

James Madison University

## JMU Scholarly Commons

---

Masters Theses, 2020-current

The Graduate School

---

5-8-2020

### A multi-input deep learning model for C/C++ source code attribution

Richard Tindell

Follow this and additional works at: <https://commons.lib.jmu.edu/masters202029>



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

---

#### Recommended Citation

Tindell, Richard, "A multi-input deep learning model for C/C++ source code attribution" (2020). *Masters Theses, 2020-current*. 46.

<https://commons.lib.jmu.edu/masters202029/46>

This Thesis is brought to you for free and open access by the The Graduate School at JMU Scholarly Commons. It has been accepted for inclusion in Masters Theses, 2020-current by an authorized administrator of JMU Scholarly Commons. For more information, please contact [dc\\_admin@jmu.edu](mailto:dc_admin@jmu.edu).

A Multi-Input Deep Learning Model for C/C++ Source Code Attribution

Richard Jeffrey Tindell II

A thesis submitted to the Graduate Faculty of

JAMES MADISON UNIVERSITY

In

Partial Fulfillment of the Requirements

for the degree of

Master of Science

Department of Computer Science

May 2020

---

FACULTY COMMITTEE:

Committee Chair: Dr. Xunhua Wang

Committee Members/ Readers:

Dr. Nathan Sprague

Dr. Brett Tjaden

## **Dedication**

This work is dedicated to my children William, Theodore, and Charles. Thanks for interrupting my work for silly things.

To my dad for gifting me with your love of learning and skill in computers.

To my wife. Thank you for your patience, support, and love during this long process. I could not have done this without you.

Love you all.

## Acknowledgments

First, I would like to thank Dr. Xunhua Wang for serving as both my adviser and thesis committee chair during this process. Thank you for expanding your knowledge base and learning something new with me. I would also like to thank Dr. Nathan Sprague and Dr. Brett Tjaden for serving on my thesis committee.

Finally, I would also like to thank Jonathan Fitch, Lynn Carr, and Matthew Wotring for their proof-reading skills. Thank you for all the effort you put into making this the best paper it can be.

## Table of Contents

<b>DEDICATION.....</b>	<b>II</b>
<b>ACKNOWLEDGMENTS .....</b>	<b>III</b>
<b>TABLE OF CONTENTS .....</b>	<b>IV</b>
<b>LIST OF FIGURES.....</b>	<b>VI</b>
<b>ABSTRACT .....</b>	<b>VIII</b>
<b>CHAPTER 1 INTRODUCTION .....</b>	<b>1</b>
OVERVIEW .....	1
PROBLEM STATEMENT.....	3
CONTRIBUTIONS .....	3
ORGANIZATION .....	3
<b>CHAPTER 2 BACKGROUND INFORMATION AND EXISTING RESEARCH .....</b>	<b>4</b>
TOOLSET AND BASIC DEFINITIONS .....	4
DEEP LEARNING.....	4
MANUAL STYLOMETRY TECHNIQUES.....	9
SOURCE CODE STYLOMETRY USING TRADITIONAL MACHINE LEARNING .....	10
EXISTING RESEARCH'S INFLUENCE ON THIS RESEARCH .....	11
BINARY SOURCE ATTRIBUTION .....	12
<b>CHAPTER 3 PREPROCESSING AND CREATING THE DATA MODEL .....</b>	<b>13</b>
PROBLEM STATEMENT.....	13
INPUT DATA.....	13
UNKNOWN DATA .....	15
INITIAL NAÏVE APPROACH .....	15
REVISED APPROACH.....	16
CONSTRUCTING THE MODEL.....	18
TRAINING THE MODEL .....	19
OTHER VARIATIONS OF THE MODEL .....	20
TEST DATA.....	25
RANDOM FOREST COMPARISON.....	26
<b>CHAPTER 4 PREDICTIONS USING THE MODEL.....</b>	<b>27</b>
OLDER VERSION OF OPENSLL.....	29
OTHER PROJECT PREDICTIONS .....	31

<b>CHAPTER 5 CONCLUSIONS.....</b>	<b>32</b>
AREAS OF FURTHER RESEARCH .....	32
<b>APPENDIX A CODE FOR PREPROCESSOR.PY .....</b>	<b>34</b>
<b>APPENDIX B CODE FOR CREATE_MODEL.PY .....</b>	<b>38</b>
<b>APPENDIX C CODE FOR UTILS.PY .....</b>	<b>42</b>
<b>APPENDIX D CODE FOR PREDICT_BITCOIN.PY .....</b>	<b>45</b>
<b>APPENDIX E TABLE OF KEYWORDS ADDED TO STOPWORDS .....</b>	<b>47</b>
<b>APPENDIX F LIST OF INPUT PROJECTS .....</b>	<b>48</b>
<b>BIBLIOGRAPHY .....</b>	<b>49</b>

## List of Figures

Figure 1 - AI Relationships. ....	5
Figure 2 - Recurrent Network .....	8
Figure 3 - Distribution of Files.....	14
Figure 4 - Distribution of Lines of Code .....	14
Figure 5 - Pandas Dataframe with Columns Array. ....	16
Figure 6 - OOV Token Table.....	17
Figure 7 - Multi-Input Model with Multiple Hidden Layers.....	18
Figure 8 - Multi-Input Loss Values throughout Epochs. ....	19
Figure 9 - Multi-Input Accuracy Values throughout Epochs. ....	20
Figure 10 - Accuracy at 96% for the training and 90% for the pre-split test data.....	20
Figure 11 - Just Source Accuracy. ....	21
Figure 12 - Just Source Loss. ....	21
Figure 13 - Just Features Accuracy. ....	22
Figure 14 - Just Features Loss. ....	22
Figure 15 - Just Comments Accuracy.....	23
Figure 16 - Just Comments Loss .....	23
Figure 17 - Multi-Input Model Without Source Text. ....	25
Figure 18 - Cryptocpp Files Matching Table.....	26
Figure 19 - File Predictions.....	28
Figure 20 - Prediction Summary.....	28
Figure 21 - File Predictions with older OpenSSL.....	29
Figure 22 - Prediction Summary with older OpenSSL. ....	30

Figure 23 - Distribution of Lines of Code with Older OpenSSL.....30

Figure 24 - New Input Data Matches.....31



## **Abstract**

Code stylometry is applying analysis techniques to a collection of source code or binaries to determine variations in style. The variations extracted are often used to identify the author of the text or to differentiate one piece from another.

In this research, we were able to create a multi-input deep learning model that could accurately categorize and group code from multiple projects. The deep learning model took as input word-based tokenization for code comments, character-based tokenization for the source code text, and the metadata features described by A. Caliskan-Islam et al. Using these three inputs, we were able to achieve 90% validation accuracy with a loss value of 0.1203 using 12 projects consisting of 5,877 files. Finally, we analyzed the Bitcoin source code using our data model showing a high probability match to the OpenSSL project.

**Keywords:** stylometry, source code attribution, deep learning

## Chapter 1

### Introduction

#### *Overview*

Being able to determine who wrote a piece of code can be an important step in analyzing source code. Scholastically, it can be used to detect plagiarism in the computer science department. In the computer security field, knowing who wrote a piece of malicious code could potentially determine the security posture for a defending organization. For example, knowing that a threat actor is state sponsored may compel an organization to seek help from a law enforcement agency while suspecting an insider threat may elicit a more internal response. Stylometry is “the statistical analysis of variations in literary style between one writer or genre and another” [1]. This can be a very arduous process to do by hand, and many have used computing models to aid with this. Human languages are a little easier to analyze in this regard. When writing, the author has a large pool of words and letters to choose from. The author’s choices ultimately reflect a bit of who he is. The more that author writes, the more likely it is that a pattern or indicator of authorship will manifest.

Stylometry is an important part of an investigation where authorship is central. In the case of WikiLeaks, the central figures were alluding that they had a large number of volunteers all with the same cause. This projected strength but the truth was far from this. Daniel Domscheit-Berg made the statement that if WikiLeaks were subjected to stylometric analysis, it would become apparent that all of their press releases were written by one of two people, himself and Julian Assange [13].

Using techniques in deep learning, it is possible to accurately identify the author of a piece of code. The source code contains clues—pieces of the author within every choice of word, variable name, and code style. Something as simple as whether a brace symbol ( { ) appears on the same line as a control structure such as an if-then block or instead on a new line could help determine the author. Style choices, in addition to word choices in the code comments or variable names in the source code, can be transformed into numeric values to be used as input in a deep learning model. The model can then classify new input code based on what was learned from previous training data.

The modern process of using machine learning techniques are usually fairly similar. A preprocessor will extract various features and convert these features into a statistical value. This extracted metadata can achieve a high level of accuracy and may be the only option when analyzing binaries; but when source code is available, analyzing the comment text with a word-based tokenizer and the source-code text with a character-based tokenizer can vastly improve accuracy.

The basic process for any deep learning application is performing any required preprocessing, training the data model, and testing the model. In our approach, we extract three separate inputs during the preprocessing phase, apply natural language processing to comments, tokenize the source code text using character-based encoding, and extract metadata features from the source code. The inputs are passed into a deep learning model where multiple layers help classify a given file. These three separate inputs provide the highest accuracy and the lowest loss value of all methods tested.

### *Problem Statement*

This thesis research paper aims to answer the following two questions:

1. Can deep learning be used to accurately determine the author of source code?
2. What contributes the most to author attribution in source code: comments, source code text, or an abstracted feature list?

### *Contributions*

The results of this thesis research paper are

1. Deep learning is an effective tool when identifying the authors of source code using comments, source code text, and abstracted feature lists.
2. Comments tend to leave the most amount of author evidence.
3. Source code text leaves the least amount of author evidence.

### *Organization*

The remainder of this thesis is organized as follows. Chapter 2 gives an overview of existing research and background information on deep learning and source code attribution. Chapter 3 discusses our approach in preprocessing and model creation. In Chapter 4, we use the model to predict the similarity of an unknown code sample. Finally, we summarize our findings in Chapter 5.

## Chapter 2

### Background Information and Existing Research

The specific topic of stylometry with deep learning does not appear to be a widely published topic. There is existing research into manual stylometry techniques and even techniques using machine learning, but not all machine learning is deep learning.

#### *Toolset and Basic Definitions*

The research represented in this paper utilizes the Python programming language and the Keras and TensorFlow frameworks. A tensor is a dynamic, n-dimensional matrix; that is, if the force or weight of a specific value changes, the rest of the tensor must change relative to the transformation. These tensors are the basic unit of data in this deep learning application. While the tool TensorFlow allows for direct manipulation of a tensor object, Keras provides a high-level API built on top of TensorFlow to make working with tensors and other machine learning objects much easier.

#### *Deep Learning*

### **Machine Learning vs Deep Learning**

An important distinction to make is the one between machine learning and deep learning. Deep learning is a subset of machine learning that attempts to turn raw data into useable information.

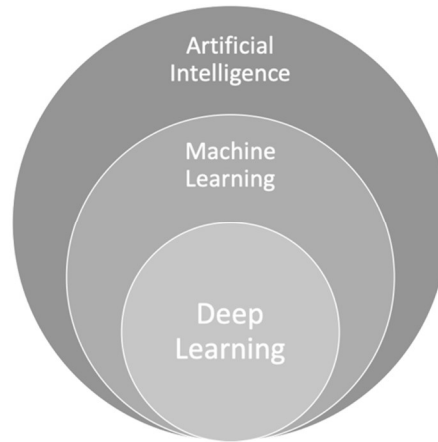


Figure 1 - AI Relationships.

One of the key advantages of deep learning when comparing it to other types artificial intelligence is outlined in “Deep learning” by Y. LeCun, Y. Bengio, and G. Hinton [2]:

Conventional machine-learning techniques were limited in their ability to process natural data in their raw form. ... Deep-learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but non-linear modules that each transform the representation at one level (starting with raw input) into a representation at a higher, slightly more abstract level. (p. 463)

While traditional machine learning would require the data to be in a structured format, the goal of the deep learning subset is to input raw data without having to heavily preprocess it into a structured format. Different layers are implemented to extract and group pertinent information [3]. A neural network is a deep neural network if it contains two or more hidden layers within the network. The output of one layer becomes the input of a subsequent layer. Each layer can be of a various type including input, convolutional, sequence, normalization, pooling, combination, and output layers.

## Layer Types and Other Important Terms

Data shape is how a data model is represented. It is tied to the number of inputs and outputs of a given layer. One way to limit the number of inputs would be through preprocessing the data. If the input data is too abstract, preprocessing can reshape it to focus on only what is important in the data.

Embedded layers transform “positive indexes into dense vectors of fixed size” [4]. Because much of this project deals with language processing, this layer is essential. By converting the input data (words or characters in our case) into a smaller dense vector, the process time should be dramatically faster.

Dense layers are also called “fully connected layers” because every “cell” or “neuron” within the neural network is connected. These layers take inputs of a specified shape and produce outputs of a different specified shape. In addition to the input and output units, these layers can have different activation functions. The activation function acts to produce a weighted output for a given input in the layer using a mathematical equation as a gate. One of the simplest activation functions is a linear activation, which allows multiple inputs to be mapped to multiple outputs through a linear equation. This project uses dense layers with both rectified linear unit (ReLU) and softmax activations.

ReLU activation is similar to a linear activation function, except that all negative input values are outputted as zeros. This is generally much faster, as fewer subsequent neurons are firing, and it allows for back propagation. This back propagation is used to calculate weights of specific outputs for use throughout the neural network.

Softmax activation is used to produce an output of probabilities. These probabilities correspond to the likelihood an input can be placed into a specific category of a number of outputs. In a classification problem, this is typically the last step since the output corresponds

to the various categories. The output probabilities all sum to the value of 1, so this output is not only able to categorize inputs but also to give a ranked categorization.

The benefit of using deep learning is that a computer is able to apply different weights to different data transformations and adjust these weights based on outputs of the loss function and an optimizer. The loss function calculates the distance between an output and what was expected. The output of this function is called the “loss score” or “loss value” and is used with an optimizer to adjust the weights of another layer. Categorical cross-entropy is the loss function utilized in this project, and it is also called “softmax loss” because it combines a softmax activation and a cross-entropy loss. Cross-entropy loss is used for probability applications which makes it a great choice for our final output [3, page 73].

The loss function feeds the loss score into an optimizer. RMSprop is the optimizer used in this project. It uses a moving average over the root mean squared (RMS) [5]. It is useful for training very large datasets, as it is fast. Our dataset is large enough to see a benefit by using this type of optimizer.

## **Types of Input Data**

Training a machine learning model requires data of three different types—training, validation, and test. The training data is somewhat self-explanatory, it is used to train the data model. The important thing about selecting this data is making sure there is enough training data to teach the model about a specific set of characteristics.

Validation data is the data used by the model to verify the training values and adjust them during the training process. It is important to contrast this data with test data which is only used once the model is complete. Validation data is sometimes a subset of training data used to adjust weights. Test data is used afterward to verify the accuracy of the model created.



## Recurrent Neural Networks

A layer is described as recurrent if it needs to have a memory of previous inputs or states. A recurrent neural network (RNN) attempts to mimic the way a biological lifeform learns by keeping some track of state while processing a larger body of information. These layers are necessary when a specific piece of data cannot be processed in isolation and are used quite extensively on text data, speech data, and classification problems. The basic features behind an RNN are its use of a loop and its ability to keep track of state.

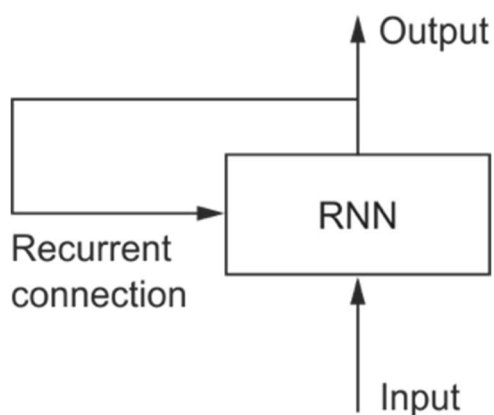


Figure 2 - Recurrent Network. Reprinted from Deep Learning with Python (196), by François Chollet, 2018, Manning Publications. Copyright 2018 by Manning Publications Co. Reprinted with permission.

Long Short-Term Memory or LSTM is a common recurrent layer and is one of the layers used in this project. It is a type of RNN that is used to compensate for the “vanishing gradient problem.” When a neural network becomes too large, the weight of each layer will be difficult to change; that is, the gradient will become too small. The output will be unable to change regardless of the new data [3, page 202]. LSTM compensates for the vanishing gradient problem by saving some information for later use which helps to prevent old signals from having no effect on the current output. Being an RNN, it also has a “memory” of what has been passed into it.

## Supervised vs Unsupervised

Another way to classify a machine learning neural network is by calling it “supervised” or “unsupervised.” In a supervised neural network, data is mapped to a known output. This is particularly useful for classification and regression or when the output is of a known type [3, page 94]. Unsupervised machine learning is used in data visualization problems. Usually this technique is applied to the data set to understand the data better, after which a supervised machine learning technique or traditional programming can be applied. This project is classified as supervised machine learning.

### *Manual Stylometry Techniques*

Deep learning is merely a technique applied to a problem. It is a means to an end, and for this project, the end is code attribution via stylometry. Stylometry has been around in some form since 1890 when the basics were published in a book entitled *Principes de Stylométrie*. Using a manual technique attempts to identify indicators of authorship within a text by examining the following features [12]:

- Word length
- Sentence length
- Paragraph length
- Punctuation
- Function words
- Letters
- N-grams, bigrams, trigrams (characters in a row)
- Bi-words and Tri-words (two or three words occurring in a certain order)

Every author chooses different words to convey meaning. Even if two authors were writing the same basic prose, their choice of words could reveal who they are. It is necessary

for both manual and programmatical stylometry to obtain large amounts of sample text in order to establish this pattern.

The work represented in Burrows (2010)[14] and Kalgutkar[15] make for an excellent survey in source code attribution. Burrows implements traditional statistical analysis of n-grams and stylistic features. The outcome is an accuracy of 78.86% for single authorship (p. 131). Kalgutkar outlines a brief history to authorship attribution and mentions a number of possible features including which type of control loop a code author employs. This particular paper outlines existing manual techniques and presents a comparative summary in this field.

### *Source Code Stylometry Using Traditional Machine Learning*

There have been several attempts at stylometry for source code using traditional machine learning techniques. These all generally follow a similar process: feature extraction, mapping features to the code samples, and classification usually through a decision tree. The features extracted from source code can be grouped into one of three categories: lexical, layout, or syntactic features [8].

### **Lexical Features**

The lexical features extracted from source code are similar to those in natural languages. These include things like unigram frequency, keyword usage, number of comments, number of input parameters for functions, and unigram location [6, page 258] [8, page 5]. The applications studied took the values of frequency and location of the various features and applied different averaging and logarithmic functions to them to produce a numeric value for these features as input into a machine learning algorithm.

## Syntactic and Layout Features

Layout features of source code have more to do with the style of the code itself rather than the words selected. Things like number of empty lines, whether tabs or spaces were used, and whether a curly brace appears on the same line or next line of a block of code are all syntactic features in source code. Syntactic features are extracted through an abstract syntax tree, which is also created for every function [6, page 259]. This is accomplished by essentially compiling the application. The tree provides useful information such as the maximum depth of an AST node, frequency of language keywords, and how much of the code is in a branch vs a leaf in the tree.

### *Existing Research's Influence on this Research*

For this project, we utilize much of the lexical and layout features. None of the syntactic features were used, as the abstract syntax tree could not be generated reliably from our Python application. The following features from the CHLNVYG15 paper were used (see Appendix A):

- Lexical Features
  - ln\_keyword\_length
  - ln\_unique\_keyword\_length
  - ln\_token\_length
  - avg\_line\_length
- Layout Features
  - ln\_tabs\_length
  - ln\_space\_length
  - white\_space\_ratio
  - is\_brace\_on\_new\_line
  - do\_tabs\_lead\_lines

Using these features with other inputs allows for a high level of accuracy. With a large corpus, they were able to classify 1,600 authors at 94% accuracy and 250 authors at 98% accuracy.

The problem of code authorship attribution has been addressed in a number of other papers with varying methodologies. In a paper by Junfeng Wang, et al. [16], a program dependence graph methodology is proposed. They represent data dependencies within an application for both data and control features. This method emphasizes how data flows within the control statements rather than the stylistic features of an application.

#### *Binary Source Attribution*

Finally, study has been done into binary attribution; that is, identifying the author of a software program that is already compiled with no access to the source code. The approach taken in RZM11 was to first create a control flow graph and instruction sequence so that features could be extracted. These features are the inputs into a machine learning model used to group similar groups of code. The results were 81% accuracy for ten authors and 51% accuracy for 200 authors [7]. While our research analyzes binaries rather than source code, the extracted features and the approach taken suggest a good pattern to follow even if the input data differs.

## Chapter 3

### Preprocessing and Creating the Data Model

In this chapter, we describe the model we wish to create and how we will create it including the important preprocessing step. We will start by examining the input data used to create the model.

#### *Problem Statement*

Given a C++ cryptography project with an unknown origin, can we determine who wrote the source code or perhaps what code most resembles this code, giving clues to the authorship of the new project?

#### *Input Data*

To prove the concept, it is important to limit the type and scope of input data. For our research, we selected eleven C/C++ projects of similar, closely related projects. Additionally, the input data is limited to source files only. While readme text files and markdown files might aid in authorship attribution, our original problem statement deals with source code only. In all, this encompasses 5,877 different files. For a full list of which projects were selected and where to find them, see Appendix F.

The number of files is not distributed equally. About 64% of the files are found in the OpenSSL, NSS, and Botan projects. If the distribution is calculated by number of lines of code rather than number of files, the OpenSSL project is no longer the most probable project. Even though the distribution is not equal, this doesn't really affect the probability of selecting the correct project at random. If a random guesser knew the percentage distribution, with no other

information, this guesser would have no reason to guess any project other than the one with the highest probability. This would establish a baseline of 22.1% to 26.1% for a random guess if the distribution was known, depending on the distribution model.

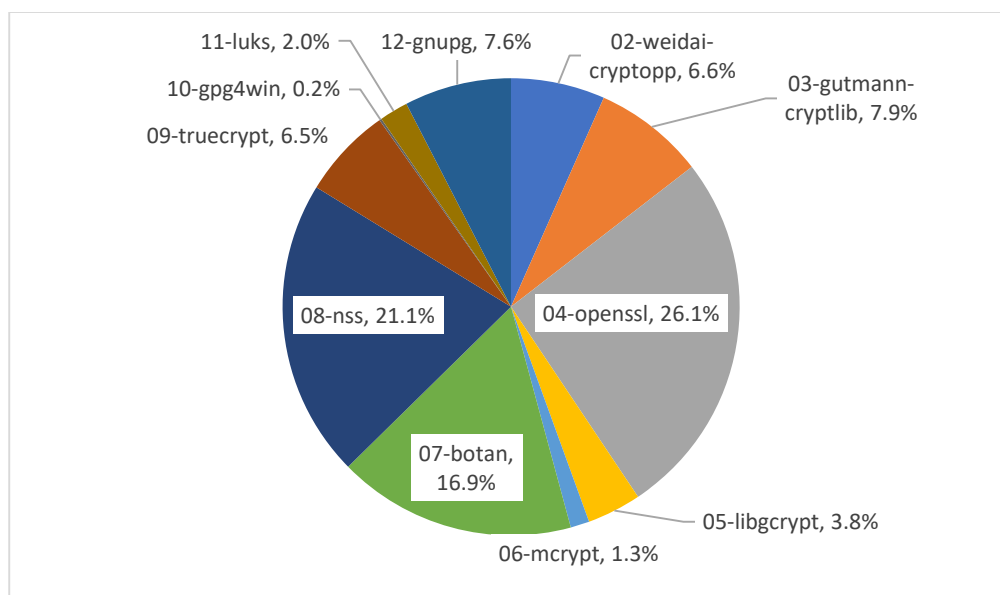


Figure 3 - Distribution of Files

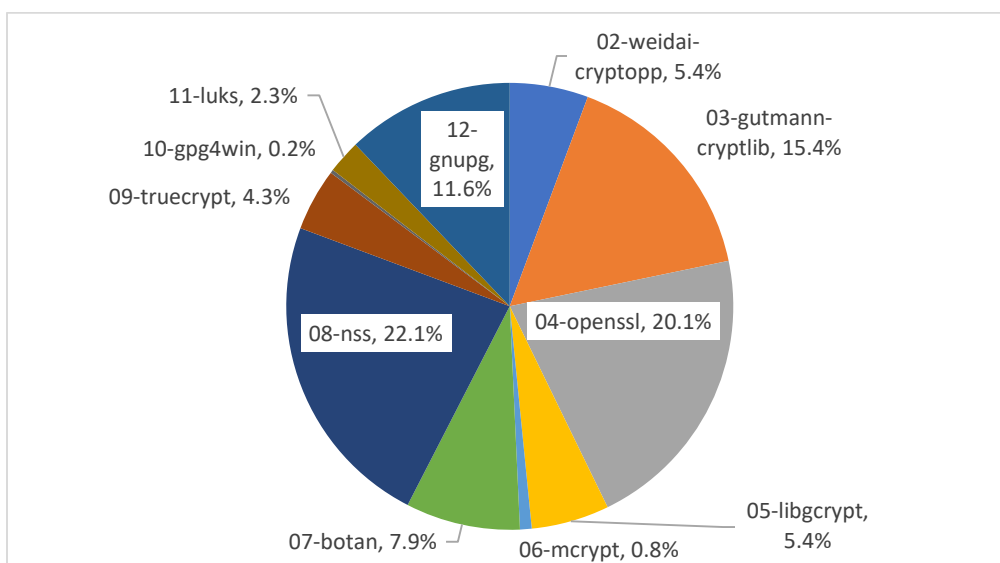


Figure 4 - Distribution of Lines of Code

### *Unknown Data*

To further test the data model, we selected source code with an unknown author, the Bitcoin source code. We selected the earliest version of this available to us, v0.01 ALPHA. The code is attributed to Satoshi Nakamoto, but many believe this to be a pseudonym. In addition, there are enough lines of code in this project to make it a viable data source. The unknown author and sizable code base make this an interesting project to analyze.

A pre-requisite for the input data was that there had to be enough data in the sample to produce a reliable result. Hal Finney, a programmer who some think could be the author of the Bitcoin source code, was also considered for an input to this project. Unfortunately, the only data source available written by Hal Finney consisted of only one file with 507 lines of code. Training using this project produced results of 0% accuracy. This should be expected as we train using a whole file. This one file would be in either our training data or our testing data, but not both. There would be no way to test this file after training. For this reason, the project 01-halfinney was removed from the input data.

### *Initial Naïve Approach*

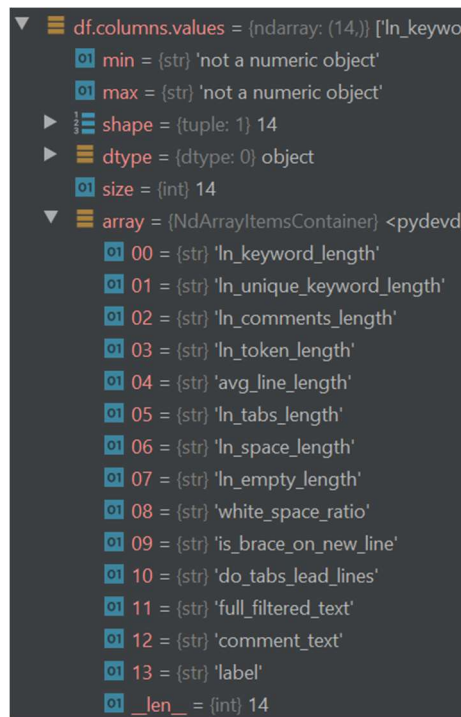
Initially, we decided to tokenize every word of the source code and perform a traditional natural language approach. We quickly identified several challenges, the first being how long the model would take to train. Trying to accommodate most of the tokens from all files would run for hours without finishing. When we attempted to capture all the tokens, we ran into the other main issue, that is, running out of memory. When we limited the data source



to a small sampling of words, the accuracy was fairly low. Because of these issues, we decided to preprocess the data.

### *Revised Approach*

Abstracting a smaller dataset that can still represent the larger dataset is the goal of preprocessing the data. We wanted to test three different inputs separately to see which adds the most value and then combine them all to get a result. For a given file, the source code is separated into source-only and comment-only strings. These are saved in a data frame column to be tokenized later. The feature set is then extracted from the source code by analyzing the important statistics. This is the subset of features mentioned in CHLNVYG15. Each feature is placed into a column in the data frame table.



```

df.columns.values = (ndarray: (14,)) ['ln_keywo
01 min = {str} 'not a numeric object'
01 max = {str} 'not a numeric object'
▶ shape = (tuple: 1) 14
▶ dtype = (dtype: 0) object
01 size = (int) 14
▼ array = (NdArrayItemsContainer) <pydevd
01 00 = {str} 'ln_keyword_length'
01 01 = {str} 'ln_unique_keyword_length'
01 02 = {str} 'ln_comments_length'
01 03 = {str} 'ln_token_length'
01 04 = {str} 'avg_line_length'
01 05 = {str} 'ln_tabs_length'
01 06 = {str} 'ln_space_length'
01 07 = {str} 'ln_empty_length'
01 08 = {str} 'white_space_ratio'
01 09 = {str} 'is_brace_on_new_line'
01 10 = {str} 'do_tabs_lead_lines'
01 11 = {str} 'full_filtered_text'
01 12 = {str} 'comment_text'
01 13 = {str} 'label'
01 _len_ = (int) 14

```

Figure 5 - Pandas Dataframe with Columns Array.

Next, the source code text and comment text are tokenized. Both tokenizers do similar things, but it is worth noting their differences. For the comment text tokenizer, we selected word-based encoding; and for the source code text tokenizer, we went with character-based encoding. Because the comments are written in a natural human language, we processed them using many standard methods. This included treating each word as a token. We then removed what are called “stopwords,” or common words from the English language, and included some C and C++ keywords (see Appendix E). The words that remain reflect the individuality of the author and will help isolate the author’s identity.

The source code text is tokenized with character-based encoding. This is done for several reasons. The first is to avoid what we call an “out-of-vocabulary” word when the words are tokenized. This is a word that is unknown to our tokenizer during a test phase or during our prediction phase. It is more likely to happen in source code because variable names or packages that may not exist in other code.

Filename	Word Based Encoding			Character Based Encoding		
	OOV	Tokens	Percent OOV	OOV	Tokens	Percent OOV
base58.h	53	134	39.55%	0	851	0%
bignum.h	122	505	24.16%	0	2729	0%
db.cpp	333	569	58.52%	0	4589	0%
db.h	255	468	54.49%	0	3413	0%
headers.h	2	29	6.90%	0	199	0%
irc.cpp	142	267	53.18%	0	1922	0%
irc.h	5	8	62.50%	0	74	0%
key.h	43	138	31.16%	0	908	0%
main.cpp	1430	2303	62.09%	0	19833	0%
main.h	805	1131	71.18%	0	9706	0%
market.cpp	108	171	63.16%	0	1293	0%
market.h	110	149	73.83%	0	1340	0%
net.cpp	500	877	57.01%	0	7897	0%
net.h	484	731	66.21%	0	6481	0%
script.cpp	345	677	50.96%	0	5134	0%
script.h	139	384	36.20%	0	2301	0%
serialize.h	599	1641	36.50%	0	10550	0%
sha.cpp	7	701	1.00%	0	2511	0%
sha.h	0	159	0.00%	0	1088	0%
ui.cpp	1713	3166	54.11%	0	28947	0%
ui.h	237	503	47.12%	0	5631	0%
uibase.cpp	826	3583	23.05%	0	40417	0%
uibase.h	185	668	27.69%	0	6707	0%
uint256.h	94	493	19.07%	0	1896	0%
util.cpp	96	329	29.18%	0	2299	0%
util.h	157	375	41.87%	0	2745	0%
<b>Total</b>	<b>8790</b>	<b>20159</b>	<b>43.60%</b>	<b>0</b>	<b>171461</b>	<b>0.00%</b>

Figure 6 - OOV Token Table.

When word-based encoding was used, 43.6% of the tokens were out of vocabulary. While the result to our final prediction was negligible, character-based encoding produced the same result with no out-of-vocabulary tokens.

Finally, the data is split into training and validation data and testing data. The split chosen was a 25%-75% split for testing to training. This allowed for enough data to train the model and a good amount of data to verify the model after training.

### *Constructing the Model*

With preprocessing done, we have three main inputs into our program. We construct a multi-input model using the tokenized source code text, tokenized comment text, and source code features.

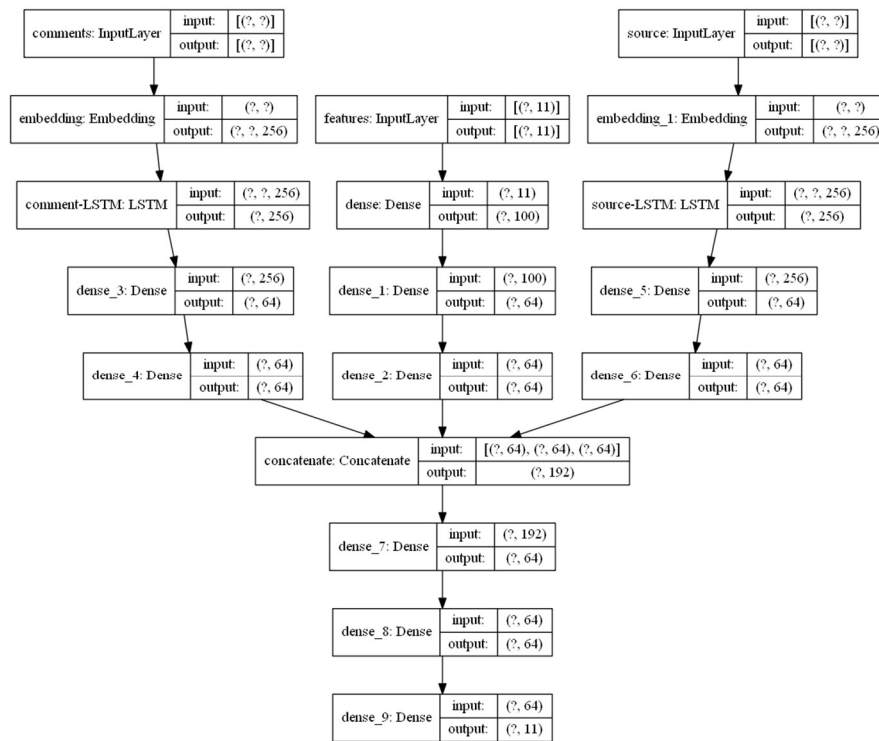


Figure 7 - Multi-Input Model with Multiple Hidden Layers.

The comments and source code go through an embedding layer, then an LSTM layer and two dense layers. The features go through three dense layers. After this, all three are joined with a concatenate layer and go through final processing into a final dense layer with an output size equal to the number of input projects. Above, the model is shown with 12 final outputs.

Several other configurations were tried as well. Having more layers did not seem to increase the model's accuracy, and it increased the time it took to train the model. Thus, any extra layers seemed to detract from the overall application. Fewer layers would also detract from the application, resulting in reduced accuracy and higher loss.

### *Training the model*

The code runs through the input data, preprocesses it, and begins training using 75% of the data for training and 25% of the data for testing and validation. Keras runs through the configured number of epochs. We selected 20 epochs, as fewer noticeably diminished the accuracy, and more added little in the way of accuracy.

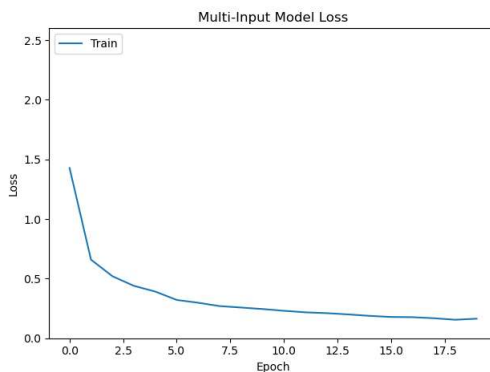


Figure 8 - Multi-Input Loss Values throughout Epochs.

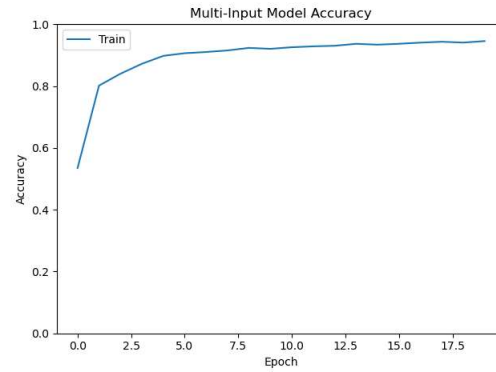


Figure 9 - Multi-Input Accuracy Values throughout Epochs.

As expected from a deep learning model, the loss value decreases and the accuracy increases as the model is trained. The model trained well on the training data, achieving around 96% with a loss value of 0.1203 and the test data that was split in the beginning achieved 90% accuracy.

```
4761/4761 [=====] - 14s 3ms/sample - loss: 0.1203 - acc: 0.9605
Validation Accuracy: 0.9
Execution_time is : 00:04:44
```

Figure 10 - Accuracy at 96% for the training and 90% for the pre-split test data

### *Other Variations of the Model*

To see which input had the most impact on the overall accuracy, we removed portions of the script used to create the model. The results would help us determine which portions of the code were most necessary to increase the overall accuracy and minimize the loss value in our testing.

## Just Source Text

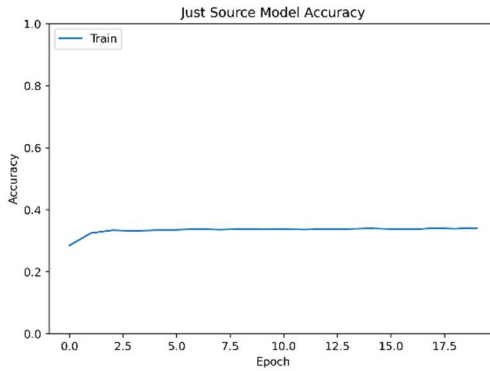


Figure 11 - Just Source Accuracy.

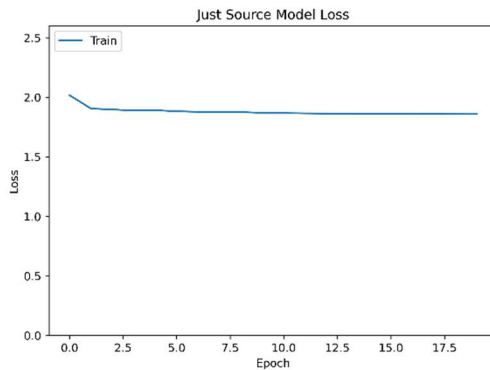


Figure 12 - Just Source Loss.

When the source code text alone was used, it achieved a validation accuracy rate of only 35% and a loss value of 1.8. Given a file at random and asked to guess what project the file belongs in without any analysis, random chance would give a 9.09% chance given 11 projects. If the distribution were known, we could hope for 22.1% to 26.1%, assuming the guesser did nothing but guess the most probable project. The “just source” model does appreciably better than random chance would, but not by much. Additionally, with such a high loss value, further training would not benefit the model. As can be seen in Figures 11 and 12, the training gets a little more accurate after the first epoch, then stays nearly flat. While this input does seem to

contribute the least, it does still contribute to the overall model. Neither word nor character-based tokenization seemed to have any effect on this comment-only output.

### Just Source Features

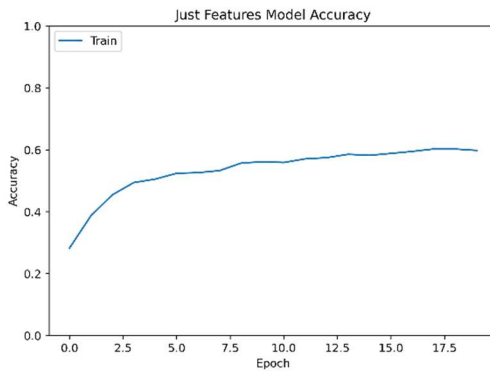


Figure 13 - Just Features Accuracy.

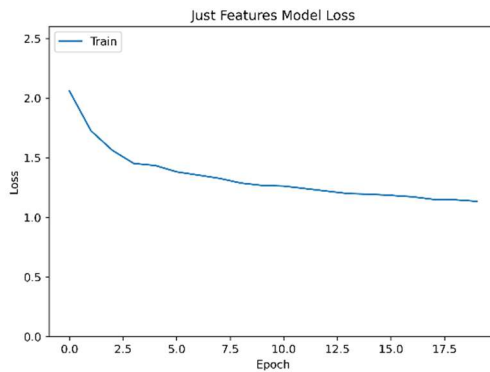


Figure 14 - Just Features Loss.

The extracted features performed the next best. The validation accuracy was about 56%, and the loss ended at 1.1097. The loss value is still high, and at 56% accurate, it needs some improving if it were to be the only input. One of the main benefits of this model is just how fast it trained. After extracting all the features, this model took only six seconds to train.

In addition, extracting this information was much faster than tokenizing every word of the comment text and every character of the source text.

To reiterate, the features extracted here are a subset of the ones outlined by CHLNVYG15. If all features were extracted, we might expect this portion of the model to contribute significantly more than it currently does.

### Just Comment Text

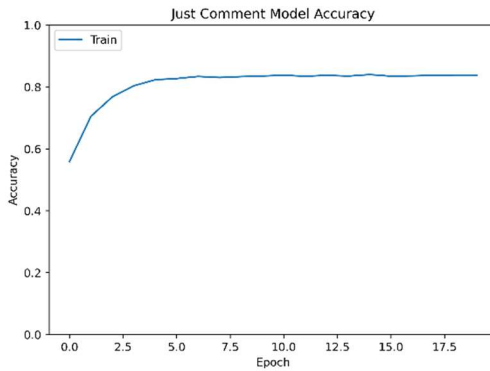


Figure 15 - Just Comments Accuracy.

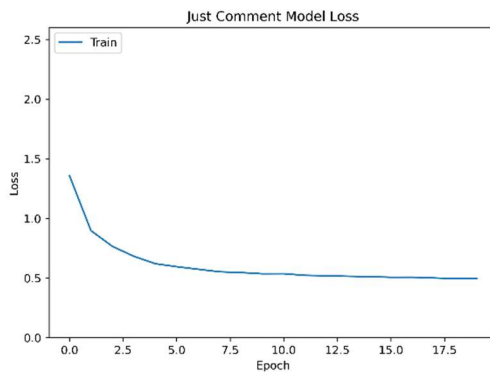


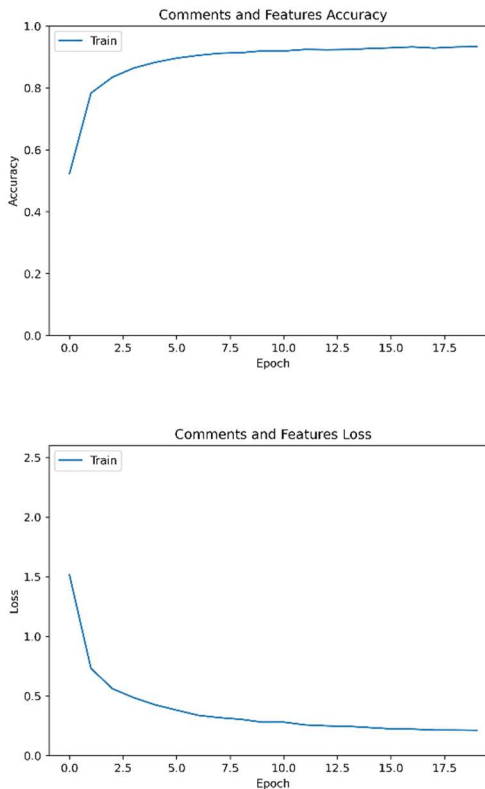
Figure 16 - Just Comments Loss

The comment contributed the most to the overall accuracy of the model. This result might be intuitive, as writing comments gives the author the most opportunity to add his own



words. The end validation accuracy was about 77% with a loss value of 0.4940. While using the comments alone is the most beneficial, there are a few files that have minimal or no comments at all. This is just one of many necessary inputs into our multi-input model.

### Features and Comments Without Source Text



Since the source text seemed to contribute the least, a model with only the source features and comment text was created to see if removing it would perform just as well as the full multi-input model. This model was identical to the multi-input model, just without the tokenized source text. The multi-input model with three inputs achieved 90% validation accuracy, and this model was 88% accurate.

```
4761/4761 [=====] - 8s 2ms/sample - loss: 0.2128 - acc: 0.9332
Validation Accuracy: 0.88
Execution time is : 00:02:55
```

Figure 17 - Multi-Input Model Without Source Text.

This accuracy was only marginally worse than the three-input model and is possibly negligible. One thing that was different in the output was the loss value. The loss value of this model was 0.2128 compared to the loss value of 0.1203 of the three-input model. Because of this, the source text is significant enough to merit retaining it as one of our inputs.

### *Test Data*

In addition to separating some of the files programmatically for test data, it was important to verify our result with test data with a known author. The project cryptocpp was chosen to be split in half as test data. The authors of this project are known and should map directly to the cryptocpp bucket if our model is trained properly.

After the project was split into two folders, the model was retrained using the remaining data. Nearly every file in the split directory matched the proper directory with a high level of certainty.

02-weidai-cryptopp	182	93.81%
03-gutmann-cryptlib	3	1.55%
04-openssl	0	0.00%
05-libgcrypt	1	0.52%
06-mcrypt	1	0.52%
07-botan	5	2.58%
08-nss	0	0.00%
09-truecrypt	2	1.03%
10-gpg4win	0	0.00%
11-luks	0	0.00%
12-gnupg	0	0.00%

Figure 18 - Cryptocpp Files Matching Table

### *Random Forest Comparison*

Finally, a random forest classifier was used to compare the results of the deep learning model to a more traditional machine learning approach. The same preprocessing was applied to the input data and the same stylometric features were extracted. Using just the stylometric features, the random forest classifier achieved a validation accuracy of 59.7% and when using all the same inputs the validation accuracy peaked at 76.9% accuracy.

## Chapter 4

### Predictions Using the Model

One of the main reasons to create such a data model is to use it in other applications for making predictions. As stated before, we selected the Bitcoin code base as our subject for prediction. We wrote an application (see Appendix D) that would use the same tokenizer and models created previously. Reusing the same tokenizer values is very important, as we want a new file to be tokenized with the same values as all previous files. More specifically, a new tokenizer would create a new word index. When words are separated for their numeric values, each file would use a different word index. The token “myVariable” might be indexed at the value 5 for one file and at the value 237 in another. The word index must remain constant throughout all files analyzed.

Each file in the Bitcoin code was separately passed through the application. The file went through the same preprocessing and was then passed through the data model; and a list of predictions was generated, one for each labelled project in our training data. For each file, the results came back as a highly probable match to the OpenSSL codebase. In fact, most of the files were over a 90% probable match.

File Name	Project Predicted	Percentage Match
base58.h	04-openssl	95.10%
bignum.h	04-openssl	99.53%
db.cpp	04-openssl	98.78%
db.h	04-openssl	98.85%
headers.h	09-truecrypt	84.43%
irc.cpp	04-openssl	97.71%
irc.h	04-openssl	52.38%
key.h	04-openssl	97.30%
main.cpp	04-openssl	98.55%
main.h	04-openssl	98.52%
market.cpp	04-openssl	95.65%
market.h	04-openssl	97.70%
net.cpp	04-openssl	98.66%
net.h	04-openssl	98.55%
script.cpp	04-openssl	99.38%
script.h	04-openssl	99.41%
serialize.h	04-openssl	97.97%
sha.cpp	06-mcrypt	52.65%
sha.h	06-mcrypt	70.09%
ui.cpp	04-openssl	82.81%
ui.h	03-gutmann-cryptlib	69.31%
uibase.cpp	04-openssl	91.95%
uibase.h	04-openssl	59.01%
uint256.h	04-openssl	98.82%
util.cpp	04-openssl	98.17%
util.h	04-openssl	63.24%

Figure 19 - File Predictions.

Additionally, only four files did not match the OpenSSL codebase as the most likely candidate for authorship.

Project Name	Number of Predictions
02-weidai-cryptopp	0
03-gutmann-cryptlib	1
04-openssl	22
05-libgcrypt	0
06-mcrypt	2
07-botan	0
08-nss	0
09-truecrypt	1
10-gpg4win	0
11-luks	0
12-gnupg	0

Figure 20 - Prediction Summary.

Noteworthy here is the fact that the other projects guessed were not the highest probability choices. The mcrypt project is one of the smallest code bases regardless of whether the distribution is by lines of code or by file, at 0.8% or 1.2% respectively.

#### *Older Version of OpenSSL*

While these results are notable, the OpenSSL project on Github has over 400 contributors. An older version of OpenSSL should have fewer contributors and fewer years of precedent in the code. OpenSSL version 0.8.1b was added to the project in addition to the version already in the project (version 3.0.0). After removing the old tokenizer data and retraining the model, the results indicated that Bitcoin was more similar to the older version of OpenSSL than the new version.

File Name	Project Predicted	Percentage Match
base58.h	01-openssl-0.8.1b	34.15%
bignum.h	04-openssl	31.43%
db.cpp	01-openssl-0.8.1b	25.06%
db.h	04-openssl	21.11%
headers.h	08-nss	53.98%
irc.cpp	04-openssl	40.00%
irc.h	01-openssl-0.8.1b	66.02%
key.h	02-weidai-cryptopp	27.24%
main.cpp	01-openssl-0.8.1b	32.67%
main.h	04-openssl	21.94%
market.cpp	01-openssl-0.8.1b	34.13%
market.h	04-openssl	29.20%
net.cpp	01-openssl-0.8.1b	28.74%
net.h	04-openssl	24.73%
script.cpp	01-openssl-0.8.1b	23.94%
script.h	04-openssl	22.14%
serialize.h	01-openssl-0.8.1b	30.97%
sha.cpp	05-libgcrypt	49.78%
sha.h	05-libgcrypt	36.97%
ui.cpp	04-openssl	55.17%
ui.h	04-openssl	74.40%
uibase.cpp	01-openssl-0.8.1b	
uibase.h	02-weidai-cryptopp	56.55%
uint256.h	01-openssl-0.8.1b	24.02%
util.cpp	01-openssl-0.8.1b	29.93%
util.h	01-openssl-0.8.1b	20.68%

Figure 21 - File Predictions with older OpenSSL.

Project Name	Number of Predictions
01-openssl-0.8.1b	12
02-weidai-cryptopp	2
03-gutmann-cryptlib	0
04-openssl	9
05-libgcrypt	2
06-mcrypt	0
07-botan	0
08-nss	1
09-truecrypt	0
10-gpg4win	0
11-luks	0
12-gnupg	0

Figure 22 - Prediction Summary with older OpenSSL.

If the distribution of files and lines of code are recalculated with this new project added, the older version of OpenSSL only makes up 7.4% of the total number of files and 4.3% of the lines of code. It ends up being one of the smaller projects in our training set.

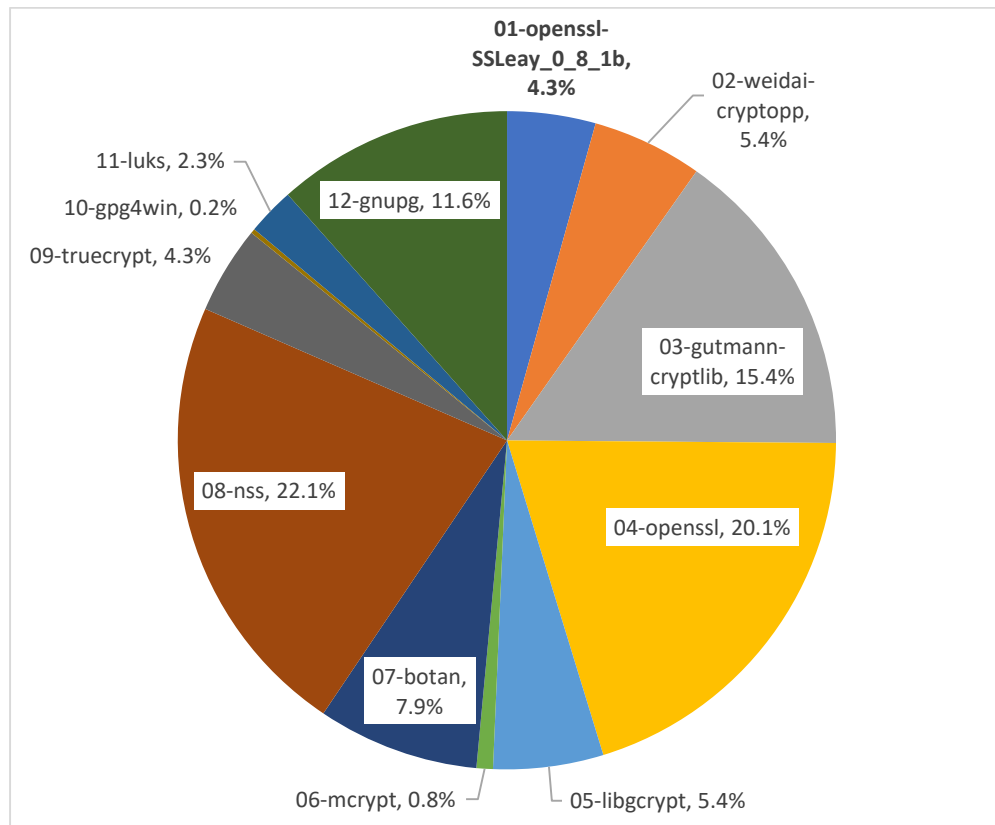


Figure 23 - Distribution of Lines of Code with Older OpenSSL

While this result appears definitive, it does not mean that one of the original authors of OpenSSL wrote Bitcoin. It does mean that of the code samples we analyzed, Bitcoin was most like the OpenSSL projects.

#### *Other Project Predictions*

A valid question one may have would be if this model would classify all large projects into the same buckets regardless of the code contained in them. To investigate this theory, a number of large projects were chosen as test data after the model was created. The projects chosen were curl, DeepSpeech, jq, linux-0.01, msgpack, Mosaic 2.7, SFML, and Whisper Yaffs. All of these projects are written in C or C++, have a large number of files, and are not known to be written by any of the authors in the original input data. These new test projects matched a few different input projects. Only curl and DeepSpeech had over 90% of their files match only one project. The other remaining projects generally had the matches spread across three or more projects.

<b>Project Name</b>	<b>Closest Match</b>	<b>Percent of Files Matching</b>	<b>Next Closest Match</b>	<b>Percent of Files Matching</b>
<b>curl</b>	07-botan	97.23%	02-weidai-cryptopp	1.80%
<b>DeepSpeech</b>	07-botan	94.70%	04-openssl	2.54%
<b>jq</b>	07-botan	36.62%	02-weidai-cryptopp	23.94%
<b>Linux</b>	06-mcrypt	35.53%	02-weidai-cryptopp	30.26%
<b>msgpack</b>	07-botan	37.68%	06-mcrypt	27.56%
<b>Mosaic 2.7</b>	09-truecrypt	29.18%	08-nss	17.08%
<b>SFML</b>	06-mcrypt	55.39%	08-nss	16.39%
<b>Whisper Yaffs</b>	11-luks	41.18%	07-botan	19.61%

Figure 24 - New Input Data Matches

This shows that the model not only matches projects to one specific project, but also attempts to classify an input file according to the features and encodings extracted during training.



## Chapter 5

### Conclusions

Code stylometry to discover authorship is an important analytical step when reviewing code. The research this thesis represents shows deep learning is another valuable asset in determining the authorship of source code.

A strictly machine learning approach used by CHLNVYG15 was able to produce great results. The source code features they describe are a great way to represent a larger data set and provide a good baseline accuracy. We have shown that in the case where an analyst is given access to the full source code, the specific word choices in both source code and comments add valuable insights into who wrote the code. We have also shown that the comments of a code seem to be the most telling piece of information when determining authorship as this allows the author to have more selection at his choice of words. Source code text analysis is the least telling piece of information as many of these choices have been made by the compiler.

With 90% validation accuracy and a relatively low loss value of 0.1203, we have shown that deep learning is a viable way to show similarities between code bases. In addition, with such a low loss value, it appears that combining all three inputs into a deep learning model is the best approach of the options presented.

#### *Areas of Further Research*

As discussed earlier, we could not determine an accurate way of generating an abstract syntax tree from the C and C++ code short of compiling it. Having the full metadata feature

set would likely improve the accuracy of the model. This would be a good area for further research.

The project was limited to C and C++, but the general model should be applicable to multiple languages. The values in some of the columns would be different in regard to file length, but this could be counteracted by adding in another column in the metadata stating what language the original source code was in or, more simply, what file extension the original file had. Possible research could include seeing which, if any, language was more susceptible to this type of analysis and if the same author could be determined across different languages.

The type of source code chosen for this project was also limited in scope. A very practical application for this type of software would be to try to identify who wrote a piece of malware. Malware analysis is its own field of study, but one thing that might aid in this application would be an additional input of indicators of compromise. To put this succinctly, if a piece of code calls out to the same domain or IP address or it targets the same domain or IP address, it is likely related. This could be a fourth input in the model as tokenized input, or another column in the metadata. In either case, more research is needed to determine usefulness in a specific application.

Finally, nearly all deep learning programs benefit from more training data. The final prediction model here was able to identify code if it belonged to one of the eleven projects it trained on. More samples with the same author, or sample depth, would be beneficial.

## Appendix A

### Code for preprocessor.py

The following is the Python code used as the preprocessor in this project. The code extracts features, source code text, and comment text.

```
import re
from collections import Counter

from keras_preprocessing.text import Tokenizer
import nltk

nltk.download('punkt')
nltk.download('stopwords')

class FeatureSet:
    """
    Adapted from the CSFS presented in De-anonymizing Programmers via Code
    Stylometry by:
    Aylin Caliskan-Islam, Drexel University; Richard Harang,
    U.S. Army Research Laboratory;
    Andrew Liu, University of Maryland;
    Arvind Narayanan, Princeton University;
    Clare Voss, U.S. Army Research Laboratory;
    Fabian Yamaguchi, University of Goettingen;
    Rachel Greenstadt, Drexel University
    """

    # LEXICAL FEATURES
    ln_keyword_length = 0
    ln_unique_keyword_length = 0
    ln_comments_length = 0
    ln_token_length = 0
    avg_line_length = 0

    # LAYOUT FEATURES
    ln_tabs_length = 0
    ln_space_length = 0
    ln_empty_length = 0
    white_space_ratio = 0
    is_brace_on_new_line = False
    do_tabs_lead_lines = False

    comment_text = ''
    full_filtered_text = ''

    def __init__(self):
        self.ln_keyword_length = 0
        self.ln_unique_keyword_length = 0
        self.ln_comments_length = 0
        self.ln_token_length = 0
        self.avg_line_length = 0

        self.ln_tabs_length = 0
```

```

        self.ln_space_length = 0
        self.ln_empty_length = 0
        self.white_space_ratio = 0
        self.is_brace_on_new_line = False
        self.do_tabs_lead_lines = False

def get_features(input_file):
    input_file_text = ''

    num_empty_lines = 0
    lines = ''

    braces_on_new_lines = 0
    braces_not_on_new_lines = 0

    lines_starting_with_tabs = 0
    lines_starting_with_spaces = 0
    num_lines = 0

    with open(input_file, 'r', encoding="ISO-8859-1") as f:
        for line in f:
            input_file_text += line
            num_lines += 1
            if line.startswith(' '):
                lines_starting_with_spaces += 1
            elif line.startswith('\t'):
                lines_starting_with_tabs += 1
            if '{' in line:
                if line.index('{') == 0:
                    braces_on_new_lines += 1
                else:
                    braces_not_on_new_lines += 1
            if line.split() == []:
                num_empty_lines += 1

    tokenizer = Tokenizer()

    tokenizer.fit_on_texts([input_file_text])

    num_word_tokens = len(tokenizer.word_counts)

    keywords = ["alignas", "alignof", "and", "and_eq", "asm", "atomic_cancel",
                "atomic_commit", "atomic_noexcept", "auto", "bitand", "bitor",
                "bool", "break", "case", "catch", "char", "char8_t", "char16_t",
                "char32_t", "class", "compl", "concept", "const", "constexpr",
                "constexpr", "constinit", "const_cast", "continue", "co_await",
                "co_return", "co_yield", "decltype", "default", "delete", "do",
                "double", "dynamic_cast", "else", "enum", "explicit", "export",
                "extern", "false", "float", "for", "friend", "goto", "if",
                "inline", "int", "long", "mutable", "namespace", "new",
                "noexcept", "not", "not_eq", "nullptr", "operator", "or",
                "or_eq", "private", "protected", "public", "reflexpr", "register",
                "reinterpret_cast", "requires", "return", "short", "signed",
                "sizeof", "static", "static_assert", "static_cast", "struct",
                "switch", "synchronized", "template", "this", "thread_local",
                "throw", "true", "try", "typedef", "typeid", "typename", "union",
                "unsigned", "using", "virtual", "void", "volatile", "wchar_t",
                "while", "xor", "xor_eq", "include"]

    # prepare the stopwords. extend them to include common keywords in c/c++
    stopwords = nltk.corpus.stopwords.words('english')
    stopwords.extend(keywords)
    stopwords = set(stopwords)

    num_keywords = 0
    num_unique_keywords = 0

    for keyword in keywords:

```

```

keyword_count = tokenizer.word_counts.get(keyword)
if keyword_count:
    num_keywords += keyword_count
    num_unique_keywords += 1

def comment_remover(text):
    def replacer(match):
        s = match.group(0)
        if s.startswith('/'):
            return " "
        else:
            return s

    pattern = re.compile(
        r'//.*?\n|/\*.*?*/',
        re.DOTALL | re.MULTILINE
    )
    return re.sub(pattern, replacer, text)

def comments(text):
    pattern = re.compile(
        r'//.*?\n|/\*.*?*/',
        re.DOTALL | re.MULTILINE
    )
    result = re.findall(pattern, text)
    return result

comment_text = comments(input_file_text)
text_without_comments = comment_remover(input_file_text)

tokens = nltk.word_tokenize(text_without_comments)
# remove all tokens that are not alphabetic
source_words = [w for w in tokens if w.isalpha()]
source_words = [w for w in source_words if w not in stopwords]

num_of_comments = len(comment_text)
char_count = len(input_file_text)

comment_text = '\n'.join(comment_text)

import numpy as np

features = FeatureSet()

features.full_filtered_text = source_words

# LEXICAL FEATURES
if (char_count):
    if num_keywords: features.ln_keyword_length = np.log(
        num_keywords / char_count)
    if num_unique_keywords: features.ln_unique_keyword_length = np.log(
        num_unique_keywords / char_count)
    if num_of_comments: features.ln_comments_length = np.log(
        num_of_comments / char_count)
    if num_keywords: features.ln_token_length = np.log(
        num_word_tokens / char_count)

# start layout features
char_counter = Counter(input_file_text)

num_of_spaces = char_counter[' ']
num_of_tabs = char_counter['\t']
num_of_new_lines = char_counter['\n']
num_of_white_spaces = num_of_spaces + num_of_tabs + num_of_new_lines

# LAYOUT FEATURES
if char_count:
    if num_of_tabs > 0: features.ln_tabs_length = np.log(

```

```

        num_of_tabs / char_count)
    if num_of_spaces > 0: features.ln_space_length = np.log(
        num_of_spaces / char_count)
    if num_of_white_spaces:
        features.ln_empty_length = np.log(
            num_of_white_spaces / char_count)
        features.white_space_ratio = num_of_white_spaces / (
            char_count - num_of_white_spaces)
    avg_line_length = char_count / num_lines
    features.avg_line_length = avg_line_length
    features.is_brace_on_new_line = braces_on_new_lines > braces_not_on_new_lines
    features.do_tabs_lead_lines = lines_starting_with_tabs > lines_starting_with_spaces

    features.comment_text = comment_text

    return features

# EXAMPLE HOW TO RUN on it's own

# files = ['sample.c', 'sample.cpp']

# features = []
#
# for file in files:
#     features.append(get_features(file))
#
# df = pd.DataFrame([t.__dict__ for t in features])

```

## Appendix B

### Code for create\_model.py

The following is the python code used to create the model used in this project. It utilizes functions in preprocessor and the utils.py file.

```
import os
# PROFILING METHODS
import time
from time import gmtime
from time import strftime

import click as click
from keras.utils import plot_model
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras import Model
from tensorflow.keras.layers import Dense, Embedding, LSTM, concatenate, Input
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.python.keras.utils import np_utils

from project2.new_take.config import DEFAULT_MAX_WORDS, file_exts
from project2.new_take.utils import ingest_files

def get_start_time():
    # import time
    start_time = time.time()
    return (start_time)

def get_end_time():
    # import time
    end_time = time.time()
    return (end_time)

def get_execution_time(start_time, end_time):
    return strftime("%H:%M:%S", gmtime(int('{:.0f}'.format(float(str((end_time - start_time)))))))

start_time = get_start_time()

from project2.new_take.utils import max_length # TODO: rename packages

@click.command()
@click.option('-i', '--input_directory', help='The input root directory to read all files for training the data model.',
              type=click.Path(exists=True, file_okay=False, dir_okay=True, resolve_path=True))
@click.option('-e', '--extensions', help='List of file extensions to read.',
```

```

        default='.'.join(file_exts),
        show_default=True,
        type=click.STRING
    )
@click.option('-nm', '--num_words', help='Set the num_words variable for the Keras Tokenizer',
              default=DEFAULT_MAX_WORDS,
              show_default=True,
              type=click.INT)
def create_model(input_directory, extensions, num_words):
    extensions = [e.strip() for e in extensions.split(',')]
    print('Using extensions: ', extensions)

    if os.path.isdir(input_directory):
        # print('Labelled and ingested files.')
        print('Preprocessed files.')
        df = ingest_files(input_directory, extensions)
        # Load into a pandas dataframe
        print(df)

        # Tokenize the comments
        comment_texts = df.comment_text.values
        comment_tokenizer = Tokenizer(num_words=num_words)
        print('Tokenizing comments...', end='')
        comment_tokenizer.fit_on_texts(comment_texts)
        comment_seq = comment_tokenizer.texts_to_sequences(comment_texts)
        # # get Max size of a list to know how much to pad
        comment_max_len = max_length(comment_seq)
        comment_train_vals = pad_sequences(comment_seq, maxlen=comment_max_len, padding='post')
        comment_vocab_size = len(comment_tokenizer.word_index) + 1
        x = comment_tokenizer.word_counts.get('the')
        print('Done.')

        # Tokenize the source words
        print('Tokenizing source...', end='')
        source_tokenizer = Tokenizer(num_words=num_words)
        source_texts = df.full_filtered_text.values
        source_tokenizer.fit_on_texts(source_texts)
        source_seq = source_tokenizer.texts_to_sequences(source_texts)
        # # get Max size of a list to know how much to pad
        source_max_len = max_length(source_seq)
        source_train_vals = pad_sequences(source_seq, maxlen=source_max_len, padding='post')
        source_vocab_size = len(source_tokenizer.word_index) + 1
        print('Done.')

        # Drop the labels off the x values
        x = df.drop('label', 1)
        x.comment_text = comment_train_vals
        x.full_filtered_text = source_train_vals

        labels = df['label'].values # Also known as Y

        # split the x and y into test and train
        x_train, x_test, y_train, y_test = train_test_split(
            x, labels, test_size=0.3, random_state=1337)

        # split the train and test into comments and feature data sets
        comment_train = x_train.comment_text.values
        source_train = x_train.full_filtered_text.values
        feature_train = x_train.drop('comment_text', 1)
        # todo: check this

        comment_test = x_test.comment_text.values
        source_test = x_test.full_filtered_text.values
        feature_test = x_test.drop('comment_text', 1)

        # convert the y_train to a one hot encoded variable
        encoder = LabelEncoder()
        encoder.fit(labels) # fit on all the labels
        encoded_Y_train = encoder.transform(y_train) # encode on y_train

```



```

one_hot_y_train = np_utils.to_categorical(encoded_Y_train)

encoded_Y_test = encoder.transform(y_test) # encode on y_test
one_hot_y_test = np_utils.to_categorical(encoded_Y_test)

embedding_dim = 256 # This is the number of units in a hidden layer. Tune this
accordingly
# BUILD THE MODELS
# We will be using two branches and concatenating them.
# One branch for the comments and one for the code features
n_cols = feature_train.shape[1]

# Input layers
comment_input = Input(shape=(None,), name='comments')
source_input = Input(shape=(None,), name='source')
features_input = Input(shape=(n_cols,), name='features')
# embedding layer
features_f = Dense(100, activation='relu')(features_input)
comment_f = Embedding(input_dim=comment_vocab_size,
output_dim=embedding_dim)(comment_input)
source_f = Embedding(input_dim=source_vocab_size, output_dim=embedding_dim)(source_input)

# memory layers
# features_f = LSTM(32, name='features-LSTM')(features_f) #ndims don't match
comment_f = LSTM(64, name='comment-LSTM')(comment_f)
source_f = LSTM(64, name='source-LSTM')(source_f)

# dense layers
features_f = Dense(64, activation='relu')(features_f)
features_f = Dense(512, activation='relu')(features_f)

comment_f = Dense(64, activation='relu')(comment_f)
comment_f = Dense(512, activation='relu')(comment_f)

source_f = Dense(64, activation='relu')(source_f)
source_f = Dense(512, activation='relu')(source_f)

merge = concatenate([features_f, comment_f, source_f])

# Post merge layers
hidden1 = Dense(64, activation='relu')(merge)
hidden2 = Dense(512, activation='relu')(hidden1)
# todo: dynamic output
output = Dense(encoder.classes_.size, activation='softmax')(hidden2)

model = Model(inputs=[features_input, comment_input, source_input], outputs=output)

plot_model(model, to_file='multit-input-model.png', show_shapes=True)
model.summary()

model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])

history = model.fit({'comments': comment_train, 'features': feature_train, 'source':
source_train},
                    one_hot_y_train, epochs=20, batch_size=64)

model.save('saved_new-take.h5')
model.save_weights('saved_new-take-weights.h5')

import matplotlib.pyplot as plt

# Plot training & validation accuracy values
plt.plot(history.history['acc'])
# plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')

```

```

plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

# Plot training & validation loss values
plt.plot(history.history['loss'])
# plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

loss, acc = model.evaluate({'comments': comment_test, 'features': feature_test, 'source':
source_test},
                           one_hot_y_test, verbose=False)
print("Training Accuracy: ", acc.round(2))

end_time = get_end_time()
print("Execution_time is :", get_execution_time(start_time, end_time))

if __name__ == '__main__':
    create_model()

```

## Appendix C

### Code for utils.py

The following is the code in utils.py. The functions are called throughout the codebase and exist as a convenience to tidy up the code.

```
import os
import pickle

import pandas as pd
from tensorflow import zeros
from tensorflow.keras.preprocessing.sequence import pad_sequences

from preprocessor import FeatureSet, get_features

class LabeledSourceFeatures:
    label = ''
    features = FeatureSet()

    def __init__(self, label, features):
        self.label = label
        self.features = features

    def flat_features(self):
        return self.features.__dict__

def get_label(full_path, base_path):
    """
    Given a full path, and a base path, this subtracts the base path from the
    full path and returns the parent-most folder.
    This is a bit brittle of a function, but it should work for our purposes.
    """
    try:
        # idx = full_path.index(base_path)
        label = list(filter(None, full_path[len(base_path):].split(os.sep)))[0]
        return label
    except ValueError:
        return 'Unknown'

def get_file_list(input_path, extensions):
    file_list = []
    for root, dirs, files in os.walk(input_path):
        for file in files:
            for ext in extensions:
                if file.endswith(ext):
                    file_list.append(os.path.join(root, file))
    return file_list

def max_length(lst):
    """
    Returns a list of lengths for a list.
    """
    maxList = max(lst, key=lambda i: len(i))
    maxLength = len(maxList)
```

```

    return maxLength

def ingest_file(input_file, base_dir):
    # START BY INGESTING SOURCE CODE WITH LABELS
    labeledFeatures = []

    feature = get_features(input_file)
    labeledFeatures.append(
        LabeledSourceFeatures(
            get_label(input_file, base_dir), feature))

    intermediate_data = [(t.label, t.flat_features()) for t in labeledFeatures]
    final_data = []
    for row in intermediate_data:
        new_row = row[1]
        new_row['label'] = row[0]
        final_data.append(new_row)

    df = pd.DataFrame(final_data)

    # Converting bool columns to binary:
    df.is_brace_on_new_line = df.is_brace_on_new_line.astype(int)
    df.do_tabs_lead_lines = df.do_tabs_lead_lines.astype(int)

    # bar.finish()
    return df

def ingest_files(input_dir, ext):
    # START BY INGESTING SOURCE CODE WITH LABELS
    print('Scanning directory: ', input_dir)
    file_list = get_file_list(input_dir, ext)
    number_of_files = len(file_list)
    print('Scanning ', str(number_of_files), ' files...')

    labeledFeatures = []
    for file_name in file_list:
        feature = get_features(file_name)
        labeledFeatures.append(
            LabeledSourceFeatures(get_label(file_name, input_dir), feature))

    intermediate_data = [(t.label, t.flat_features()) for t in labeledFeatures]
    final_data = []
    for row in intermediate_data:
        new_row = row[1]
        new_row['label'] = row[0]
        final_data.append(new_row)

    df = pd.DataFrame(final_data)

    # Converting bool columns to binary:
    df.is_brace_on_new_line = df.is_brace_on_new_line.astype(int)
    df.do_tabs_lead_lines = df.do_tabs_lead_lines.astype(int)

    return df

# CREATE THE TOKENIZERS
def tokenize_file(file_path):
    """
    returns a list of x values.
    """
    base_path = os.path.dirname(file_path)
    base_path = os.path.basename(base_path)

    df = ingest_file(file_path, base_path)
    # Tokenize the comments
    comment_texts = df.comment_text.values

```

```

with open('pickles/comment_tokenizer.pickle', 'rb') as ctp:
    comment_tokenizer = pickle.load(ctp)
comment_seq = comment_tokenizer.texts_to_sequences(comment_texts)
# get Max size of a list to know how much to pad
comment_max_len = max_length(comment_seq)
comment_train_vals = pad_sequences(comment_seq, maxlen=comment_max_len,
                                   padding='post')

if not comment_train_vals.any():
    comment_train_vals = zeros(1)

# Tokenize the source words
with open('pickles/source_tokenizer.pickle', 'rb') as stp:
    source_tokenizer = pickle.load(stp)
source_texts = df.full_filtered_text.values
source_seq = source_tokenizer.texts_to_sequences(source_texts)
# get Max size of a list to know how much to pad
source_max_len = max_length(source_seq)
source_train_vals = pad_sequences(source_seq, maxlen=source_max_len,
                                   padding='post')

if not source_train_vals.any():
    source_train_vals = zeros(1)

# Drop the labels off the x values
x = df.drop('label', 1)
x.comment_text = comment_train_vals
x.full_filtered_text = source_train_vals
# pull out the comment_text and source code text out to their own values
x_comment_val = x.comment_text.values
x_source_val = x.full_filtered_text.values
x_feature_val = x.drop('comment_text', 1)
x_feature_val = x_feature_val.drop('full_filtered_text', 1)

return [x_feature_val, x_comment_val, x_source_val]

```

## Appendix D

### Code for predict\_bitcoin.py

The following code was used to generate predictions for each file in the sample bitcoin code.

```
import os

import numpy as np
from tensorflow.keras import models

from config import file_exts, DEFAULT_MAX_WORDS, default_data_backup_dir
from utils import tokenize_file

num_words = DEFAULT_MAX_WORDS

model = models.load_model('models/saved_new-take.h5')
model.summary()

# test_dir = sys.argv[1]
test_dir = './data-backup'
test_texts = []

labels_index = [
    '01-openssl-0.8.1b',
    '02-weidai-cryptopp',
    "03-gutmann-cryptlib",
    "04-openssl",
    "05-libgcrypt",
    "06-mcrypt",
    "07-botan",
    "08-nss",
    "09-truecrypt",
    "10-gpg4win",
    "11-luks",
    "12-gnupg"
]

current_dir = ''

guesses = {
}

for l in labels_index:
    guesses[l] = 0

for root, dirs, files in os.walk(default_data_backup_dir):
    t = 0
    r = root.split(os.sep)[-1]
    if r in labels_index:
        print(r)
        current_dir = r
        for file in files:
            for ext in file_exts:
                if file.endswith(ext):
                    file_path = os.path.join(root, file)
                    predictions = model.predict(tokenize_file(file_path))
                    p = np.argmax(predictions[t])
```

```
guessed_dir = labels_index[int(p)]
percentage = "{:.2%}".format(np.max(predictions))

print("\t%s ==> %s - %s " % (file, guessed_dir, percentage))
guesses[guessed_dir] += 1

import pprint

pp = pprint.PrettyPrinter()
print('/n')
pp.pprint(guesses)
```

## Appendix E

### Table of Keywords Added to Stopwords

This was the list of words commonly found in C and C++ applications that I added to my stopword list to filter out of the source code.

alignas	alignof	and	and_eq	asm	atomic_cancel
atomic_commit	atomic_noexcept	auto	bitand	bitor	char16_t
bool	break	case	catch	char	char8_t
char32_t	class	compl	concept	const	constexpr
constexpr	constinit	const_cast	continue	co_await	if
co_return	co_yield	decltype	default	delete	do
double	dynamic_cast	else	enum	explicit	export
extern	FALSE	float	for	friend	goto
inline	int	long	mutable	namespace	new
noexcept	not	not_eq	nullptr	operator	or
or_eq	private	protected	public	reflexpr	register
reinterpret_cast	requires	return	short	signed	typename
sizeof	static	static_assert	static_cast	struct	wchar_t
switch	synchronized	template	this	thread_local	union
throw	TRUE	try	typedef	typeid	xor_eq
unsigned	using	virtual	void	volatile	include
while	xor				



## Appendix F

### List of Input Projects

Project	URL	Notes
01-halfinney	<a href="https://github.com/halfinney/bc_key">https://github.com/halfinney/bc_key</a>	This project consisted of one file. It was removed as an invalid data set.
02-weidai-cryptopp	<a href="https://github.com/weidai11/cryptopp">https://github.com/weidai11/cryptopp</a>	
03-gutmann-cryptlib	<a href="https://cryptlib-release.s3-ap-southeast-1.amazonaws.com/cryptlib345.zip">https://cryptlib-release.s3-ap-southeast-1.amazonaws.com/cryptlib345.zip</a>	
04-openssl	<a href="https://github.com/openssl/openssl">https://github.com/openssl/openssl</a>	
05-libgcrypt	<a href="https://gnupg.org/ftp/gcrypt/libgcrypt/libgcrypt-1.8.5.tar.bz2">https://gnupg.org/ftp/gcrypt/libgcrypt/libgcrypt-1.8.5.tar.bz2</a>	
06-mcrypt	<a href="https://sourceforge.net/projects/mcrypt/files/Libmcrypt/2.5.8/libmcrypt-2.5.8.tar.gz/download">https://sourceforge.net/projects/mcrypt/files/Libmcrypt/2.5.8/libmcrypt-2.5.8.tar.gz/download</a>	
07-botan	<a href="https://github.com/randombit/botan">https://github.com/randombit/botan</a>	
08-nss	<a href="https://ftp.mozilla.org/pub/security/nss/releases/NSS_3_9_2_RTM/src/nss-3.9.2.tar.gz">https://ftp.mozilla.org/pub/security/nss/releases/NSS_3_9_2_RTM/src/nss-3.9.2.tar.gz</a>	
09-truecrypt	<a href="https://github.com/FreeApophis/TrueCrypt">https://github.com/FreeApophis/TrueCrypt</a>	
10-gpg4win	<a href="https://files.gpg4win.org/gpg4win-3.1.10.tar.bz2">https://files.gpg4win.org/gpg4win-3.1.10.tar.bz2</a>	
11-luks	<a href="https://www.kernel.org/pub/linux/utils/cryptsetup/v2.2/cryptsetup-2.2.2.tar.xz">https://www.kernel.org/pub/linux/utils/cryptsetup/v2.2/cryptsetup-2.2.2.tar.xz</a>	
12-gnupg	<a href="https://www.gnupg.org/ftp/gcrypt/gnupg/gnupg-2.2.17.tar.bz2">https://www.gnupg.org/ftp/gcrypt/gnupg/gnupg-2.2.17.tar.bz2</a>	
01-openssl-SSLey_0_8_1b	<a href="https://codeload.github.com/openssl/openssl/zip/SSLey_0_8_1b">https://codeload.github.com/openssl/openssl/zip/SSLey_0_8_1b</a>	This project was later added to compare the results of the bitcoin code with both this and the older OpenSSL.

## Bibliography

- [1] Stylometry. (2020). In *Oxford Online Dictionary*. Online Edition. Retrieved from URL <https://www.lexico.com/definition/stylometry>.
- [2] Lecun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436–444.
- [3] François Chollet. (2018). *Deep learning with Python*. Shelter Island, NY: Manning Publications Co.
- [4] Embedding Layers - Keras Documentation. Retrieved December 2019, from <https://keras.io/layers/embeddings>.
- [5] RmsProp. (2013). Climin Documentation. Retrieved December 2019, from <https://climin.readthedocs.io/en/latest/rmsprop.html>.
- [6] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt. (2015). De-anonymizing programmers via code stylometry. In *Proceedings of the 24th USENIX Security Symposium*, pages 255–270, August 12 - 14 2015. Retrieved from <https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/caliskan-islam>.
- [7] N. Rosenblum, X. Zhu, and B. Miller. (2011) Who Wrote this Code? Identifying the Authors of Program Binaries. In *Proceedings of the 16th European Conference on Research in Computer Security*, pages 172–189.

- [8] G. Shearer and F. Nelson. (2017). Source-code stylometry improvements in python. Technical Report ARL-TN-0860, Army Research Lab.
- [9] S. Alrabaee, P. Shirani, L. Wang, M. Debbabi, and A. Hanna. (2018). On leveraging coding habits for effective binary authorship attribution. In 23rd European Symposium on Research in Computer Security, pages 26 – 47.
- [10] M. Brennan, S. Afroz, and R. Greenstadt. (2012). Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity. *ACM Transactions on Information and System Security*, 15(3):12:1–12:22.
- [11] S. MacDonell, A. Gray, G. MacLennan, and P. Sallis. (1999) Software Forensics for Discriminating Between Program Authors Using Case-Based Reasoning, Feedforward Neural Networks and Multiple Discriminant Analysis. In *Neural Information Processing, Volume 1*.
- [12] Stylometry Methods and Practices: Methods. (2018). Temple University Libraries. Retrieved from <https://guides.temple.edu/stylometryfordh/methods>.
- [13] F. Brunton, & H. F. Nissenbaum (2016). *Obfuscation: A Users Guide for Privacy and Protest*. Cambridge, MA: MIT Press.
- [14] Burrows, S. (2010, November 4). Source Code Authorship Attribution. Retrieved from <https://researchbank.rmit.edu.au/eserv/rmit:10828/Burrows.pdf>
- [15] V. Kalgutkar, R. Kaur, H. Gonzalez, N. Stakhonova, & A. Matyukhin. Code Authorship Attribution: Methods and Challenges. Retrieved from <https://dl.acm.org/doi/fullHtml/10.1145/3292577>

- [16] F. Ullah, J. Wang, F. Al-Turjman, et al. (n.d.). Source Code Authorship Attribution Using Hybrid Approach of Program Dependence Graph and Deep Learning Model. Retrieved from <https://ieeexplore.ieee.org/abstract/document/8848478>