall	JEDI-Recall
Django	0.467	0.001	0.687	0.003
numpy	0.424	0.009	0.783	0.006
mock	0.252	0.000	0.472	0.000
sqlalchemy	0.551	0.092	0.871	0.419
PyQt4	0.559	0.000	0.896	0.000
theano	0.674	0.000	0.930	0.000
wx	0.568	0.000	0.842	0.000
google	0.638	0.001	0.910	0.002
flask	0.481	0.000	0.819	0.000

Table 5 – PyReco Results for Python Third-Party Libraries (D’Souza et al., 2016)

D’Souza, Yang, and Lopes (2016) have demonstrated that the use of a nearest neighbor classification algorithm, combined with a large corpus of training data can provide improved results for Python code completion. They do identify some potential issues with their results. First, they only used 20 libraries, and while broader than single library tests, it may not expand well to a more generalized use case. Second, the manual evaluation was performed only once, and the results may change over time. Third, bugs in the scanned source code in the repositories used as the input dataset could skew the training model, and thus the results may contain some false positives. Despite these issues, D’Souza, Yang, and Lopes (2016) have shown the application of typed code completion approaches to untyped languages is possible.

## Long Short-Term Memory Neural Networks Pythia

Svyatkovskiy, Zhao, Fu, and Sundarsan (2019) state that existing models do not fully leverage the capabilities of Natural Language Processing (NLP) using the long-range sequential characteristics of source code represented in ASTs. Prior research has focused on the use of vectorized source code fragments, rather than introducing techniques from NLP to vectorize the AST. Svyatkovskiy, Zhao, Fu, and Sundarsan (2019) developed the Pythia model to leverage the word2vec algorithm developed by Mikolov, Corrado, Chen, and Dean (2013) at Google to vectorize flattened sequences extracted from the AST representation of source code. These word2vec vectors are then used as the input to a Long Short-Term Memory (LSTM) Recurrent Neural Network (RNN) as the training data.

Pythia's novel approach to vectorizing source code using word2vec is an evolution over the prior Boolean vector model used by Bruch, Monperrus, and Mezini (2009) for their Best Matching Neighbors (BMN) model and shared by D'Souza, Yang, and Lopes (2016) for their modified BMN approach, or the frequency vector model used by Proksch, Lerch, and Mezini (2015) for their Pattern Based Matching (PBM) model. The key evolution is using an established model for representing complex word embedding rather than the minimally required vectors they needed to drive their models.

Svyatkovskiy, Zhao, Fu, and Sundarsan (2019) leverage the word embeddings to drive their LSTM training model, which provides richer historical context for the use of specific syntax tokens. They show that by using this richer historical context, they were able to improve the accuracy of code completion suggestions provided to the developer. They demonstrate this improvement by measuring the Top-k accuracy and the Mean

Reciprocal Rank (MRR) against a Markov chain code completion system, which they consider as the current state of the art. The Top-k accuracy is defined as

$$Acc(k) = \frac{N_{top-k}}{Q}, (1)$$

$$MRR = \frac{1}{Q} \times \sum_{i=1}^Q \frac{1}{rank_i}, (2)$$

Where  $N_{top-k}$  represents the top  $k$  suggestions,  $Q$  is the total set of data samples, and  $rank_i$  is the prediction rank of a recommendation. Using Top-k, the top-1 represents how often the first code completion suggestion is correct, and top-5 measures how often the correct code completion suggestion is in the first 5 suggestions. The results of Pythia and Markov Chain for the same data set are shown in Table 6.

Class Name	Top-5 Accuracy Markov Chain	Top-5 Accuracy Pythia
os	0.863	0.950
numpy	0.575	0.697
list	0.978	0.989
str	0.974	0.988
os.path	0.895	0.957
sys	0.821	0.959
wx	0.272	0.533
logging	0.846	0.914
time	0.951	0.980
tensorflow	0.511	0.754

Table 6 – Accuracy of Pythia and Markov Chain (Svyatkovskiy et al., 2019)

Svyatkovskiy, Zhao, Fu, and Sundarsan (2019) tried four different neural network models and made it a hyperparameter for their automated testing. First is a fully connected LSTM RNN. Second is the Gated Recurrent Units RNN (GRU) with predicted embedding. Third is LSTM with predicted embedding. Fourth, is LSTM with attention for temporal data. In terms of Top-5 accuracy, all four models were close to each other, ranging between 0.91 and 0.93. However, the size of the resulting models was significantly different, as is visible in Table 7.

Model Architecture	Top-5 Accuracy	Model Size (MB)
LSTM + fully connected	0.91	202
GRU + predicted embedding	0.91	152
LSTM + predicted embedding	0.92	152
LSTM + attention	0.93	164

Table 7 – Top-5 Accuracy of Different RNN Models for Pythia (Svyatkovskiy et al., 2019)

The Pythia model chose to use LSTM with predicted embedding for their final model. This provided a small loss in Top-5 accuracy for a reduction of 12 megabytes in the trained model. The slight increase in quality using LSTM with attention is supported by the findings of Li, Wang, Lyu, and King (2018) in their paper exploring using Neural Attention and Pointer Networks to improve code completion suggestions. Li, Wang, Lyu, and King (2018) explore a similar approach to Pythia, with a key difference between the two models the use of attention instead of predicted embedding.

Svyatkovskiy, Zhao, Fu, and Sundarsan (2019) provide the results showing overall accuracy and MRR of four different code strategies compared to Pythia. These results are visible in Table 8 below. The results clearly show that Pythia is significantly improved

against the other models. The different models tested were Alphabetic, which is the complete list of all possible results sorted alphabetically; Frequency, which is the complete list of all possible results sorted by frequency of use; Frequency-if, which is the complete list of all possible results sorted by frequency of use taking if-else blocks into account; and Markov Chain modeling the relationship between different sequences of method invocation chains.

<b>Model</b>	<b>Top-1 Accuracy</b>	<b>Top-5 Accuracy</b>	<b>MRR</b>
Alphabetic	0.36	0.47	0.372
Frequency	0.38	0.64	0.495
Frequency-If	0.40	0.67	0.521
Markov Chain	0.58	0.83	0.704
Pythia	0.71	0.92	0.814

Table 8 – Comparison of Accuracy and MRR of Python and Four Models (Svyatkovskiy et al., 2019)

Svyatkovskiy, Zhao, Fu, and Sundarsan (2019) used 2,700 top-stared and non-forked Python projects as the corpus of source data. This is significantly less than the 20,000 used by D’Souza, Yang, and Lopes (2016) for their Python code completion model, but with significantly better results. Pythia had an overall MRR of 0.814, but PyReco averaged approximately 0.500, with a max MRR of 0.727 (D’Souza et al., 2016; Svyatkovskiy et al., 2019). Pythia achieves superior results trained on less data for more general-purpose cases including Python standard and third-party libraries.

The results for Pythia are an improvement over prior models, but it continues to demonstrate higher accuracy for standard libraries over third-party libraries. The

reduction in data set size and increased accuracy is an opportunity to improve third-party accuracy by introducing more source code repositories into the source corpus.

## Measuring Code Completion

### Prefix Scoring with Change Replay

Robbes and Lanza (2008) developed a novel approach to comparing multiple code completion strategies as a means of comparing their different strategies presented in their paper. Their approach is the capturing of changes creating during the development effort and use that information as a source of history similar to a version control history model they called an evolving AST. As the program is changed, the AST will reflect those changes. They capture the atomic AST events reflecting adding, changing, removing, and moving nodes within the AST and composite changes that represent multiple atomic events. These events are stored in the order of execution so they can be replayed in order, duplicating the developer's context.

Replaying the stored AST events in order but stopping prior to the point a method name is captured, allows the replay engine to try different code completion algorithms at the same contextual point the developer experienced when the original context was captured. In trying different code completion engines, Robbes and Lanza (2008) can capture key metrics useful in demonstrating the accuracy and efficiency of each algorithm for the same contextual situation and feed this data into their evaluation model. Instead of the traditional precision and recall metrics for measuring prediction algorithm, they give shorter prefixes and higher ranked results more weight. They seek to calculate  $G_i$

for each prefix, where  $I$  is the prefix length (Robbes & Lanza, 2008) using the following formula:

$$G_i = \frac{\sum_{j=1}^{10} \frac{results(i,j)}{j}}{attempts(i)} \quad (3)$$

In this calculation,  $i$  is the prefix length;  $j$  is the index;  $results(i, j)$  are the number of correct predictions at index  $j$  for prefix length  $i$ , and  $attempts(i)$  is the times the benchmark was run for the prefix length  $i$ . An accuracy of 100% for a given algorithm, would have a grade of 1 for that prefix length. With the graded accuracy, a score is calculated favoring shorter prefixes of 2 to 8 and multiplied by 100 for easier reading.

$$S = \frac{\sum_{i=1}^7 \frac{G_i + 1}{i}}{\sum_{k=1}^7 \frac{1}{k}} \times 100 \quad (4)$$

Robbes and Lanza's (2008) premise is that with a standard means of calculating code completion accuracy taking the developers context into account, different strategies for generating code completion suggestions can be measured and compared. However, they assumed that user input is anchored at the front as a prefix. Code completion strategies since their paper was published support user input for filtering as a prefix, postfix, infix, as individual character filters, and combinations of these models. Robbes and Lanza's (2008) formulas might be adaptable for any one of those user input models but would not be as easily adaptable for mixed user input models. Finally, their grading and scoring models are not just looking for the highest ranked solution, but the highest rank in the shortest suggestion list and with the shortest user supplied prefix. The size of the suggestion list is not relevant to the accuracy if the relevant results are at the top of the

list, and a user supplied prefix is no longer an applicable measure given the different filtering models in modern code completion systems.

### Mean Reciprocal Rank

Mean Reciprocal Rank (MRR) captures the rank of the relevant result that the developer considers accurate. The Mean Reciprocal Rank defines  $Q$  as the number of data queries,  $rank_i$  as the ranked position of the first relevant result for the  $i$ -th query, and is calculated with the following formula:

$$MRR = \frac{1}{Q} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \quad (5)$$

The MRR calculation is used by D'Souza, Yang, and Lopes (2016) to evaluate PyReco; by Asaduzzaman, Roy, Schneider, and Hou (2016) to evaluate their Context-sensitive Code Completion; by Svyatkovskiy, Zhao, Fu, and Sundarsan (2019) to evaluate Pythia; and by He, Xu, Zhang, Hao, Feng, and Xu (2021) to evaluate PyArt. This makes the use of MRR one of the more consistently used measurables for code completion suggestion accuracy.

### Top-k Accuracy

Top- $k$  accuracy measures the likeliness that a result will be in first  $k$  results from a code completion suggestion request. Results in when  $k = 1$  are the most accurate, since the developers expected result was the first entry in the result set. Different research used different values for  $k$  in addition to top-1. All of the papers using a top- $k$ , reported the results of the top-1. This measure is used by Asaduzzaman, Roy, Schneider, and Hou (2016) to evaluate their Context-sensitive Code Completion; by Nguyen, Hilton,



Codoban, Nguyen, Mast, Rademacher, Nguyen, and Dig (2016) to measure the accuracy of their APIREC model; by Svyatkovskiy, Zhao, Fu, and Sundarsan (2019) to evaluate Pythia; and by He, Xu, Zhang, Hao, Feng, and Xu (2021) to evaluate PyArt. Top- $k$  is another one of the more consistently used measurables for code completion suggestion accuracy.

### Recall

Bruch, Schäfer, and Mezini (2008) define Recall as the ratio between relevant recommendations made by a system for a given query and the total number of recommendations that should have been made. While Recall is used as part of the evaluation of the Pattern-Based Bayesian Network model (Proksch et al., 2015); of Context-sensitive Code Completion (Asaduzzaman et al., 2016); of PyReco (D'Souza et al., 2016); and PyArt (He et al., 2021); it is rarely used as a primary evaluation metric. Further, a number of papers, including the research by Svyatkovskiy, Zhao, Fu, and Sundarsan (2019) for Pythia do not use Recall.

### Precision

Bruch, Schäfer, and Mezini (2008) define Precision as the ratio between relevant recommendations made by a system for a given query and the total number of recommendations actually made by the system. Like recall, this metric is used by a number of research papers, but not with consistency. Further, it does not address the key question of is the relevant response earlier in the list. As the code completion systems have improved, the objective is not to measure if the right response is in the suggestions list, but where it is in the suggestions list. Hence, Pythia does not use Precision.

## Replicating Pythia

Schuster, Song, Tromer, and Shmatikov (2020) replicated the Pythia model replacing PTVS (*Microsoft/PTVS*, 2015/2020) with `46steroidd` (*Welcome to Astroid's Documentation!* — *Astroid 2.5.9.Dev10+g432aa99 Documentation*, n.d.) for parsing Python source files into an AST, and replacing ML.NET (*ML.NET / Machine Learning Made for .NET*, n.d.) with PyTorch (*PyTorch*, n.d.). This replicated model was used to demonstrate data poisoning vulnerabilities when using neural nets for code completion. To demonstrate the viability of those attacks, they implemented Pythia, using 2,800 top-starred repositories from GitHub after some filtering for minimum and maximum AST node counts. This demonstrates that the Pythia model is replicable. Further, it demonstrates that key elements of the Pythia implementation, such as the AST parser and the machine learning library used, are replaceable with comparable frameworks.

## Chapter 3

### Methodology

#### Introduction

This research proposed a strategy to increase the code-completion accuracy of third-party Python libraries in the Pythia prediction model. To deliver this increased accuracy, the input data used for training and testing was increased to include more examples of Python code using targeted third-party libraries.

Pythia is an approach to code-completion that has demonstrated a significant improvement over other approaches for accuracy (Svyatkovskiy et al., 2019). It has also been proven to be reproduceable using alternate but comparable tooling to the original approach (Schuster et al., 2020). Schuster, Song, Tromer, and Shmatikov (2020) provided details about their hyperparameters for Pythia training that differ from the paper from Svyatkovskiy, Zhao, Fu, and Sundaresan (2019), such as the vocabulary criteria, number of hidden units, and training epochs for the LSTM neural network. This allowed the model generation to be significantly accelerated compared to the original Pythia approach with only a small loss in accuracy. Schuster, Song, Tromer, and Shmatikov (2020) state the accuracy difference may be related to the Pythia's reliance on undocumented internal APIs in Visual Studio. Given that the approach was to show a relative increase in accuracy between third-party Python libraries and Python standard libraries, this overall accuracy degradation was not relevant.

The overall workflow of the proposed new model is reflected in figure 13 below. This model consists of four stages of operation: Data Collection; Preprocessing; Offline Model Training; and Testing. Each of these four stages of operations are discussed below. The general operational flow is modeled on Pythia’s workflow with key changes to the selection of the AST data passed to the Matrix Builder and the Test Suite instead of Pythia’s integration with IDEs. Specific differences from the Pythia workflow are discussed below.

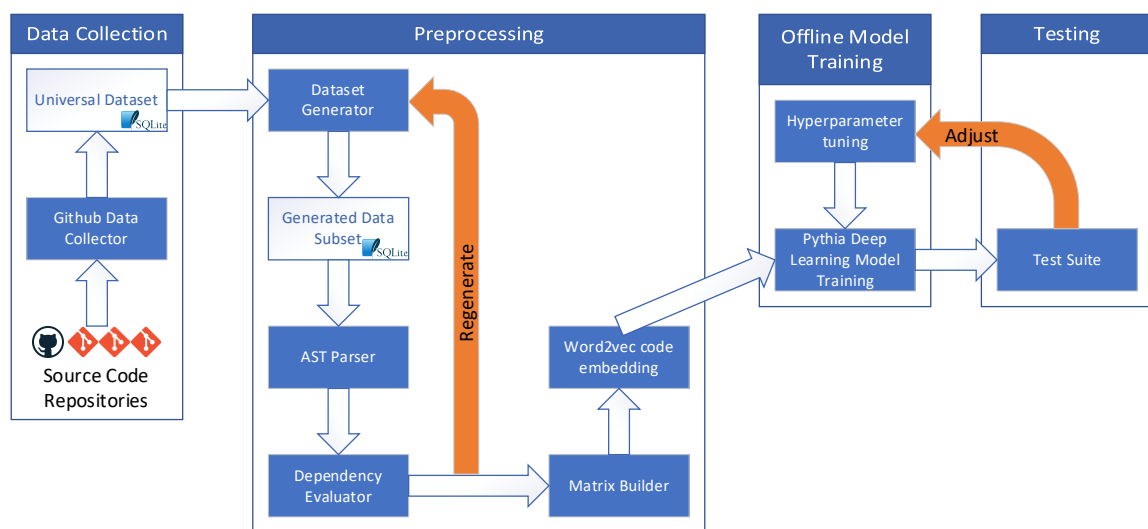


Figure 13— Workflow from source code to testing.

## Data Collection

The core proposal in this dissertation is to increase the number of GitHub Python projects evaluated by the model generator. The challenge was that it is not known prior to evaluating the initial candidate set of Python projects and the resulting accuracy if more projects are required in the dataset or not. Further, GitHub searches are non-deterministic

due to the ever-changing number of projects, files within the projects, and stars associated with a project.

### GitHub Data Collector

To create a fixed set of GitHub projects to work with over multiple queries, a command line tool was created to search GitHub ordered by number of stars descending and store the results in an SQLite database. This was the Universal Dataset used as a static foundation for all further dataset generations. Since GitHub is not deterministic between search queries, multiple overlapping queries were run sequentially to collect as many matching projects as possible, while ignoring duplicate projects. These multiple runs were for projects with different minimal sizes in kilobytes. This process was repeated until the maximum number of target projects was reached or exceeded.

GitHub was searched for the 30,000 projects ordered by number of stars descending at the following eight different minimum projects sizes in kilobytes in descending order: 0, 1, 10, 100, 1000, 5000, 10000, 100000. For each project, detailed information was collected based on the available GitHub data fields and meta information available. These fields included, but were not limited to, the GitHub ID, result sort order, creation date, last update date, Git clone URLs, branch information, fork status, stars, size, owner ID, and programming language. For each project discovered, the GitHub project tree was walked to discover all Python files associated with the default branch of the repository, and the GitHub access URL, SHA code, and path name was stored. When these searches were completed, a snapshot of GitHub at a moment in time had been created. That snapshot allowed for the unique identification of distinct repositories based on the

GitHub ID. For each of those distinct repositories identified, all files relevant to the AST processing were located using GitHub's URL to the raw source code file. This allowed all files to be accessed without having to clone the projects. Further, this model allowed for consistently repeating the rest of the model without being impacted by constant changes to GitHub over time.

## Preprocessing

### Dataset Generator

A key requirement of this workflow was the ability to regenerate a new dataset that is a subset of the universal dataset based on specific input criteria. The initial dataset utilized the same criteria defined by Svyatkovskity, Zhao, Fu, and Sundaresan (2019). This was defined as the 2,700 non-forked top-starred Python projects. The Pythia dataset was identified as containing 15.8 million method calls, but without knowing the specific projects and their sizes, and if the method call count was in the input dataset or the testing dataset, it is not possible to use both the repository count, and method count as a target for the input dataset. The repository count was chosen since it was the broadest match that could be reasonably replicated. The subset of data was then divided into development and test sets at 70% and 30% respectively as was used by Pythia. Further, the development test set was split at random into training and validation sets with 80% and 20% respectively. These ratios were used for all datasets generated and not just the initial dataset. All the information required to generate the dataset, and the usage classification of each repository was stored as a set in the database. This allowed for rerunning with the same data on demand.

This need for regeneration on demand using different parameters was supported by the creation of a command line tool that regenerated a new dataset from the universal dataset, dataset configuration parameters, and an optional source sub-dataset. When a sub-dataset was supplied, the new dataset extended the supplied sub-dataset with unused unique repositories found in the universal dataset. This allowed for the input dataset to be incrementally extended. The newly generated dataset was stored in the database as a unique dataset, including the identity of the source dataset and the update parameters. This allowed for reusing the same set of data and an audit trail of how it was generated.

### Code Processing

All source files identified in the input dataset were parsed into their representative Abstract Syntax Tree (AST). This parsing phase removed non-code nodes and retained nodes representing member access expressions and method invocations. Meta information about the file, such as module imports information and type definitions, if any, were captured and stored with the file information to assist with non-training analysis. This information allowed the dependency evaluation process to identify how many files depend on specific Python standard and third-party modules. Further, this assisted in type predictions when the AST was flattened for consumption by the matrix builder.

Each syntax node and token name were mapped to an integer between 1 and  $V$  to assign a token index to each token.  $V$  is an autotuned parameter representing the vocabulary size. All tokens in the vocabulary were mapped in the range of  $1..V$ , while all tokens outside of the vocabulary were mapped in the range of  $V+1..\infty$ . The size of the

vocabulary was defined based on the frequency of the tokens being greater than the tunable hyperparameter for the token frequency threshold. The initial default for the token frequency threshold was 500 as used in Pythia. Any token equal to or greater than that threshold was mapped in the  $1..V$  range.

Variable names were normalized into typed or untyped equivalents that simplified their uniqueness to a generated name. As an example, the variable named `myText` with a type of string is relabeled as `string_1`. Variables of undeterminable type were assigned the type of untyped. Union types were assigned the type of all relevant types identified. Aliases were normalized so multiple aliases for the same type will be treated equivalently. This simplified the matching process and minimized the impact of variable name uniqueness during the training process. Further, eliminating the unique variable names reduced the vocabulary size significantly.

Each AST was generated using pre-order depth-first traversal of the Python tokens. Tokens were partitioned based on their relevance leading towards a method call, with irrelevant tokens, such as comments, being discarded. The scan was used to generate an in-order sequential sequence of relevant AST tokens with a “.” as the end of sequence termination character. These token sequences were used by word2vec as training sequences during testing to represent the user generated code when generating a completion projection, and during evaluation for the correctness of the prediction. When generating training sequences, up to  $T$  lookback tokens prior to the method call were retained.  $T$  is a tunable hyperparameter for the model, and the initial default was 1000 as used by Pythia. This was tuned based on performance and results of testing.



## Dependency Evaluator

There was a need to determine if the input dataset needed to be grown with more projects using specific modules. A Dependency Evaluator that examined parsed AST meta information, the results of any existing prior test results, and configuration information to determine if the dataset needed to be expanded and regenerated was planned. If a new dataset was required or a revision to an existing dataset, the Dependency Evaluator was going to generate a new dataset configuration and trigger dataset regeneration. Two tunable hyperparameters were planned to be added to support this stage. The first was a tolerance threshold to control when a dataset is out of or in tolerance ranges. The second was a maximum number of regenerations before the generated dataset automatically accepted the current results and moved to the next phase of processing.

## Encoding Code Snippets

The sequence of nodes from flattened AST were mapped into matrices as input to word2vec. The AST is walked pre-order depth first to retain a representation of the contextual relationship between the code tokens. Method calls were used for the labels and were one-hot encoded. The matrices were then used by word2vec to produce low-dimensional dense vectors that were used by the LSTM. This preserved the semantic relationships discovered through AST parsing. The output softmax produced by word2vec was used as the input for the LSTM training and as the output classification matrix, as implemented in Pythia (Svyatkovskiy et al., 2019).

## Offline Model Training

### Pythia Deep Learning Model Training

The Pythia model training uses a Long Short-Term Memory (LSTM) Recurrent Neural Network (RNN) to generate the prediction softmax (Svyatkovskiy et al., 2019). The diagram of Pythia's RNN architecture is shown in Figure 14. Svyatkovskiy, Zhao, Fu, and Sundaresan (2019) used 2 RNN layers with 100 hidden units per layer and a dropout rate of 0.8. Schuster, Song, Tromer, and Shmatikov (2020) replicated Pythia using the same 2 RNN layers, but with 8 hidden units per layer and a dropout rate of 0.75 with a small decrease in overall accuracy. However, the time to generate the models using Schuster, Song, Tromer, and Shmatikov's (2020) method was reduced to 15 hours for a single GPU from approximately 41 hours with multiple GPUs using Svyatkovskiy, Zhao, Fu, and Sundaresan's (2019) method. This single GPU method also reduced the complexity of the model generation by removing the need to pad and batch the training buffers used in the multi-GPU LSTM.

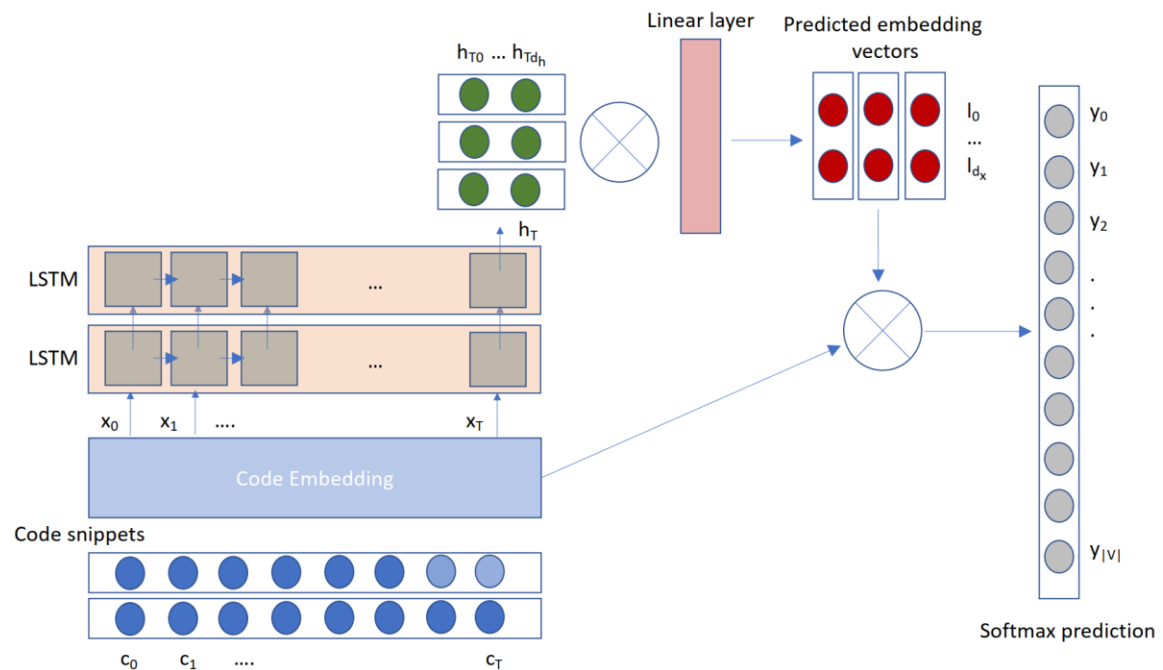


Figure 14— Pythia Neural Network Architecture

This dissertation measured the ratio of Python standard library accuracy to Python third-party library accuracy using Schuster, Song, Tromer, and Shmatikov's (2020) method. This ratio was similar to the data shared by Svyatkovskity, Zhao, Fu, and Sundaresan's (2019), and the simpler method was used to reduce the overall complexity and model generation time. This allowed for faster feedback from the prediction testing to the dataset generation to adjust dependencies for another model generation cycle.

### Hyperparameter Tuning

Svyatkovskity, Zhao, Fu, and Sundaresan's (2019) have defined the hyperparameters for their training of Pythia in figure 15. Most of these hyperparameters were the initial settings for this dissertation's model. Schuster, Song, Tromer, and Shmatikov's (2020) replication of Pythia showed that some of these hyperparameters can be eliminated or changed to reduce complexity and generation time with little to no impact on overall accuracy. Their hyperparameters were considered where appropriate.

Hyperparameter	Explanation	Best value
$\lambda_0$	Base learning rate	0.002
$\gamma$	Learning rate decay per epoch	0.97
$N$	Number of recurrent neural network layers	2
$d_h$	Number of hidden units in LSTM, per layer	100
$T$	Number of lookback tokens, timesteps through which backpropagation is run	1000
$T_{RNN}$	Number of timesteps through which backpropagation is run	100
RNN type	Type of RNN	LSTM
$d_b$	Batch size	256
Loss function	Type of loss function	Categorical cross-entropy
$d_x$	Embedded vector dimension	150
Optimizer	Stochastic optimization scheme	Adam
Dropout	Dropout keep probability	0.8
L2 Regularization	Weight regularization of all layers	10
Clip norm	Maximum norm of gradients	10
Token frequency threshold	Minimum frequency of syntax token in the corpus for inclusion in vocabulary	500

Figure 15— Pythia Hyperparameters (Svyatkovskiy et al., 2019)

## Evaluation

### Key Questions

To evaluate the results of this dissertation, the following questions were answered:

1. Does the replicated implementation of Pythia produce a ratio between Python standard libraries and Python third-party libraries similar to the original Pythia analysis by Svyatkovskiy, Zhao, Fu, and Sundaresan's (2019)?
2. Will increasing the number of projects in the input dataset to include more projects with references to specific Python third-party libraries improve the accuracy of those third-party libraries?
3. How many additional projects are required to be added to increase the accuracy significantly?

## Evaluation Metrics

The two primary metrics that were captured to measure the improvement of the generation model are the Top-k accuracy and the mean reciprocal rank (MRR).

Svyatkovskity, Zhao, Fu, and Sundaresan's Pythia paper (2019) publish an overall MRR and Top-5 accuracy, and the Top-5 for the 10 most frequent libraries their dataset included. This dissertation uses the Pythia overall MRR and Top-5 as a baseline to compare against.

The secondary metric used to indicate improvement was the shift in the K counts from missing to  $K > 5$  to K5 to K1. The shifting of K counts shows how the dataset is improving that is sometimes not visible in the Top-K and MRR. As an example, a shift from missing to  $K > 5$  is an improvement of the model and was visible in the MRR, but not reflected in the Top-5. Similarly, a move from K5 to K1, was not significantly reflected in the MRR for larger datasets.

Given the initial dataset for Pythia was not identified, it was not possible to ensure the recreation of their top-10 Python libraries given a fully random dataset. Therefore, the dataset generator supported encouraging more projects with the same libraries identified in Pythia's results. This was used to measure against the same libraries. Finally, the model was run without the dataset generation biased to include the Pythia top-10 libraries to demonstrate the ability to generate improved results for a random set of 10 Python libraries.

## Differences from Pythia

### Preprocessing

The preprocessing phase differed from the original Pythia model by introducing the ability to regenerate the input dataset to include more projects that use third-party Python modules. This involved the new data collection process, dataset generator, and the dependency evaluator. Other aspects of the Pythia model remained, though the implementation differed slightly due to alternative tooling used.

### Model Training

The offline model training was simplified by removing the parallel processing and reducing the number of hidden layers. This reduced the cycle time for each model test with a generated dataset and reduced the time to implement. The reduced complexity also removed other potential variables from the comparison that could adversely impact the results.

### Model Quantization

Model quantization was not implemented. This process's purpose was to reduce the physical size of the stored model as an optimization for serving results to clients. It is not significant to the accuracy results of predictions.

## Serving Recommendations

The ability to serve recommendations to an IDE is not required to test the prediction accuracy of the implemented model. This is only necessary to demonstrate the integration with developer tools. Since it is not required to evaluate the predictions, it was not implemented.

## Conclusion

This research showed that by increasing the number of projects into the Pythia dataset using guidance based on the frequency of Python third-party modules included, the accuracy of Pythia for those third-party modules is increased. The successful demonstration of that improvement justifies the additional effort to include this guided dataset adjustment into existing Pythia based code recommendation systems.

## Chapter 4

### Results

#### Introduction

The Pythia model was replicated using Python 3.11, gensim 4.3.1 for Word2Vec processing, keras 2.12.0, and tensorflow 2.12.0. Numerous hardware configurations in AWS and Google Cloud were tried, but the below configurations were stable and produced results reliably in reasonable timeframes. Based on memory, CPU, GPU, and disk needs, the following runtime environments were used:

- Local computer with AMD Ryzen 9 5960X 16 core processors / 32 cores, 64 GB RAM, NVIDIA GeForce RTX 3070 Ti with 40GB GPU memory, and approximately 8TB SSD storage.
- Google Collab Pro+ with NVIDIA A100 GPU with 40GB GPU RAM, 83.6GB CPU RAM and 166.8 SSD.
- AWS r6a.4xlarge EC2 with 16 vCPU, 128GB RAM, and 2TB storage.

The tests were successfully completed without exhausting the initial GitHub repositories collected during the proposal phase.



## Changes from Planned Methodology

The automated dependency evaluator planned for in the methodology was not implemented. The processing of data on different platforms (local, Google Collab, and AWS EC2s) required the frequent migration of data between the platforms. This migration drove up the time needed and the costs of each pass. It was necessary to manually select the dependencies and use broader step sizes in the dataset instead of the planned auto-incrementing steps in repository sizes based on the results of individual tests.

## Baseline Model

The Pythia paper from Svyatkovskity, Zhao, Fu, and Sundaresan (Svyatkovskiy et al., 2019) list ten Python libraries with the Top-5 accuracy results. They do not list individual MRR results, but instead provide an overall Top-1 accuracy score, Top-5 accuracy score, and MRR for Pythia. They identify the source data for the overall scores as the “test set for the Pythia neural model and various baselines.”

This left a few possibilities for how they computed their over results:

- The calculation is the aggregation over the entirety of the test data set, but not the validation or training data sets.
- The calculation is the aggregation over a different, but unidentified, combination of the test, validation, and training data sets.
- The calculation is an aggregation of the ten libraries used in the Top-5 Pythia results.

The first two options are impossible to replicate without the original source input data used by Pythia and the exact computation used. Neither of this information was provided. The last calculation cannot be replicated without the exact formula and the original source input libraries used by Svyatkovskiy.

To circumvent this gap in information, the baseline version of this dissertation's implementation was compared against the Pythia K5 accuracy only for the ten libraries with measurable results by Svyatkovskiy. One of the ten libraries had no measurable results in this dissertation's baseline, which generated an outlier situation that skewed those results. To compensate for that scenario, the baseline was compared against all ten and the nine libraries with results in both models. The average change in K5 accuracy over the libraries with and without the outlier were calculated between Pythia and this dissertation's baseline. The comparison between Pythia and the baseline model used in this dissertation is shown in Table 9 below. This table shows the Python libraries used in Svyatkovskiy's paper, the K5 accuracy reported for Pythia, the K5 accuracy generated by this dissertations baseline model, and improvement percentage over the Pythia K5 achieved by the baseline model.

<b>Library</b>	<b>Pythia K5 Accuracy</b>	<b>Dissertation K-5 Accuracy</b>	<b>Improvement</b>
os	0.863	0.976	+2.71%
numpy	0.575	0.843	+21.02%

list	0.978	0.975	-1.40%
str	0.974	0.968	-2.01%
os.path	0.895	0.980	+2.39%
sys	0.821	0.947	+1.61%
wx	0.272	0.000	-100.00%
logging	0.846	0.970	+6.11%
time	0.951	0.977	-0.33%
tensorflow	0.511	0.838	+75.96%

Table 9 - Comparison of Pythia and Baseline Model for K5 Accuracy

The average change across all ten libraries identified by Pythia is an improvement of 0.61%. When the outlier of the wx library is removed, the average of the improvement across the nine remaining libraries is 11.79%. Considering the full set of ten libraries shows that this dissertation's baseline model is comparable to Pythia's results in terms of K5 accuracy, the only per-library metric provided by Svyatkovskiy. When the outlier of the wx library is removed, the baseline model shows a measurable improvement over the results presented by Svyatkovskiy.

Since the MRR results of Svyatkovskiy are not directly reproducible, the MRR for the ten Pythia identified libraries, the MRR for the Pythia libraries without the outlier of wx, and the MRR for all libraries included in the test data set was calculated and compared against the MRR provided by Svyatkovskiy. The results of this MRR calculation are provided in Table 10 below.

<b>Pythia</b>	<b>Baseline w/ Outlier</b>	<b>Baseline w/o Outlier</b>	<b>Baseline w/ All Libraries</b>
0.814	0.765	0.849	0.705

Table 10 - Comparison of Pythia and Baseline Model for MRR

The table shows that when comparing the baseline model MRR, including the outlier, against Svyatkovskiy's MRR result, leads to a 6.02% decrease in MRR. However, when comparing the baseline model MRR, excluding the outlier, against Svyatkovskiy's MRR result, leads to a 4.30% increase in MRR. Comparing the MRR average of all libraries in the baseline model's test set against Svyatkovskiy's MRR results in a 13.39% decrease in MRR. The challenge here is that without being able to use Svyatkovskiy's input dataset to calculate a baseline MRR, it is not possible to have a meaningful comparison in absolute MRR values.

Given this limitation in showing accurate MRR comparisons against Svyatkovskiy, this dissertation uses the K5 accuracy comparison to establish the validity of the baseline model being able to replicate the capabilities of Pythia. Comparable model parity is shown for the libraries tested by Svyatkovskiy with the outlier with a difference of 0.61%. While excluding the outlier model raises this difference to 11.79% showing a significant improvement. The expectation is that for the same libraries, the baseline model will generate similar or better results than the Pythia model as documented by Svyatkovskiy.

This dissertation calculated the impact of increasing the quantity of specific libraries based on their relative change over the baseline model's in K1, K5, and MRR. It also calculated the improvement in the target counts where a prediction moves from missing to greater than K5, to K5, and to K1. Given the lack of required information to precisely replicate the Svyatkovskiy Pythia model, the baseline model was used as a comparable stand in for the Pythia model, and improvements against the baseline model are expected to replicate against Svyatkovskiy Pythia in a similar fashion.

#### Expansion by 100 Repositories

The first expansion set test used a maximum quantity of 100 new repositories per target library. This represents a 3.70% increase in the total number of repositories in the dataset, which expanded from 2,700 repositories to a maximum of 2,800 repositories.

The following libraries were selected to be expanded by 100: markdown, yaml, and tqdm. The reason for their selection was because the ratio of count to missing was significant, the current MRR had room for improvement, and the test data sets were not significant in size. This allowed for the models to be recreated and tested frequently during the hyperparameter tuning phase of the model generation.

#### markdown Library

The markdown library in the baseline model had a total of 2,167 identified method calls in the test data set, with 876 of those method calls being K1 accurate, 907 being K5

accurate, 3 being greater than K5 accurate, and 1,257 missing from the prediction list. This resulted in a K1 accuracy rate of 40.42%, a K5 accuracy rate of 41.86% and an MRR of 0.410 as seen in table 11.

Count	K1	K5	Kn	Kmissing	K1 Acc.	K5 Acc.	MRR
2,167	876	907	3	1,257	0.40425	0.41855	0.41015

Table 11 - Markdown Library Results using Baseline Model

The dataset was expanded by 100 repositories that included imports of the markdown library. This increase only applied to the training and validation dataset, and not to the testing dataset. The same testing dataset was used in all tests to ensure that the changes to the results were not impacted by the tests, only by the input datasets. The results of adding 100 additional markdown including repositories resulted in a significant improvement, with the raw results visible in table 12 below.

Count	K1	K5	Kn	Kmissing	K1 Acc.	K5 Acc.	MRR
2,167	941	1,009	11	1,147	0.43424	0.46562	0.44733

Table 12 - Markdown Library Results using Baseline+100 Model.

The change between the baseline dataset and the baseline+100 dataset is visible in table 13 below. The results show a significant left shift improvement in the counts of predictions. The K1 count increased by 7.42%, K5 count increased by 11.25%, the

greater than K5 (Kn) counts increased by 266.67%, and the missing predictions count (Kmissing) reduced by 8.75%. K1 and K5 accuracy directly correlates to their count improvements, so they improved by 7.42% and 11.25% respectively. MRR improved by 9.07%. This is provided in table 13 below.

$\Delta K1$	$\Delta K5$	$\Delta Kn$	$\Delta K_{missing}$	$\Delta K1 \text{ Acc.}$	$\Delta K5 \text{ Acc.}$	$\Delta MRR$
+7.42%	+11.25%	266.67%	-8.75%	+7.42%	+11.25%	+9.07%

Table 13 - Markdown Library Improvement Results of Baseline v Baseline+100

#### tqdm Library

The tqdm library in the baseline model had a total of 1,475 identified method calls in the test data set, with 902 of those method calls being K1 accurate, 977 being K5 accurate, 17 being greater than K5 accurate, and 481 missing from the prediction list. This resulted in a K1 accuracy rate of 61.15%, a K5 accuracy rate of 66.24% and an MRR of 0.635 as seen in table 11.

Count	K1	K5	Kn	Kmissing	K1 Acc.	K5 Acc.	MRR
1,475	902	977	17	481	0.61153	0.66237	0.63451

Table 14- TQDM Library Results using Baseline Model

The dataset was expanded by 100 repositories that included imports of the tqdm library. This increase only applied to the training and validation dataset, and not to the

testing dataset. The same testing dataset was used in all tests to ensure that the changes to the results were not impacted by the tests, only by the input datasets. The results of adding 100 additional repositories including the tqdm module resulted in a minor improvement that was less significantly visible than shown with the markdown library. The raw results are visible in table 15 below.

Count	K1	K5	Kn	Kmissing	K1 Acc.	K5 Acc.	MRR
1,475	905	987	7	481	0.61356	0.66915	0.63830

Table 15 – TQDM Library Results using Baseline+100 Model.

The change between the baseline dataset and the baseline+100 dataset is visible in table 16 below. The results show a minor left shift improvement in the counts of predictions. The K1 count increased by 0.33%, K5 count increased by 1.02%, the greater than K5 (Kn) counts decreased by 58.82%, and the missing predictions count (Kmissing) remained the same. K1 and K5 accuracy directly correlates to their count improvements, so they improved by 0.33% and 1.02% respectively. MRR improved by 0.60%. This is provided in table 16 below.

$\Delta K1$	$\Delta K5$	$\Delta Kn$	$\Delta Kmissing$	$\Delta K1$ Acc.	$\Delta K5$ Acc.	$\Delta MRR$
+0.33%	+1.02%	-58.82%	0.00%	+0.33%	+1.02%	+0.60%

Table 16 – TQDM Library Improvement Results of Baseline v Baseline+100



## yaml Library

The yaml library in the baseline model had a total of 564 identified method calls in the test data set, with 114 of those method calls being K1 accurate, 114 being K5 accurate, 0 being greater than K5 accurate, and 450 missing from the prediction list. This resulted in a K1 accuracy rate of 20.21%, a K5 accuracy rate of 20.21% and an MRR of 0.202 as seen in table 17.

Count	K1	K5	Kn	Kmissing	K1 Acc.	K5 Acc.	MRR
564	114	114	0	450	0.20213	0.20213	0.20213

Table 17 - Yaml Library Results using Baseline Model

The dataset was expanded by 100 repositories that included imports of the yaml library. This increase only applied to the training and validation dataset, and not to the testing dataset. The same testing dataset was used in all tests to ensure that the changes to the results were not impacted by the tests, only by the input datasets. The results of adding 100 additional yaml including repositories resulted in no change to the results. The raw results are shown in table 18 below. It was expected based on the experience with the tqdm library, that the results of adding 100 yaml repositories would yield little improvement. The result of no improvement was not expected.

Count	K1	K5	Kn	Kmissing	K1 Acc.	K5 Acc.	MRR
-------	----	----	----	----------	---------	---------	-----

564	114	114	0	450	0.20213	0.20213	0.20213
-----	-----	-----	---	-----	---------	---------	---------

Table 18 - Yaml Library Results using Baseline+100 Model.

## Results Analysis

The results with 100 additional repositories were mixed across the three tests performed. The markdown library had significant improvements with an increase in K1 and K5 accuracy, the MRR score, and a left shift in the counts from missing through to K1. However, tqdm's results were marginal at best. The K1 and K5 accuracy improved, but not significantly. Similarly, the MRR increased by not significantly. The only left shift was from the already identified predictions from Kn to K5 to K1. Finally, the results for yaml were disappointing with no change in any of the calculated values.

Numerous different hyperparameters were tried to improve the model with no improvement, and most yielded worse results. Examining the new repositories included into input dataset showed that additional code being analyzed was very similar to the existing code being analyzed. There were some additional patterns of usage for tqdm, but there were no significant usages difference for yaml. This lack of diversity in the code base was visible in the testing data by showing no significant improvements in identifying new patterns. There was nothing new for the training to learn with the current dataset over what it had learned. It improved on what it had already learned but did not find something new.

This is not a fault of the model, but of the input dataset. The model was able to improve the predictability of items it could already find, the Kmissing to K5 and K1, but it could not find new predictions. The working theory being that there is a floor to the improvement value of an additional 100 repositories, which is only a 3.70% increase, that it is not bringing enough diversity to the training set to improve the results. Given the small size of tqdm and yaml tests relative to the entire dataset, 0.039% and 0.015% respectively, of the overall test dataset, it is probable they required more usage patterns for training to identify the small number of unique usage patterns in the test dataset. For reference, markdown represented 0.058% of the test dataset and saw significant improvement. The tqdm library saw a small improvement with 100 additional repositories at 0.039% of the test dataset, and yaml saw no improvement with 100 additional repositories and 0.015% of the test dataset. There is a floor to the improvement with 100 additional repositories that likely has a correlation to the size of the test dataset.

#### Expansion by 270 Repositories

Based on the results of testing with 100 additional repositories, two options were available to move the analysis forward: increase the size of the tests for the target libraries relative to the overall test dataset or increase the number of libraries being added into the input dataset. The former option would require re-baselining the model and would create bespoke testing channels based on the specific library sets. This was not conducive to proving the general improvement of the Pythia model through increasing the input dataset and was discarded as an option.

It was decided to increase the number of libraries added to the input dataset. The analysis in the prior section shows that 100 additional libraries, or 3.70%, was too small. The new number of additional libraries would be 10% of the repository count, or 270 additional libraries. This would be a significant enough increase to provide more input data diversity.

#### markdown Library

The markdown library in the baseline model had a total of 2,167 identified method calls in the test data set. The model was enhanced with 270 additional repositories containing the markdown library. The result was with 1,042 of those method calls being K1 accurate, 1,096 being K5 accurate, 8 being greater than K5 accurate, and 1,063 missing from the prediction list. This resulted in a K1 accuracy rate of 48.08%, a K5 accuracy rate of 50.58% and an MRR of 0.493 as seen in table 19 below.

<b>Model</b>	<b>K1</b>	<b>K5</b>	<b>Kn</b>	<b>Kmissing</b>	<b>K1 Acc.</b>	<b>K5 Acc.</b>	<b>MRR</b>
Baseline	876	907	3	1,257	0.40425	0.41855	0.41015
Baseline+100	941	1,009	11	1,147	0.43424	0.46562	0.44733
Baseline+270	1,042	1,096	8	1,063	0.48085	0.50577	0.49256

Table 19 – Markdown Library Results for Baseline, Baseline+100, and Baseline+270 Models

This dataset with 270 additional repositories yielded significant improvements over the baseline and the baseline with 100 additional repositories. The raw results of all 3

tests are below in table 19. The baseline+100 yielded improvements of 7.42%, 11.25%, and 9.06% over the baseline for K1 accuracy, K5 accuracy, and MRR, respectively. The baseline+270 yielded improvements of 18.95%, 20.84%, and 20.09% over the baseline for K1 accuracy, K5 accuracy, and MRR, respectively. The baseline+270 improved over baseline+100 for K1, K5, and MRR by 10.73%, 8.62%, 10.11% respectively. Table 20 below shows the percentage change from baseline to the baseline plus 100 and 270 additional markdown repositories. The table also shows the percentage change between 100 and 270 markdown repositories.

The increased repositories also impacted the predictions greater than K5 and the missing predictions. The missing predictions were reduced by 15.43% with 270 additional repositories compared to the baseline, and 7.32% when compared to the baseline+100. The predictions above K5 went down by 27.27% between baseline+270 and baseline+100. This shows that more of the expected predictions were now being captured by both the baseline+100 and baseline+270 models. The baseline+100 had K>5 predictions increase by 266.67 over baseline. This implies that many of the missing predictions that are now identified are still in the K>5 predictions list. However, with baseline+270, this number is reduced by 27.27% over the baseline+100. This shows that the baseline+270 is not only finding more predictions, but significantly more of them are in the K1 to K5 prediction range.

<b>Model</b>	<b>K1</b>	<b>K5</b>	<b>Kn</b>	<b>Kmissing</b>	<b>K1 Acc.</b>	<b>K5 Acc.</b>	<b>MRR</b>
--------------	-----------	-----------	-----------	-----------------	----------------	----------------	------------

Baseline+100 v Baseline	7.42%	11.25%	266.67%	-8.75%	7.42%	11.25%	9.07%
Baseline+270 v Baseline	18.95%	20.84%	166.67%	-15.43%	18.95%	20.84%	20.90%
Baseline+270 v Baseline+100	10.73%	8.62%	-27.27%	-7.32%	10.73%	8.62%	10.11%

Table 20 – Markdown Library Change % for Baseline, Baseline+100, and Baseline+270 Models

The MRR reinforces this conclusion by showing the improvement in the rank of the predictions by 10.11% using the baseline+270 compared to from the baseline+100 model. When compared to baseline, the MRR shows a 20.90% improvement in the rank of the predictions. These MRR improvements reinforce the improvements visible in the K1 and K5, plus the shift from Kmissing to K>5.

#### tqdm Library

The tqdm library in the baseline model had a total of 1,475 identified method calls in the test data set. The baseline model results were significantly better than markdown, and the resulting improvement to tqdm by adding 100 repositories was not as significant. Adding 100 tqdm repositories yielded improvements of 0.33%, 1.02%, and 0.60% over baseline for K1, K5, and MRR respectively. The number of missing predictions was unchanged. The number of K>5 predictions reduced by 58.82%, the only significant change, but in raw numbers the change was less significant, being a 17 to 7 reduction.

The addition of 270 tqdm repositories improved over the baseline and baseline+100, but not significantly. K1 improved by 2.00%, K5 by 1.13%, K>5 by 64.49%, and MRR by 1.78%. Kmissing was still unchanged. Compared to baseline+100, the baseline+270 improvement was 1.66%, 0.10%, 14.29%, and 1.18% for K1, K5, K>5, and MRR, respectively. These results show that the baseline+270 has improved prediction rank for already known predictions but did not identify any of the missing predictions. It is an improvement, but not a significant improvement. The raw results are in table 21 below. The comparative percentage improvements of the models are in table 22 below.

<b>Model</b>	<b>K1</b>	<b>K5</b>	<b>Kn</b>	<b>Kmissing</b>	<b>K1 Acc.</b>	<b>K5 Acc.</b>	<b>MRR</b>
Baseline	902	977	17	481	0.61153	0.66237	0.63451
Baseline+100	905	987	7	481	0.61356	0.66915	0.63830
Baseline+270	920	988	6	481	0.62373	0.66983	0.64580

Table 21 - TQDM Library Results for Baseline, Baseline+100, and Baseline+270 Models

<b>Model</b>	<b>K1</b>	<b>K5</b>	<b>Kn</b>	<b>Kmissing</b>	<b>K1 Acc.</b>	<b>K5 Acc.</b>	<b>MRR</b>
Baseline+100	7.42%	11.25%	266.67%	-8.75%	7.42%	11.25%	9.07%
v Baseline							
Baseline+270	18.95%	20.84%	166.67%	-15.43%	18.95%	20.84%	20.90%
v Baseline							

---

Baseline+270	10.73%	8.62%	-27.27%	-7.32%	10.73%	8.62%	10.11%
--------------	--------	-------	---------	--------	--------	-------	--------

---

v

Baseline+100

---

Table 22 – Yaml Library Change % for Baseline, Baseline+100, and Baseline+270 Models

---

The results for tqdm show improvement using more input data, but with a higher starting prediction quality for tqdm combined with less diversity of code usage, means that finding new prediction patterns are harder for this library. The improvements from more data do exist, but not with the same significance as seen with markdown.

#### yaml Library

The yaml library in the baseline model had a total of 1,475 identified method calls in the test data set, and a poor prediction quality with a K1 and K5 accuracy of 20.21% and 20.21% respectively. The addition of 100 yaml repositories to the baseline model resulted in no change to any of the measurements.

The addition of 270 yaml repositories yielded a significant improvement over the baseline and baseline+100 models with K1, K5, Kmissing, and MRR improving by 34.21%, 78.95%, 24.00%, and 52.67% respectively. The baseline+270 model improved the ability to find missing predictions, which resulted directly in improved K1 and K5 results. The K>5 metric increased instead of reducing because of missing predictions not being in the K1 or K5 prediction lists. The result is that the MRR improved by 52.67%.



The raw metrics are in table 23 below and comparative percentages for the models are visible in table 24 below.

<b>Model</b>	<b>K1</b>	<b>K5</b>	<b>Kn</b>	<b>Kmissing</b>	<b>K1 Acc.</b>	<b>K5 Acc.</b>	<b>MRR</b>
Baseline	114	114	0	450	0.20213	0.20213	0.20213
Baseline+100	114	114	0	450	0.20213	0.20213	0.20213
Baseline+270	153	204	18	342	0.27128	0.36170	0.30860

Table 23 - Yaml Library Results for Baseline, Baseline+100, and Baseline+270 Models

<b>Model</b>	<b>K1</b>	<b>K5</b>	<b>Kn</b>	<b>Kmissing</b>	<b>K1 Acc.</b>	<b>K5 Acc.</b>	<b>MRR</b>
Baseline+100 v Baseline	0.00%	0.00%	N/A	0.00%	0.00%	0.00%	0.00%
Baseline+270 v Baseline	34.21%	78.95%	N/A	-24.00%	34.21%	78.95%	52.67%
Baseline+270 v Baseline+100	34.21%	78.95%	N/A	-24.00%	34.21%	78.95%	52.67%

Table 24 – Yaml Library Change % for Baseline, Baseline+100, and Baseline+270 Models

The results for yaml show significant improvement using more input data. However, it required more input data than markdown to show improvement. Further, starting with a

lower initial prediction quality allowed for a more significant improvement than markdown or tqdm.

### Results Analysis

The results with 270 additional repositories showed improvements at various levels of significance. Libraries with a higher prediction quality, such as tqdm with an MRR of 0.63451, have less room for improvement, so a less significant improvement was expected. Similarly, libraries with lower prediction quality, such as yaml with an MRR of 0.20213, have more room for improvement, so a more significant improvement was expected. The markdown library had a prediction quality between the other two, with an MRR of 0.41015, and it showed improvement in both models using additional libraries. Increasing the expansion size to 270 repositories yielded better results for all three libraries. While the three did not have the same level of improvement, there was an inverse correlation between the lower the initial quality and a higher improvement in the quality of the predictions. A deeper analysis of this correlation is left to future research.

### Impact Beyond Target Module

The expansion of additional repositories for a specific target module impacts the predictions of the model for the specific target module, but the additional repositories bring in other modules than the specific target module. The additional repositories must not degrade the overall performance of the model for the non-target modules to maintain the expected performance of the generated model.

The Pythia paper provides three overall metrics when comparing against other models, K1 Accuracy, K5 Accuracy, and MRR. However, it does not provide the details of the input or test data sets that would allow for these overall metrics to be replicated using the same technique. To account for this insufficient information, this dissertation selected the 10 modules that were clearly used by Pythia in Svyatkovskiy's paper and added 31 more modules. These 41 modules were used in all tests against all models. For each module in each test suite, the K1 Accuracy, K5 Accuracy, Kn Count, and K-missing Count, and MRR were calculated. The sum of all method calls in the test suite is 3,747,192. This total method count is constant for all models tested with the other measures being variable by model. The sum of K1, K5, Kn, and K-missing are each calculated along with its percentage against the constant total. Finally, the average MRR is calculated for each model across all 41 modules. These metrics allow comparing the results of each model against the baseline and each other.

The tables 25 and 26 below show the raw counts, MRR, and percentages for the overall results for the baseline and the 10 derivative models that were tested. While only markdown, tqdm, and yaml were used to measure the impact of expanding a specific module, other modules were run through the same process to ensure that the overall baseline remained stable independent of the module selected.

<b>Module</b>	<b>K1 Calls</b>	<b>K5 Calls</b>	<b>Kn</b>	<b>Kmissing</b>	<b>MRR</b>
Baseline	2,772,862	3,464,843	249,932	32,417	0.70453
markdown-100	2,763,246	3,460,939	254,261	31,992	0.70719
markdown-270	2,782,032	3,467,626	249,261	30,305	0.71785
tensorflow.python.ops. array_ops-270	2,782,032	3,467,626	249,261	30,305	0.71785
numpy.random-270	2,793,479	3,473,222	243,785	30,185	0.70911
tqdm-100	2,768,410	3,462,466	252,593	32,133	0.70342
tqdm-270	2,774,217	3,463,070	252,470	31,652	0.70510
yaml-100	2,762,635	3,461,007	253,955	32,230	0.70314
yaml-270	2,775,664	3,466,599	249,027	31,566	0.70969
flask-270	2,777,943	3,467,538	249,540	30,114	0.72756
pytest-270	2,764,001	3,463,086	253,646	30,460	0.71162

---

Table 25 - Overall Raw Counts and MRR for Baseline and 10 Derivative Models

<b>Module</b>	<b>K1%</b>	<b>K5%</b>	<b>Kn%</b>	<b>Kmissing%</b>
Baseline	74.00	92.47	6.67	0.87
markdown-100	73.74	92.36	6.79	0.85
markdown-270	74.24	92.54	6.65	0.81
tensorflow.python.ops. array_ops-270	74.24	92.54	6.65	0.81
numpy.random-270	74.55	92.69	6.51	0.81
tqdm-100	73.88	92.40	6.74	0.86
tqdm-270	74.03	92.42	6.74	0.84
yaml-100	73.73	92.36	6.78	0.86
yaml-270	74.07	92.51	6.65	0.84
flask-270	74.13	92.54	6.66	0.80
pytest-270	73.76	92.42	6.77	0.81

Table 26 - Overall Percentages for Baseline and 10 Derivative Models

The data shows that across the 10 variants of the baseline model that were tested, the overall metrics remained relatively stable. The K1% changed slightly for all variants, but within a small range of the baseline. Only 3 modules degraded K1%, and they were no more than 0.37% away from the baseline. Similarly, 7 modules improved their K1%, but no more than 0.74%. Examining the K5% shows comparable results, but over a narrower range with 5 modules degrading and 5 modules improving. The 5 modules that degraded were within 0.11% of the baseline and the 5 modules that improved were within 0.24% of the baseline.

The changes were more noticeable in the  $K_n\%$  and  $K_{missing}\%$  metrics, but still small. However, the  $K_n\%$  is not a clear good or bad indicator since a reduction in  $K_n\%$  could mean an increase in  $K1\%$  or  $K5\%$  or it could mean an increase in  $K_{missing}\%$ . Similarly, an increase in  $K_n\%$  could mean a decrease in  $K1\%$  or  $K5\%$  or it could mean a decrease in  $K_{missing}\%$ . The more relevant number is  $K_{missing}\%$ , which decreased for all variants, with a maximum decrease of 7.1%. This decrease in  $K_{missing}\%$  means that all variants improved their ability to produce more predictions.

The analysis of the Mean Residual Rank (MRR) supports the overall data analysis of  $K1\%$ ,  $K5\%$ , and  $K_{missing}\%$ . The MRR only degraded for 2 variants, and both were within 0.20% of the baseline. However, the other 8 variants improved, with a maximum overall improvement of 3.27%. The two variants that degraded in MRR were expanded with 100 repositories, and both of those showed positive improvements when 270 repositories were used.

These overall metrics show that the additional repositories had a minor impact overall on the  $K1\%$ ,  $K5\%$ , and MRR accuracy but did help to improve the number of predictions found. The additional repositories targeted a specific subset of the modules included, which represents a subset of the overall model data. Since the additional repositories bring with it unpredictable and effectively random additional modules, the overall minor

impact is the expected result. Figure 16 shows the distribution of the 3,747,192 method calls across K5, Kn, and Kmissing for all model variants.

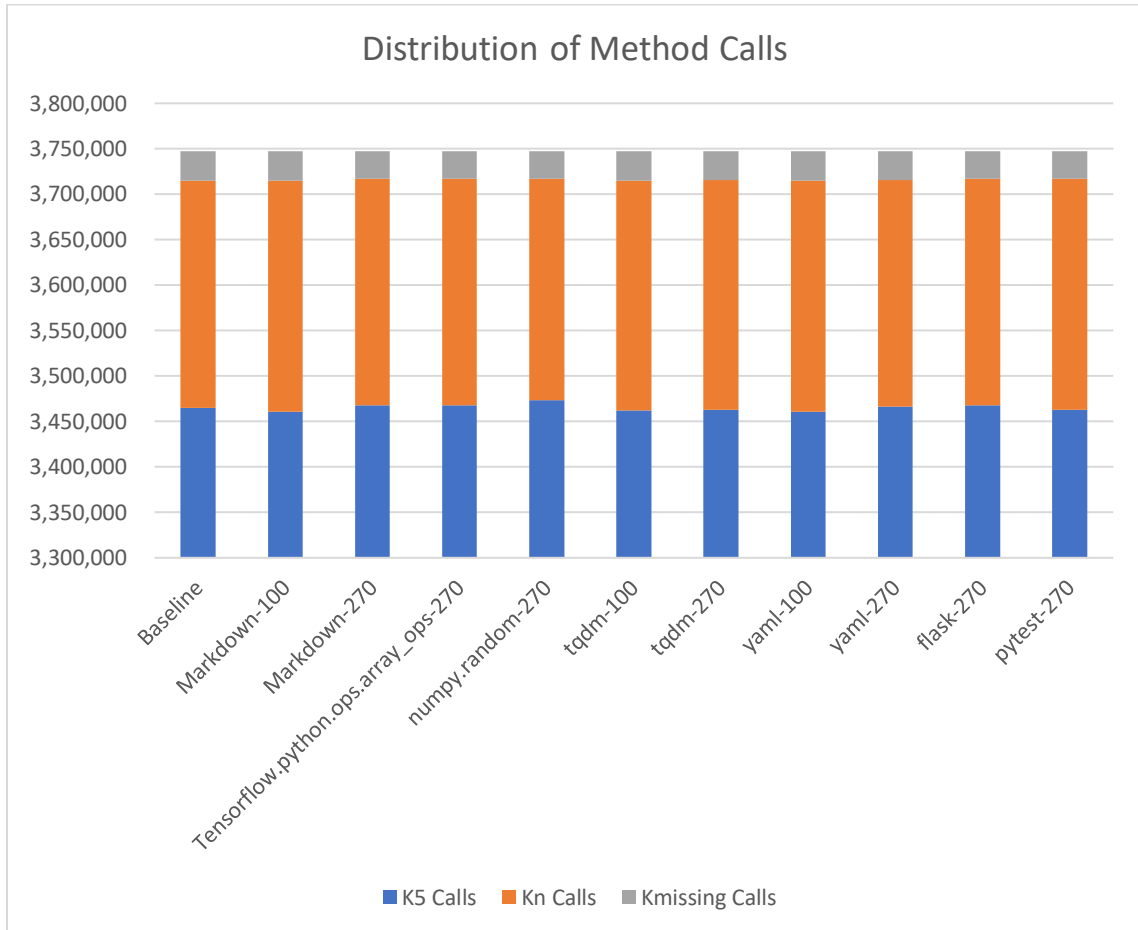


Figure 16 - Distribution of Method Calls

## Chapter 5

### Conclusions, Implications, Recommendations, and Summary

#### Conclusions

Analysis of the results clearly shows improvement of predictions with additional repositories containing a specified target library, while maintaining the same overall predictive quality across the entire test suite. Figure 16 above shows the stability of predictive quality across 10 variant models, Figures 17 and 18 below show the improvement specific to the markdown module, Figures 19 and 20 below show the improvement specific to the tqdm module, and Figures 21 and 22 below show the improvement specific to the yaml module.

The Accuracy by Model charts show the percentage of accuracy for K1, K5, and MRR metrics for the different models. These charts show the change in accuracy from the baseline model to the baseline with 100 additional repositories model, to the baseline with 270 additional repositories model. The higher the result, the higher the accuracy of the prediction. The Missing Predictions by Model charts show changes in the count of missing predictions between the same models. The lower the value, the less predictions missing from the model.



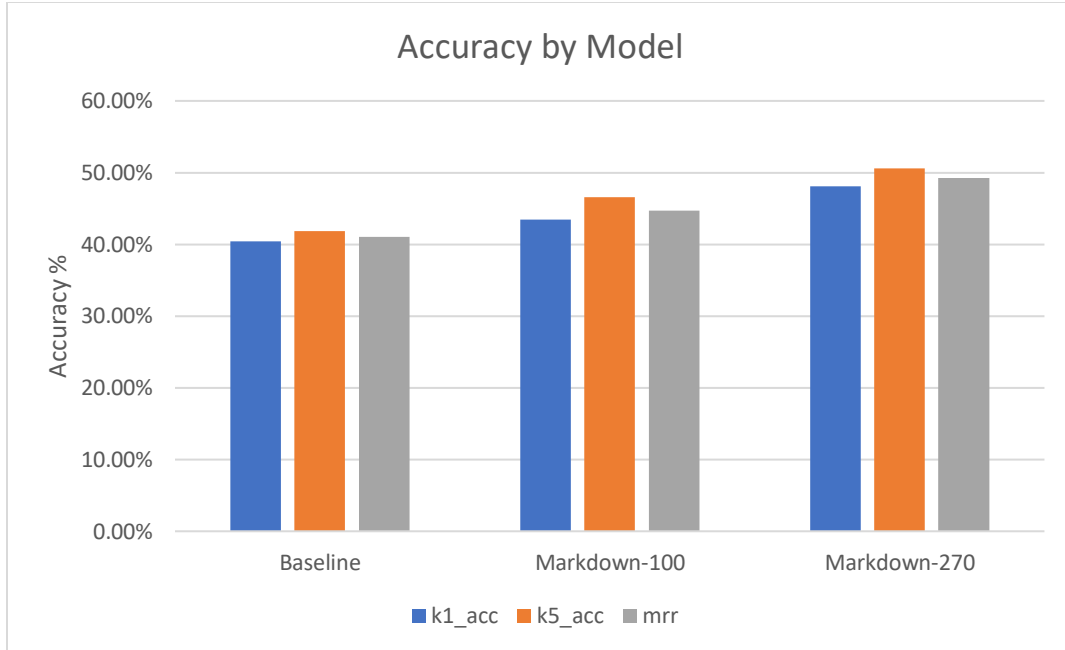


Figure 17 - Markdown Accuracy for Baseline, Baseline+100, and Baseline+270

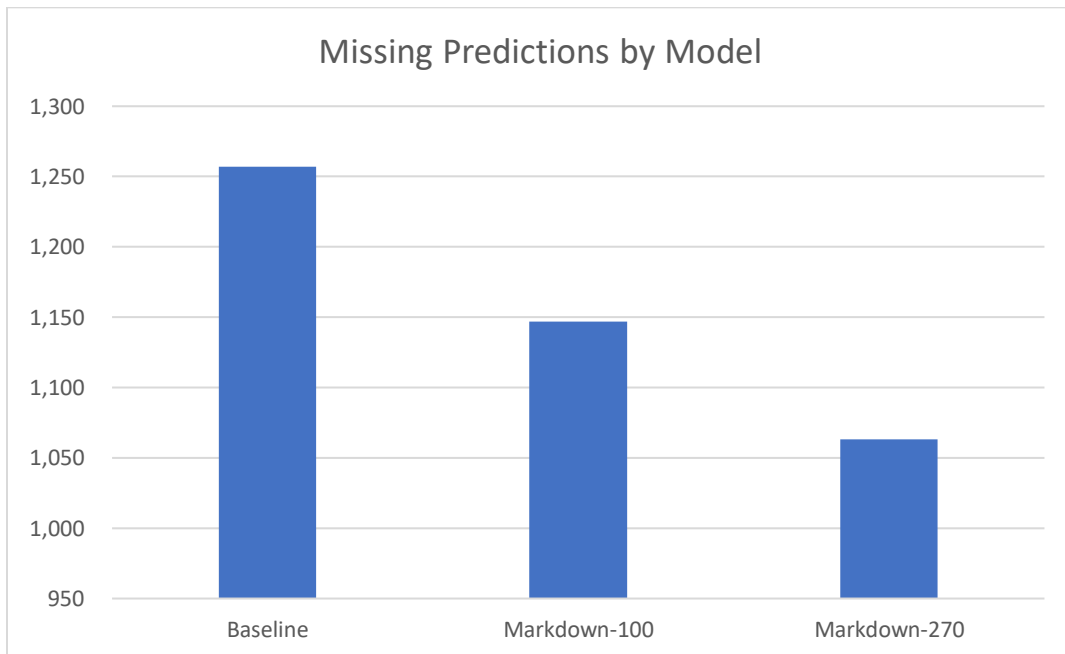


Figure 18 - Missing Markdown Predictions for Baseline, Baseline+100, and Baseline+270

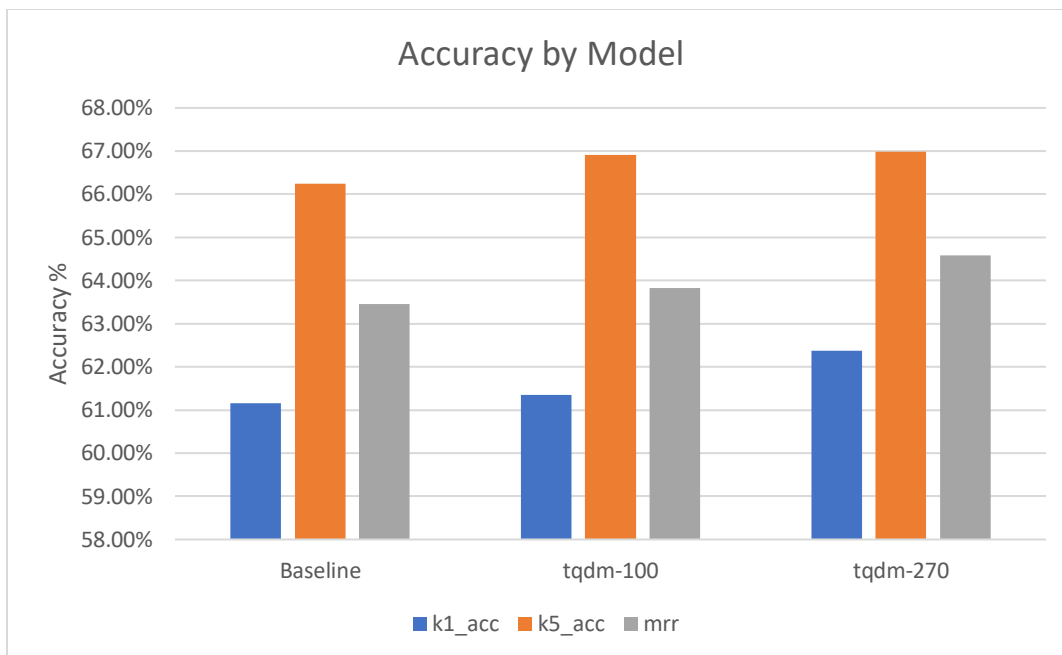


Figure 19 - tqdm Accuracy for Baseline, Baseline+100, and Baseline+270

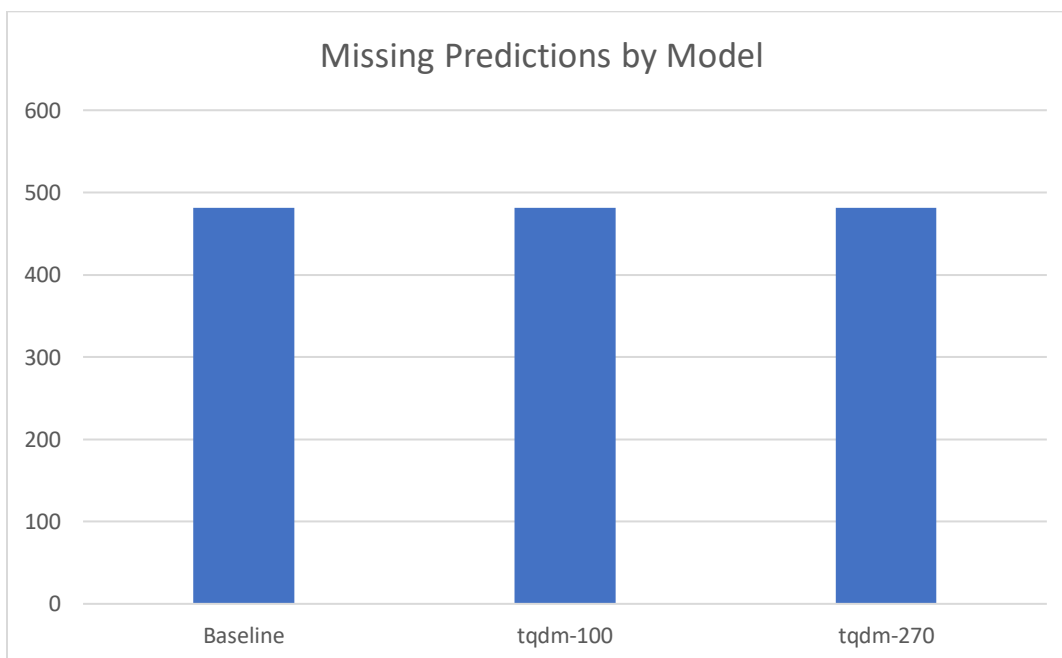


Figure 20 - Missing tqdm Predictions for Baseline, Baseline+100, and Baseline+270

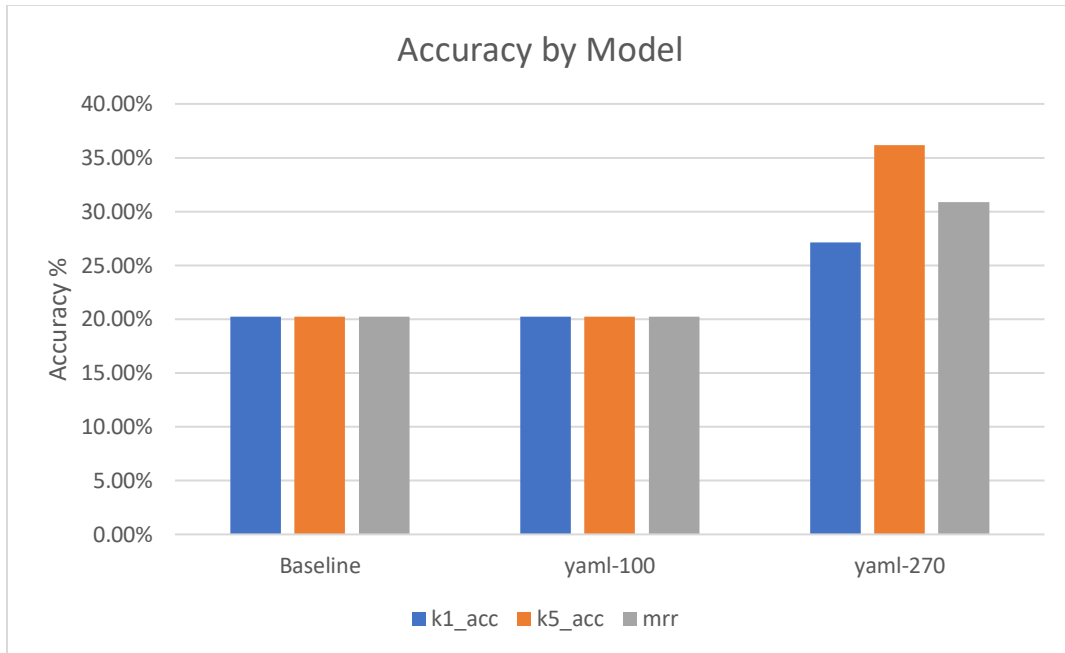


Figure 21 - yaml Accuracy for Baseline, Baseline+100, and Baseline+270

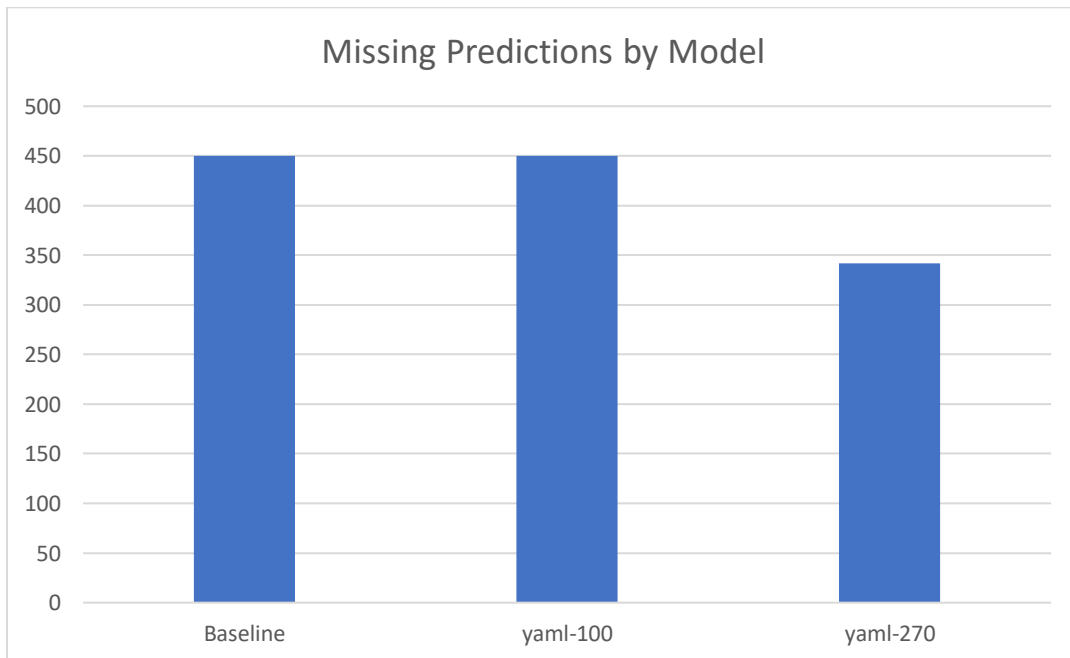


Figure 22 - Missing yaml Predictions for Baseline, Baseline+100, and Baseline+270

These results prove the hypothesis that increasing the input quantity of repositories containing specific third-party Python modules increased the predictive quality of the Pythia model. Furthermore, the results show that this increase in predictive quality results in no loss of overall predictive quality of the variant models.

### Implications

This dissertation demonstrated the ability to increase the accuracy of code prediction for a specific third-party Python module. This concept should be applicable to internal Python modules and other languages beyond Python. The ability to create targeted models for specific use cases could improve the accuracy of predictions in focused development efforts with highly specialized third-party libraries, such as game development, finance, statistics, and LSTM model development.

Instead of a single general purpose predictive model being used for all use cases, a targeted model could be generated by use case. While the model generation time is not short enough to enable real or near-time, a specialized model is achievable within hours using consumer grade computing resources locally and on the cloud.

### Recommendations

The parsing function narrowly focused specific elements of the Python code and did not account for more complex constructs, such as polymorphism and specialized Python methods. Further, this dissertation leveraged Gensim for tokenization and type identification. Both could be addressed with a more specialized approach to the Python parser, as was done by Svyatkovskiy for Pythia.

During the analysis of the data for this dissertation, a correlation between the number of exposed methods in a given module to the accuracy was noticed. Specifically, when the module had a lower number of exposed methods, a higher accuracy was achieved using less unique usages in the training set. This correlation was observed, is intuitive, but not proven, and would be worthy of exploration in future research.

The dynamic nature of available open-source code available in online hosted repositories combined with the webhooks available from similar repository hosts, such as GitHub, provide an opportunity for automated model generation. Specifically, repositories for specific languages could be monitored in a repository host, and when enough repositories have been added or updated, these repositories could be cloned and used as additional input content for model generation. Further, older repositories that have grown stale could be eventually expunged from the input data set. This would allow for the prediction models to reflect active code usage patterns and language versions.

## Summary

This dissertation shows that by adding additional repositories containing a specific module into the Pythia input dataset, it is possible to increase the accuracy of the third-party method calls within the specific module without negatively impacting the other predictions provided by the Pythia model. While increasing the repository count by 3.7% resulting in some positive results, a 10% increase yielded more consistent improvements across multiple third-party Python modules. The prediction improvement was demonstrated across 3 different Python third-party modules using 6 different model variants. The lack of negative impact to predictive quality was demonstrated using 10 different model variants.

## Appendices

## Appendix A – Hyperparameters

Hyperparameter	Description	Value Used
LookbackTokens	Number of lookback tokens	100
GensimWorkers	Gensim Worker Threads	15
WindowSize	word2vec window size	5
EmbeddingVectorDimensions	word2vec embedded vector dimensions	150
EmbeddingEpochs	word2vec epochs	10
MinimumFrequencyThreshold	Minimum frequency of a token	50
MinimumLabelFrequencyThreshold	Minimum frequency of a method call	25
MinimumModuleFrequencyThreshold	Minimum frequency of a Python module	500
TokenPadding	Token used for padding	<PAD>
TokenUnknown	Token used for unknown tokens	<UNK>
TokenEOS	Token used for end of sentence	.

Figure 23 - Gensim word2vec Hyperparameters

Hyperparameter	Description	Value Used
LookbackTokens	Number of lookback tokens	100
BatchSize	Batch sample size	512
HiddenDimensions	Number of LSTM hidden dimensions	256
LearningRate	Learning rate for LSTM	0.0001
LRWeightDecay	Decay rate per epoch	0.97
ClipNorm	Clipping rate	3
LSTMEpochs	Number of LSTM epochs	20
Dropout	Dropout rate for individual LSTM units	0.4
L2Regularizer	LSTM L2 regularizer	0.0001
LRWarmupEpochs	Number of warmup epochs	0
LossMethod	LSTM loss function	sparse_categorical_crossentropy
Activation	LSTM activation method	softmax
Optimizer	LSTM optimization method	Adam

---

Figure 24 - LSTM Hyperparameters



## Appendix B – Experiment Result Data

module_name	count	k1	k5	kn	k_missing	mrr
numpy	962,356	575,828	811,728	149,603	1,025	0.70719
os	440,614	390,136	429,917	9,297	1,400	0.92554
list	412,277	347,572	402,037	8,302	1,938	0.90275
str	393,713	327,275	381,164	10,744	1,805	0.89528
posixpath	292,597	212,912	285,803	6,730	64	0.83346
tensorflow	201,677	121,076	168,920	31,068	1,689	0.70626
dict	178,616	135,055	169,185	8,026	1,405	0.84095
sys	142,761	115,432	139,118	3,025	618	0.88147
re	94,908	65,983	91,871	2,899	138	0.80840
logging	67,673	49,796	65,635	1,585	453	0.83169
numpy.random	67,418	45,281	62,852	4,521	45	0.78067
unittest	60,321	58,351	59,981	213	127	0.97990
google.protobuf.descriptor	57,798	57,535	57,751	9	38	0.99704
time	54,017	46,458	52,761	777	479	0.91084
datetime	52,695	36,086	50,292	1,899	504	0.79893
json	45,274	37,573	45,010	46	218	0.90833
pandas	43,799	33,079	41,345	1,790	664	0.83785
tensorflow.python.ops.array_ops	21,939	10,294	17,206	4,496	237	0.60823
tensorflow.python.framework.ops	21,509	18,401	21,192	87	230	0.91200
pandas._testing	19,087	14,981	18,423	193	471	0.86389
numpy.testing	17,458	11,698	17,117	201	140	0.79290
six	16,609	14,397	16,300	152	157	0.91819
ui_mainwindow	14,907	1,263	2,391	1,255	11,261	0.12043
jax.numpy	12,551	5,741	9,065	2,256	1,230	0.57521
os.path	11,498	9,336	11,267	100	131	0.88422
requests	11,289	8,655	10,940	195	154	0.85736
emitter	10,142	8,453	8,965	13	1,164	0.85726
keras	5,241	4,604	5,124	59	58	0.92290
jax	4,247	2,847	3,739	188	320	0.76006
scipy	4,004	3,289	3,726	88	190	0.86941
dataframe	2,876	1,289	1,718	93	1,065	0.51543
markdown	2,167	876	907	3	1,257	0.41015
tqdm	1475	902	977	17	481	0.63451
yaml	564	114	114	0	450	0.20213
flask	454	70	70	0	384	0.15419
beautifulsoup	337	85	88	0	249	0.25668
werkzeug	133	69	73	2	58	0.52986
djangocache	80	42	42	0	38	0.52500

pytest	61	28	29	0	32	0.46449
sqlalchemy	50	0	0	0	50	0.00001

Figure 25 - Baseline Data

module_name	count	k1	k5	kn	k_missing	mrr
numpy	962,356	570,721	808,643	152,734	979	0.70282
os	440,614	389,157	429,975	9,239	1,400	0.92428
list	412,277	348,418	402,326	8,035	1,916	0.90425
str	393,713	326,316	381,177	10,731	1,805	0.89416
posixpath	292,597	212,228	285,812	6,721	64	0.83193
tensorflow	201,677	120,450	168,148	31,840	1,689	0.70365
dict	178,616	134,587	168,881	8,354	1,381	0.83894
sys	142,761	115,232	139,116	3,051	594	0.88097
re	94,908	66,220	91,940	2,830	138	0.80942
logging	67,673	49,384	65,599	1,621	453	0.82806
numpy.random	67,418	45,185	62,986	4,387	45	0.78056
unittest	60,321	58,167	59,982	212	127	0.97804
google.protobuf.descriptor	57,798	57,536	57,749	11	38	0.99714
time	54,017	46,449	52,796	742	479	0.91116
datetime	52,695	35,418	50,148	2,043	504	0.79098
json	45,274	37,523	45,001	55	218	0.90769
pandas	43,799	32,903	41,216	1,974	609	0.83471
tensorflow.python.ops.array_ops	21,939	10,010	17,095	4,607	237	0.59778
tensorflow.python.framework.ops	21,509	18,212	21,196	83	230	0.90761
pandas._testing	19,087	15,036	18,392	224	471	0.86515
numpy.testing	17,458	12,066	17,136	182	140	0.80662
six	16,609	14,458	16,301	193	115	0.91996
ui_mainwindow	14,907	1,223	2,390	1,256	11,261	0.11779
jax.numpy	12,551	5,726	9,031	2,290	1,230	0.57191
os.path	11,498	9,232	11,228	139	131	0.87887
requests	11,289	8,745	10,986	194	109	0.86370
emitter	10,142	8,510	8,993	9	1,140	0.86128
keras	5,241	4,586	5,119	64	58	0.92020
jax	4,247	2,831	3,728	199	320	0.75697
scipy	4,004	3,221	3,708	106	190	0.85695
dataframe	2,876	1,214	1,709	102	1,065	0.49820
markdown	2,167	941	1,009	11	1,147	0.44733
tqdm	1475	917	979	15	481	0.64103
yaml	564	110	114	0	450	0.19708
flask	454	78	85	0	369	0.17952
beautifulsoup	337	84	88	0	249	0.25520

werkzeug	133	82	86	7	40	0.63357
djangocache	80	42	42	0	38	0.52500
pytest	61	28	29	0	32	0.46722
sqlalchemy	50	0	0	0	50	0.00001

Figure 26 - Markdown-100 Variant

module_name	count	k1	k5	kn	k_missing	mrr
numpy	962,356	579,634	812,792	148,723	841	0.71007
os	440,614	390,304	430,203	9,160	1,251	0.92608
list	412,277	348,600	402,152	8,286	1,839	0.90421
str	393,713	328,251	381,516	10,435	1,762	0.89708
posixpath	292,597	214,001	286,163	6,370	64	0.83563
tensorflow	201,677	121,232	168,461	31,551	1,665	0.70593
dict	178,616	136,051	169,496	7,784	1,336	0.84442
sys	142,761	116,107	139,221	2,946	594	0.88475
re	94,908	66,409	91,827	2,951	130	0.81042
logging	67,673	49,574	65,532	1,705	436	0.82955
numpy.random	67,418	45,416	63,046	4,327	45	0.78216
unittest	60,321	58,378	59,964	230	127	0.97984
google.protobuf.descriptor	57,798	57,563	57,747	13	38	0.99746
time	54,017	46,287	52,741	797	479	0.90896
datetime	52,695	36,153	50,276	1,934	485	0.80018
json	45,274	37,421	45,013	43	218	0.90651
pandas	43,799	33,062	41,395	1,921	483	0.83826
tensorflow.python.ops.array_ops	21,939	9,980	17,095	4,607	237	0.59788
tensorflow.python.framework.ops	21,509	18,091	21,156	123	230	0.90392
pandas._testing	19,087	14,906	18,378	238	471	0.86083
numpy.testing	17,458	11,777	17,174	168	116	0.79651
six	16,609	14,402	16,314	180	115	0.91866
ui_mainwindow	14,907	1,244	2,380	1,288	11,239	0.11837
jax.numpy	12,551	5,480	8,816	2,505	1,230	0.55334
os.path	11,498	9,344	11,243	124	131	0.88519
requests	11,289	8,719	10,952	228	109	0.86147
emitter	10,142	9,413	9,716	21	405	0.94189
keras	5,241	4,575	5,109	74	58	0.91776
jax	4,247	2,805	3,708	219	320	0.75049
scipy	4,004	3,225	3,702	112	190	0.85706
dataframe	2,876	1,148	1,679	132	1,065	0.48243
markdown	2,167	1,042	1,096	8	1,063	0.49256
tqdm	1475	897	970	24	481	0.63162
yaml	564	153	194	28	342	0.30478

flask	454	84	85	0	369	0.18613
beautifulsoup	337	150	156	0	181	0.45352
werkzeug	133	83	87	6	40	0.63757
djangocache	80	42	42	0	38	0.52500
pytest	61	29	29	0	32	0.47541
sqlalchemy	50	0	0	0	50	0.00001

Figure 27 - Markdown-270 Variant

module_name	count	k1	k5	kn	k_missing	mrr
numpy	962,356	572,735	809,899	151,432	1,025	0.70454
os	440,614	388,782	429,835	9,402	1,377	0.92369
list	412,277	348,679	402,153	8,186	1,938	0.90406
str	393,713	326,943	381,104	10,804	1,805	0.89478
posixpath	292,597	213,138	285,778	6,755	64	0.83359
tensorflow	201,677	120,740	168,731	31,357	1,589	0.70557
dict	178,616	135,661	169,515	7,696	1,405	0.84331
sys	142,761	115,703	139,025	3,118	618	0.88244
re	94,908	65,700	91,742	3,028	138	0.80548
logging	67,673	49,536	65,556	1,664	453	0.82976
numpy.random	67,418	45,603	62,812	4,561	45	0.78269
unittest	60,321	58,348	59,982	212	127	0.97983
google.protobuf.descriptor	57,798	57,520	57,738	22	38	0.99699
time	54,017	46,332	52,770	768	479	0.90995
datetime	52,695	35,946	50,280	1,911	504	0.79730
json	45,274	37,342	45,011	45	218	0.90600
pandas	43,799	32,950	41,275	1,903	621	0.83569
tensorflow.python.ops.array_ops	21,939	9,830	17,069	4,633	237	0.59311
tensorflow.python.framework.ops	21,509	18,321	21,178	101	230	0.91015
pandas._testing	19,087	15,013	18,401	215	471	0.86443
numpy.testing	17,458	11,538	17,139	179	140	0.78872
six	16,609	14,419	16,311	141	157	0.91765
ui_mainwindow	14,907	1,245	2,383	1,263	11,261	0.11918
jax.numpy	12,551	5,624	8,989	2,398	1,164	0.56708
os.path	11,498	9,382	11,270	97	131	0.88747
requests	11,289	8,624	10,936	211	142	0.85658
emitter	10,142	8,509	8,956	22	1,164	0.85951
keras	5,241	4,626	5,109	74	58	0.92344
jax	4,247	2,869	3,761	206	280	0.76573
scipy	4,004	3,349	3,736	78	190	0.87937
dataframe	2,876	1,228	1,717	94	1,065	0.50194
markdown	2,167	868	905	5	1,257	0.40836

tqdm	1475	905	987	7	481	0.63830
yaml	564	110	110	4	450	0.19600
flask	454	70	70	0	384	0.15419
beautifulsoup	337	86	88	0	249	0.25817
werkzeug	133	65	74	1	58	0.51116
djangocache	80	42	42	0	38	0.52500
pytest	61	29	29	0	32	0.47541
sqlalchemy	50	0	0	0	50	0.00001

Figure 28 - tqdm-100 Variant

module_name	count	k1	k5	kn	k_missing	mrr
numpy	962,356	574,582	810,056	151,299	1,001	0.70586
os	440,614	389,210	429,723	9,514	1,377	0.92422
list	412,277	349,107	402,375	7,990	1,912	0.90499
str	393,713	327,723	381,354	10,572	1,787	0.89605
posixpath	292,597	212,514	285,771	6,762	64	0.83263
tensorflow	201,677	121,560	168,858	31,448	1,371	0.70785
dict	178,616	135,722	169,075	8,173	1,368	0.84275
sys	142,761	115,370	139,135	3,008	618	0.88086
re	94,908	65,941	91,700	3,070	138	0.80742
logging	67,673	49,707	65,579	1,641	453	0.83053
numpy.random	67,418	45,695	62,937	4,436	45	0.78405
unittest	60,321	58,374	60,009	185	127	0.98018
google.protobuf.descriptor	57,798	57,493	57,751	9	38	0.99674
time	54,017	46,693	52,749	789	479	0.91336
datetime	52,695	36,314	50,211	1,980	504	0.80124
json	45,274	37,870	45,021	35	218	0.91174
pandas	43,799	33,593	41,358	1,844	597	0.84399
tensorflow.python.ops.array_ops	21,939	10,062	17,354	4,348	237	0.60451
tensorflow.python.framework.ops	21,509	18,463	21,170	114	225	0.91376
pandas._testing	19,087	14,981	18,401	215	471	0.86352
numpy.testing	17,458	11,727	17,146	172	140	0.79461
six	16,609	14,401	16,305	147	157	0.91776
ui_mainwindow	14,907	1,321	2,392	1,254	11,261	0.12290
jax.numpy	12,551	5,404	8,798	2,589	1,164	0.55085
os.path	11,498	9,252	11,252	126	120	0.88103
requests	11,289	8,722	10,954	193	142	0.86106
emitter	10,142	8,412	8,966	12	1,164	0.85425
keras	5,241	4,555	5,132	67	42	0.91922
jax	4,247	2,825	3,750	217	280	0.75916
scipy	4,004	3,243	3,715	99	190	0.86244

dataframe	2,876	1,148	1,681	130	1,065	0.48102
markdown	2,167	863	905	5	1,257	0.40659
tqdm	1475	920	988	6	481	0.64580
yaml	564	167	198	18	348	0.32100
flask	454	67	70	0	384	0.15015
beautifulsoup	337	82	88	0	249	0.25223
werkzeug	133	68	73	2	58	0.53052
djangocache	80	42	42	0	38	0.52500
pytest	61	24	28	1	32	0.42214
sqlalchemy	50	0	0	0	50	0.00001

Figure 29 - tqdm-270 Variant

module_name	count	k1	k5	kn	k_missing	mrr
numpy	962,356	571,319	809,310	152,021	1,025	0.70329
os	440,614	389,267	429,997	9,217	1,400	0.92448
list	412,277	346,815	401,612	8,749	1,916	0.90113
str	393,713	326,705	381,067	10,841	1,805	0.89441
posixpath	292,597	213,427	285,980	6,553	64	0.83481
tensorflow	201,677	120,775	168,380	31,655	1,642	0.70445
dict	178,616	133,425	168,710	8,523	1,383	0.83480
sys	142,761	116,023	139,326	2,841	594	0.88481
re	94,908	65,753	91,875	2,895	138	0.80567
logging	67,673	49,721	65,668	1,552	453	0.83113
numpy.random	67,418	43,871	62,672	4,701	45	0.76690
unittest	60,321	58,385	59,985	209	127	0.98008
google.protobuf.descriptor	57,798	57,562	57,742	18	38	0.99737
time	54,017	46,475	52,785	753	479	0.91130
datetime	52,695	36,043	50,253	1,938	504	0.79856
json	45,274	37,608	45,011	45	218	0.90891
pandas	43,799	33,242	41,358	1,825	616	0.84002
tensorflow.python.ops.array_ops	21,939	9,671	17,128	4,574	237	0.59185
tensorflow.python.framework.ops	21,509	18,059	21,191	88	230	0.90303
pandas._testing	19,087	15,003	18,420	196	471	0.86435
numpy.testing	17,458	11,597	17,124	194	140	0.79001
six	16,609	14,420	16,315	137	157	0.91840
ui_mainwindow	14,907	1,259	2,427	1,219	11,261	0.11986
jax.numpy	12,551	5,621	8,927	2,394	1,230	0.56445
os.path	11,498	9,286	11,248	119	131	0.88137
requests	11,289	8,604	10,960	175	154	0.85622
emitter	10,142	8,620	8,990	12	1,140	0.86725
keras	5,241	4,526	5,093	90	58	0.91299

jax	4,247	2,798	3,694	233	320	0.75015
scipy	4,004	3,282	3,714	100	190	0.86668
dataframe	2,876	1,286	1,745	66	1,065	0.51724
markdown	2,167	870	905	5	1,257	0.40806
tqdm	1475	904	978	16	481	0.63641
yaml	564	114	114	0	450	0.20213
flask	454	70	70	0	384	0.15419
beautifulsoup	337	88	88	0	249	0.26113
werkzeug	133	70	74	1	58	0.53733
djangocache	80	42	42	0	38	0.52500
pytest	61	29	29	0	32	0.47541
sqlalchemy	50	0	0	0	50	0.00001

Figure 30 - yaml-100 Variant

module_name	count	k1	k5	kn	k_missing	mrr
numpy	962,356	575,680	811,451	149,880	1,025	0.70688
os	440,614	390,268	430,143	9,159	1,312	0.92593
list	412,277	349,909	402,572	7,809	1,896	0.90634
str	393,713	327,625	381,834	10,074	1,805	0.89647
posixpath	292,597	212,660	285,920	6,613	64	0.83324
tensorflow	201,677	121,148	168,913	31,235	1,529	0.70660
dict	178,616	135,289	169,460	7,796	1,360	0.84267
sys	142,761	116,377	139,264	2,903	594	0.88591
re	94,908	66,191	91,755	3,035	118	0.80831
logging	67,673	49,900	65,699	1,541	433	0.83316
numpy.random	67,418	45,109	62,820	4,553	45	0.77947
unittest	60,321	58,421	59,974	220	127	0.98024
google.protobuf.descriptor	57,798	57,532	57,743	17	38	0.99705
time	54,017	46,498	52,754	784	479	0.91141
datetime	52,695	36,097	50,277	1,930	488	0.80013
json	45,274	37,693	45,019	37	218	0.91012
pandas	43,799	33,213	41,473	1,777	549	0.84089
tensorflow.python.ops.array_ops	21,939	9,786	17,233	4,469	237	0.59537
tensorflow.python.framework.ops	21,509	18,321	21,184	95	230	0.91058
pandas._testing	19,087	15,008	18,417	199	471	0.86473
numpy.testing	17,458	11,375	17,152	166	140	0.78431
six	16,609	14,445	16,320	153	136	0.91936
ui_mainwindow	14,907	1,244	2,434	1,212	11,261	0.11935
jax.numpy	12,551	5,375	8,892	2,429	1,230	0.55097
os.path	11,498	9,231	11,271	107	120	0.88006
requests	11,289	8,652	10,992	186	111	0.85869

emitter	10,142	8,601	8,978	24	1,140	0.86567
keras	5,241	4,571	5,114	69	58	0.91861
jax	4,247	2,770	3,660	267	320	0.74306
scipy	4,004	3,276	3,702	112	190	0.86334
dataframe	2,876	1,100	1,674	137	1,065	0.47100
markdown	2,167	901	950	6	1,211	0.42637
tqdm	1475	904	981	13	481	0.63570
yaml	564	153	204	18	342	0.30860
flask	454	70	70	0	384	0.15419
beautifulsoup	337	131	156	0	181	0.42335
werkzeug	133	69	73	2	58	0.52919
djangocache	80	42	42	0	38	0.52500
pytest	61	29	29	0	32	0.47541
sqlalchemy	50	0	0	0	50	0.00001

Figure 31 - yaml-270 Variant



## References

- “19. Adding to the Stdlib — Python Developer’s Guide.” Accessed October 21, 2020. <https://devguide.python.org/stdlibchanges/>.
- Asaduzzaman, Muhammad, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou. “A Simple, Efficient, Context-Sensitive Approach for Code Completion: A Simple, Efficient, Context-Sensitive Approach for Code Completion.” *Journal of Software: Evolution and Process* 28, no. 7 (July 2016): 512–41. <https://doi.org/10.1002/smr.1791>.
- “Boto · PyPI.” Accessed October 21, 2020. <https://pypi.org/project/boto/>.
- Bruch, Marcel, Martin Monperrus, and Mira Mezini. “Learning from Examples to Improve Code Completion Systems.” In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering on European Software Engineering Conference and Foundations of Software Engineering Symposium - ESEC/FSE '09*, 213. Amsterdam, The Netherlands: ACM Press, 2009. <https://doi.org/10.1145/1595696.1595728>.
- Bruch, Marcel, Thorsten Schäfer, and Mira Mezini. “On Evaluating Recommender Systems for API Usages.” In *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering - RSSE '08*, 16. Atlanta, Georgia: ACM Press, 2008. <https://doi.org/10.1145/1454247.1454254>.
- D’Souza, Andrea Renika, Di Yang, and Cristina V. Lopes. “Collective Intelligence for Smarter API Recommendations in Python.” *arXiv:1608.08736 [Cs]*, August 31, 2016. <http://arxiv.org/abs/1608.08736>.

“GitHub - Dspinellis/Unix-History-Repo at Research-V1-Snapshot-Development.” Accessed June 6, 2021. <https://github.com/dspinellis/unix-history-repo/tree/Research-V1-Snapshot-Development>.

Halter, Dave. “Davidhalter/Jedi.” Python, October 21, 2020. <https://github.com/davidhalter/jedi>.

He, Xincheng, Lei Xu, Xiangyu Zhang, Rui Hao, Yang Feng, and Baowen Xu. “PyART: Python API Recommendation in Real-Time.” *arXiv:2102.04706 [Cs]*, February 9, 2021. <http://arxiv.org/abs/2102.04706>.

Kaiser, Gail E., Peter H. Feiler, and Steven S. Popovich. “Intelligent Assistance for Software Development and Maintenance.” *IEEE Software* 5, no. 3 (1988): 40–49.

Li, Jian, Yue Wang, Michael R. Lyu, and Irwin King. “Code Completion with Neural Attention and Pointer Networks.” *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, July 2018, 4159–65. <https://doi.org/10.24963/ijcai.2018/578>.

“Linux in 2020: 27.8 Million Lines of Code in the Kernel, 1.3 Million in Systemd - Linux.Com.” Accessed June 6, 2021. <https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/>.

Mărășoiu, Mariana, Luke Church, and Alan Blackwell. *An Empirical Investigation of Code Completion Usage by Professional Software Developers*, 2015.

“Microsoft/PTVS.” C#. 2015. Reprint, Microsoft, October 21, 2020. <https://github.com/microsoft/PTVS>.

“Microsoft/Python-Language-Server.” C#. 2018. Reprint, Microsoft, October 21, 2020.

<https://github.com/microsoft/python-language-server>.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. “Efficient Estimation of Word Representations in Vector Space.” *arXiv:1301.3781 [Cs]*, September 6, 2013.

<http://arxiv.org/abs/1301.3781>.

“ML.NET | Machine Learning Made for .NET.” Accessed June 13, 2021.

<https://dotnet.microsoft.com/apps/machinelearning-ai/ml-dotnet>.

Murphy, Gail C. “Beyond Integrated Development Environments: Adding Context to Software Development.” In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 73–76. Montreal, QC, Canada: IEEE, 2019. <https://doi.org/10.1109/ICSE-NIER.2019.00027>.

Murphy, G.C., M. Kersten, and L. Findlater. “How Are Java Software Developers Using the Eclipse IDE?” *IEEE Software* 23, no. 4 (July 2006): 76–83.

<https://doi.org/10.1109/MS.2006.105>.

Nguyen, Anh Tuan, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N. Nguyen, and Danny Dig. “API Code Recommendation Using Statistical Learning from Fine-Grained Changes.” In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*, 511–22. Seattle, WA, USA: ACM Press, 2016. <https://doi.org/10.1145/2950290.2950333>.

“Numpy · PyPI.” Accessed October 20, 2020. <https://pypi.org/project/numpy/>.

Proksch, Sebastian, Johannes Lerch, and Mira Mezini. “Intelligent Code Completion with Bayesian Networks.” *ACM Transactions on Software Engineering and Methodology* 25, no. 1 (December 2, 2015): 1–31. <https://doi.org/10.1145/2744200>.

“PyCharm: The Python IDE for Professional Developers by JetBrains.” Accessed October 21, 2020. <https://www.jetbrains.com/pycharm/>.

“PyTorch.” Accessed June 13, 2021. <https://pytorch.org/>.

Robbes, Romain, and Michele Lanza. “How Program History Can Improve Code Completion.” In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 317–26. L’Aquila, Italy: IEEE, 2008. <https://doi.org/10.1109/ASE.2008.42>.

Schuster, Roei, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. “You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion.” *arXiv:2007.02220 [Cs]*, October 8, 2020. <http://arxiv.org/abs/2007.02220>.

“Search Results · PyPI.” Accessed October 21, 2020.

<https://pypi.org/search/?c=Topic+%3A%3A+Software+Development+%3A%3A+Libraries+%3A%3A+Python+Modules>.

Stylos, Jeffrey, and Steven Clarke. “Usability Implications of Requiring Parameters in Objects’ Constructors.” In *29th International Conference on Software Engineering (ICSE’07)*, 529–39. Minneapolis, MN: IEEE, 2007. <https://doi.org/10.1109/ICSE.2007.92>.

Svyatkovskiy, Alexey, Ying Zhao, Shengyu Fu, and Neel Sundaresan. “Pythia: AI-Assisted Code Completion System.” In *Proceedings of the 25th ACM SIGKDD International*

*Conference on Knowledge Discovery & Data Mining*, 2727–35. KDD '19. New York, NY, USA: Association for Computing Machinery, 2019.

<https://doi.org/10.1145/3292500.3330699>.

“The NumPy Open Source Project on Open Hub.” Accessed October 21, 2020.

<https://www.openhub.net/p/numpy>.

“The Python Standard Library — Python 3.9.0 Documentation.” Accessed October 21, 2020.

<https://docs.python.org/3/library/>.

“Welcome to Astroid’s Documentation! — Astroid 2.5.9.Dev10+g432aa99 Documentation.”

Accessed June 13, 2021. <http://pylint.pycqa.org/projects/astroid/en/latest/>.

Winograd, Terry. “Breaking the Complexity Barrier Again.” *ACM SIGIR Forum* 9, no. 3

(November 4, 1973): 13–30. <https://doi.org/10.1145/951761.951764>.