GRADUATE THESIS APPROVAL FORM AND SIGNATURE PAGE

Instructions: This form must be completed by all doctoral students with a thesis requirement. This form MUST be included as page 1 of your thesis via electronic submission to ProQuest.

Thesis Title: Applying Natural Language Processing Techniques to Code

Author's Name: Aviel Justice Stein

Submission Date: 06/16/2022

The signatures below certify that this thesis is complete and approved by the Examining Committee.

Role: Chair Name: Spiros Mancoridis
Title: Professor
Institution: Drexel
Approved: Yes Date: 06/22/2022

Role: Member Name: Ali Shokoufandeh
Title: Professor
Institution: Drexel
Approved: Yes Date: 07/06/2022

Role: Member Name: Jake Williams
Title: Associate Professor
Institution: Drexel
Approved: Yes Date: 06/27/2022

Role: Member Name: Edward Kim
Title: Associate Professor
Institution: Drexel
Approved: Yes Date: 06/20/2022

Role: Member Name: Rachel Greenstadt
Title: Associate Professor
Institution: NYU
Approved: Yes Date: 07/06/2022

Office of Graduate Studies

Dissertation / Thesis Approval Form

This form is for use by all doctoral and master's students with a dissertation/thesis requirement. Please print clearly as the library will bind a copy of this form with each copy of the dissertation/thesis. All doctoral dissertations must conform to university format requirements, which is the responsibility of the student and supervising professor. Students should obtain a copy of the Thesis Manual located on the library website.

Dissertation/Thesis Title: Applying Natural Language Processing Techniques to Code

Author: Aviel J. Stein

This dissertation/thesis is hereby accepted and approved.

Signatures:

Examining Committee

Chair _____

Members _____

_____

_____

_____

_____

Academic Advisor _____

Department Head _____

Office of Graduate Studies • 3141 Chestnut St. • Randell Hall 240 • Philadelphia, PA 19104
Tel.: 215-895-0366 • **Fax**: 215-895-0495 • **Email**: graduate@drexel.edu • **Web**: www.drexel.edu/provost/graduatestudies

Applying Natural Language Processing Techniques to Code

A Thesis

Submitted to the Faculty

of

Drexel University

by

Aviel J. Stein

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy

May 2022

Dedications

We give your name glory, for your mercy and truth.

Acknowledgments

I would like to express my deepest appreciation to my committee for their time, support, and generosity. To professor Spiros Mancoridis for his guidance and wisdom. To professor Rachel Greenstadt for her gracious insights. To professor Ali Shokoufandeh for his kindness and optimism. To professor Jake Williams for his linguistic acumen and helpful tips. To professor Edward Kim for his encouragement and good will.

I am also grateful for the insights and enthusiasm of my colleagues who helped inspire this work. This would not have been possible without my family and friends who cheered me on throughout this process. And I would like to thank Drexel for the community it fostered and its commitment to excellence.

This research was funded by the Auerbach Berger Chair in Cybersecurity held by Spiros Mancoridis

Table of Contents

List of Tables

List of Figures

Abstract

Applying Natural Language Processing Techniques to Code
Aviel J. Stein

Analyzing source code is becoming a helpful method as the amount of code and the number of coders increases. Developing software is a mentally strenuous activity because it involves understanding and formalizing solutions to complex problems. As solutions are adapted and the problem themselves change, the code changes with it. Traditionally, source code analysis (SCA) uses methods and tools that rely on the rigid structure of code to work, and while useful, are not always able to adapt to new information and code. By comparison, the natural language processing (NLP) field leverages naturally occurring patterns within natural language text to help people understand and interpret text robustly on a large scale. This works shows that methods from NLP can be applied to code to extract information automatically in broad and more adaptable ways. We show several examples of how techniques developed for NLP can be successfully applied to software to create source code tools that can help developers improve source code.

Chapter 1: Introduction

Language, whether spoken, written, or compiled, pervades our life. The last one is a relatively new way of using language and math with technology to create novel systems of communication. The creation of documents in the formal language of source code is a large driving factor in our economy. In addition to providing the means by which we communicate, computer science and data science give us principled methods for understanding data and acting on information. Advances in technology and computing have paved the way for applying computationally intense analytic methods to the contents of the Internet. Of particular interest is the information we create "naturally", i.e., data created by humans for humans, mostly in the form of natural language. Natural Language Processing (NLP) is the science of mining human communication for patterns and using learning paradigms to train systems that imitate and interact in a natural manner. These methods use artificial intelligence to analyze, imitate, and ultimately help humans communicate better and more efficiently (e.g., Google translate, interactive voice response, etc.). Applications of NLP have been successfully used in many contexts, due largely to their ability to handle a multitude of unstructured data. It is worth exploring how well these techniques can be applied to other domains such as source code. As a formal language, programming source code can be tested for its ability to perform specific functions correctly and efficiently, with or without the need for human evaluators. In addition to having linguistic qualities, there is ample source code data accessible on the Internet.

1.1 Motivation

Good code comes from developers who can use and write useful programs. The best software often comes from collaborative efforts between programmers and practitioners from diverse backgrounds. What they have in common is the ability to understand and communicate the problem and its solution. The study of how we communicate (linguistics) has recently been explored with a computational approach. This, in hand with the massive amount of text data available has resulted in the

explosive growth of the NLP field. Code is written in programming languages, which are types of formal languages. This begs the question, how well do the techniques developed in NLP work for code? In this work we demonstrate the efficacy of applying NLP approaches to formal language in the form of source code and the techniques can be used for source code analysis (SCA) tasks. We have curated source code data sets from GitHub to accomplish baseline measurements and train models that can be used by programmers. We use a combination of NLP, SCA, and Artificial Intelligence (AI) to identify factors in code such as annotation in the form of commented code and semantic cohesion.

Formal languages (FL) are inspired by natural language (NL) and are formed and used in similar ways. NL is constructed using words and letters (symbols), grammar (syntax), and meaning (semantics). By studying how these linguistic structures are used, we hope to uncover patterns that are useful for understanding programs. FL are a specialization of NL and avoid the ambiguities of NL to convey precise meaning. Source code is the primary example of a FL, although others exist such as chemical formulas. Code has many of the same linguistic characteristics found in NL, including a grammar that defines a syntax, and a vocabulary of words (key words or variables). The primary difference is that FL is less ambiguous, and, in the case of code, it can be executed and can process data. Because code has discrete symbolic notation, is sequential, and is often paired with NL, it can be studied with similar techniques as those used in NLP. To date, there has been some attention to code's similarities to NL[2–4], and even some who go as far to say that "natural language is a programming language."[5] This research has been effective, in part, because of the ability to store, share, and serve software solutions on the web.[6;7] Tools built to analyze these new data corpora can be created from NLP-inspired techniques.

Code offers several advantages over natural language. While many natural language tasks must be evaluated or tweaked by hand, with indefinite precision, code can be assessed more concretely and objectively. If we have a sufficient test set, human evaluation becomes less critical. Also, qualities and behavior beyond correct functionality include speed of execution, memory efficiency, and robustness to bad input. The SCA field is interested in many of these questions. In particular, SCA focuses on

the correctness and optimization of software. Optimization can be in terms of execution performance, resource usage, or how quickly a developer can understand the code. Correctness refers to code being structured according to the rules of the language and computing correct answers. Robustness is the ability to handle bad input. Despite SCA employing Machine Learning (ML) and Deep Learning (DL) techniques, current methods and tools have two limitations; firstly, they are often treated as black boxes making them difficult to evaluate outside custom data sets, and, secondly, they do not consider natural patterns, which, as we have seen from NLP, can provide effective answers despite ambiguities and irregularities. Evaluating code quality via SCA is valuable, nevertheless. We can transfer lessons learned from NLP to FL. This offers a new approach to current SCA tasks and creates new SCA opportunities. As open-source projects and the abundance of NL and FL continue to become well developed and available on the Internet, ML and DL become well suited to leverage this data. FL has been less explored from a linguistic-data point of view. Applications in this area can lower the barrier to entry for understanding the purpose and mechanisms of code. The goal for our research is to close the contextual gap between the automated analysis of formal and natural language.

1.2 Thesis Outline

We use NLP as our blueprint for studying large repositories of source code. One key aspect of NLP that makes it suitable for this work is that it offers principled methods for handing multiple ambiguous sources of information. Code is often paired with natural language and similar pieces of code can be used for various tasks. While NLP is a large field with many facets, its tasks can broadly be categorized into two types: analysis and transformation. Analysis tasks typically try to understand some object of interest, usually text, and determine emotion or meaning, for example. Transformation involves shifting the context of the text. For example, a translation algorithm shifts the context between different languages. Analysis tasks score very well with current AI tools. However, transformation tasks are harder and, thus, present more opportunity for improvement. We start by comparing how well analysis for attribution works for code and then focus on how to process and transform code with NLP tools.

Chapter 2 explores analyzing text and code for attributes of interest. Text and code may have both soft and technical attributes such as the language, author, topic, word count, etc. We show how to process the style and topic of text on news articles. After showing that signals for both can be present in one text, we show that you can categorize code documentation based on tutorial topic and source code by author and behavior. These methods show a range of robust results across several generic model and feature types, even with language agnostics models. This suggests the NLP approach can be used easily and effectively for analyzing code and may be transferable to new domains. Text attribution can be used for security and quality purposes on code, such as detecting malware or determining the run time of an algorithm. Moreover, it shows that simple NLP techniques can process code in interesting and meaningful ways.

Chapter 3 discusses paraphrasing and how it can be used to modify and augment code. It is a text transformation technique that retains the semantic essence of the text, but also creates a new formation of words that conveys a similar meaning in the original language. With code, applications include creating code transformation tools that could be used for conforming with code style guidelines.[8;9]

Chapter 4 describes text segmentation as it deals with the tokenization of text data into identifiable groups. This involves identifying units of interest such as letters, words, morphemes, sentences, intents, or topics inside a document. Topic segmentation is of particular interest but is difficult to annotate because it can be ambiguous and difficult to evaluate. As to how this method applies to code and formal language, it can be used to segment code into parts, such as functions or related files, or associating documentation with the related code.

Chapter 5 discusses text translation as the process of converting text from one language into another while retaining the meaning of the original text. There is not always a unique answer, because human translators may choose different words to convey the same meaning. With the application to formal language, translation becomes even more interesting. For example, one could translate between C++ and Python or translate between FL and NL, such as generating comments that can be inserted into code, or generating code functions from NL descriptions. The latter task is

much harder and commonly results in code that does not function properly or has incorrect syntax. An advantage of FL is that it can be compiled to verify that it is well-formed and executed using a test suite to verify the correctness of the code.

Chapter 6 covers the use of tagging for identifying relationships between words in the text. In the case of NL, this is usually done for information retrieval such as Part of Speech (POS)[10] tagging and Named-Entity Recognition (NER)[11] which are both NLP tasks which chunk and label areas of interest. We also use dependency parsing to evaluate the relationship between nodes of the abstract syntax tree (AST). These methods are done for information retrieval and feature enhancing purposes. For example finding articles related to a specific event or parsing input for question answering. In terms of the structure of the text, dependency parsing identifies the relationships between words, their part of speech, and the use in the sentence. With code we have lexical and syntactic information. We explore ways of inferring these attributes in a similar manner to identify the lexical chunks within the code and reconstruct the AST of a program as a robust and high fidelity feature extraction technique. This will be helpful in enhancing the techniques we have already covered as the field matures and helps us perform analysis on incorrect and incomplete code.

Human achievement rests largely on our ability to communicate and collaborate with one another via language. With the advent of modern communication and network technology we communicate more, and faster, than ever before. However, with so much information, we need new ways of managing the volume and velocity of the data we are expected to process. NLP brings together linguistics and data science to provide tools for dealing with large and messy sets of linguistic data. State-of-the-art NLP tools can analyze and transform natural language at near-human ability. Our thesis is that similar techniques can be applied to formal languages such as source code. Formal language differs from natural language in that the rules for correctness are rigid and rarely change, its meaning is manifest by its behavior, and its modality directly operates on data or produces data. In this work, we explore how analytic and transformative methods, with origins in NLP, are affected when applied to code.

Table 1.1: In some cases we are able to reuse NLP methods directly or create analogous features

Approach	Reuse	Analogous features
Attribution	ML models, NLP tokenizer/featurizer	Multi-lingual, behavioral
Paraphrasing	Edit distance, parallel corpora	Code transformations, test suite validation
Segmentation	LSTM, SentincePiece	Comment localization, code tokenization
Translation	Neural machine translation, attention,	Stop words, related tokens
Tagging	Dependency parsing, part of speech	Conll for code, syntax, lexical

1.3 Assumptions

Our goal in this work is to leverage the natural elements of source code in conjunction with the limitations and advantages of its formal nature. We draw analogies between NLP tasks to software use cases to help developers automate tedious tasks and aid in decision making for more complex ones. We may reuse methods directly, or modify techniques and create analogous features (see Table 1.1).

For attributing qualities of a program, stylometry had been shown to be very effective on source code and extends well to other qualities such as using behavior as analogous to topic. We want to be able to identify qualities of the code that are interesting to us. A standard English tokenizer and common NLP features showed similar results to formal methods. We chose to use paraphrasing to try to generate new altered code that retained the behavior. The NLP literature uses parallel corpora to find different text talking about the same thing. We find similarities in the code using an edit distance. So this analogous task can go one step further and verify that the behavior is retained with a test suite. This helps us find ways to perturb the code from what could be considered common phrases.

Since programmers often use written documentation of code to help their understanding we consider how might that information be processed or generated. In order to extract high level meaning from code, a programmer might decide what pieces of code belong together and write comments and summarize with a header file. In order to segment source code by its contextual components we identified where we would expect to see comments. We create analogous features

by fitting an NLP tokenizer to limit the vocabulary size and retain meaningful chunks and trained a sequential model that identified contextual breaks in those tokens. For generating annotations for the code we decided that automatic translation was a promising analogous task. We were able to use many translation models but applying them directly to code is more difficult because comments are not necessarily direct translations of code. We used analogous features by checking the comment for stop words and compared similar tokens between them.

Even natural language has formal elements that help indicate specific meanings in uses of words. While we can get very far implicitly learning this structure, methods for inferring the structure of text has helped improve advanced NLP methods. We assume that programmers initially are able to get an idea about what is going on in the program by reading it similarly to how we read natural language. Of particular importance are the lexical tokens and syntactic relationships. We speculated that part of speech tags are a similar paradigm to the lexical token and also conclude that learning relationships between words would work as an effective parser.

1.4 Research Contributions

This dissertation demonstrates that methods used for natural language processing can be applied to solve source code analysis tasks, such as:

- Method for authorship and behavior attribution for Python, C++, and Java source code with the use of plaintext tokenizers for simple and efficient feature extraction.
- Method for discovering functionally equivalent source code transformations in C++ using techniques for paraphrases identification.
- Method for delineating between cohesive portions of Python, C++, and Java source code with the use of contextual elements as a way of selecting good places to insert comments.
- Method for automatically creating informative English annotations (e.g., comments) for blocks of Python source code using a translation approach.
- Method for determining the lexical and syntactic structure of Python source code, even if the

source code is incomplete or incorrect (i.e., under construction).

Chapter 2: Attribution: Author, Topic, and Behavior

2.1 Introduction

The Internet is replete with text that may contain irrelevant or unhelpful information, therefore, a means of processing and distilling content is important and useful to human readers as well as information extracting tools. Natural Language Processing (NLP) techniques use statistical and computation-driven methods to analyze large bodies of text. Some common questions we may want to answer are "what is this article about?" and "who wrote it?". In this chapter we compare ML models for evaluating two common NLP attribution tasks, identifying author and topic. We start with news articles from the 2017 Vox Media data set as it represents a common form of media. We then extend the topic modeling approch to the more niche form of software tutorials from ProjectHub and Hackseter.io. Finally, we apply the methods directly to source code and find that both authorship and code behavior are highly attributable.

Section 2.2 describes existing work around news articles, topic, and authorship using NLP. Two attributes of text that NLP scientist commonly want to label are the authorship (stylometry) or the topic of the text. The authorship signal is useful for detecting plagiarism and fake accounts, among other things. Topic classification can be helpful for sorting or searching a data set. For news topic classification, we use the 2017 Vox Media[12], which is an understudied data set that has advantages over other contemporary news article data sets in terms of the number of articles as well as labeled topics and authors. We then use the technique in the domain of technical programming jargon by apply it to online coding tutorials. Finally, we extend it directly to source code programs from Google Code Jam (GCJ)[13] and use code behavior in place of natural language topic. Additionally, Most studies only explore either authorship attribution or topic classification but not both. One advantage of this work is that it explores both tasks side-by-side in the same context, and, thus, shows that they can be solved using comparable techniques for natural language text and source code. These NLP techniques are helpful for many academic and industrial applications. Because

contexts may differ, it is important to have baselines and reusable data sets to compare results or build models for transfer learning.

Section 2.3 describes the methods used to extract features and classify text. Text classification generally relies on ML to provide high accuracy results when applied to large data sources. For our authorship attribution and topic classification, we extracted several types of features such as word n-gram, term frequency inverse document frequency (TFIDF), and POS features, but found that n-gram word count resulted in the best performance. We also use various common ML models to compare their effectiveness for this task (see section 2.3.3).

Section 2.4 describes the experiments demonstrating NLP efficacy for Vox articles. After performing the attribution, we inspected our models by creating a confusion matrix and conducting a feature analysis, to understand how the classification may be affected by a confluence of signals. Generally, though, authors tend to write about the same topics as they have in the past. For the 10-class data set we attained 74% accuracy for topic attribution and 86% accuracy for author attribution. Finally, Section 2.5 describes the limitations of these approaches, discusses the implications of our work, and suggests ways that future research can improve upon them.

2.2 Background

There are many aspects of text that can be attributed beyond topics for example, classifying news based on bias[14] and credibility[15] or fake news[16]. One approach classified news articles based on their source and attributed Fox, Vox, or PBS with at best 94% accuracy, but is it because of the text's style or the content signal?[17] The approach used by Yirey et al.[18] focuses on distinguishing between articles on Finance, Stocks, Education, and Environment and scores around 81% with support vector machine (SVM) and with a similar number of articles per topic. However, one drawback was that the data set had to be well balanced. Another use of topic analysis is tracking topics that a user may be interested in and can help suggest future articles for the user to read.[19] This process has to do two things, 1) track articles a user reads, and 2) identify the topics within articles. Topics of past articles will likely be similar to topics in future articles. An open question is whether a user likes articles written by certain sources, authors, or if they are affected by other factors. A related

problem involves attributing individual authorship to documents. While this is not strictly an NLP task, as it has also been applied to other things where authorship is relevant such as art [20], music [21], source code authorship [22], etc., it is most prevalent with text. Authorship attribution can be used to determine if someone plagiarized or helped preserve the anonymity of the author. Researchers performed DL authorship attribution on a data set of 10 authors and lengthy articles achieving 95% accuracy, and, on shorter articles, 77% accuracy. [23] However, in the case of a news organization, they may have tens or hundreds of authors, so it may not be as robust in those circumstances. In less edited and smaller text portions, whiteprints scored 95% accuracy on eBay comments. [24] The level of professional editing could influence how much style is present. Another difference between this and other text corpora, is that the authors of articles likely pass their work through editors, and also likely have a style guide. This may create an organizational signal or a decreased authorship signal.

2.3 Methods

This section describes the data, feature extraction, and ML classifying models used for our results. We chose to use balanced data sets (i.e., those with approximately the same number of samples per class) for the sake of visualizations, though the results remain about the same with natural distributions. We used common strategies for feature extraction including term frequency inverse document frequency (TFIDF) and n-gram. Additionally, we compared different ML algorithms to see how they performed under a variety of conditions.

2.3.1 Data Preprocessing

We handle the skewed data by constructing two curated subsections of the data containing a balanced number of articles per class (i.e., author or topic). One contains 10 classes, each with 300 articles, the other contains 50 classes, each with 50 articles. Having the data set balanced is useful for dissecting the results in the confusion matrices (Figures 3-4), but does not significantly improve overall accuracy. We also chose to ignore topics such as “Life”, “Identities”, and “The Latest” because we found that they tend to act as a miscellaneous category for Vox instead of focusing on

a topic. We also filtered out the topics "Xpress" and "Vox Sentence" which tend to have very short articles, which makes them unsuitable for this task in addition to often being vague. This data set has many favorable features such as being well curated for ML, including author and topic labels, and including inductive summary, which most other data sets such as 20 news organization do not have.

2.3.2 Features

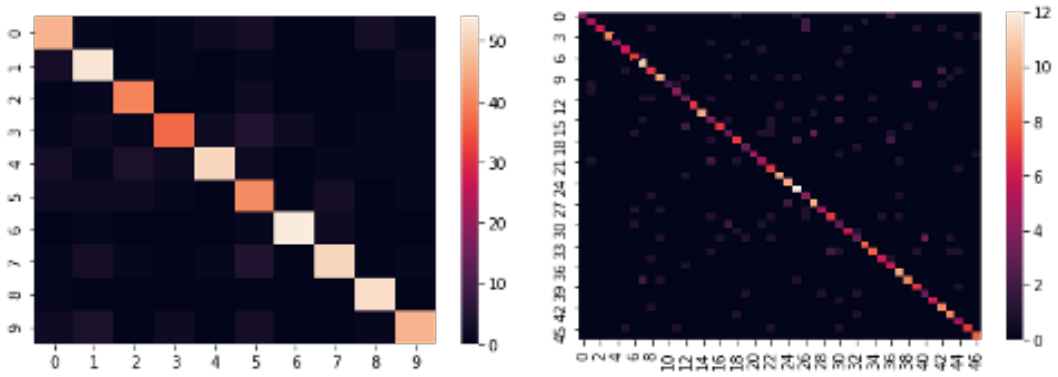
ML models use features from the text to learn the class signatures. To this end, we extract three types of features. First, we use word count and word bi-gram count. We also use word and word bi-gram TFIDF. We also use Natural Language Tool Kit's (NLTK) built in TreebankWordTokenizer and tagger to do PoS. We limit the number of features in order to train the models more efficiently. We ignore features that are either very common or very rare as they are prone for overfitting, specifically by limiting features with term frequency between 0.01 and 0.99. We use the Random Forest (RF) model for feature importance evaluation. We also exclude all non-alphabetic characters besides spaces and periods. We exclude some features that are artifacts of the web embedding. Finally, we use the RF feature importance metric to look at what features are most important for distinguishing classes. Though we try various features types, we find that n-gram term frequency perform best, while also not causing overfitting and use the same feature construction parameters for both authorship and topic attribution.

2.3.3 Classifier Models

Discrete classification is a ML task with many classifiers readily available. We use several different learning algorithms and techniques as a comparative opportunity. We use Naïve Bayes (NB), Decision Trees (DT), RF, Support Vector machines (SV), and multi-layer perceptron neural networks (NN). These learning algorithms are supported by many open source libraries. We use a one vs. all (OvA) strategy with the NN to get slightly better results. They were built in Python with scikit-learn.

Table 2.1: Authorship attribution accuracy with various ML models with n-gram word counts features.

Model	Dense % Accuracy	Dense % Accuracy
Naïve Bayes	81	64
Decision Tree	53	30
Random Forest	74	51
Support Vector	74	32
Neural Netowrk	83	51
Neural Network One-vs-All	86	70



(a) **(b)**

Figure 2.1: Authorship confusion matrix for dense using NN_OvA model (a) and sparse using NN_OvA model (b).

2.4 Results

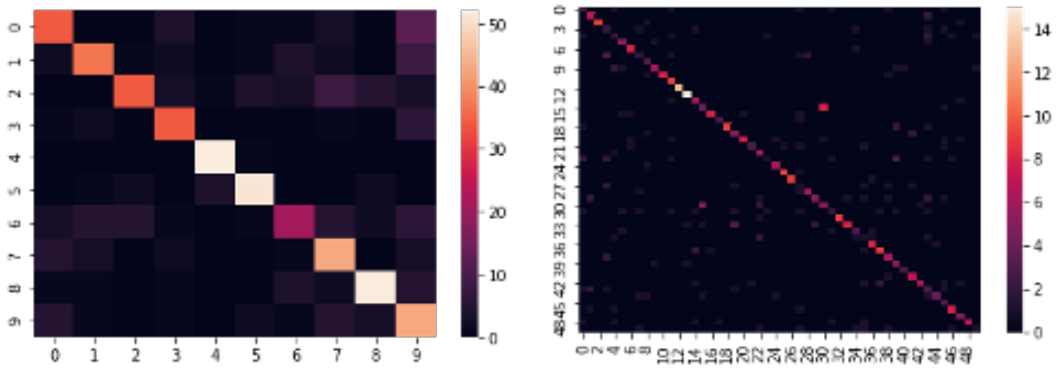
2.4.1 NLP Authorship vs. Topic Attribution

We start with stylometry. We can detect the style of an author statistically by doing the feature extraction as described previously. In the dense data set, we trained with articles of the top 10 most prolific authors of Vox. With the dense data set, we used 80% for training and saved 20% for testing, and attained up to 84% accuracy using a neural network with the OvA (NN_OvA); though the other methods also behave fairly well for this task. With the Sparse data set we have some loss in signal but still strong considering there we are classifying 50 authors with 70% accuracy, whereas the baseline for guessing is 2%.

Topics can be classified between 62%-74% accuracy. Therefore, given similar information, topics are 10% less accurate with dense information and 8% less accurate with sparse information. Looking

Table 2.2: Topic classification accuracy with various ML models on with n-gram word counts features.

Model	Dense % Accuracy	Dense % Accuracy
Naïve Bayes	73	61
Decision Tree	55	45
Random Forest	70	62
Support Vector	65	38
Neural Netowrk	72	53
Neural Network One-vs-All	74	62



(a) **(b)**

Figure 2.2: Topic confusion for dense using NN_OvA model (a) and sparse for dense using NN_OvA model (a).

at the confusion matrix of topics with dense information, it appears that one topic tends to dominate, and in the sparse data set there appears to be two that were misclassified as each other. Whereas in the authorship attribution case, the errors are more scattered. Additionally to compared this approach to other work[18] which use fewer numbers of topics with more articles to score above 90% accuracy in distinguishing between "Politics Policy", "Science Health", "Culture", and "Business Finance".

2.4.2 Tutorials

We apply the same techniques used in NLP to code related data in the form of tutorial text. We use the classification methods on both text and code in order to attribute topic, author and behavior. The open-source movement in technology has enabled hobbyists, researchers, and innovators to share and iterate on ideas in a decentralized and asynchronous way. The internet of things (IoT) is

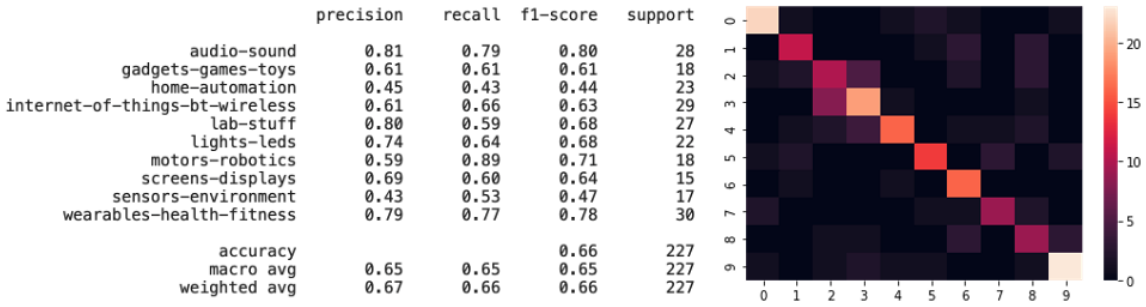
Table 2.3: Topic classification accuracy with various ML models on with n-gram word counts features for project hub.

Model	small % Acc	medium % Acc	large % Acc	Natural % Acc
Naïve Bayes	26	47	55	50
Random Forest	40	53	59	56
Support Vector	24	56	59	60
Neural Netowrk	40	54	57	53
One-vs-All	62	**66**	59	61

becoming more common place and becoming more accessible with technologies such as raspberry pi and Arduino. Two popular sites for sharing projects that build on these technologies are ProjectHub and Hacker.io. We also use the GCJ data set to attribute code authorship and function. We see that classifying project descriptions by topic achieves similar accuracy to classifying news articles by topic.

Arduino is an "Open-source electronic prototyping platform enabling users to create interactive electronic objects." Project hub is a site hosted by Arduino, which contains projects that use its technology for creative purposes. They have identified many different types of projects. These different types of projects represent a similarity to the topics in news articles, but instead of being articles about different themed events, they are instructions (often with stories) that describe why and how to build something for a specific task, from "Motors and Robotics" to "Wearables, Health, and Fitness". We scraped the article text from these topics (see Appendix A), although some have much more information than others. We created three balanced data-sets for small (197 samples), medium (907 samples), and large (2594 samples). We also made one extra-large data-set (4,315 samples) that contained more of a natural distribution of the topic types. We trained on 75% of the data and tested on 25%. We saw that we can train models that can get around 60% using the OVA technique for all sizes and even for non-balanced data. But even with plenty of data it is not able to separate more effectively. This may be because several of the topics are closely related (e.g., "Home Automation" and "Internet of Things", "Bluetooth" and "Wireless"). The choice of label is determined by the author of each article.

Hackster.io is another project hosting site with similar projects to Project Hub. One advantage



	precision	recall	f1-score	support
audio-sound	0.81	0.79	0.80	28
gadgets-games-toys	0.61	0.61	0.61	18
home-automation	0.45	0.43	0.44	23
internet-of-things-bt-wireless	0.61	0.66	0.63	29
lab-stuff	0.80	0.59	0.68	27
lights-leds	0.74	0.64	0.68	22
motors-robotics	0.59	0.89	0.71	18
screens-displays	0.69	0.60	0.64	15
sensors-environment	0.43	0.53	0.47	17
wearables-health-fitness	0.79	0.77	0.78	30
accuracy			0.66	227
macro avg	0.65	0.65	0.65	227
weighted avg	0.67	0.66	0.66	227

Figure 2.3: The SVM scored the highest of our methods for the project hub data so we used the one-vs-all approach which gave us a score of 66%.

of Hackster.io is that it has more topics and projects. This is because it does not focus exclusively on one hardware platform, like Arduino. In fact, one can search projects by product, which makes it handy almost like a cookbook with supplies. This is an additional feature we can use for classifying the project by its type. However, while not explored here, one may describe a project and predict what kind of products will be needed to complete it.

One unique feature of this data set is that each article may have several tags. It is advantageous in that it makes each article more distinct and provides more information. But it is disadvantageous because it makes classification more ambiguous. This could have a causal advantage where we assume every tag has a reason, but missing tags may not be wrong and could be used to learn what parts of the project a tag may be referring to. We choose to focus on top 5 topics: "Home Automation", "Internet of Things (IoT)", "Robotics", "Display", and "Environmental-Sensing". About 20% of the articles include more than one of the topics and each of these topics shares several articles with each of the other topics, and there are many that share several. To remedy this, we can ignore all articles that are in overlap areas. This leaves us with 453 samples for training and 152 samples for testing. There is some uncertainty with which we can say these categories capture the semantics content of the articles.

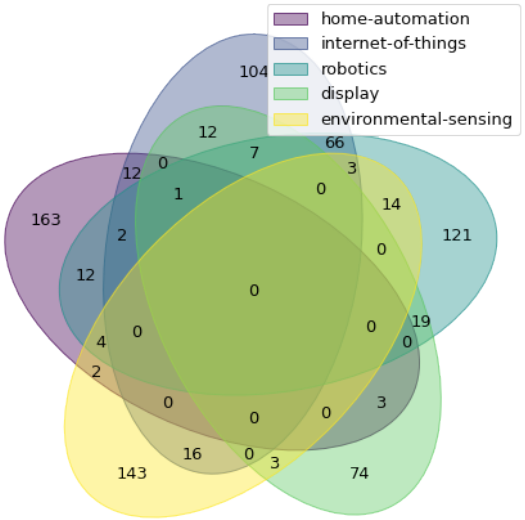


Figure 2.4: Venn-diagram with top 5 topics and their overlaps showing that there is significant overlap between all topics.

Table 2.4: Topic classification accuracy with various ML models with n-gram word counts features for hackser.io articles.

Model	Accuracy
Naïve Bayes	52
Decision Tree	72
Random Forest	73
Support Vector	61
Neural Netowrk	66
One-vs-All RF	75

2.4.3 Source Code Behavior and Style

In this section we look at how using generic and code-specific feature extraction techniques operates for these NLP tasks. We use the Google Code Jam (GCJ)[13] data set because it has labels for both code and author. The function in this case is similar to several people writing about something specific such as recipes for the same food or reviews of a movie. They have the same intent which is to communicate something specific. We use different segments of the data, focusing on C++, Java, Python, and Multi which includes all the previous languages. Detecting authorship signals by the style in which the code is written is a useful task and has been explored pretty widely[22]. It has both security and privacy implications. It can be helpful for tracking malicious actors who deploy malware

Table 2.5: Topic classification accuracy across 50 tasks, each with between 50-100 sample files with various ML models with n-gram word count features for different programming languages and mix of languages.

Model	Python % Acc	Java % Acc	C++ % Accuracy	Multi-lingual % Acc
Naïve Bayes	75	73	77	65
Random Forest	89	88	91	88
Support Vector	80	81	82	78
Neural Netowrk	87	87	89	86
One-vs-All	85	85	89	84

Table 2.6: Author classification accuracy across 50 tasks, each with between 50-100 sample files with various ML models with n-gram word count features for different programming languages and mix of languages.

Model	Python % Acc	Java % Acc	C++ % Accuracy	Multi-lingual % Acc
Naïve Bayes	87	91	89	82
Random Forest	99	99	99	98
Support Vector	94	96	96	93
Neural Netowrk	97	98	98	97
One-vs-All RF	99	99	98	97

but it can also be dangerous if the government is a bad actor. It also has implications for software copyright. Classifying by function is a unique quality of this data-set because all the programs perform the same function, pass a test-suite, and have many samples across several languages. Most of the time people use projects to do something different than the original purpose.

Both authorship and behavioral attribution do well when operating on similar data sets. Authorship does better than function attribution across the board as we saw with the natural language authors. However, Both methods for code do better than the cases of natural language. This is interesting because we use the same approach for both, but there is more information about both the style of the writer and the content being written about in the case of source code. This high fidelity of data could account for several things. One difference is that in the natural language all the topics were about different events, whereas the parallel here is that they were all discussing the same event. Even so, using these techniques shows that high quality models can be made for distinguishing between code that functions differently. Moreover, this is a task that would be difficult for non-expert humans to distinguish.

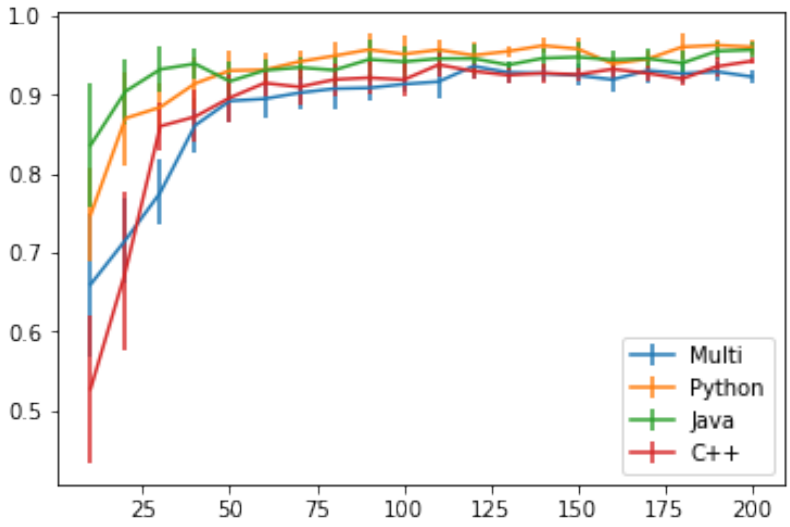


Figure 2.6: Topic classification accuracy on 10 tasks, the x-axis represents number of samples per topic and the y-axis is accuracy. We see that the quality of the model quickly increases and levels off between 90-95% and works well for all languages and multiple languages.

2.5 Discussion

For both NLP text and code the approach is the same and we can have some certainty of how well it will do depending on its situation. This may determine if you have enough data to create a good model. Another thing to consider about this data is that there are only a limited number of tasks per year, but each task has many options. In contrast to natural language, in code each author only has so many samples but there are many authors. This means that for interest we can test the robustness of the style signal across many authors and that we can extend the amount of samples for the task situation. Another interesting observation is that, for both natural text and code, authorship has a stronger signal per sample even though objectively the behavior is deterministic.

An avenue for future work in the topic space could go in several directions. For the multi-label hackster.io data, it could be useful to create a model that can identify what parts of the project relate to each tag. Or use the GCJ data with generative adversarial model to figure out out how to generate good code that retains the function of the original. The adversarial model would have to pass a test suite to prove the behavior remained the same. This type of model could be used to

create a large scale of pretrained weights to be used for transfer learning.

2.6 Conclusions

In this chapter we compare machine learning models and common NLP techniques on text and code. We find that authorship and topic can be distinguished on NLP text for news. We use the same technique on technology project descriptions and it performs similarly. There are some differences between the data sets which makes it a good place to learn about how NLP text relates to source code repositories. We also explore how to mine similar data from code itself to classify both on author and topic and find that both these methods perform as well or better than NLP classification.

Chapter 3: Paraphrasing: Semantic Code Transformations

Automatically identifying and generating equivalent semantic content to a word, phrase, or sentence is an important part of NLP. The research done so far in paraphrases in NLP focuses exclusively on textual data, but has significant potential if it is applied to formal languages like source code. In this Chapter, we present a novel technique for generating source code transformations via the use of paraphrases. We explore how to extract and validate source code paraphrases. The transformations can be used for stylometry tasks and processes like source code refactoring. A ML method that identifies valid transformations has the advantage of avoiding the generation of transformations by hand and is more likely to have more valid transformations. Our data set is comprised of 27,300 C++ source code files, consisting of 273 topics, each with 10 parallel files. This generates approximately 152,000 paraphrases. Of these paraphrases, 11% yield valid (compilable) code transformations. We then train a random forest classifier that can identify valid transformations with 83% accuracy. In this chapter we also discuss some of the observed relationships between linked paraphrase transformations. We depict the relationships that emerge between alternative equivalent code transformations in a graph formalism.

3.1 Introduction

People use paraphrasing to convey information they have heard or read using different words from those used by the original writer or speaker. We paraphrase for better clarity or concision or just to protect the identity of the writer or speaker. In Section 3.2.1 we discuss how this process has been explored in NLP and has been crucial to many applications such as information extraction, information retrieval, query and pattern expansion, summarization, question answering, machine translation, semantic parsing, etc.[25] NLP paraphrases have been shown to be more effective at retaining semantic correctness and imitating author style than other ML techniques including generative adversarial networks (GANS).

In Section 3.2.2, we discuss some of the differences between formal language and natural language. Natural languages have more redundancy and ambiguity, but share attributes with formal languages such as grammar, syntax, and words. Section 3.2.3 covers how ML approaches such as Random Forests, DL, GANS, attribution, and imitation have also been applied to computer source code effectively. However, there has not been significant research in source code paraphrasing, despite its potential. In this work, we explore source code paraphrasing by defining and generating source code paraphrases and exploring the potential of source code paraphrasing. We loosely define semantically equivalent code paraphrases as those that result in programs that pass exactly the same test cases as the original code. Source code paraphrasing can be used to simplify code, generate new ways to solve problems, create pseudo-code, translate between programming languages, and support authorship attribution. We posit that the source code transformations we generate can be used in tasks such as authorship obfuscation, code refactoring and optimization, and guideline enforcing.

In Section 3.3, we consider source code to be parallel if they are both solutions to the same problem and validated by a test suite. We use the Levenshtein edit distance[26] to measure the likelihood that two lines of code are paraphrases. Once we have found the candidate lines of source code, we verify that semantics were preserved for the task by comparing the outputs of a test suite that is assumed to exist for the original (non-paraphrased code).

In Section 3.4, our method employs the use of parallel corpora of source code that solve the same problem. The Google Code Jam challenge problems, which have been used in NLP for source code stylometry, also fit well for this inquiry. This paraphrase method results in the generation of approximately 4,500 candidate phrases and yields approximately 11% semantic-preserving code transformations. The time required to generate these transformations depends on the number of semantically valid phrases. We evaluate our method by training a Random Forest (RF) classifier to distinguish between valid and invalid lines of code. We discuss the impact this work may have on NLP and Formal Language Processing (FLP) in Section 3.5. There are various ways to judge the quality and use of our generated paraphrases and we recommend some. We also describe future work and its benefits. Section 3.6 concludes the chapter by presenting the important findings, namely

that paraphrase techniques work with formal languages to produce source code transformations and that models can be made to evaluate the quality of the transformations automatically.

3.2 Background

3.2.1 Natural Language Processing

The analysis of text is useful in the context of the Internet, where data-drenched forums abound. Many of the tasks trained and tested on in this space are of news and social media data. Language models are built in order to extract useful information from these resources.[27] Language models are systematic approaches in the development of useful features for ML algorithms.[28] The model can build relationships between words and phrases through an analogy method. For example, a model can identify pairs of text that may be similar. To determine the extent that two pairs overlap in their meaning, we present the following analogy. Clothing:shirt and dish:bowl is: "clothing is to shirt as dish is to bowl," and ask how valid such an anlogy is.[29] ML algorithms can use the features generated by these models to make predictions. Dolan et al present a study in which human judges with 448 sentence pairs generated by these algorithms believed that approximately 26% of these pairs were correct, 34% were partially correct and 40% were not related.[30] Stylometry explores the extent to which style (writing, music, art, etc) can be measured for attribution, imitation, obfuscation and other purposes. For example, you can build a language model for distinguishing authors. This task can be applied to many authors with high accuracy and has even been used to imitate and obfuscate their styles. Moreover, previous work used a sequence-to-sequence language model, inspired by machine translation, that was able to retain some of the semantics of the original text.[31] The generative adversarial network tries to imitate the style of another author while retaining the original semantics. While these methods can fool a ML model, they do not guarantee semantic equivalence that could be easily identified by a human judge.

3.2.2 Natural versus Formal Languages

Natural language has evolved over time and continues to do so. Even though there are tools for communication, confusion and ambiguity are sometimes impossible to avoid. Formal languages

are precise tools that we have created to solve these limitations. Formal languages have been developed and used by scientists who want to formulate specific rules about what they are trying to communicate. They have grammar, vocabulary, and syntax that are less ambiguous and redundant than natural language. There is still some debate about the specific qualities of formal languages. In 2018, the International Journal of Computer Mathematical Sciences published an article discussing the difference between formal languages and natural languages stating that formal languages do not have semantics. Thus in formal languages, importance is given to syntax only, not to the meaning of the sentence.[32] However, all source code is written in formal languages and the NLP methods that utilize the semantics of language can be used on them. What is important to take away from this section is that formal languages still have a well-defined structure, are defined using grammar, vocabulary in the form of tokens, and semantics as defined by the mapping of well-formed syntactic structures to executable machine instructions. In contrast to NLP, we will refer to methods applied to formal languages as Formal Language Processing (FLP).

3.2.3 Formal Language Processing

The overlap between NLP tasks and code analysis is increasing. Stylometry techniques generate similar results for both source code and textual data. Language-agnostic methods can apply fairly simple language analysis such as term frequency inverse document frequency (TF-IDF) to build a language model with the ability to perform accurate authorship.[33] Even binary code can contain traces of style. With source code, the method collects features from abstract syntax trees (ASTs), layout, and lexical features, and achieves a 94% accuracy when testing the style of 1,600 different authors[22]. Generating paraphrases can be used to change the features extracted, potentially obfuscating or imitating in an adversarial context. However, semantic preserving GANS have been able to reduce the attribution accuracy of traditional stylometry methods significantly.[31] They rely on 5 types of code transformation: Control, Declaration, API, Template, and Miscellaneous, using ASTs, control flow graphs, use-define chains, and declaration reference mappings for generating files. Their obfuscation reduced attribution accuracy to 1%. Code transformations are crucial for GANS because they rely on generating new data to try to misdirect the detective network and improve it.[34]

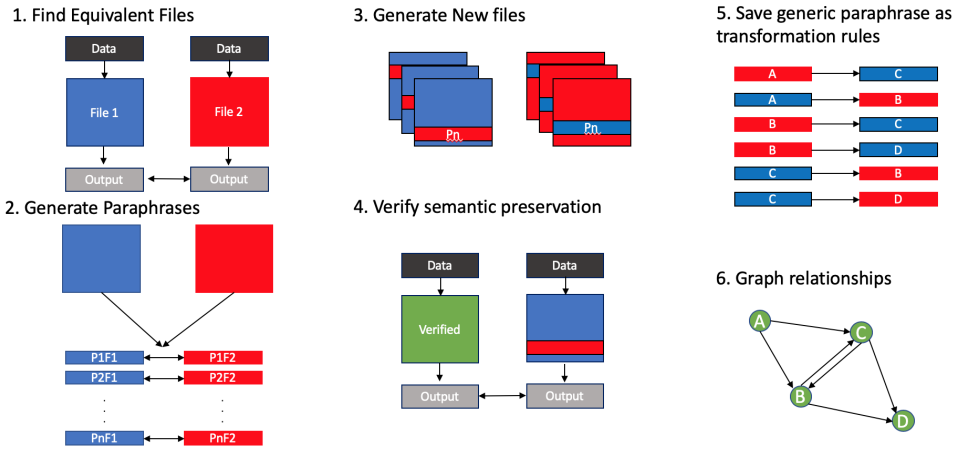


Figure 3.1: Our method of identifying, validating, and comparing paraphrases.

The generation of paraphrases has the potential to be used to change the features extracted. Given the success of Code Stylometry we are hopeful that paraphrasing can have an impact on privacy and code improvement as well.

3.3 Methods

We explore how readily paraphrasing techniques can be applied to formal languages. We build our technique by analogy to NLP approaches. These approaches vary by the type of corpus and goals for the results. To find semantic preserving code transformations, we use methods appropriate for a monolingual parallel corpora.

A survey of the most frequently used methods for generating phrasal and sentential paraphrases by NLP researchers in the past two decades categorizes approaches by corpus type.[22] Four techniques used on monolingual parallel corpora are presented, two of which rely on distributional similarity while the other two focus on more direct non-distributional methods.[35 36 25 37] The last approach uses statistical machine translation tools to generate novel paraphrases using different news articles from the web, which is the approach we built our method on. Quirk et al. use the edit distance (Levenshtein distance) to compare pairwise sentences and determine new paraphrases. They use word alignment algorithms to improve this process and measure results.

If the data set has several versions of the same text written in the same language, the corpus

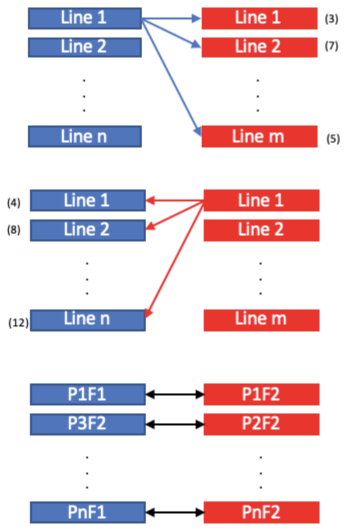


Figure 3.2: Computing line edit distance between two files to identify paraphrases.

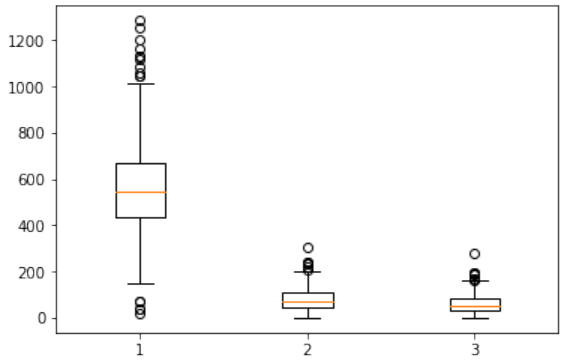


Figure 3.3: Box plots of number of invalid (1), compileable (2), and valid (3) paraphrases.

can be called parallel monolingual. A strict real-world example of a parallel corpus is famous books that have several translations in another language. The Illiad, for example, has many English translations. Each of them contains slight variations depending on the author, their interpretation, and the primary ideas they want to emphasize, but they are all written in English and derived from the same original text. These texts are thereby parallel and can be used to extract English paraphrases. By the same token, we use source code solutions that solve the same problem, but are written by different programmers as our monolingual parallel corpora.

Next, to generate the paraphrases we could decompose each source code file to varying levels of

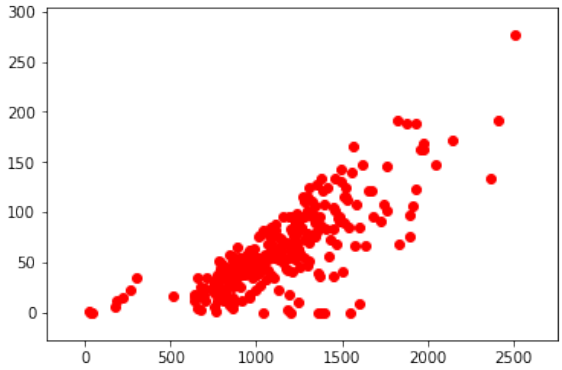


Figure 3.4: Time required to extract valid paraphrases based on the number of candidate pairs, where the x-axis is time in seconds and the y-axis is the number of transformations extracted.

granularity including source lines, statements, functions, classes, etc. We decided to begin at the source code line level because it is the most elementary level of abstraction. Following other works, we decided to compute the Levenshtein edit distance between each pair of lines in two distinct files.[25] This distance is defined as the sum of all mappings, deletions, substitutions, and insertions needed to ensure an equivalence between two lines. The cost of mapping a character is 0 and 1 for all other operations. We pair each line of the first script with the line from the second script that grants the smallest Levenshtein distance (see Figure 3.2). Similarly, we pair each line of the second script. Occasionally some threshold of distance is used as the filter by which to permit pairs. We decided to only consider lines pairs as potential paraphrases if they are mutually close to each other, yet syntactically distinct. To validate the paraphrases, we must show that the phrases still obey the syntactic rules and do not cause compile and run-time issues. To be useful as code transformations they also must retain the original behavior. To accomplish this, we can modify the solutions to use the new paraphrase instead. Variable names are a problem that arises. We must keep the original variable and function names unchanged. If the solutions are determined by their ability to pass a test suite, we can use the same test suite to determine whether the paraphrased code still constitutes an acceptable solution. If the paraphrased code produces the same output, we assume that the phrase transformation is a viable code transformation.

3.4 Results

The Google Code Jam data set has been valuable to code analysis research and has been used to develop and benchmark results under a controlled setting.[13] In this work, we show how to identify and validate semantically equivalent paraphrases between pairs of C++ files. Each of the solutions of the tasks in our data set is written in C++ and consists of approximately 50 to 200 lines. Our data set consists of 27,300 files consisting of 273 topics each with 10 parallel files i.e., those that solved the same challenge question. The computational complexity for computing the edit distance is $O(n^2)$. Using the initial C++ scripts and substituting some lines with the new paraphrases generated, creates new solutions to the problems solved by the initial two C++ scripts. Since the solution size per problem is 10 files, there were 90 permutations of size two, each producing paraphrases. To ensure this, we tokenize all paraphrased pairs into keywords, variables, numerical values, etc. using a set of regular expressions. For each syntactically unique paraphrase pair, a new pair is created by swapping all tokens of the two paraphrases, except variable and function names. We identified 152,658 candidate paraphrase pairs and filtered out syntactically identical pairs. Google Code Jam provides a website to practice with a test set to check if we solved the problem that was used to validate the pairs. 63,264 paraphrases failed to compile, meet run-time requirement, or pass the test suite, and 16,096 paraphrases passed all of these criteria.

To formalize the phraphrases into a graph form we generalize a paraphrase to have generic variables names such as a, b, c...aa, ab, ac...zz, aaa, etc. We used each phrase as a node and the edge represents a transformation from one to the other (see Figure 3.5). Many transformations occur frequently, but a large majority only appear once or twice. The expectation of a distribution closer to a normal distribution suggests that a larger data-set set should be used but also that there are both common practices and idiosyncrasies represented (see Figure 3.6). We were also curious about how dense the connected networks were (see Figure 3.7). The more nodes, the more chance for a highly connected graph. But they remained relatively constant in their density. While this graph looks barren, many of the graphs are quite sparse. Figure 3.7 shows that most of the graphs have fewer than 10 nodes. The number of paraphrases and size of graphs are heavily skewed. Future

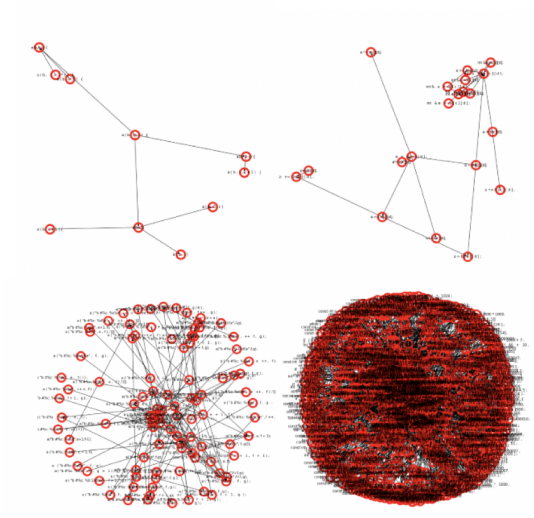


Figure 3.5: Paraphrases graphed to show related phrases and how you can get from one to another. Here we are showing graphs with 10, 20, 101, and 1,435 nodes respectively.

studies consisting of a larger sample size may yield a more normal distribution.

To speed up the paraphrase extraction process or extract paraphrases without a test suite we trained a Random Forest model on the verified transformations. Limited to just the Levenshtein Distance, we get 20% valid paraphrases, which is competitive with state-of-the-art NLP technique's semantic retention. We generalized the phrases by relabeling all the variable names to be a, b, c, ... z, aa, ab, ... zz, aaa, aab, etc. to avoid repetition and make it adaptable to new code. Additionally, we removed all instances in the overlap between positive and negative samples, which ended up being 3% of the data. Some of the overlap could be due to the context in, or time sensitivity of, the task. The data set contained 7,300 positive instances and 28,374 negative ones which reduced the data-set to 44% size of original with similar distribution of positive to negative observations. We trained and tested a random forest model with 10-fold train test split and having 100 estimators and no max depth. 10 trials of this experiment resulted in an average of 83.7% and a standard deviation of .03%. The ROC curve in Figure 3.8 also shows a balance between true positive and false positives.

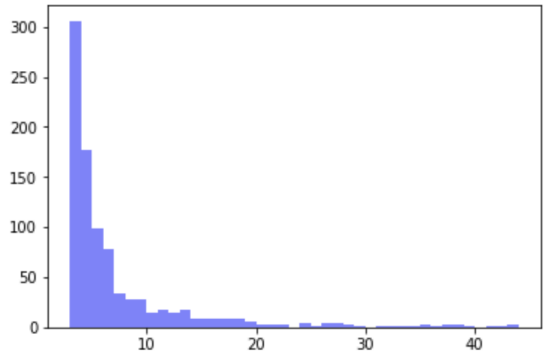


Figure 3.6: The x-axis is the number of times a paraphrase is seen; the y-axis shows the number of phrases fall into this category (cardinally).

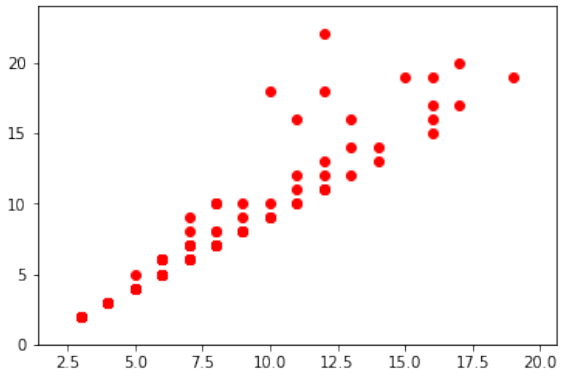


Figure 3.7: The density of the graphs increases as the number of nodes increases.

3.5 Discussion

These code transformations have the potential to improve code in various ways such as formatting, refactoring, and run-time execution optimization based on evaluation of the transformation. Formatting, for example, is as simple as prefering lines of code in form A instead of form B so when we see phrase B, replace with phrase A. We can extend this idea to refactoring if we consider A to be better than B by some refactoring or optimization metric. These transformations can also be evaluated for stylometric purposes. For example, an author may always use one idiom, but in this particular instance, he or she does not want people to know that they wrote this code. In this case, a recommender system could suggest ways that other people use as alternatives. The benefit of this

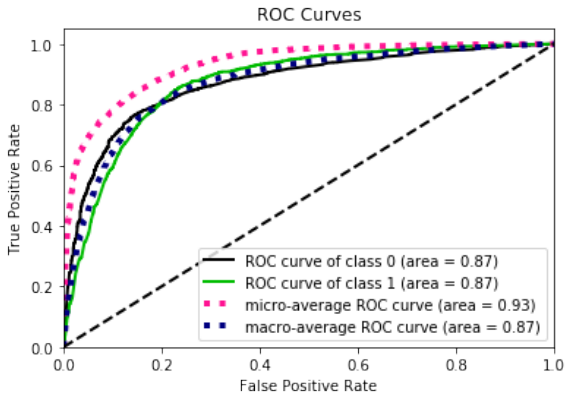


Figure 3.8: ROC Curve.

Table 3.1: C++ Token Types.

String	Close Parentheses
TernaryOpen	Curly Brace
Float Literal	Comparator
Open Parentheses	Whitespace
Close Bracket	Comment
Assignment Operator	String Literal
Operation	Annotation
Bit Operation	Open Bracket
Preprocessor Statement	Close Curly Brace
Keyword	Assignment
Integer literal	Punctuation

method is that it is amenable to automation and does so quickly (at least faster and cheaper than source code transformations generated manually by a human).

This work opens many avenues of possible interest. The path directly forward suggests work on applying this method to high level paraphrases such as statements, functions, etc. The transformations can be used to impact FLP stylometry and is particularly well-suited for GANS. Both the refactoring and optimizing techniques use transformations that are commonly found in source code developed by humans rather than code that is generated automatically. Finally, they may help illuminate complicated or obfuscated code.

3.6 Conclusions

This work demonstrates that paraphrase techniques from NLP are also suitable for FL and that the paraphrases identified can be used as code transformations with similar accuracy as NLP techniques. The RF model we trained can accurately discern valid source code transformations from invalid ones with accuracy much higher than comparable methods in NLP and without the need of a test suite making them suitable for production quality code. To generate these automatically is desirable for applications such as Stylometry and other systematic approaches to changing source code.

Chapter 4: Segmentation: Blocks of Source Code

Source code segmentation is the process of dividing the source code of a program into meaningful pieces. Our goal is to segment code based on the semantics of its content. In other words, the segments reflect locations that are good candidates for the insertion of manually composed comments or automatically generated comments. Instead of focusing on syntactic boundaries for segmentation, such as function and class declarations, we exploit the semantic content of the code. We use code snippets mined from Github as known semantic segments to train a LSTM Neural Network model. It is able to infer locations in the code where a programmer would likely insert comments. The model can operate on any text and performs well across multiple different programming languages to detecte candidate segment boundaries within blocks of code. This semantic code segmentation is especially useful for incomplete code repositories under development, which may also be written in more than one programming language. Additionally, our technique supports a detection threshold parameter so users can adjust the number and site of suggested segments provided by our tool.

4.1 Introduction

As we discussed in Chapter 3 on paraphrases, we may want to go beyond single lines of code. We discuss how code segmentation can help create multi-line segments framed as segments that constitute good locations for comments. Comments in source code help developers document and communicate important information about a program. While frequent and descriptive comments are encouraged as a best practice, they take time to write. Thus, the task of writing comments is often neglected and old comments may become stale or misleading as the source code changes. We discuss how automatically generated comments can keep documentation up to date in chapter 5. The problem that must be tackled first is knowing where to add comments, which is hard to automate in long code files. Code segmentation is a technique that can determine the boundaries of source code segments that are semantically cohesive. These boundaries provide good candidate

locations in the source code to insert a manually composed comment or a automatically generated comment. We create models for segmenting code into short, medium, and long code snippets.

Segmentation breaks larger text into smaller segments. This is similar to receiving the full text of a long book without chapters, and having a text segmentation model discover likely breakpoints and suggest chapter divisions. We refine the code segmentation process of[38]. Our enhancements to this method include using a new source corpus for training, NLP inspired preprocessing methods, and scope level target models as follows: (1) we use the NLP text tokenizer, called Sentencepiece[39], which is a tool that learns what is a token without the use of formal regular expressions; thus, enabling users to parse source code using tokens without processing the text on a per character basis; (2) we use the accessible and vast "Source Code Analysis data set" (SCAD)[40], which consists of code-comment pairs mined from reputable Github repositories instead of using Stack Overflow snippets; and, (3) we use several granularities of segments for different levels of comment specification. Our source code segmentation pipeline is capable of operating on arbitrarily long source code files. Additionally, it can return a variable number of potential segments by changing a configurable threshold for more or fewer source code segment recommendations. This pipeline is illustrated in Figure 4.1.

In Section 4.2, we discuss the NLP inspiration for topic and source code segmentation and motivations for improving it. In Section 4.3, we review the data, how we create multi-segment code samples, and how we tokenize and preprocess them for our neural network. Section 4.3.3 gives details about our choice of neural network and its structure, as well as other design decisions. We chose to use a bi-directional long short-term memory (LSTM) model because of its success in NLP and related natural language segmentation tasks. In Section 4.4, we show the results of the LSTM for specialized programming languages as well as for the multi-lingual model. We show that the multi-lingual model performs almost as well as the specialized models and has the advantage of being able to process code repositories written in different languages. Finally, in Sections 4.5 and 4.6 we discuss the value of this approach, its potential impact on other research, and future work.

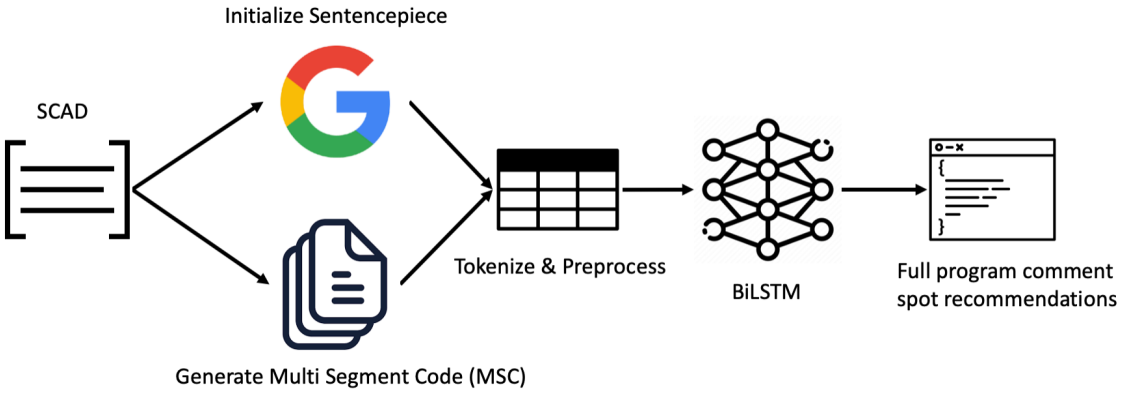
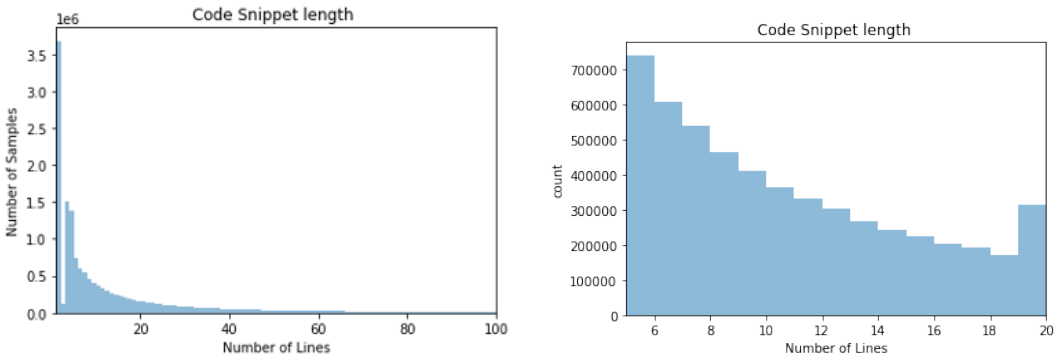


Figure 4.1: This figure outlines our pipeline for source code segmentation: First, we use Google's sentencepeice algorithm on SCAD to create a tokenizer. Second we generate code blocks with known segmentation points. Thrid, we preprocess the data by extracting features with the tokenizer. Last, we use the data to train and test a LSTM neural network to segment programs.

4.2 Background

Code Segmentation can be thought of as a classification problem where we predict the likelihood of a token marking the point between two logically separate series of tokens. We are interested in delineated semantically cohesive segments of source code. In natural language, partitioning by topic was demonstrated with Wikipedia articles[41]. Segmentation is important for downstream NLP tasks such as neural machine translation, paraphrase detection, source code and documentation summarizing. Source code segmentation is the process that partitions programs into manageable and semantically cohesive segments. However, segmenting code is different from segmenting natural language because code is a formal language that has specialized notations to mark the beginning and ending of syntactic code segments. The structure of formal languages, such as source code, is strict, or else the code will not compile. Source code also is less ambiguous. Previous work[38] explored a model for source code segmentation, but their model implementation is not readily available. We have identified several areas where it could be modified and refined. Our refined source code segmentation model is available at: https://github.com/Avielstein/code_segmentation.



(a) **(b)**

Figure 4.2: (a) Histogram of code segment length between 0-100 lines of code. (b) Histogram of code segment length between 4-20 lines of code.

4.3 Methods

Our goal is to create a model that can segment a program into semantically coherent pieces; yet, a model is only as good as the data it is trained on. First, we discuss the origin of this data and how it compares to data used elsewhere. We then discuss how we generate blocks of code for which we know the point where the segments change. Then, we explain how we use the NLP tool for creating a vocabulary and tokenizer adapted for source code.

4.3.1 Data

We choose to use the "Source Code Analysis data set" (SCAD), which is a data set of code-comment pairs mined from 108,568 projects downloaded from Github that have a redistributable license and at least 10 stars in C, C++, Java, and Python. This data set has several advantages over the data set used in[40], which used mined segments from Stack Overflow. The segments from SCAD are longer and from reputable GitHub repositories. Furthermore, Stack Overflow snippets are often much shorter and less likely to be functionally complete. Also, since the code snippets seen on Stack Overflow posts are often taken out of context, it is less likely that the code included represents the code one might see in an actual program. Snippets of code in Stack Overflow usually represent a problem in code with solutions recommended from the community resulting in the chance of unreliable code. It is unclear how[40] addresses these vulnerabilities from using Stack Overflow. We

Table 4.1: Number of Suitable Code snippets per language. Short comment segments are considered to be 1-5 lines of code, medium comments are considered to be 3-10 lines of code, long comments are considered to be 10-20 lines of code.

Short Code Segments			
Language	File extension	# Snippets	# lines of code
Python	py	384,274	1,379,956
C++	cpp, h	1,579,746	2,593,695
Java	java	2,234,085	5,200,971
Multi	combined	4,198,103	9,174,622
Medium Code Segments			
Language	File extension	# Snippets	# lines of code
Python	py	1,090,472	6,496,023
C++	cpp, h	648,940	3,623,966
Java	java	1,030,612	5,912,465
Multi	combined	2,770,022	16,032,454
Long Code Segments			
Language	File extension	# Snippets	# lines of code
Python	py	355,397	4,717,632
C++	cpp, h	161,779	2,203,111
Java	java	281,307	3,757,412
Multi	combined	798,481	10,678,155

limit our samples to those with more than four and fewer than twenty lines of code. We create 9 data sets of code snippets representing different languages and by length as measured by the number of new lines. The short data set includes snippets with 1 to 5 lines of code, the medium snippets have 3 to 10 lines of code, and the long code snippets have 10 to 20 lines of code. We also limited the text and comment length to be fewer than 500 and 250 characters, respectively. This results in over a million snippets samples from each language.

4.3.2 Preprocessing

To extend the NLP analogy to the problem of source code segmentation, we are able to take advantage of an NLP tokenizer for our code. We use Sentencepiece[39], a language-independent sub-word tokenizer designed for Neural-based text processing, from Google. As code is written using a programming language, our tokenizer pays special attention to spaces, tabs, and new lines which are "SPACE", "TAB", and "NEWLINE" tokens, respectively. We use the entire SCAD corpus to train the tokenizer. This creates a vocabulary, which we limit to 10,000 words. While there are many more words in the corpus, this vocabulary contains sufficient sub-words to parse any text it receives.

This method is useful for large vocabularies with the potential of unknown words and makes source code processing more robust. We generated blocks of code from sample segments within SCAD. These generated multi-segment code (MSC) blocks are our ground truth for training. Each MSC consists of three segments from the data set. We then tokenize each MSC with the custom tokenizer and label the beginning of each segment with a 1 and all others with a 0. This process has the advantage of avoiding the inclusion of a line break between segments as in [40]. Each MSC consists of segment samples from SCAD from the same language so there are no segments with multiple languages within one MSC. We do this to avoid our model learning from separating between languages since this is unlikely in real world code. We have a vast amount of data to use but only need a small fraction to achieve competitive results. We generate 10,000 MSCs for each language and at each length. From this we use 5000 MSCs for training, 1,500 for validation, and 1,500 for testing. To train the multi-lingual model, we use 2,500 from each language, 1500 for training, 500 for validation, and 500 for testing.

4.3.3 Learning Model

Once we have preprocessed the data we can train a neural network. Recurrent neural networks, and in particular bidirectional long short term memory (LSTM) [42] models have been shown to be effective for NLP tasks [43;44]. We chose to use a LSTM model because it is effective for the sequence-to-sequence data we have generated. We built our model with the PyTorch library. We chose to focus on classifying the center token and use the 20 tokens before and after as context, resulting in a window size of 41 tokens, compared to the 100 character window size of [38]. We label each of the tokens in the code as either a break point (1) or not a break point (0) and we choose to use Sentencepiece to learn code tokens that can be used to parse the code. We used a sliding window with a center token and twenty tokens before and twenty tokens after. If a window included an area outside the code tokens, we replaced them with the vocabulary data token (unk). Each segment included many windows and two break points, representing where the snippets of the segment were merged. However, with 10,000 MSC, we ended up with over 2 million windows. Therefore, we decided to randomly sample one window from each MSC as training for our model which may include one,

```
def unlock_and_close(self):
  if self._locked:
    fcntl.lockf(self._fh.fileno(), fcntl.LOCK_UN)
  self._locked = False
  if self._fh:
    self._fh.close()
```

```
def _create_file_if_needed(self):
  if not os.path.exists(self._file.filename()):
    old_umask = os.umask(0177)
    try:
      open(self._file.filename(), 'a+b').close()
    finally:
      os.umask(old_umask)
```

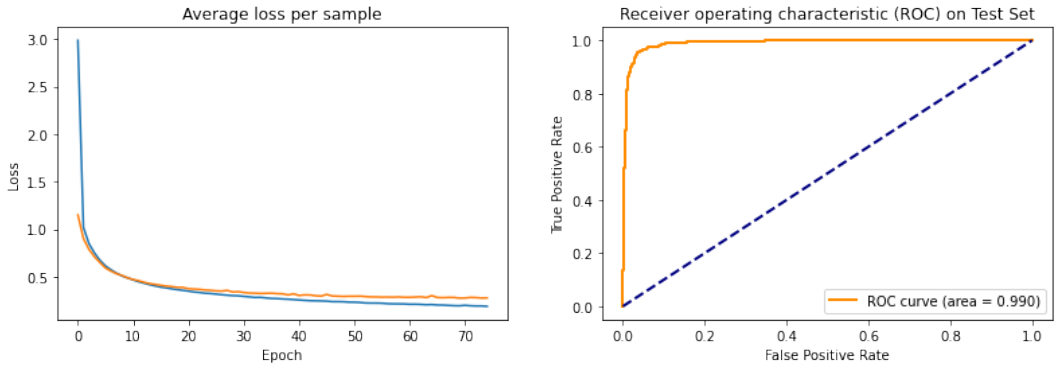
```
def _unlock(self):
  self._file.unlock_and_close()
  self._thread_lock.release()
```

Figure 4.3: Example of code from SCAD, by combining several logically distinct code segments we create blocks of code with known break points that can be tokenized and used to train a neural network.

two, or no breakpoints. This is desirable because it means that it is more likely to generalize when working on programs of arbitrary length. The LSTM model had a hidden dimension of 200 nodes, 2-LSTM layers, a dropout of 0.05, learning rate of 0.0005, and clipping threshold of 0.25. We batched the data in sets of 50 samples windows.

4.4 Results

We trained the LSTM until there was no improvement in the validation loss for 5 consecutive epochs. Models typically required between 60-80 epochs to complete training or about 30-40 minutes on a personal computer. The results of the model on the test sets are referenced below in Table II. Also, when evaluating the accuracy and ROC, we only consider NEWLINE tokens, since breakpoints are always known to be at a NEWLINE (NL) token. Java consistently scored above 99% in NL accuracy,



(a) **(b)**

Figure 4.4: (a) Example of model training (with long Python snippets) that trained for over 70 epochs. Our model levels out and the validation loss is very close suggesting that it generalizes well. (b) We evaluate based on results for New Lines (NL) only and as expected, when we apply the model to the training set it generalizes well (with long Python snippets). The area under the curve (AUC) for the ROC curve, shows good coverage and balance between true and false positive rates.

Python scored above 98% NL accuracy, but C++ did not perform as well. It scored around 90% NL accuracy and 96% for long segments, which may be in part due to a smaller portion of NL which represent a point of segmentation. It is also worth noting that this data has far more tokens and even NL that do not indicate a break point than do. The ROC show that the results are balanced between True and False positive results.

We also train a multi-lingual model that is capable of segmenting C++, Java, and Python indiscriminately. This suggests that its easy to use a tokenizer and model across most common code, making it more viable for code found in the wild. The multi-lingual model scored between 95-97% and was limited to similar constraints as the single language models (*i.e.,* same vocabulary size and training data, actually a bit less).

4.5 Discussion

Regarding the source code segmentation, we found that the LSTM presented with a byte-pair encoding tokenizer worked well for segmenting code. It offered several advantages including a sliding window, fuzzy parsing, multi lingual properties, and a customizable threshold for more or less gran-

Table 4.2: Comparing segmentation with specific languages and with all combined.

Short Segments Efficacy			
Language	% Token Accuracy	% NL Token Accuracy	NL AUC
Python	99.8%	98.9%	99.8%
C++	99.5%	90.0%	96.1%
Java	99.9%	99.9%	99.9%
Multi	99.7%	95.1%	98.8%
Medium Segments Efficacy			
Language	% Token Accuracy	% NL Token Accuracy	NL AUC
Python	99.9%	99.3%	99.8%
C++	99.3%	90.8%	89.8%
Java	99.9%	99.4%	99.9%
Multi	99.7%	96.3%	96.3%
Long Segments Efficacy			
Language	% Token Accuracy	% NL Token Accuracy	NL AUC
Python	99.9%	98.5%	99.0%
C++	99.7%	96.2%	85.9%
Java	99.9%	99.5%	99.9%
Multi	99.7%	97.1%	92.3%

ular comments. A sliding window is important for being able to evaluate the code of long programs. The fuzzy parsing means we can analyze code even if the code is not correctly written (*i.e.,* the source code contains syntax errors). Since byte-pair encoding is a compression algorithm, it can easily be adapted to work on entire documents, regardless of the programming language. And finally, we were able to change the value of our threshold for comment insertion to add fewer or more comments. This model worked well for all the languages and can even be used in combination in the form of the multi-lingual models. C++ does the worst, but the difference is minor so it can still be used reliably on source code and any discrepancy could be addressed with more data. We end up only using about 450,000 of the 10,000,000+ snippets available, so that is about 4.5% of our data but due to the training time cost, we limited it to 10,000 MSC per model. Also, it is likely that the multi-lingual model is learning all three models but it is not clear if it could do well on a new language. The source code vocabulary discovered by Sentencepeice contains many common source code words. These include keywords and other special words, but also common vocabulary names. It would be interesting to see what would happen if the segments had generic variable names (*e.g.,* var1, var2,... *etc.*). Would this result in our model learning longer tokens of common phrases?

4.6 Conclusions

Our method for segmenting source code accounts for its semantic content by leveraging NLP tools for learning where comments should be inserted. Source Code paraphrasing, as discussed in Chapter 3, could be expanded to operate on multi-line segments and is necessary before comment generation, as dicussed in Chapter 5. These methods have the advantages of operating on any text and being multi-lingual. We segment source code files into functionally distinct snippets by training on a generated data set of multi-segment code blocks that have been processed with a Sentencepeice tokenizer. The learned vocabulary is reflective of common words in code. This model is capable of being used off the shelf for recommending comment locations on full multi-lingual code repositories, which are common in places such as Github. In addition to recommending locations for comments, it can be used as a precursor to downstream SCA tasks such as paraphrase discovery and comment generation. Even though we used less than 5% of our available data, we achieved high accuracy for Python and Java. C++ was harder to learn, but this could be resolved by training with more data. Additionally the multi-lingual model scored higher than 95% on test data across all levels of segment size.

Chapter 5: Translation: Code Snippet to Comment

5.1 Introduction

Software source code can be complex and, therefore, difficult to comprehend, even for experienced programmers. Comments are one of the essential tools available to software engineers. Unfortunately, comments are often absent or outdated. Deep Learning (DL) and probabilistic methods have been developed in an attempt to comment code automatically. The focus of these models has transitioned from the more difficult problem of translating source code into natural language to instead, the more tractable problem of generating useful abstract representations of the source code. The state-of-the-art approach introduced in [45] uses a Recurrent Neural Network (RNN) with an Attention cell within a Long Short-Term Memory module (LSTM) that feeds the representation of code snippets into downstream nodes of the network. This model, when trained on a collection of Stackoverflow code snippets in C++ generates meaningful summaries correctly about 37% of the time and have a BLEU score of about 0.2. In our work, we used SCAD [40], which is a collection of code-comment pairs mined from reputable GitHub repositories, to train and validate a Neural Machine Translation (NMT) model. We focus exclusively on Python, ignoring code-comment pairs in C++, and Java, which may be considered in future work. SCAD also may have an advantage over the Stackoverflow data because Stackoverflow often discusses problems with the code. SCAD comments are the result of the programmer's intentionally conveying information regarding the code to other programmers. In our analysis we initially obtained a BLEU score of .29 and, subsequently, after applying some custom filters increased the score to .49, which is considered a high quality translation.

5.2 Background

5.2.1 Neural Language Processing Techniques

Neural Machine Translation (NMT) relies heavily on DL to achieve good results. In particular, it uses sequence-to-sequence (Seq2Seq) models such as RNNs and LSTM architectures. Additionally,

it can include other layers such as attention, encoders-decoders, and embeddings. Both the input and output text is encoded to make it easily readable by a machine. In the context of source code translation, the encoding of data refers to mapping the token strings to integer values. This numerical encoding of the source language becomes the output of the first segment of a Seq2Seq model, often referred to as the encoder. The decoder takes the encoding as input, and outputs the target language in encoded form, which is then converted to human readable text as the final step in the process. The use of word embeddings in NMTs is often referred to as the process of eliciting similar network outputs for similar words/phrases. This is accomplished by configuring an embedding layer within the ML model that represents tokens as dense, fixed-length vectors. This setup, as opposed to something like a one-hot encoding, results in similar outputs for synonymous tokens[46].

5.2.2 Deep Neural Architectures

Simple neural network models are often insufficient for processing sequential data. Recurrent Neural Networks (RNNs) are a category of neural network that accounts for temporal information. There are several ways of implementing RNN's, including LSTM's, bidirectional RNN's, and hierarchical RNN's. Essentially, an RNN feeds its output to itself at each subsequent time step forming a loop and passing down information as needed. At each time step, the input from the previous time step is encoded and used in conjunction with the current input[47]. To avoid the vanishing gradient we use a Gated Recurrent Unit (GRU) introduced in[46]. The GRU uses update and reset gates to learn how much of the past information should be passed to the future. This is a significant part of the unit since it can reduce the chance of a vanishing gradient occurrence[48]. Sequence-to-Sequence (Seq2Seq) models are a special class of RNNs used to solve complex language problems. Seq2Seq models are used to convert sequences of one type to sequences of another type. For example, translation of English sentences to German sentences is a possible Seq2Seq task. Seq2Seq models are traditionally composed of an encoder-decoder architecture where the encoder maps the input sequence to a context vector of fixed length and the decoder generates the transformed output given this vector. This tends to work well for short sequences, but as the length of a sequence increases,

the compressed vector becomes a bottleneck and struggles to encode all the data of a long sequence into the vector. Attention[49] helps to focus on a specific set of tokens in the input sequence. With an attention module included in a NLP task, it is apparent which tokens in the output sequence depend on specific tokens in the input sequence and can normally be represented in terms of a heat map, deemed an attention map.

5.2.3 Evaluation of NMT Methods

Machine translation dates back to the 17th century[50] but it was not until the 2010's that modern DL methods started to be applied to translation[47;51] now known as Neural Machine Translation. Google Translate relies on artificial neural networks, typically ones with RNN's and LSTM layers[52]. For long sequences, attention mechanisms help improve accuracy. There are many considerations one must take into account when attempting to automate translation. For instance, the lexical, syntactic, and semantic information between the source and target language must be maintained. Generally, translations are measured by their fluency (correct syntax and spelling) and adequacy (semantics); however, this evaluation is challenging because there are no single correct answers, and even human evaluators often disagree on the quality of automatic translations.

Evaluation metrics can be used to quantify the quality of translations. An example of a widely-used translation metric is Bilingual Evaluation Understudy (BLEU), which considers the N-gram overlap between machine translation output and reference translations. Another translation metric is Metric for Evaluation of Translation with Explicit ORdering (METEOR), which gives partial credit for matching stems, synonyms, and the use of paraphrases. These metrics are not perfect however. Specifically, these metrics treat all words as equally relevant; thus, the scores are not always informative, and even human translators can score low. However, while they are not exact measurement, they do tend to correlate with human judges in quantitative surveys, as illustrated in Figure 5.1.

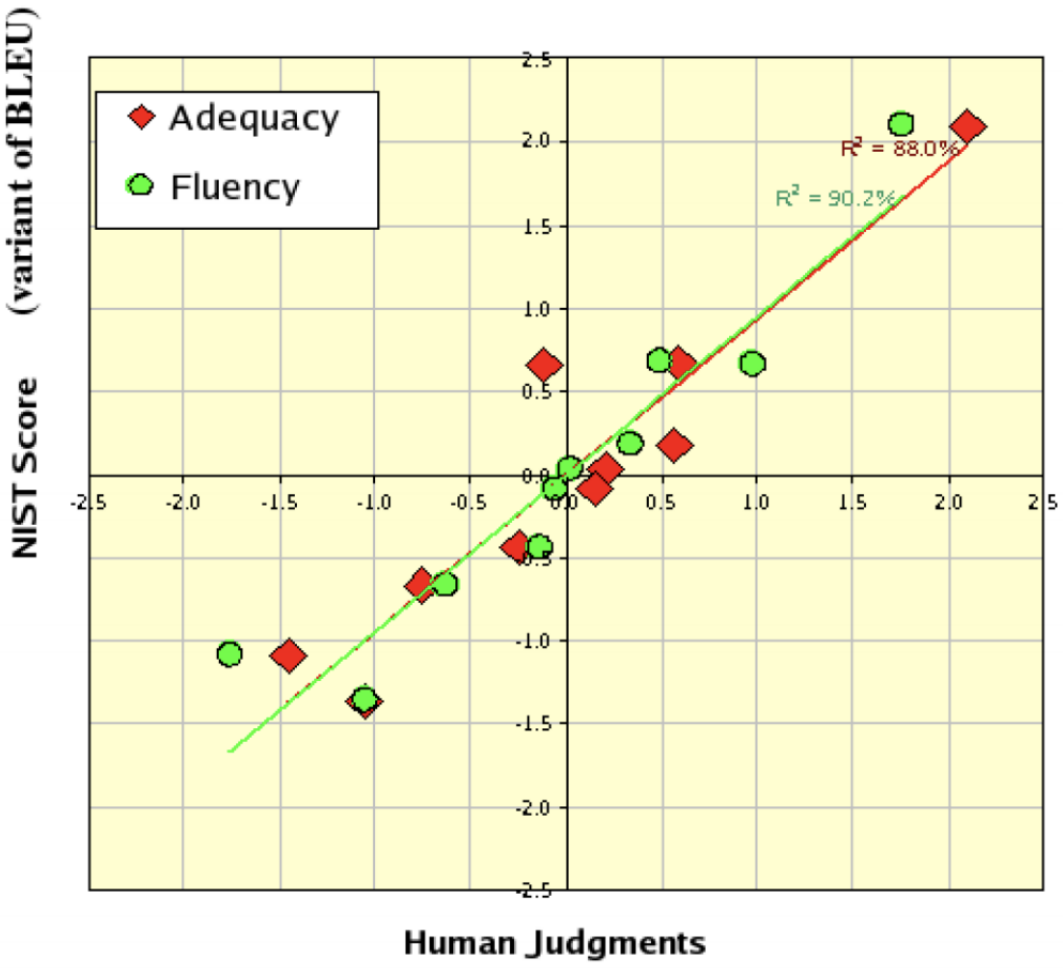


Figure 5.1: Metric vs human judgement.[1]

5.3 Methods

5.3.1 Preprocessing

For each input code snippet, we prefixed a start token (`<start>`) and post-fixed an end token (`<end>`) so the model can identify the length of the sequence. Next, we filtered the code snippets for sequences with a maximum length of 100 characters between 4 and 20 lines of code. We used Google's Sentencepiece[39] to tokenize the code and limit the vocabulary to fewer than 10,000 words. In this step, we generated dictionaries that map each token to a unique id as well as reverse dictionaries that map each id back to its corresponding word. Lastly, the input sequences were padded to a maximum

length (100 characters) so that every sequence is consistent and there are no variable-length inputs.

Good comments should convey the meaning of the code, be easy to read, and ideally have a consistent style[53]. Therefore we also filter the code-comment pairs by the cohesion and naturalness. In terms of cohesion the sequences were parsed so that the comment and code share at least 3 tokens. Unlike in natural language where each language uses different words, many of the tokens in code are used as, or as part of, variable and function names. We found comments that shared some tokens with tokens in the code were more focused on the function of the code than the code's function in its context. This makes it more suitable for direct translation of code snippets without context. Additionally, some comments were overly technical and did not convey an abstract meaning of the code in a uniform way. To circumnavigate this, we also verify that the comment is fluent English and lacks many stop words, which are words that do not contain much semantic information but connect fragments, such as the words: and, or, but, the, etc. This is a modified variant from traditional NLP techniques, which often avoid using stop words because they crowd the important words. However, we found that by making sure some stop words were included (at least 3) we were able to get more sensible comments that were easier to understand. On the last point of having a consistent style, we do not impose any kind of restraints. However, something like that could be done post processing on the text, such as with domain adaptive text style transfer[54].

5.3.2 Neural Machine Translation

The model is implemented as an encoder-decoder architecture that follows the inherent attributes of the Seq2Seq model[51]. We utilized the Python programming language and PyTorch library to construct our project. In this model, the input is fed to a GRU encoder model that outputs shape $(batch\_size, max\_length, hidden\_size)$ and generates a hidden state in the shape of $(batch\_size, hidden\_units)$. We define the hyperparameters of the batch size as 64, embedding dimension as 256, and the number of hidden units as 1024. We arrived at using these values for the hyperparameters of the GRU after experimentation and found that these values provided efficient and accurate results. We used the adaptive momentum optimizer (Adam), used a teacher forcing ratio of 0.5, hidden dimension size of 1000, batch size of 5, learning rate of 0.001, and 1

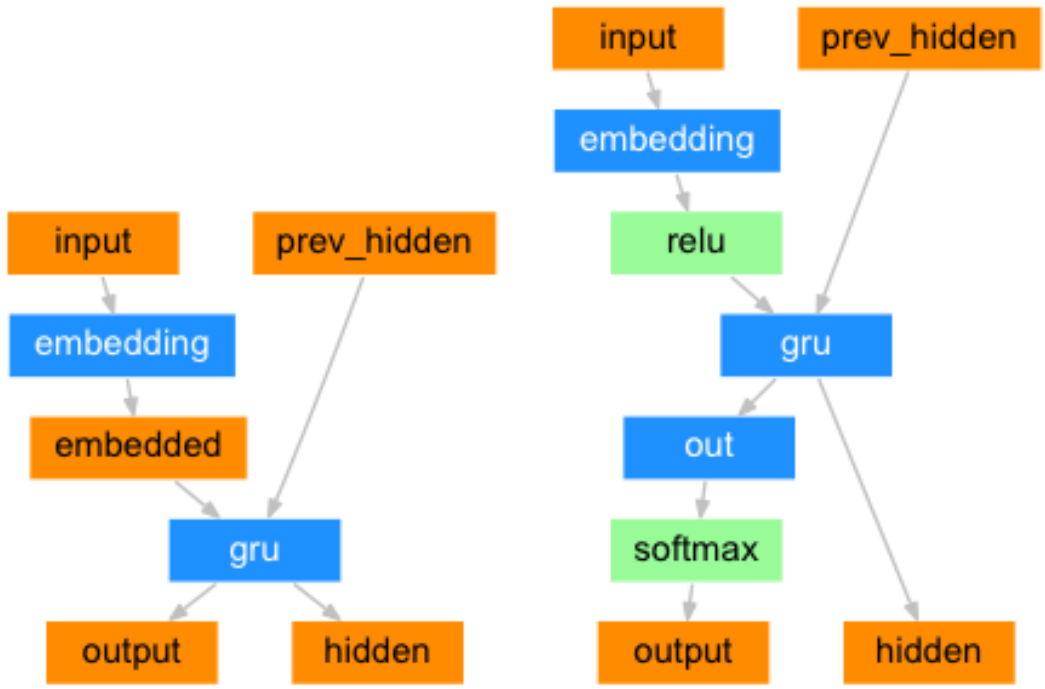


Figure 5.2: (a) The encoder of a seq2seq network is a RNN that outputs some value for every word from the input sentence. (b) The decoder produces the output as a sequence of words (i.e., the translation) from the output vector of the encoder.

percent dropout. The use of the Adam optimizer resulted in a faster convergence rate compared to traditional Stochastic Gradient Descent (SGD) and used the NLP technique of teach forcing to propagate the ground truth as input in further time steps in case the prediction from the previous time step was incorrect. To address the overfitting encountered during the experimentation of the model while tuning the hyperparameters, we included dropout to the GRU, which is a common practice to reduce overfitting.

The implemented Seq2Seq model uses an attention mechanism applied on top of the GRU encoder model. The weights of the attention mechanism are computed by a softmax function applied length of the input. The score is used to compute how much weight a given hidden state is given in the attention is measured by the Bahdanau's additive style in scoring as detailed in[55]. The first part of the Seq2Seq model is the encoder, which takes as input the context vector from the attention

module as well as the output of the input passed to an embedding layer (see Figure 5.3). The embedding used on the input maps the input from the size of the vocabulary (10,000) to the size of the embedding dimension (256) by transforming the positive integers into dense vectors of a fixed size. These two inputs: (1) embedding output and (2) context vector, are concatenated and the merged vector is then fed into a GRU. The main component of the encoder is the GRU with weights initialized according to the Glorot uniform distribution with a smaller standard deviation to avoid vanishing gradients[56]. The decoder follows a similar architecture and begins with the attention module computing the context vector and the attention weights (see Figure **??**). Again, input is passed through an embedding layer and then concatenated with the context vector passed through the GRU with the same initialization technique. The main difference between the encoder and decoder is that the output of the GRU in the decoder is flattened and passed through an additional FC layer to the dimension of the vocabulary size with a linear activation function. This additional layer acts as the final downstream task of mapping the output decoding to an English sentence representing a comment that is easily readable by a human.

The training of this model is traditional in terms of Seq2Seq learning where the input is passed through the encoder to generate its output, as well as the hidden state of the encoder. These two outputs, encoder output and encoder hidden state, are passed to the decoder along with the decoder input, which is represented as the start token. The decoder then outputs the predictions as well as the hidden state of the decoder. For each prediction, the hidden states and the prediction are fed back into the model to compute the loss of that given sample. As an alternative to back-propagation through time, we use teacher forcing to train the RNN. This technique uses actual or expected output from the training data at the current time step as the input in the next time step rather than the output generated by the network. This method converges more quickly, and without teacher forcing, the hidden states of the model could be updated by a sequence of wrong predictions and cause the errors to accumulate when the model is struggling to learn. Finally, the gradients are computed with respect to the sparse categorical cross-entropy loss function and the model is optimized using Adam gradient descent.

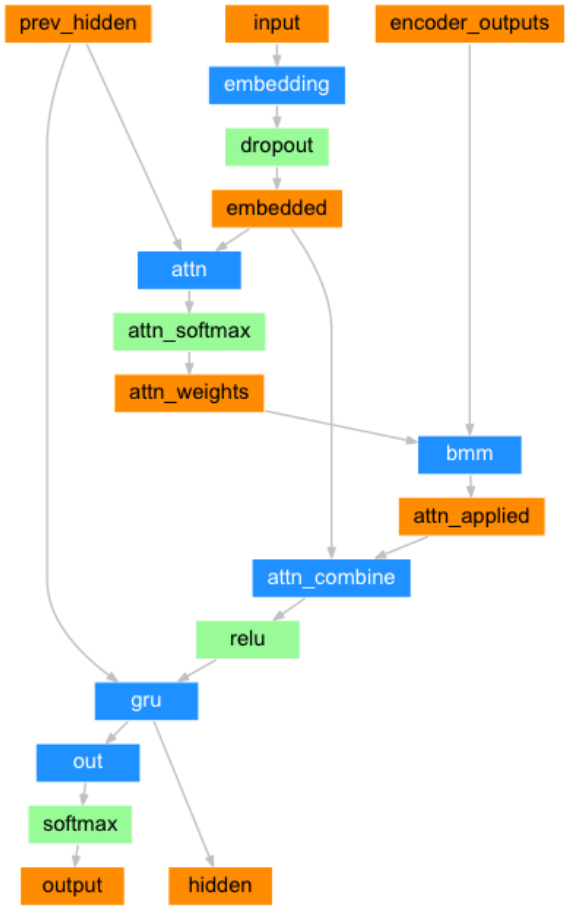


Figure 5.3: We calculate the attention weights with another feed-forward layer which uses as input the decoder's input and hidden state.

The last important component of our architecture is the translation or the evaluation of the embedded output. The evaluation function is similar to the process followed during training, but excludes the use of teacher forcing and instead the input to the decoder is the prediction at the previous time step, along with its hidden state and the output of the encoder. The output of the evaluation halts its prediction once the model predicts the end token (`<end>`). Also, to note, the attention weights are stored at each time step to generate a map of the attention for each code snippet and translation pair.

5.4 Results

5.4.1 Neural Machine Translation

The model is quite computationally expensive and both the English and Python code languages have a vocabulary of 10,000 tokens, and there are 20,000 samples in the training set, 100 in the validation set, and 1,000 in the test set. The model was it trained for 35,000 epochs and the categorical cross-entropy loss fell from 6.41 to a final value of 0.94.

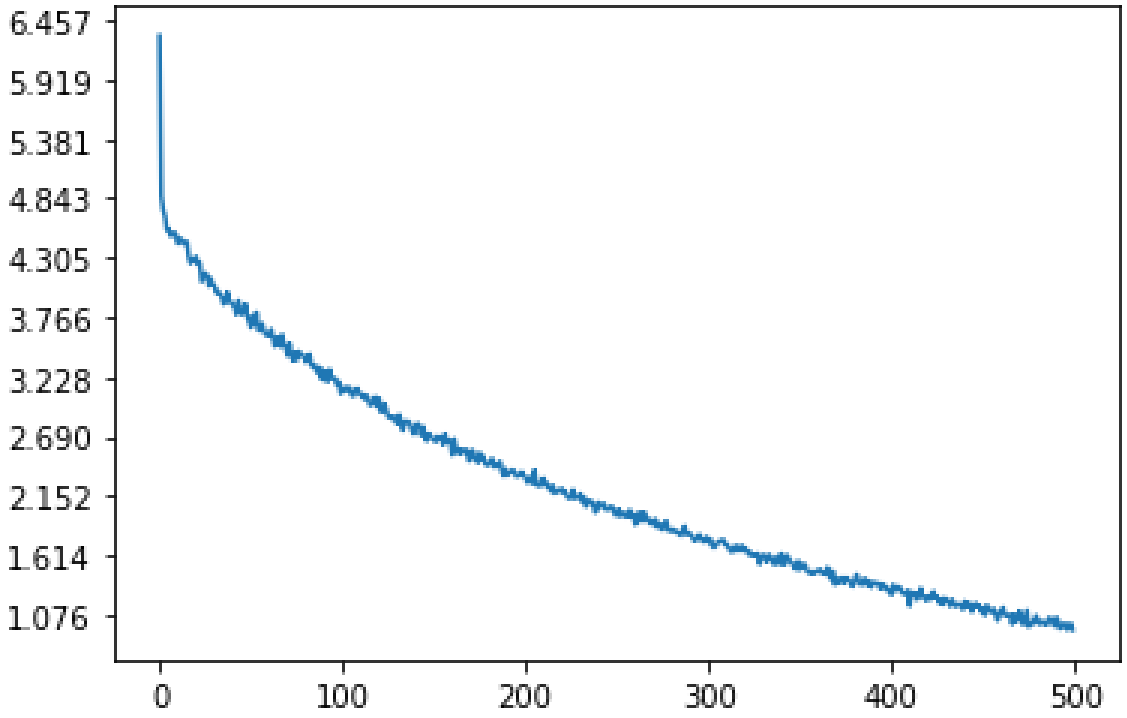


Figure 5.4: Plot of the loss incurred by the model across epochs.

5.4.2 Translation metrics

We used automated translation metrics common in NLP. Namely, we used BLEU[57], CHRF[58], GLUE[59], METEOR[60], and RIBES[61]. These methods are language independent and correlate with human evaluation. The Bilingual Evaluation Understudy score (BLEU), defined in Equation 5.1, was first introduced in 2002 and was one of the first metrics to exhibit high correlation with human translations. The BLEU equation where the N-gram precision p_n with N-grams up to length N, positive weights W_n for each N-gram length, and a brevity penalty BP for sentences that are too

short. We also use the character N-gram F1 score (CHRF) which analyzes a sentence on the character level instead of the word level. The Google-BLEU (GLEU) score is a variation of the BLEU score, which calculates the recall as the ratio of the number of matching N-grams in the reference translation to the total number of N-grams in the proposed translation, this score works better on the sentence level. The Rank-based Intuitive Bilingual Evaluation Score (RIBES) tackles the problem of word ordering, rewarding sentences which maintain the correct word ordering which is important for causal semantics. METEOR improves on the BLEU metric by stemming words and considering synonyms in WordNet[62].

$$\text{BLEU} = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{5.1}$$

Automated Evaluation

Measuring the efficacy of NMT models can be difficult due to the subjectivity of evaluating language tasks. We choose to use automatic methods over human evaluation since it is the common practice in the literature and human evaluation can take a long time. Our choice of BLEU, CHRF, GLEU, and RIBES gives us a wide range of perspectives about the outcome of our work.

Table 5.1: Filtering for shared tokens between code and generated comments can help improve confidence of a good BLEU score but limits number of generated comments. While a low BLEU score does not guarantee a good translation, it does help improve confidence that the comment is producing relevant material. Filtering for 3 shared words is a good compromise.

Shared tokens	% Qualified	Average BLEU
0	100	.28
1	93.3	.30
2	70.6	.35
3	37.6	.47
4	11.4	.57

Filtering for some number of shared words can be helpful to assure that the comment is relevant. We recommend using a filter of 3 shared words because it provides reliable translation according to the BLEU scores and are focused on relevant portions of the code. The result of the RIBES score of 0.41 emphasises causal consistency, suggesting that our comments were attentive to the order they were generated. The distributions for CHRF both look to be fairly normally distributed with

a mean score of 0.42 for the at character scale. BLEU and GLEU have the highest mean scores at 0.47 and 0.50 respectively, which are at the token scale. We see similar distributions for very good and very bad results with most distributed favorably well given the difficulty of the code-comment pairs. Good natural language translations often score between 0.4 and 0.7 even with human made translations since there are many ways to translate a sentence. This is also true of code, which can be described in many ways resulting in direct translations to be very rare. Additionally code—as well as the comments describing them—tend to be neologistic, *i.e.,* the names of variables, functions, classes, and libraries are frequently made up and may not be found in any common lexicon. This could have significant effects on the NMT metric scores.

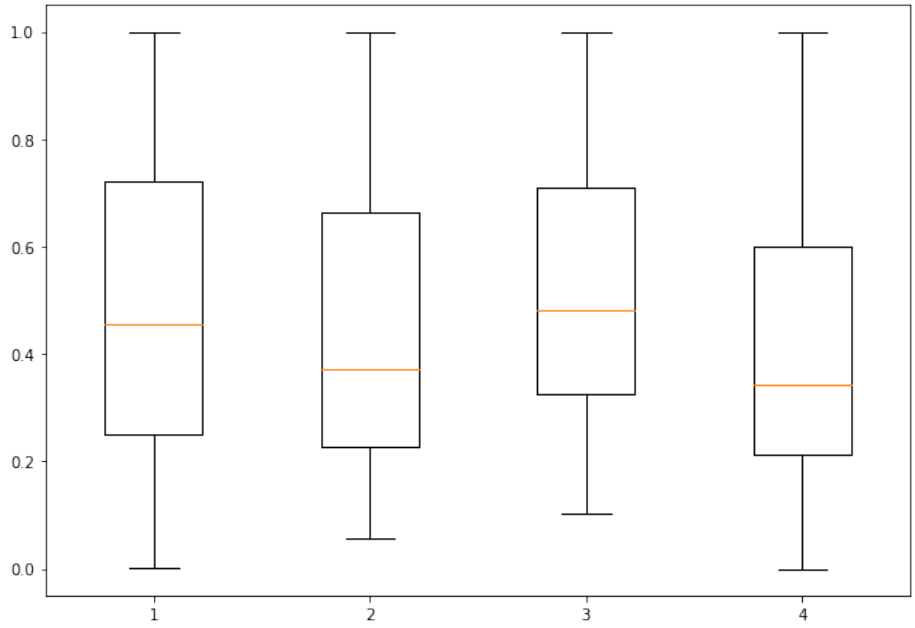


Figure 5.5: These are the results from filtering with at least 3 shared words. 1, 2, 3, and 4 represent the distributions of BLEU, CHRF, GLEU, and RIBES scores. A comparison of translation metrics show a distribution and generally positive results.

Human Level Evaluation

Due to the extensive length of our training and testing samples, it would take too long to examine each output; we therefore detail a few samples and provide some analysis. We noticed that sometimes the generated comments are more succinct or do not go into as much detail as the original comment and focus more on the core function. Occasionally, our technique would produce output that was

very close to that in the test data set, but our technique would choose different words such as “get” in place of “return”. We also noticed that some translations with a very low BLEU score still described the function of the code fairly well.

```
--------
Code
 def now_reign_year():

    now_ = datetime.datetime.now()
    return now_.year - 2015



Original Comment:  return now year reign for king .
Generated Comment: return the current year in date .
BLEU: 0.38570315774665637

--------
Code
     def get_used_size_from_instance(self, instance):

        raise NotImplementedError()


Original Comment: method used in update infra instances sizes . return use
d size in bytes from instance .
Generated Comment: get a size of the instance .
BLEU: 0.07868235647726078

----------
Code
     def device_state_attributes(self):

        return self._state_attributes

Original Comment: return all the state attributes .
Generated Comment: return the state attributes of the device .
BLEU: 0.6354719482589886

----------
Code
     def legacy_event_type(self):

        return self.name_to_event_type_map[self.name]


Original Comment: return the legacy event type of the current event
Generated Comment: return the string name type event .
BLEU: 0.4147310221083959
-----------
```

Figure 5.6: Generated comments that show that the translation score may be low but still be a good description of the code. In these cases they tend to be more concise than the original.

5.5 Discussion

This work draws on several active areas of research in NLP and their applications to code comprehension. Additionally, our classification results are less prone to language-specific abnormalities that make NMT metrics hard to compare across languages. Regarding the NMT model we used, we found that natural language translation (*i.e.,* French to English) is easier than formal language translation (*i.e.,* Python source code to English). We tried several simple Seq2Seq models that performed adequately for natural language but were not able to produce reliable results for comment generation from corresponding source code. We found that an attention mechanism is required for reliable results.

Below are some of the generated attention plots showing the relationship between code tokens and the corresponding predicted output English text. While these examples exhibit fairly accurate comment results, their attention maps are much sparser than expected. We did not find the attention maps to be always informative; it seemed like attention focused on unrelated parts of the code and not on the shared words as we saw in our experience with NLP.

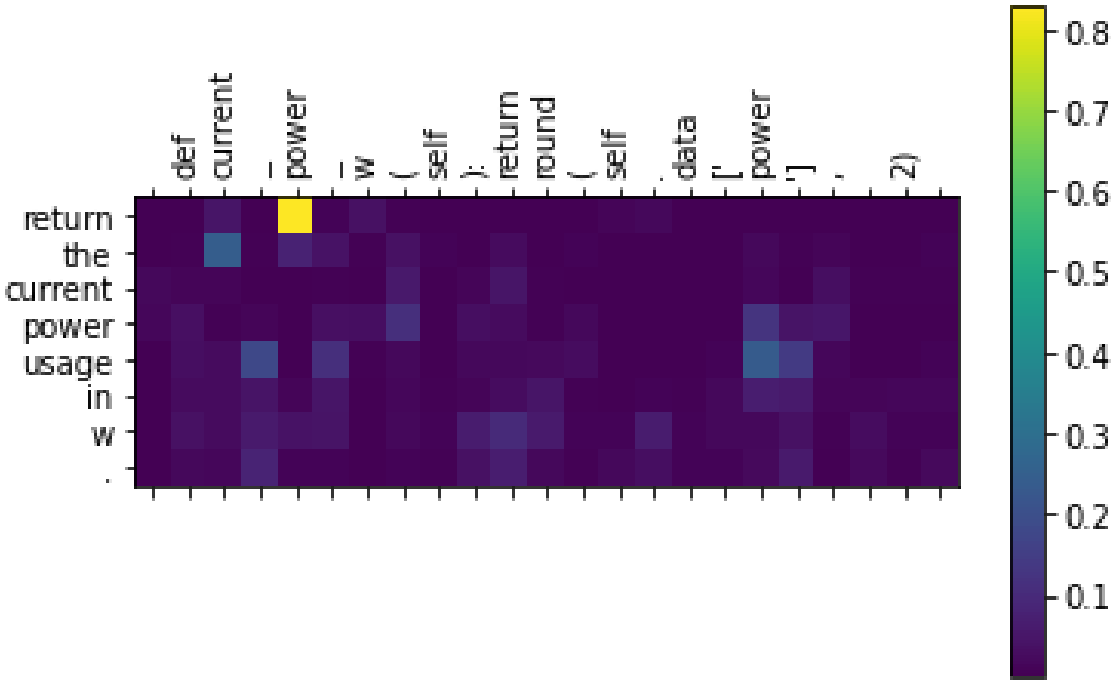


Figure 5.7: Attention map for small function, the focus of the return is correlated with the power, but not between shared words.

A next step in our work could be to generate longer comments across large pieces of code; for example, in the form of a README file, that document entire code repositories (*i.e.,* several related source code programs). Another avenue to pursue is to exploit the structure of Abstract Syntax Trees (ASTs) as a representation of the code, rather than relying solely on sequential source code tokens.

Additionally, another direction to extend this work could be using the Seq2Seq and transformer models outside of translation, but to find security vulnerabilities in code. This could be done by using differences (diffs) in GitHub repositories, and using those diffs to learn before and after comparisons such that if you identify a vulnerability, you could suggest a solution.

5.6 Conclusions

NMT models have been valuable for the study of natural language and text repositories. When applied to converting code to English text, we see the NMT model is automatically able to describe the function of the code. Even though the act of automatically creating comments is not the same as a direct translation between "natural" languages, we can still use it to annotate code with some assurance that the quality of the generated code is good enough to be useful to a software engineer. In our work, we obtain comparable results from traditional NLP translation metrics such as BLEU when filtering for shared words. This score is on par, or better, than in natural language domains found in the literature. We found that NMT models take a longer time to train than NLP models and require attention modules in addition to merely using a simple Seq2Seq model.

Chapter 6: Tagging: Lexical and Syntactic infrenencing

6.1 Introduction

This chapter covers the use of tagging source code tokens for identifying relationships between words to infer the abstract syntact tree (AST) structure of Python source code. Code is highly structured and we leverage this fact to improve feature extraction. In particular, we are interested in what can be inferred from just analyzing the text of the code. This chapter covers how one can infer the lexical tokens and AST structure of code from source code using part of speech (POS) tagging and dependency parsing without the use of a source code parser. We explore ways of inferring these attributes to reconstruct the AST of a program. This will be helpful in enhancing the techniques we have covered in this thesis as the field matures. These models do not require the text to be formally correct, meaning they can work on incomplete code and code segments. POS methods have been applied to code before, specifically for text-based software engineering tools[63] and to identify security vulnerabilities[64], however, to our knowledge this is the first time dependency parsing has been used to infer the ASTs of programs without the use of a parser.

Throughout this thesis we have primarily treated source code as plain text. The structure of source code languages are well defined and source code can be decomposed into components. This chapter discusses the difference between the surface level information and what can be infered about the structure of the text. We use POS and dependency parsing to infer the lexical and syntactic elements from plain text code. In the case of NL, this is usually done for information retrieval such as POS[10] tagging and Named-Entity Recognition (NER).[11] Both are NLP tasks that abstract and label areas of interest. Other similar tasks are dependency parsing[65], which identifies which words are related to on anothe, and arc labeling[66], which identifies the relationship. These techniques are use to annotates sentences for humans and computers and is useful for number of downstream NLP tasks. We show how this kind of information can be extracted from code using deep learning models based off of POS tagging, to identify the lexical tokens in the code, and Dependency parsing, to

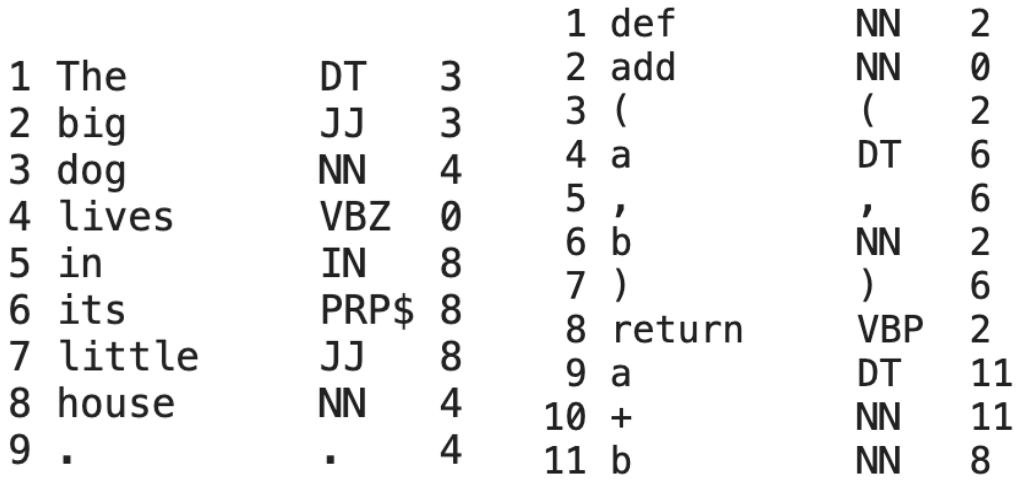


Figure 6.1: Output of the dependency parser showing the Conll formatting of (a) the relationships between words in the English sentence "The big dog lives in its little house." and (b) its attempt at doing the same thing to the code snippet "def add(a,b) return a + b".

infer the AST structure.

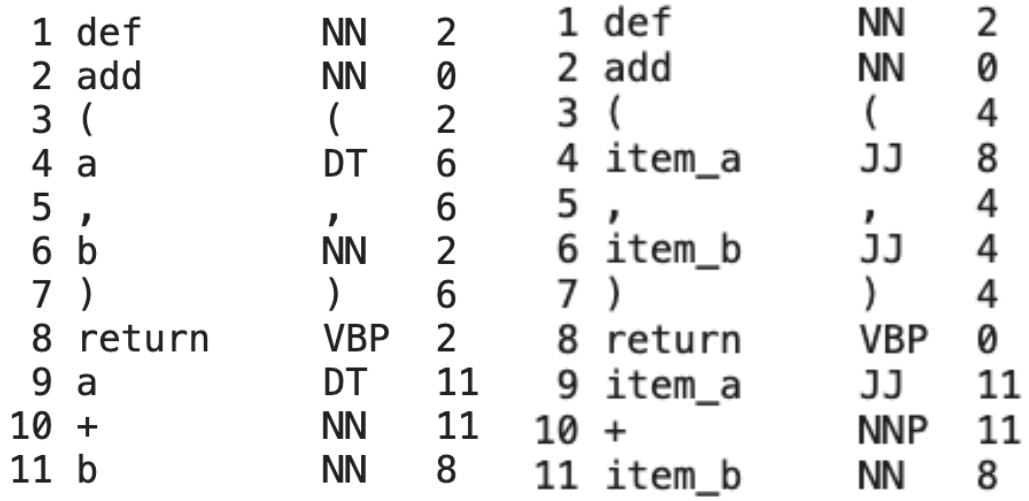
6.2 Background

One tenet of this work is that we can use models that assume little about the formal rules of the language and are instead designed for natural, especially the English, language. This makes sense because most keywords in programming languages are English. Natural language also has some degree of grammatical structure. The formal structure of natural language has been studied extensively. In NLP, structural aspects of the text are extracted using POS, NER, and Dependency Parsing. These methods are done for information retrieval and feature enhancing purposes. They can be used for high-fidelity information extraction and it has been shown that using POS can improve NMT performance[67]. One of the first methods to automatically parse natural language sentences was the Penn Treebank[68] and another more recent one is CONLL[69]. In the latter, format is favored because it was designed to provide a universal datatype for NLP tasks.

6.3 Methods

In this work we use a multi-task approach for creating models that can infer the underlying information about the structure of the code. For the POS task we use a dual LSTM tagger, one for character level and one for word level tagging. We obtain ground truth data using Python's tokenize function, which outputs the lexical tokens of well-formed source code. We then use a format similar to the Penn Treebank, which is a list of tuples with each literal word with its part of speech, or in the case of source code, the token type (e.g., "The big dog lives in its little house." becomes [('The', 'DT'), ('big', 'JJ'), ('dog', 'NN'), ('lives', 'VBZ'), ('in', 'IN'), ('its', 'PRP$'), ('little', 'JJ'), ('house', 'NN'), ('.', '.')]. We use NLTK to process our English text and in this case it uses 'DT' to mean determiner, 'JJ' as adjective, 'NN' as singular noun, 'VBZ' to mean 'verb, 3rd person sing. present tense', 'IN' as its literal (preposition/subordinating conjunction), 'PRP$' to mean possessive pronoun, and '.' to be the literal period. There are many other such token types 'EX' meaning "existential there" and others that are more similar to what we see in code such as 'JJR' and 'JJS' which represent comparative and superlative adjectives respectively. Python has its own set of lexical token types which define the language form, this allows us to pair them the same way (e.g., "print(1+2)" is interpreted as [('print', 'NAME'), ('(', 'LPAR'), ('1','NUMBER'), ('+','PLUS'), ('2','NUMBER'), (')', 'RPAR')]. We can then construct an LSTM model that works for POS and apply it to identify lexical tokens the same way. Our LSTM POS parser has two layers, the first working on the character level and next working on the word level.

For POS, we choose to use a simple annotation method similar to the Penn Treebank because it worked well with NL and ported easily over with the Python tokenizer. For dependency parsing you need to find the relationships between the words in the text. We based our work on [70], which uses an LSTM with Biaffine attention. The paper reports 95.8% AUS when we ran it on a personal computer with the English Web Treebank (EWT) [71] we trained for 50 epochs to get 88% AUS. To interpret the code in the CONLL format we extract the lexical and AST tokens. Although there are many columns, we only need ID, Lexical, POS, Parent_ID. We can choose to either feed the POS from our previous model or use the tokenizer model and then use the Python AST module



1	def	NN	2
2	add	NN	0
3	(	(	2
4	a	DT	6
5	,	,	6
6	b	NN	2
7	)	)	6
8	return	VBP	2
9	a	DT	11
10	+	NN	11
11	b	NN	8

(a)

1	def	NN	2
2	add	NN	0
3	(	(	4
4	item_a	JJ	8
5	,	,	4
6	item_b	JJ	4
7	)	)	4
8	return	VBP	0
9	item_a	JJ	11
10	+	NNP	11
11	item_b	NN	8

(b)

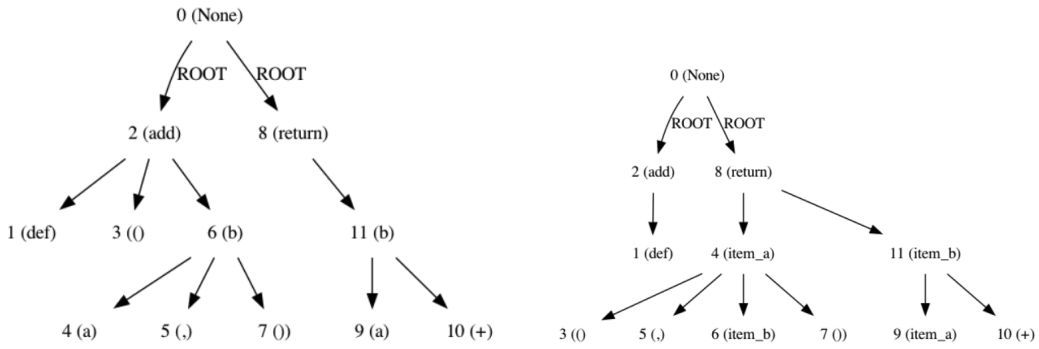
Figure 6.2: Output of the dependency parser showing the Conll formatting of (a) "def add(a,b) return a + b" we see that it thinks "a" is an article adjative, where as in (b) "def add(item_a,item_b) return item_a + item_b" we can use the "item_" to help clarify.

to generate our ground truth ASTs. Not all AST tokens have a literal equivalent, and vice versa, so some of the data is lost. We use a limited number of lexical tokens and AST node tokens when extracting from Python and have to align them because information about dependency comes from the AST but the order comes from the lexical extraction. This is because the AST module does not always save the literals but instead is able to interpret them within the data structure. We end with a subset of tokens that we can then match to the ones produced by the POS model.

6.4 Results

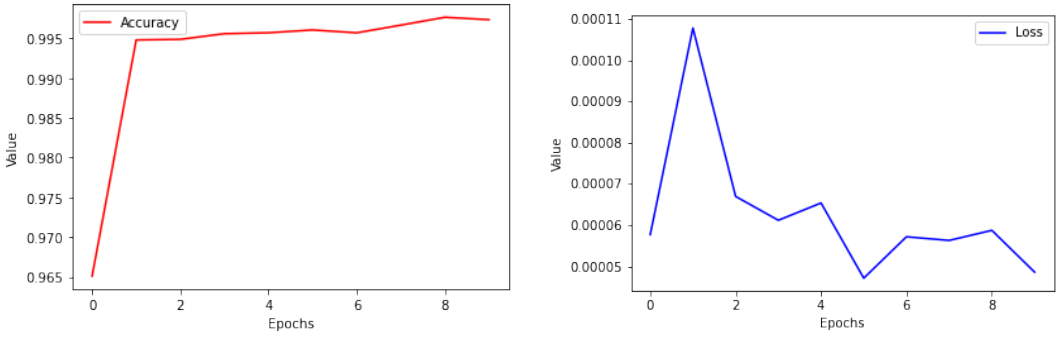
We sample around one thousand code snippets from SCAD and used about two thirds (645) for training and the remainder (333) for testing. For the lexical features our technique is very effective, reaching above 99% accuracy in less than 5 epochs, whereas on English, the tokenization takes almost 70 epochs to get around 97%. We also reduce the model size by a couple orders of magnitude (32 vs. 1024 dimensions for source code vs. English). This is likely because formal language is well structured so it is easy for the network to learn.

We sample and preprocess 4,774 train and 1,592 test Python samples and train our model for 50



(a) **(b)**

Figure 6.3: Tree output of (a) "def add(a,b) return a + b" we see that it thinks "a" is an article adjective, where as in (b) "def add(item_a,item_b) return item_a + item_b" we can use the "item_".



(a) **(b)**

Figure 6.4: Extracting the lexical tokens using POS methods (a) quickly reaches above 99% accuracy with (b) low loss.

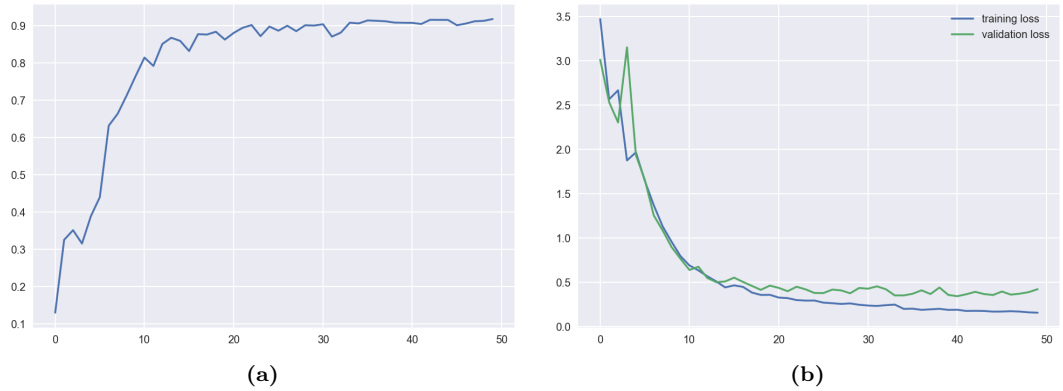


Figure 6.5: This shows the UAS (a) and the loss (b) for training and testing of the dependency parser on natural language.

epochs. We use the Unlabeled Attachment Score (UAS) which is the percentage of words that get the correct Parent_ID to evaluate our model at each epoch on the train and test data sets. This is the common way to evaluate the dependency parsing with the CONLL format. The test accuracy levels out at around 91% which is even a little better than how we do with the EWT. This model takes much longer to train than the POS model.

6.5 Discussion

There are several material observations that emerge from this work. As we pointed out earlier, we can actually use the English dependency parser on source code. We see that our choice of variable name impacts the interpretation of the structure of the code. The use of "item_a" instead of just "a" and makes the tree closer to the correct AST. Also, when we strip away the parentheses and comma, as we do when getting the lexical and AST tokens, the tree becomes flatter and further from the correct (ground truth) AST. Words that make the English parse tree similar to the AST version might be easier to read as a programmer. Variable names can be long and specific but can become cumbersome to write over and over again so people usually opt for short and simple ones. However, there can be more information stored inside these tokens and they could be further broken down into parts such as with the sentence piece tokens, since it is a subword tokenizer. And some parts of the code which are common could be grouped together such as a loops and common abbreviations

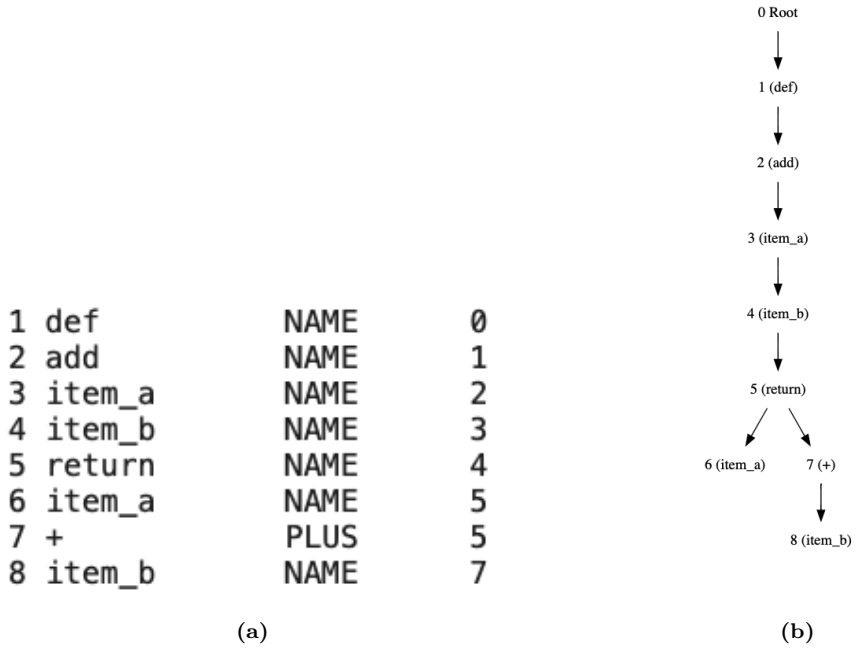


Figure 6.6: (a) shows the CONLL format for the code with the lexical tokens in place for the POS tokens (only two types) and the heads from the AST and in (b) we see that the AST representation for this simple function is much simpler than the NLP interpretation.

(e.g., import random as r). The best case scenario is that the surface level information aligns with the underlying functionality of the code.

The Python feature extractor could be improved for the AST to get more information about more of the tokens and keep some of the ones we choose to ignore. In future work we can use this information to improve some of the models we have worked with in the past such as the translation or paraphrase models. This is an area where you will do much better if you make it language specific, but the fact that it can run on plaintext makes it language agnostic. As we saw, we can interpret code as English and get some structural analysis of the code. We could try to remove pieces and see how our model would compensate for the missing parts. We should look to the NLP and fuzzy parsing literature on this point; however further development is beyond the scope of this chapter.

6.6 Conclusions

We have shown that we can infer the underlying structure of code in similar ways as we can with natural linguistics. We use an LSTM model based on POS extraction to recover the lexical tokens. This is similar to the segmentation model in Chapter 3 and is able to score over 98% accuracy with minimal data and training. We then abstract the code into the CONLL format using the lexical tokens and structural information from the AST. We train a dependency parser model which is able to infer the structure with a UAS of over 90%, on par with English equivalents. This shows that NLP methods for language parsing can be used to uncover underlying structures found in source code. This finding likely explains, to some extent, why NLP methods for segmentation, translation, etc., work well for source code as well.

Chapter 7: Conclusions

Throughout this thesis we have approached the medium of code with the tools of natural linguistics. Our objective was to see how far we can take NLP techniques and apply them to these specialized software problems. We found that several SCA tasks can be solved effectively with trained models and data designed for plain text. Moreover, we have presented a wide array of methods as they apply to code and tools that can be used by other developers to improve their work.

In this work we primarily focuse on the Python programming language because it is a popular and versatile language. We tackle problems of code analysis via attribution on code with standard NLP tools, which show high performance across several programming languages. Code classification for both behavior and author are very robust, and classifying plain text documents (in the form of tutorials) related to code appeared to do just as well with news article classification. We then used NLP methods to manipulate code to mine for correct code transformations, find ideal locations for comments within a program, generate comment annotations for code chunks, and infer the lexical token types and syntax tree structure of code. There are several tasks in which language agnostic models were effective, however others rely on specific rules governing how the language operates such as the tagging and paraphrasing. It is uncertain if language agnostic models work for generating comments. For example, a human reader may be able to make an educated guess about the code behavior based on surface level information and assuming the structure based on common traits between programming languages, but it will always be uncertain unless the true rules of the

Table 7.1: Application of NLP Techniques by Programming Language.

Method	Python	C++	Java	Multi
Attributing	Yes	Yes	Yes	Yes
Paraphrasing	No	Yes	No	No
Segmenting	Yes	Yes	Yes	Yes
Translating	Yes	No	No	No
Tagging	Yes	No	No	No

language are known, as embodied by the compiler. We found that for attribution and segmentation standard methods are quite apt. Other methods such as tagging and translation, advanced DL methods and tailoring had to be deployed in order to work effectively. Even in these cases, we were able to use NLP-inspired solutions to improve the state of the art.

7.1 Conclusions and contributions

Our conclusion is that many of the tools from NLP can be easily and effectively used in the context formal language. Our choice of source code was based on the ability to impact such a large community. Code analysis stands to gain the most when these approaches are applied to quality and security monitoring, especially to improve tools that make code easier to develop and maintain. For example, once a bug or vulnerability has been identified, can a system help suggest a solution? Tagging could become better when faced with adversarial conditions, for example, with the use of code transformation for exploring code mutation. We could also use these methods in combinations, for example creating an abstractive summary of a program. Our conjecture is that creating code summaries will speed up software development and maintenance time and reduce the number of bugs in code. We will help developers write better code by providing information and suggestions with innovative searching methods. There are many potential extensions to this work, and as both the fields of DL and NLP continue to expand, the more we can leverage them for purposes of analyzing and adapting code. Additionally, we stand to learn about our own use of language from processing code, and perhaps new ways of using language naturally will be adopted. For example if using the AST and lexical tokens for aiding in translation turns out to benefit the model, it is likely that we can use similar structural principles for processing text. The AST and lexical tokens can then also be inferred using POS and dependency parsing models.

We found that most tasks in NLP had some analog when applied to code, and many non-trivial tasks could be approached when interpreting the code as plaintext. Our conclusion is that NLP is an effective means to study code and related data for making high quality models and assessments of software. The following is the list of our research contribution in this work:

- Method for authorship and behavior attribution for Python, C++, and Java source code with the use of plaintext tokenizers for simple and efficient feature extraction.
- Method for discovering functionally equivalent source code transformations in C++ using techniques for paraphrases identification.
- Method for delineating between cohesive portions of Python, C++, and Java source code with the use of contextual elements as a way of selecting good places to insert comments.
- Method for automatically creating informative English annotations (e.g., comments) for blocks of Python source code using a translation approach.
- Method for determining the lexical and syntactic structure of Python source code, even if the source code is incomplete or incorrect (i.e., under construction).

7.2 Limitations and future work

This work has explored several avenues that accomplish code analysis tasks with NLP. However, just as with NLP, processing code can go through several iterations and be designed with many results in mind. Each of the tasks we explored had opportunities to be reapplied to related areas. For example, paraphrasing and tagging are largely secondary tasks that provide more information or augment the code in interesting ways. Paraphrasing could allow for perturbations aimed at standardization or optimization. These perturbations and structural information could help improve translation information or segmentation to improve paraphrasing or could apply to identifying vulnerabilities in the code. Translation itself could be applied in many other ways such as trying to generate code from descriptions or changing between languages and versions of programming languages. Similar techniques could be extended to recommend variable names or API usages. Since code is effectively bimodal, on the one hand written for humans and built to be processed by a machine, more advanced and yet intuitive tools should be developed. Standardized tools such as NLTK, Numpy, TensorFlow, etc. are all useful open source software that allow developers to communicate about their work in a way that is easy to be shared and developed. This could accelerate learning and make it easier to become a better developer.

Bibliography

[1] George Doddington. Automatic evaluation of machine translation quality using n-gram co-occurrence statistics. In *Proceedings of the second international conference on Human Language Technology Research*, pages 138–145, 2002.

[2] Pavol Bielik, Veselin Raychev, and Martin Vechev. Programming with" big code": Lessons, techniques and applications. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[3] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[4] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016.

[5] Michael D Ernst. Natural language is a programming language: Applying natural language processing to software development. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[6] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. Stackoverflow and github: Associations between software development and crowdsourced knowledge. In *2013 International Conference on Social Computing*, pages 188–195. IEEE, 2013.

[7] David Lorge Parnas. Software aging. In *Proceedings of 16th International Conference on Software Engineering*, pages 279–287. IEEE, 1994.

[8] Mohit Iyyer, John Wieting, Kevin Gimpel, and Luke Zettlemoyer. Adversarial example generation with syntactically controlled paraphrase networks. *arXiv preprint arXiv:1804.06059*, 2018.

[9] Joachim Wagner, Jennifer Foster, and Josef van Genabith. A comparative evaluation of deep and shallow approaches to the automatic detection of common grammatical errors. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 112–121, 2007.

[10] Adwait Ratnaparkhi. A maximum entropy model for part-of-speech tagging. In *Conference on empirical methods in natural language processing*, 1996.

[11] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. Neural architectures for named entity recognition. *arXiv preprint arXiv:1603.01360*, 2016.

[12] Vox media, workshop for data science + journalism (ds+j) 2017, howpublished = `https://data.world/elenadata/vox-articles`,.

[13] Code jam. URL `https://codingcompetitions.withgoogle.com/codejam`. Accessed: 2019-05-01.

[14] Johannes Kiesel, Maria Mestre, Rishabh Shukla, Emmanuel Vincent, Payam Adineh, David Corney, Benno Stein, and Martin Potthast. Semeval-2019 task 4: Hyperpartisan news detection. In *Proceedings of the 13th International Workshop on Semantic Evaluation*, pages 829–839, 2019.

[15] Dimitrios Bountouridis, Mónica Marrero, Nava Tintarev, and Claudia Hauff. Explaining credibility in news articles using cross-referencing. In *SIGIR workshop on ExplainAble Recommendation and Search (EARS)*, 2018.

[16] Junaed Younus Khan, Md Khondaker, Tawkat Islam, Anindya Iqbal, and Sadia Afroz. A benchmark study on machine learning methods for fake news detection. *arXiv preprint arXiv:1905.04749*, 2019.

[17] Stephen M Lee. Variation in political news. 2019.

[18] Yirey Suh, Jaemyung Yu, Jonghoon Mo, Leegu Song, and Cheongtag Kim. A comparison of oversampling methods on imbalanced topic classification of korean news articles. *Journal of Cognitive Science*, 18(4):391–437, 2017.

[19] Kamaldeep Kaur and Vishal Gupta. A survey of topic tracking techniques. *International Journal*, 2(5), 2012.

[20] James M Hughes, Dong Mao, Daniel N Rockmore, Yang Wang, and Qiang Wu. Empirical mode decomposition analysis for visual stylometry. *IEEE transactions on pattern analysis and machine intelligence*, 34(11):2147–2157, 2012.

[21] Michael Mara. Artist attribution via song lyrics, 2014.

[22] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 255–270, 2015.

[23] Hoshiladevi Ramnial, Shireen Panchoo, and Sameerchand Pudaruth. Authorship attribution using stylometry and machine learning techniques. In *Intelligent Systems Technologies and Applications*, pages 113–125. Springer, 2016.

[24] Ahmed Abbasi and Hsinchun Chen. Writeprints: A stylometric approach to identity-level identification and similarity detection in cyberspace. *ACM Transactions on Information Systems (TOIS)*, 26(2):1–29, 2008.

[25] Chris Quirk, Chris Brockett, and William Dolan. Monolingual machine translation for paraphrase generation. In *Proceedings of the 2004 conference on empirical methods in natural language processing*, pages 142–149, 2004.

[26] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[27] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[28] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[29] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, 2013.

[30] Bill Dolan, Chris Quirk, and Chris Brockett. Unsupervised construction of large paraphrase corpora: Exploiting massively parallel news sources. In *Proceedings of the 20th international conference on Computational Linguistics*, page 350. Association for Computational Linguistics, 2004.

[31] Rakshith Shetty, Bernt Schiele, and Mario Fritz. A4nt: author attribute anonymity by adversarial training of neural machine translation. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1633–1650, 2018.

[32] Sunita N. Deore and Manisha N. Sawant. Formal and natural languages: The difference analysis. volume 7, pages 135–138, 2018.

[33] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. Large-scale and language-oblivious code authorship identification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 101–114. ACM, 2018.

[34] Erwin Quiring, Alwin Maier, and Konrad Rieck. Misleading authorship attribution of source code using adversarial learning. *arXiv preprint arXiv:1905.12386*, 2019.

[35] Regina Barzilay and Kathleen R McKeown. Extracting paraphrases from a parallel corpus. In *Proceedings of the 39th annual meeting of the Association for Computational Linguistics*, pages 50–57, 2001.

[36] Ali Ibrahim, Boris Katz, and Jimmy Lin. Extracting structural paraphrases from aligned monolingual corpora. In *Proceedings of the second international workshop on Paraphrasing-Volume 16*, pages 57–64. Association for Computational Linguistics, 2003.

[37] Bo Pang, Kevin Knight, and Daniel Marcu. Syntax-based alignment of multiple translations: Extracting paraphrases and generating new sentences. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 102–109. Association for Computational Linguistics, 2003.

[38] Jacob Dormuth, Ben Gelman, Jessica Moore, and David Slater. Logical segmentation of source code. *arXiv preprint arXiv:1907.08615*, 2019.

[39] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*, 2018.

[40] Ben Gelman, Banjo Obayomi, Jessica Moore, and David Slater. Source code analysis dataset. *Data in brief*, 27:104712, 2019.

[41] Noam Mor, Omri Koshorek, Adir Cohen, and Michael Rotman. Learning text segmentation using deep lstm. 2017.

[42] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

[43] Kanchan M Tarwani and Swathi Edem. Survey on recurrent neural network in natural language processing. *Int. J. Eng. Trends Technol*, 48:301–304, 2017.

[44] V Preethi et al. Survey on text transformation using bi-lstm in natural language processing with text data. *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, 12 (9):2577–2585, 2021.

[45] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2073–2083, Berlin, Germany, August 2016. Association for Computational Linguistics. doi: 10.18653/v1/P16-1195. URL `https://www.aclweb.org/anthology/P16-1195`.

[46] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[47] Nal Kalchbrenner and Phil Blunsom. Recurrent convolutional neural networks for discourse compositionality. *arXiv preprint arXiv:1306.3584*, 2013.

[48] Ralf C Staudemeyer and Eric Rothstein Morris. Understanding lstm–a tutorial into long short-term memory recurrent neural networks. *arXiv preprint arXiv:1909.09586*, 2019.

[49] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[50] Daniel Stein. Machine translation: Past, present and future. *Language technologies for a multilingual Europe*, 4(5), 2018.

[51] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. *arXiv:1409.3215 [cs]*, December 2014. URL `http://arxiv.org/abs/1409.3215`. arXiv: 1409.3215.

[52] Davide Castelvecchi. Deep learning boosts google translate tool. *Nature News*, 2016.

[53] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407, 2018.

[54] Dianqi Li, Yizhe Zhang, Zhe Gan, Yu Cheng, Chris Brockett, Ming-Ting Sun, and Bill Dolan. Domain adaptive text style transfer. *arXiv preprint arXiv:1908.09395*, 2019.

[55] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[56] Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

[57] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, Pennsylvania, USA, July 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL `https://www.aclweb.org/anthology/P02-1040`.

[58] Maja Popović. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, Lisbon, Portugal, September 2015. Association for Computational Linguistics. doi: 10.18653/v1/W15-3049. URL `https://www.aclweb.org/anthology/W15-3049`.

[59] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL `http://arxiv.org/abs/1609.08144`.

[60] Alon Lavie and Abhaya Agarwal. METEOR: An automatic metric for MT evaluation with high levels of correlation with human judgments. In *Proceedings of the Second Workshop on Statistical Machine Translation*, pages 228–231, Prague, Czech Republic, June 2007. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/W07-0734`.

[61] Hideki Isozaki, Tsutomu Hirao, Kevin Duh, Katsuhito Sudoh, and Hajime Tsukada. Automatic evaluation of translation quality for distant language pairs. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 944–952, 2010.

[62] Carlo Strapparava, Alessandro Valitutti, et al. Wordnet affect: an affective extension of wordnet. In *Lrec*, volume 4, page 40. Citeseer, 2004.

[63] Samir Gupta, Sana Malik, Lori Pollock, and K Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 3–12. IEEE, 2013.

[64] Sofonias Yitagesu, Xiaowang Zhang, Zhiyong Feng, Xiaohong Li, and Zhenchang Xing. Automatic part-of-speech tagging for security vulnerability descriptions. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 29–40. IEEE, 2021.

[65] Timothy Dozat and Christopher D Manning. Deep biaffine attention for neural dependency parsing. *arXiv preprint arXiv:1611.01734*, 2016.

[66] Xavier Lluís, Xavier Carreras, and Lluís Màrquez. Joint arc-factored parsing of syntactic and semantic dependencies. *Transactions of the Association for Computational Linguistics*, 1:219–230, 2013.

[67] Jan Niehues and Eunah Cho. Exploiting linguistic resources for neural machine translation using multi-task learning. *arXiv preprint arXiv:1708.00993*, 2017.

[68] Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Using Large Corpora*, page 273, 1994.

[69] Sabine Buchholz and Erwin Marsi. Conll-x shared task on multilingual dependency parsing. In *Proceedings of the tenth conference on computational natural language learning (CoNLL-X)*, pages 149–164, 2006.

[70] Timothy Dozat and Christopher D Manning. Deep biaffine attention for neural dependency parsing. *arXiv preprint arXiv:1611.01734*, 2016.

[71] Natalia Silveira, Timothy Dozat, Marie-Catherine De Marneffe, Samuel Bowman, Miriam Connor, John Bauer, and Christopher D Manning. A gold standard dependency corpus for english. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*, pages 2897–2904, 2014.

Vita

Vita

Aviel J. Stein

E-mail: ajs568@drexel.edu
Phone: (717) 585-9330

EDUCATION

Drexel University, College of Computing and Informatics, Philadelphia, PA **May 2022**
Ph.D. in Computer Science

Drexel University, College of Engineering, Philadelphia, PA **May 2019**
MS in Cybersecurity

Susquehanna University, Selinsgrove, PA **May 2017**
BS in Computer Science
Minors in Math and Physics

RESEARCH EXPERIENCE

Source Code Analysis Aug 2020– Present
Drexel University, College of Computing and Informatics, Philadelphia, PA
- conducting novel experiments deep learning and machine learning algorithms
- Designing and creating AI and decision tools for developers and security experts.
- Communicating ideas with colleges at the university and global researchers at conferences by writing reports, papers, presentations, posters, and videos.
- Presented findings at International Conference on Natural Language Processing (NLP20), IEEE International Conference on Semantic Computing (ICSC'22),

Privacy, Security, and Automation Laboratory (PSAL) Aug 2017 – Aug 2020
Drexel University, College of Computing and Informatics, Philadelphia, PA
- Collaborated with the Army Research Laboratory (ARL) and Defense Advanced Research Projects Agency (DARPA) to develop software analysis algorithms for authorship attribution and function identification
- Created system for detecting social media manipulation by identifying behaviors of individuals in influence pods.
- Presented findings related at Distributed System Security Symposium (NDSS18), the Web Conference (WWW20), and International Conference on Semantic Computing (ICSC20)

Mathematical Sciences Department Sep 2016 – May 2017
Susquehanna University, Selinsgrove PA
- Initiated research project between Computer Science & Biology departments that resulted in software usage in curricula.
- Interpreted technical documents about abdominal aortic aneurysms and ran statistical analyses to find trends among variables.
- Located and modified existing EKG anomaly detection software as part of departmental honors program.
- Analyzed the program's efficacy using the CinC Challenge 2000 data set on Physio Net for departmental presentation.

National Science Foundation Projects May 2015 – Aug 2016
Susquehanna University, Selinsgrove PA
- Acted as principal investigator on two research projects related to computer vision (2015) and machine learning (2016).
- Designed and conducted experiments using generated data from MATLAB programs to create point light displays and visual disturbances.
- Used hyperspectral bathymetry data related to 61 wavelengths to estimate the depth of shallow water using TensorFlow which achieved better degree of accuracy than related analytic models.
- Presented results at the Landmark Symposium, Susquehanna Valley Symposium, and National Conferences on Undergraduate Research (NCUR) both years.
- Created a website as a repository for the codes, posters, papers, and biographies for fellow researchers.

TEACHING EXPERENCE

Mathematical Sciences Department | Teaching Assistant Jan 2015 – May 2017
Susquehanna University, Selinsgrove PA
- Assisted 25-30 students in each of the following courses: Introduction to Programming; Introduction to Statistics; Data structures; Principles of Computer Science.
- Aided individual students during class and conduct review sessions prior to exams.
- Reviewed student work to ascertain areas of weakness and provide tutoring and mentoring directed to such areas.

University Innovation Fellow Aug 2016 – Present
Stanford University, Stanford, CA
- Trained in lean startup and design thinking methodologies, and engage in annual meetup at Stanford
- Analyzed Susquehanna University's campus entrepreneurial and innovative ecosystem
- Founded entrepreneurial organization at Susquehanna, "DICE (Design. Innovation. Creativity. Entrepreneurship.)," to facilitate initiatives on campus such as developing a makerspace, hosting DICE+ (a TEDx style event) and collaborating with the health center, athletic center and counseling center regarding innovative health strategies

WORK EXPERENCE

Information Systems and Science Laboratory | AI research Scientist Jul-Sep 2021
HRL Laboratories, Malibu CA
- Distill massive data from multiple intelligence sources to provide fast and explainable recommendations to aid command decision-making for Naval command center personnel
- Utilized cutting-edge algorithms for unsupervised computer vision experiments for visual causality
- Presented regular internal updates, prepared manuscript for publication, reports for intellectual property, and documentation for decimation for use within the US Navy

Ayyeka Technologies | Business Development Intern Jun 2014 – Aug 2014
Hebrew University, Jerusalem Israel
- Discovered and wrote applications for grants, collaborations, and projects related to remote machine-to-machine water monitoring and international expansion.
- Researched field related data and synthesized findings into comprehensive packages for purposes of promoting on social media.

- Developed crucial elements of the business plan for the purposes of growing into an international company.

Residence Life | Resident Assistant Sep 2015 – May 2017
Susquehanna University, Selinsgrove PA
- Developed a since of community and academic rigor by creating and promoting events monthly
- Meet with students and handled sensitive and private information in high emotional situations
- Maintained building with 30 - 150 students and inspected grounds and facilities and scheduled regular repairs when needed

Career Development Center | Career Ambassador Sep 2014 – May 2017
Susquehanna University, Selinsgrove PA
- Meet regularly with team of 6-9 other students to discuss strategies for working with students.
- Reviewed and aided students with their resumes, cover letters, and interviewing skills.
- Collaborated with university partners to create actionable marketing campaigns for 50+ career events annually including three-day career conference with 500+ students and 100+ alumni in attendance.

SKILLS & TOOLS

Microsoft Office Suite, Java, JavaScript, Python, C, C++, R, MIPS, HTML, CSS, MATLAB, Netlogo, SQL, PHP, TensorFlow, ScikitLearn, PyTorch, AWS-EC2, OpenCV

LEADERSHIP & MEMBERSHIP

ALD Honor Society 2014 – Present
Phi Mu Epsilon Honor Society 2016 – Present
Student Conduct Board 2014 – 2017
Society of Physics Students 2014 – 2017
LeaderShape 2014 – 2017
Student Activities Committee 2013 – 2015
Hillel | Vice-President 2015 – 201