Automated malware analysis for Android applications through raw bytecode

by

Joy Hauser

B.S., Kansas State University, 2018

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2021

Approved by:

Major Professor
George Amariucai

Copyright

Abstract

Securing mobile phone applications is one of the large areas of research based on the wide spread of mobile phones today. Android encourages developers to make Java applications to run on Android devices. While this provides developers with a lot of freedom, this provides the same opportunity to malware authors. Therefore, defenses need to be put in place to determine which applications are malicious or benign. Additionally, an automatic way to determine if applications are malicious needs to be put in place given the massive amount of applications that incident responders would need to review.

To address the question of how to determine if an application is malicious, this thesis approached the problem by utilizing a LSTM model. This approach was utilized to determine if treating individual Java bytecode instructions as "words" in a sentence for an NLP task would provide decent performance compared to the expectations for this dataset. A logistic regression model was utilized to provide a baseline measurement for what the expected results were.

Six different configurations were attempted for both of the models to determine which configuration provided the best performance for the applications pulled from the Androzoo repository. The LSTM model achieved very similar performance across all six experiments, with only the loss value changing. An accuracy of 0.9, a precision of 0.933, a recall of 0.83, a F1-score of 0.841, and a loss of 0.332 were the results of the best configuration for the LSTM. The equivalent logistic regression experiment resulted in 10.198 loss, 0.86 accuracy, 0.733 precision, 0.75 recall, and 0.731 F1-score. The LSTM model performed better than the logistic regression model, but increasing the amount of input may provide better results.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would, first, like to thank Dr. George Amariucai for his support throughout my Master's program. He continually pushed me to be ambitious in finding goals that would further my learning, my career, and my education. He continually encouraged me to attempt more difficult tasks, causing me to work even harder, and learn even more. Whenever I finished one challenge, George was ready to go with the next step and challenge in the process. Additionally, he always made time to speak with me when I asked, even when I was working full time toward the later half of my degree program. Thank you for all the support!

Dr. Doina Caragea also provided a great amount of support throughout my research process. She already had many contacts in the world of Android security research who she could put me in contact with, including the research coalition she was a part of with two other universities. I discussed the experiments that I was going to run, within this thesis, with her and she provided me feedback. Additionally, I utilized her AWS resources to perform the experiments outlined within this thesis.

I took two classes at Kansas State University about artificial intelligence and deep learning. I want to thank the professors of both of those courses, Dr. William Hsu and Dr. Doina Caragea, for the opportunity to learn more about these topics.

I came into my Master's degree with little knowledge about mobile application development, Android, or the field of artificial intelligence. I would not have been able to complete this thesis without the support and patience of these three individuals as I approached two completely new subject areas.

Additionally, I want to acknowledge the Androzoo repository for providing the data utilized to perform the experiments within this thesis.

Dedication

I want to dedicate this thesis to my family as they supported me throughout not only this degree program, but also my entire education. While I was working on my research, they repeatedly listened to me explain my understanding of the topic as I was still learning. Additionally, they provided me emotional support through the failures and the late nights till I got the code working the way I needed it to for the experiments.

Specifically, I want to mention my sister Rosa, who provided the brunt of the support throughout the last year of finishing out my thesis. Her emotional support was invaluable as I worked to get the experiments to run better with each iteration. Additionally, she got asked more than a few times which word or phrasing was better for a specific section of this paper.

Overall, I would like to acknowledge all the family and friends who supported me through this process. This has been amazing learning journey and I have become a more knowledgeable person for it.

Chapter 1

Introduction

Mobile phones are widely and increasingly used in today's society. Two companies dominate the phone market, Apple and Android. Apple created the iOS operating system to support their phones. This operating system and the applications on it are considered closed systems. Users are prohibited from attempting to manually introduce new content to the system without it getting approved through an official Apple process. Android has their own linux based operating system and the Android company encourages people to develop content for their systems. Android applications are written in Java and are primarily open source. While Android provides the users more freedom to develop their own applications or custom code, this capability presents a security risk. There is no way to limit this capability to users with benign intentions. Therefore, many malicious users have created applications to exploit their fellow users.

Since there are users with malicious intentions, a way to differentiate applications that perform malicious actions versus benign actions must be developed. Currently, neither the professional community, nor the research community, have developed a way to consistently, efficiently, and effectively identify malicious applications.

1.1 Problem Statement

This thesis will describe and evaluate a method for identifying malicious applications by analyzing the Java bytecode of an Android application, APK, with a deep learning model.

This topic was selected due to the timeliness of mobile security. While this topic has been reviewed by security researchers for several years, there have been recent advances in performance due to newer types of deep learning.

Similar approaches have been performed in the past. Methods for detecting if APKs are malicious have been developed by several research teams, including DroidDetector and DeepRefiner. DroidDetector is an online tool that performs a mixture of static and dynamic analysis on the attributes and behavior of the APK. DeepRefiner is a security practitioner tool that takes a two pronged approach by running a naïve model to get the initial observations and then a deep learning model. This tool takes a very different approach to the work in this thesis.

In this work, a different perspective will be taken from both of the before mentioned tools. Instead of utilizing a simplified representation of the Java bytecode instructions, this work will use the unmodified bytecode instructions from the APK.

1.2 Proposed Approach

This project utilized static analysis of the Java bytecode of the Android applications. Dynamic analysis was considered, but was disregarded for this project due to the requirements of performing this analysis. Android applications are event-based systems, similar to websites, so a tool would need to be utilized to simulate user input[1]. It is difficult to assert that all possible events have been found in this approach, when it is not coupled with static analysis. Additionally a virtual machine (VM) environment would need to be set up to run the applications for testing. This process would be time intensive and was not desirable due to the additional resource and time requirements in comparison with static analysis.

Supervised learning was the analysis approach utilized for the experiments within this

thesis. A third party tool, VirusTotal, was utilized to generate a quantitative score of how many antivirus solutions reported an APK as malicious. A threshold was set to determine what VirusTotal score would indicate an APK as malicious for this work. Only APKs that were clearly identified as benign or malicious based on the chosen threshold were considered in the dataset, since manually labeling them would have been too time-consuming for this project. A semi-supervised learning approach was considered, but ultimately rejected due to the amount of time it would require to analyze and label the APKs that were not clearly within the threshold for benign or malicious.

The overall task was reduced to a binary classification problem, where APKs were classified into the benign or malicious category.

To determine what a reasonable expectation of performance should be, or baseline, an artificial intelligence model was utilized. A binary logistic regression model was utilized to draw a naïve correlation between which instructions existed in the APK and whether the APK was malicious, which set a baseline expectation for performance metrics for this thesis. Linear regression models assume a linear relationship between the input and the label, so the performance of the deep learning model can be expected to be better than the a binary logistic regression model.

Recurrent Neural Networks (RNNs) are a type of deep learning model that specializes in handling Natural Language Processing (NLP) tasks. NLP is utilized for processing text, such as English sentences, and performing a task based on that information. One use case for NLP is autofill, where the model will attempt to predict the next word in the sentence. This is done by providing each word to the model, and it will use the past words to guess what the next word would be. Another use case is sentiment analysis, such as determining if a product review is positive or negative. The words in the sentence as well as the position of those words in the sentence are utilized to determine sentiment. For this research, the question is can individual Java bytecode instructions be treated as "words" in a sentence. Similar to sentiment analysis, this work is looking to determine if the APK is malicious, which could be similar to determining if the review has a negative sentiment. One specific type of an RNN model was used, a Long Short-Term Memory (LSTM) model, which utilizes cells to

retain memory. Given the different mechanism for memory retention between elements in the sequence, this model was very promising for determining if raw bytecode instructions could successfully be utilized as an input to an RNN model.

Chapter 2

Related Work

2.1 Key Terms

Regression[2] is a type of linear model where statistical analysis is used to predict the relationship between the input values and the labels. Linear regression[3] is a type of regression that assumes there is a linear relationship between the input and the label.

Logistic regression is a type of regression where the label value is assumed to be dichotomous, meaning a binary value. The model fits a sigmoid function to the parameter of a Bernoulli distribution from which the data is drawn.

A Long Short-Term Memory (LSTM) model is a gated recurrent neural network (RNN) model. It has the ability to retain memory by updating the hidden layer with biases, input weights, and recurrent weights for the forget gates.[4] As the model processes each bytecode instruction in the sequence, it can choose to forget old information that does not match its new understanding. In addition, it can choose whether to update its understanding based on the new information. This ability to retain information acts as a memory of previous experiences.

Natural language processing (NLP) is the practice of converting a human readable language to a language that could be understood by a computer. This is often used for comprehending text or converting text from one language to another.[4] Instead of providing the

model an embedding of words, an embedding of Java bytecode was provided. The model then processes the bytecode as it would process any other language, such as English.

LSTM models were designed to handle NLP tasks. While an input of Java bytecode instructions cannot be handed to the LSTM directly, an encoding and/or embedding of these instructions can be handed to the LSTM model.

2.2 Background

Several considerations must be taken into account when analyzing Android malware. There are some common approaches utilized today as well as some weaknesses in those approaches that should be understood when discussing this issue.

Many antivirus solutions today primarily conduct analysis of applications based on signatures. Signatures are meant to be unique identifiers for an application, and are often implemented as the hash of the entire code base of an application (i.e. MD5 hash, SHA hash). They allow antivirus applications to confirm if an application is an exact match to another file or application. While signatures are very good at identifying known malware that may have been deployed directly to the system, there are several concerns with this approach for all malware detection. New malware is developed on a daily basis, so maintaining a database of signatures for all malware is a very large challenge. Zhenlong Yuan, Yongqiang Lu, and Yibo Xue[5] called out concerns about how antivirus solutions handled malware repackaging. When testing how ten antivirus solutions handled repackaging, the analyst simply disassembled and then reassembled the code which caused the APK to have a different MD5 and SHA hash value. Repackaging the applications with the same functionality to have different signatures resulted in a large drop in the detection rate of malware in the ten antivirus solutions analyzed. Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou[6] stated in their paper that many applications use shared libraries. Signature based approaches account for the concept that if a library is compromised, all the applications that utilize that version of that library may be compromised. While the antivirus solution could create hashes of the new code base for all the affected applications, there are scalability considerations that

must be taken into account given that new versions of libraries come out on a weekly if not daily basis[7][8][9][10]. This thesis approaches identifying a dynamic way to evaluate malware based on the Java bytecode rather than relying the signatures of the malware.

In the paper "The Lifetime of Android API Vulnerabilities: Case Study on the JavaScript-to-Java Interface," Daniel R. Thomas, Alastair R. Beresford, Thomas Coudray, Tom Sutcliffe, and Adrian Taylor speculate on the amount of time it takes for an Android API vulnerability fix to propagate to every Android device. They do this by analyzing data about when people download system and API updates directly from Google. Additionally, they perform an in-depth analysis of the JavaScript-to-Java Interface vulnerability and the fix that Google attempted to make to it[11]. Additionally, they look at how long it took for this fix to be adopted by Android users. At the end, they also discuss the fact that this vulnerability can still be present in Android phones under certain Android system and API version numbers. This work showed that once a malware sample is introduced into a marketplace environment, it may be there for a significant amount of time. Enforcing that the ability to identify vulnerable or malicious APKs is critical in the marketplace environment.

2.3 Deep Learning Approaches

DroidDetector is an online Android malware detection tool developed by Zhenlong Yuan, Yongqiang Lu, and Yibo Xue[5] that performs a mixture of static and dynamic analysis. The authors chose to use both forms of analysis because they did not believe that the problem could be solved by only looking at the code or the behavior. For the static analysis, they reviewed two types of features. They pulled information related to whether the APK requested one of 120 unique permissions, such as the ability to make telephone calls or access the camera. They also reviewed how many times a total of 59 different function calls at the Java bytecode level that they considered sensitive, such as changing permissions on a file or deleting users' messages or contacts, were called. For the dynamic analysis, they utilized the DroidBox sandbox to simulate both benign and malicious user activity on the device. From this testing, they monitored a total of 13 application actions and recorded information about

whether they occurred. These features were provided into a Deep Belief Network (DBN) and were analyzed with a variety of ratios of benign to malicious APKs. One of the goals stated in this work was to develop an automated solution that could detect malicious activity in new or repackaged APKs, but there could be concerns with the approach they took from maintaining a list of sensitive API calls. The model required a manually currated list of what is considered to be a sensitive API call. This list may need to be updated over time, which requires human interaction. Additionally, the datasets utilized by this work were smaller compared to the datasets of some more recent works. While this thesis only performs static analysis, it uses features drawn from the individual Java bytecode instructions themselves instead of a restricted set of sensitive functions.

DeepRefiner[12] is a tool created by Ke Xu, Yinghio Li, Robert H. Deng, and Kai Chen that analyzes APKs and determines if they are malicious. The tool has two layers of detective models to classify the APKs.

First Layer - Attributes: The first detection layer attempts to determine if the APK is malicious by reviewing the application's attributes in the Android manifest file. It feeds these features into a Multilayer Perceptron (MLP) with multiple hidden layers in the deep neural network and then does a Softmax operation on the results. If this first detection layer is able to classify the APK as malicious or not with a confidence level above a specific threshold, the tool will stop there. However, if the first detection layer has a confidence level below the threshold, the APK is labeled as 'uncertain' and is passed onto the second detection layer.

Second Layer - Bytecode: The second detection layer reviews the Java bytecode from the application. Instead of analyzing the original instruction itself, the program converts the instruction to one of fifteen instruction categories[13] to reduce complexity. Then, these modified instructions are utilized to create bytecode pairs (2) where a certain number of instructions C are grouped together with an emphasis on method level grouping since method level semantics were considered more resilient. In each pair, a one-hot encoding representation is created based on the list of unique bytecode instructions from all the APKs. Skip-Gram modeling was then utilized to convert this data representation into a dense vector. There-

fore each application is represented as a variable length bytecode vector sequence. Shorter bytecode vectors are padded with zeros making them fixed length. This data is then inputted into multiple stacked LSTM layers followed by a Max Pooling layer and a Softmax layer at the end. The authors found that LSTM models were "especially effective at capturing application-level bytecode semantics without losing method-level semantics" [12]. The Max Pooling layer allowed for variable length bytecode vector sequences to be read as fixed length sequences and assisted the Softmax layer in focusing on the most relevant bytecode sequences for the final classification. The final classification performed by this model on whether the APK was malicious was returned as output.

Comparison: DeepRefiner was the most similar work to the research conducted in this thesis. The similarities between DeepRefiner and this thesis are:

- Both works reviewed the Java bytecode as input for the models.
- Both works utilized a LSTM model when reviewing the bytecode due to the great potential of RNNs for machine language learning.
- Both works are limited from capturing malicious content or operations injected at runtime due to the fact that both works utilize static analysis.

Some of the differences between these two works include:

- DeepRefiner considered two different models / approaches within the same tool to classify APKs. This thesis considered two different models for approaching the problem, but considered them separate from each other and utilized the more naïve model to provide a baseline for expected results. Due to the more intricate configuration of the LSTM model in how it attempts to learn the information from the APK, the researchers were expecting that the results would be at least as good as the MLP model if not better.
- DeepRefiner considered both an attribute based approach as well as a Java bytecode instruction based approach for identifying which APKs were malicious. This thesis

only considered the Java bytecode instructions, but reviewed them at a much more fine grained level than DeepRefiner did.

- DeepRefiner did not consider the raw Java bytecode when passing off the input to the LSTM model. Rather it utilizes an abbreviated representation of the instruction based on the 15 categories[13] of instructions. However, this thesis makes use of the raw bytecode instructions from the APK dex files.

One potential shortcoming of the DeepRefiner tool is that it does require some level of human intervention when new Java bytecode instructions are added to Android. Many new bytecode instructions are added each year, so in order to maintain this tool each of those new instructions must be manually placed into one of the chosen 15 categories used by the tool and the categories may need to be modified. This required human intervention impedes the long term usage of this tool without maintenance. However, the approach this thesis takes would not require human intervention as it would dynamically scale since it encounters new bytecode instructions.

2.4 Fuzzing Approaches

Android malware feature clustering was a technique attempted by Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou[6]. They created features from sequences of the Dalvik bytecode, or Java bytecode, based on a variety of n-gram lengths. These features represented capabilities offered by the application. They then created clusters, or groupings, of these features to compare against the keywords in the VirusTotal report about the applications, such as "Trojan". For each of the features found in the APKs, a vector of binary values indicating the presence or absence of a feature was created. Any code identified to be part of benign shared libraries was removed to reduce the size of the samples and reduce false positives. The concept from this work of utilizing a binary indicator of the existence of a feature was carried forward into the research described in this thesis. Additionally, the authors used a combination of data from the Android Genome malware dataset and the Androzoo dataset,

similar to how Androozoo data is utilized in this thesis. The authors utilized the APK bytecode to create their features, similar to how this thesis considers the APK bytecode. However, one goal of this thesis is to determine if a more granular view of the individual bytecode instructions improves the ability of a classifier to determine if the APK is malicious.

Another area of research involves classifying APKs into various malware families. Carl Sabottke, Eddie Tanner, and Richard Johson[14] approached this task by utilizing the metadata that VirusTotal provides about APKs. They created a set of 3402 unique feature options across several categories, such as imports, exports, DNS requests, file size, etc. They fed these features into a Support Vector Machine (SVM) as well as an agglomerative linkage-based clustering model. They then used a purity, or simple majority, process to determine which of 54 unique labels, including a NULL label, the APK fit. For this research, they used a small dataset of 11,362 APKs from the Malicia project. One consideration that the authors pointed out in their work is that they only ran the APKs through VirusTotal once to determine the features for the APK, rather than several iterations over time to ensure that the features remain consistent. In this thesis, the concern of reliability is being addressed by performing n-fold validation of the data to ensure that the results are reasonable, not one-time results. Additionally, APKs from several different years are being utilized in this thesis to show consistent behavior for applications across several years.

In their paper titled "Experimental Study of Fuzzy Hashing in Malware Clusting Analysis,"[15] Yuping Li, Sathya Chandran Sundaramurthy, Alexandru G. Bardas, Xinming Ou, Doina Caragea, Xin Hu, and Jiyong Jang discuss fuzzy hashing algorithms. In this paper, they talk about the benefits of using fuzzy hashing algorithms to compare sections of code or features. Additionally, they take a in-depth look at the currently available hashing algorithms and how effective they are. When doing the evaluation of the different hashing algorithms, they are specifically looking at whether the hashing algorithm would be effective for use in malware analysis.

2.5 Malware Families

In the paper "Evolutionary Algorithms for Classification of Malware Families through Different Network Behaviors," M. Zubair Rafique, Ping Chen, Christophe Huygens, and Wouter Joosen propose a "malware classification framework based on evolutionary classifier that uses different network behaviors." [16] They look at how to classify different malware families and their varients based on network information. They take two approaches to collect network behaviors of the malware: "protocol-aware and state-aware modeling scheme" [16]. The first approach assumes that it is a known network protocol, and the second approach assumes that it is an unknown network protocol built on TCP or UDP. The authors then use eight different malware classification algorithms to determine what family the malware belongs to. Four of the malware classification algorithms are a type of evolutionary algorithm and four were not an evolutionary method. During their analysis, they determined that the evolutionary malware classification algorithms took longer to train than the non-evolutionary algorithms. However, the malware classification algorithm that had the best performance when given new data was the UCS malware classification algorithm, which was one of the evolutionary algorithms.

2.6 Hybrid Approaches

In their paper, Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby wrote about a static analysis tool that they wrote to assess the security vulnerabilities in Android applications called "Amandroid". This tool is "an inter-component data flow analysis framework tailored for Android apps." [17] "Amandroid is the only tool which tracks data flows through inter-component communication (ICC)" which allows it to "find sophisticated data leak and data injection problems" along with other security vulnerabitilies [17]. To be able to identify these vulnerabilities, Amandroid creates inter-component dataflow graphs (ICFGs) and data dependence graphs (DDG) for each app that show which components talk to other components and what data flows between them. Since Android applications focus on inter-component

communication so heavily for operation, this information is invaluable when looking for vulnerabilities.

Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albaryrak provide an introduction to the state of Android security in their paper, "An Android Application Sandbox System for Suspicious Software Detection." They discuss the insufficient focus on Android application security and how two possible approaches to fix this, which are static and dynamic analysis[18]. They then explore how to make dynamic analysis viable with malware that can determine if it is running in a virtual environment[18]. The authors then discuss system calls being a great way to determine malicious activity from apps as they can perform very significant malicious actions on the operating system[18].

2.7 Dynamic Analysis Approaches

Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee[19] discuss different techniques that malware employ to detect the usage of malware analysis programs on Android devices. The authors focus on the concept of transparency through not deterring the actions of the malicious program. Instead, they recommend creating a program which runs at the hypervisor level that can communicate with a counterpart program in the user space level of the guest operating system. In their paper, the authors focus on the Windows XP operating system to perform their analysis. The benefit of having the analyzer program at the hypervisor level is that it can intercept actions on the guest operating system level and change their behavior, thereby hiding its own existence. This approach is called a hardware virtualization extension approach and has been proven by the authors to be a better way to evade detection by malware. The major flaw with the system that the authors found is that their analyzer is susceptible to detection if the malware is using a clock from outside of the machine's control, and therefore the the analyzer's control[19]. This is known as a timing attack.

Brandon Amos, Hamilton Turner, and Jules White[1] explore the practicality of different machine learning classifiers for malware analysis in their paper "Applying machine learning classifiers to dynamic Android malware detection at scale." In their paper, they evaluate

six commonly-used machine learning algorithms and determine if they are acceptable for practical use in malware analysis. Additionally, the paper discusses a malware analysis framework that they created called STREAM which sets up the environment for malware analysis, determines the feature vectors to use for the analysis, and trains the machine learning classifiers. The authors specifically state that their focus is "on profiling applications to obtain information used in dynamic analysis". The authors did a performance analysis on the machine learning algorithms to determine if the machine learning algorithm would be acceptable for real world use. They performed this analysis using cross validation and true testing. However, the authors state that cross validation performance analysis, a common practice among machine learning research, provides an inaccurate view of performance because it adds in bias. To address the cross validation concerns pointed out in this paper, the experiments in this thesis used two types of cross validation. First, the data was split into training, validation, and test sets so that the performance of the model could be evaluated throughout the epochs of training and once the model is fully trained. To account for some of the bias that limiting the number of applications included in the training set may introduce, n-fold cross validation was employed to split the same dataset five different ways. N-fold cross validation will ensure that the results presented in this research are not due to a "lucky" or biased configuration of the APKs.

Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis explore different ways to avoid dynamic analysis detection for Android malware in their paper "Rage Against the Virtual Machine: Hindering Dynamic Analysis of Android Malware". In their paper, they look at different aspects of Android emulators and how they could be detected by malware[20]. They look at the different aspects of emulators that are set by the user during the emulator creation to how emulators process instructions at the byte code level and how these behaviours can inform malware that its on a virtual environment. At the end of their paper, they discuss how to change an emulator to make it harder for malware to determine if it is a virtual environment or now.

Chapter 3

Data

This research utilized Java bytecode instructions from APKs to analyze whether the applications were malicious. The ground truth values utilized for analysis of whether or not the applications were malicious were taken from VirusTotal.

The APKs utilized for these experiments were downloaded from the Androzoo[21] repository. A research group from the University of Luxembourg created the repository to encourage research on Android application security by providing free and unrestricted access to up-to-date APKs. The team from the University of Luxembourg created crawlers for several markets, including the official Google Playstore, Anzhi, and AppChina. They also created crawlers for smaller Android markets and the Genome project. Additionally, a small number of APKs were collected through BitTorrent that may have been distributed without the author's consent and may have originally required payment. Based on this caveat, the Androzoo team asked that researchers kept this in mind when using the APKs that are stored within the repository. There are a few limitations of the repository that are worth noting for awareness.

- Only APKs that were available for free were included in the dataset.
- APKs are stored based on their SHA-256 hash and the market that they were collected from. So if an APK exists in two markets, it may appear twice in the repository. To determine if this is the case, the individual researcher would need to download both

APKs and compare them.

- Many of the marketplaces where APKs were collected from only allow users to download the latest version of the APK, so there may be some missing intermediate versions.

To assist researchers, all the APKs within the Androzoo repository have already been run against VirusTotal.

VirusTotal (VT)[22] is a well-known and generally trusted tool by industry professionals that is used to judge if an application or website is malicious. This tool "inspects [APKs] with over 70 antivirus scanners and URL/domain blacklisting services."[23] The VirusTotal tool does not make determinations itself about whether an application is malicious, but the user can make an assumption based on a certain score threshold. This research utilizes a threshold for the number of antivirus scanners that considered the application as malicious for ground truth values, or labels.

3.1 Procurement

This research was performed in collaboration with Dr. Doina Caragea's research group at Kansas State University as well as research groups from University of South Florida (USF) and Bowling Green State University (BGSU) that have formed a coalition working to solve the same problem through a variety of approaches. More information about the approaches that were taken by other members of the coalition can be found in the Literature Review section.

This coalition of three universities agreed to work together to download the 6 million APKs that were on the Androzoo site in February 2018. Kansas State University was assigned to download 1,920,590 (about 2 million) of the 6 million APKs. The other universities utilized their resources to download the other four million.

The research coalition made the decision to utilize the report from the most recent scan of the APK if the APK has already been scanned by VirusTotal. The concern that this approach presents is that the scan may not be a recent scan. It could be a scan from a few

months or years ago. However, in the case that no previous scan of this APK exists, a new scan will be run for that specific APK.

3.2 Data Extraction

3.2.1 The Inputs

Androguard was utilized to read the Android applications. APK files are zipped folders containing sub-folders that include the Java bytecode that the source code was compiled to. The Java bytecode is contained within one file per class of the source code. Each of these class files contains one or more methods. Each method contains a list of bytecode instructions.

The Python library Androguard was utilized to load the APK folders and then open them in an XML like format.

It is important to note for this research that the classes and methods were not read in any specific order. A static analysis approach was utilized to look at the data. Since Android applications run as event-based systems, it is difficult to predict the order that the code will be run in. Therefore, the classes were read in as based on how they appeared in the folder. The methods within each class were read in how they existed in the class file. The individual bytecode instructions within the methods were read as they existed in the method.

3.2.2 The Labels

The VirusTotal scores were utilized to indicate to the model which applications are malicious.

VirusTotal supplies each piece of software a score. This score is the number of antivirus scanners which believe the piece of software to be malicious. The score can be any number between 0 and 70, because 70 was the number of antivirus scanners used by VirusTotal at the time of this publication. The total number of antivirus scanners in use may change over time.

A hash of the application was provided to VirusTotal, and then a report of how many different antivirus scanners viewed the application as malicious was provided as output.

Since VirusTotal provides a score, but not an indication of whether an APK is malicious in their opinion, the research coalition decided to set a threshold based on the score. The threshold agreed to by the coalition is below:

- VirusTotal score of 0 means the application is benign.
- VirusTotal score of 1 means the application may be malicious or benign. Manual labeling would be required for these applications, so they were not included in this project.
- VirusTotal score of 2 or more means the application is malicious.

These definitions will be used going forward for this project, but may be changed for future research.

Several threshold values were considered, such as 9 and 10, but 2 was selected due to several researchers from BGSU and within the coalition were utilizing that threshold. Therefore, this project used the same threshold to allow for the results of the research being more comparable. For reference, the Androzoo team ran the statistics with an threshold of a VirusTotal score of 10 considered malicious in 2016[21] and only 1% of the APKs from Google Play store were considered malicious.

3.3 Preparation

To prepare the data for analysis, it had to undergo several stages of cleaning and filtering, which are described below.

During all the stages of the data preparation and analysis, a combination of AWS resources and the Kansas State University Beocat supercomputer were utilized to increase efficiency and decrease time and cost. Additionally, all of the downloaded APKs and VirusTotal reports were stored in the large storage mediums within Beocat.

3.3.1 Filter Out Data Without a Label

The first stage of data preparation was removing the APKs that didn't have a label. This stage was used while the data was being collected and sorted into the correct directories. Any APKs whose VirusTotal report didn't have a score were excluded from analysis. A score may not have been available if the download failed for some reason or if the APK had not been previously scanned by VirusTotal.

3.3.2 Determine Unique Bytecode Instructions

The second stage of the data preparation was creating a unique list of bytecode instructions from the APKs being analyzed. To perform this task, the unique set of bytecode instructions was identified for each APK. Then these lists were combined and the unique set of bytecode instructions for all APKs was created.

Chapter 4

Methodology

This chapter will discuss the methodology used in this project, including the technologies chosen to conduct the experiment and the various design decisions made while developing it.

4.1 Environment

Resources on the Amazon Web Services (AWS) were utilized to run the code for the experiments. These resources incur an hourly cost, but the scale available greatly reduced the amount of time that was required to run the necessary experiments. The Kansas State University supercomputer, Beocat[24], was originally utilized for the development of the code and the running of early experiments. This system provided the capability to download many APKs and perform a lot of parallel processing as well as large amounts of storage, but GPU availability was limited on this system. The switch to utilizing AWS was primarily made due to the high availability of GPU systems, which greatly sped up the processing of APKs.

The APKs files were stored on a Simple Storage Service (S3) bucket. The artifacts, as well as the results from the scripts, were also stored on the S3 bucket for greater availability.

Elastic Compute Cloud (EC2) Spot instances were utilized to reduce costs. However these instances present the concern of losing data, so data was periodically uploaded to the

S3 buckets, which would serve as a transaction system. The goal was to ensure that if the EC2 instance were terminated, the process would not have to restart from the beginning.

Spot EC2 instances running the generic Ubuntu 18.04 Amazon Machine Image (AMI) were utilized for extracting the artifacts from the APKs that would be provided to the models as inputs. Compute-optimized instances were prioritized in the selection of instance types due to the multiprocessing utilized to speed up the batch processes of converting the APKs into the artifacts.

Spot EC2 instance running the AWS Deep Learning AMI (based on Ubuntu 18.04) were utilized for running the models on the data. Accelerated computing instances were prioritized in the selection of instance types due to the GPUs which greatly speed up artificial intelligence models. Specficially, the p3.8xlarge instance type was utilized because it had the medium number of cores, in order to be more cost-efficient than some of the instance types with higher cores which still offered GPU capabilities.

4.2 Code

Python was the programming language utilized for the development of the code base associated with the experiments. This language was selected due to the availability of the popular PyTorch and Tensorflow libraries.

Between the PyTorch and Tensorflow libraries, the PyTorch library was selected for these experiments due to the ease of use. The PyTorch model provides a lot of built-in functionality to handle the creation and running of the models.

Due to the large amount of data that needs to be processed by the model and the limited amount of RAM for the EC2 instance, the code reads in a single APK at a time. It performs this by overriding the built-in Dataset object within PyTorch to add custom data handling guidance. Additionally, these modifications should increase readability and reduce the amount of time required for the maintenance of the code.

The code utilized for these experiments can be found on the GitHub repository [1].

[1] https://github.com/jhauserw3241/AndroidMalware

4.3 Experiments

The data distribution and configurations were changed between several experiments. Therefore, each of the possible options are discussed in detail in this section.

N-fold validation was used to ensure that the results were trustworthy. Each configuration will be run five times with different distributions of APKs with the same train, validation, and test configuration.

4.3.1 Data Distribution

For these experiments, APks from the Androzoo repository were used as inputs and VT scores were utilized as labels. To match the work being performed by other researchers within the coalition, a ratio of three benign APKs to one malicious APK was utilized. The usage of this ratio was employed due to the small number of malicious APKs that exist in the overall dataset.

Originally, 172,000 APKs from the collection that Bowling Green State University (BGSU) had gathered were used to run the experiments so that the results could be comparable with the results from other researchers experiments. Specifically, the set of APKs utilized matched the composition shown in Table 4.1.

Table 4.1: *Distribution of data with VT score of 2 threshold*

Year	Total Benign APKs	Total Malicious APKs
2016	30,000	10,000
2017	30,000	10,000
2018	30,000	10,000
2019	30,000	10,000
2020	9,000	3,000

To determine the number of instructions to consider in the LSTM data representation of a single APK, some statistics about the number of instructions in each APK were collected.

- The average number of instructions identified across all the APKs considered was 402,462.

- The median number of instructions identified across all the APKs was 312,349.
- The maximum number of instructions in a single APK was 12,221,962.
- The minimum number of instructions in a single APK was 0.

The fact that an APK had zero instructions introduced a concern about the APKs being parsed correctly by Androguard. To ensure that the APKs being analyzed were analyzed correctly, a review of the number of instructions per APK was conducted.

From the set of 172,000 APKs, many APKs were identified as having less than 100 instructions, as shown in Table 4.2. The 131 APKs with zero instructions were known to be processed incorrectly. The 130 APKs that had less than 100 instructions were considered suspicious of whether they were processed correctly by the APK parser. Therefore, all of the APKs with less than 100 instructions were discarded from analysis.

Table 4.2: *Potentially Incorrectly Processed APKs Distribution*

Year	Total Benign APKs	Benign APKs w/ 0 Instructions	Benign APKs w/ <100 Instructions	Total Malicious APKs	Malicious APKs w/ 0 Instructions	Malicious APKs w/ <100 Instructions
2016	30,000	13	25	10,000	6	53
2017	30,000	19	24	10,000	4	0
2018	30,000	28	11	10,000	2	4
2019	30,000	15	7	10,000	34	1
2020	9,000	9	4	3,000	1	1

A distribution of the number of instructions per each APK for all 172,000 instructions is shown in Figure 4.1.

After starting experiments with considering all Java bytecode instructions, it was identified that it would take over 200 days for the experiments to finish. This was determined to be an unacceptable amount of time and resource usage. To decrease the time the experiments took and the resource usage, many abbreviated experiments were run to determine what number of APKs and how many instructions from each APK could be considered with a reasonable amount of time and resource usage. From these abbreviated experiments, it

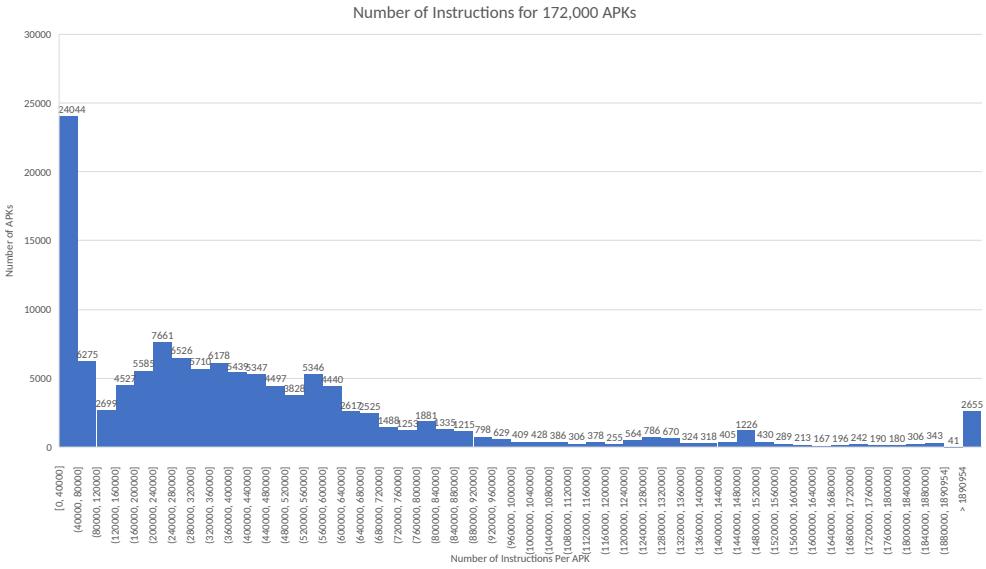

Figure 4.1: *The number of instructions per APK for all 172,000 APKs*

was determined that using a subset of 40,000 APKs and only considering the first 60,000 instructions from each APK was the best option for the time and resource constraints that were present. To determine how reducing the number of APKs and instructions per APK considered, an analysis was performed on the number of instructions in both the original set of 172,000 APK as well as the reduced set of 40,000 APKs. Figure 4.2 shows how many APKs had less than 60,000 instructions from the set of all 172,000 APKs. Similarly, the number of APKs with more than 60,000 instructions is shown in Figure 4.3. There are 27,243 APKs with 60,000 instructions or less and 96,307 that have more than 60,000 instructions. To provide another view of the data, only 597 APKs had more than 2.5 million instructions while 122,953 APKs had less than 2.5 million instructions. Having LSTM data representations that considered 2.5 million instructions would take a long time to process.

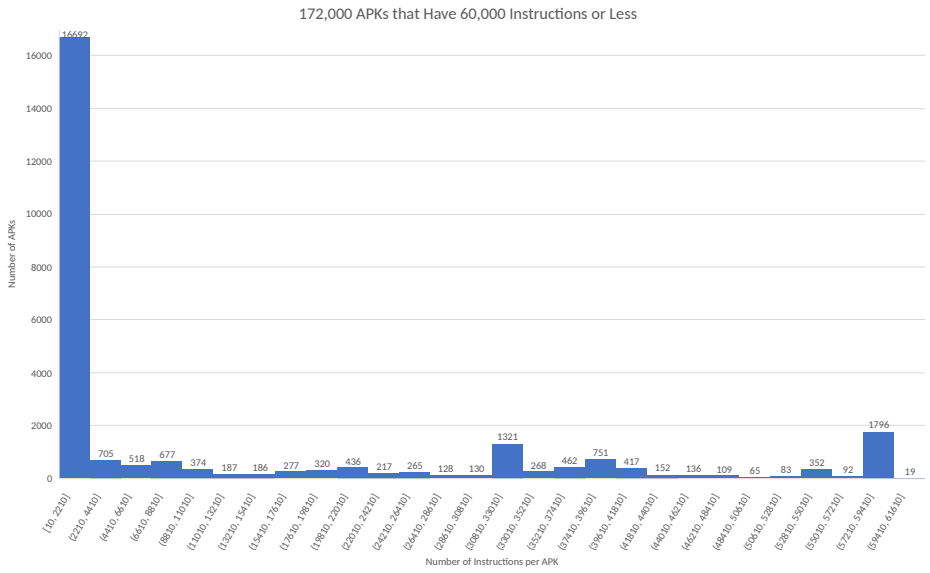

Figure 4.2: *The number of instructions per APK for the set of the 172,000 APKs that had 60,000 instructions or less*

When the dataset was reduced to 40,000 APKs, the composition of the benign to malicious APKs considered per year was captured in Table 4.3. The number of benign to malicious APKs considered for each year were the same. From this dataset, the distribution of how many instructions were present for all the APKs considered is shown in Figure 4.4.

Table 4.3: *Distribution of Data with VT Score of 2 Threshold*

Year	Total Benign APKs	Total Malicious APKs	Total
2016	6,000	2,000	8,000
2017	6,000	2,000	8,000
2018	6,000	2,000	8,000
2019	6,000	2,000	8,000
2020	6,000	2,000	8,000
Total	30,000	10,000	40,000

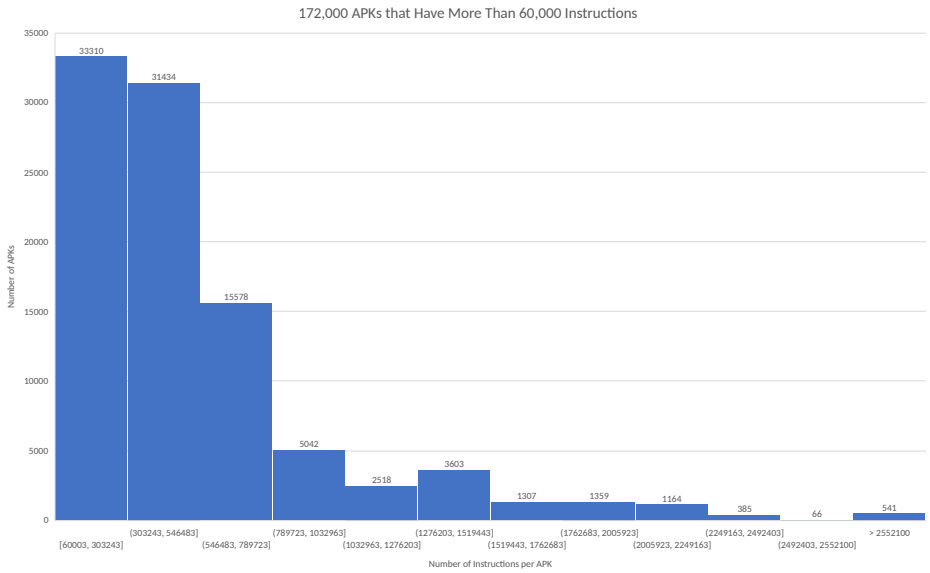

Figure 4.3: *The number of instructions per APK for the set of the 172,000 APKs that had more than 60,000 instructions*

Figure 4.5 shows how many APKs had less than 60,000 instructions from the set of all 40,000 APKs. Similarly, the number of APKs with more than 60,000 instructions is shown in Figure 4.6. There are 8,324 APKs with 60,000 instructions or less and 31,676 that have more than 60,000 instructions. To provide another view of the data, only 232 APKs had more than 2.5 million instructions while 39,768 APKs had less than 2.5 million instructions.

The data preparation steps, described in Section 3.3, were performed on the data described in Table 4.1. From this data preparation, a list of 228 unique bytecode instructions was identified.

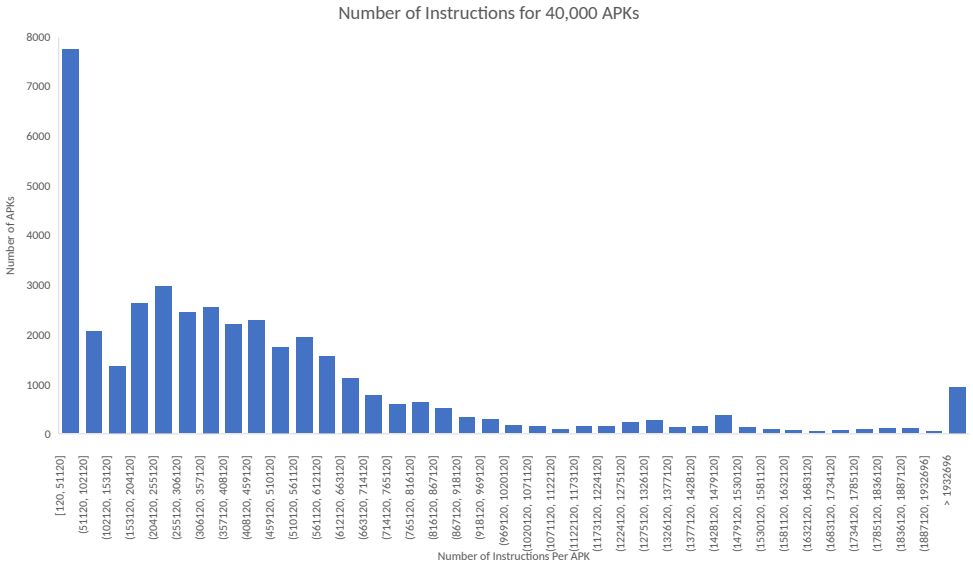

Figure 4.4: *The number of instructions per APK for all 40,000 APKs*

The data was then broken down further based on a set of percentages defined in Table 4.4 to split the data into train, validation, and test datasets. These datasets are used by the model at different stages of the model life cycle. The training dataset is used to train the model on how to identify if the APKs are malicious. The validation dataset is utilized to evaluate performance of the model throughout the training process. Both the training and validation datasets are run through the model once per epoch. The testing dataset is only run through the model once at the end of the training period to evaluate the final performance of the model.

Table 4.4: *Percentage Distribution of Data with VT Score of 2 threshold*

Data Type	Percentage	Total Benign APKs	Total Malicious APKs
Train	70%	21,000	7,000
Validation	15%	4,500	1,500
Test	15%	4,500	1,500

When the 40,000 APKs were split into training, validation, and testing sets, the ratio of three benign APKs to one malicious APK was not intentionally maintained. The APKs were selected randomly from the overall set of 40,000 APKs. However, there is a high likelihood that there is an equivalent representation of the ratio of benign to malicious APKs given the

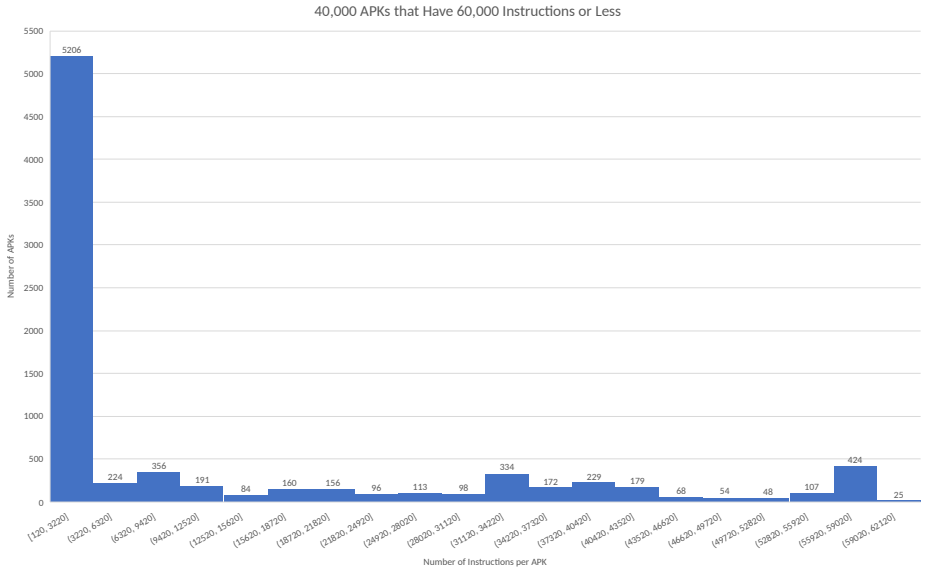

Figure 4.5: *The number of instructions per APK for the set of the 40,000 APKs that had 60,000 instructions or less*

size of the dataset being utilized and the law of large numbers from probability theory. The different datasets where the same 40,000 APKs are re-split into a different set of training, validation, and test APKs are guaranteed to be not be the same exact split of APKs across the three subsets.

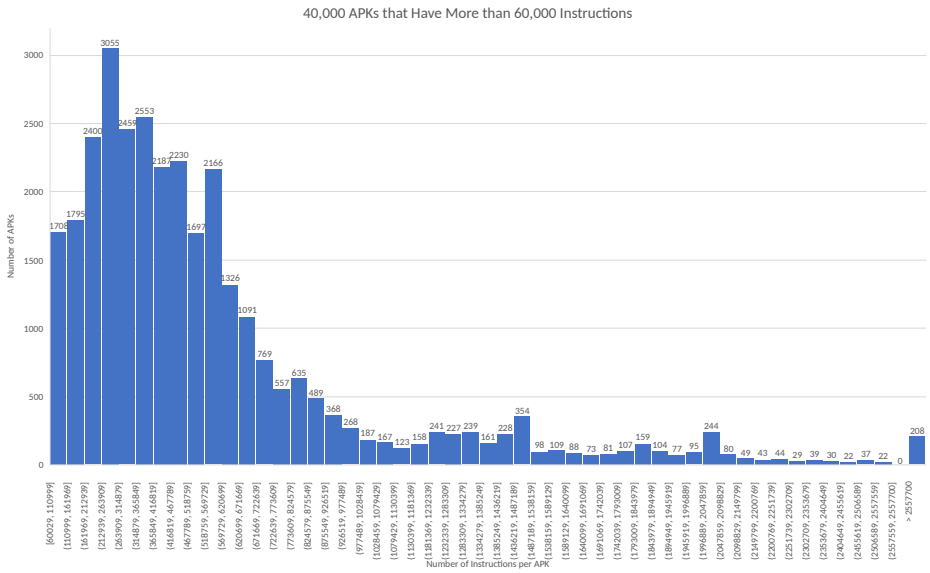

Figure 4.6: *The number of instructions per APK for the set of the 40,000 APKs that had more than 60,000 instructions*

4.3.2 Configurations

Both the logistic regression and LSTM models contained the below configuration options. Some of the configuration options were unchanged between experiments, while others were dynamically changed using several sets of values to identify the best configuration for the models.

Static Configurations

The logistic regression input dimension is set to the number of unique bytecode instructions, 228. More information on how the data is set up for the logistic regression model can be found in Section 5.1.

The LSTM input dimensionality is defined by a combination of the input dimension and the sequence length. The input dimension for the LSTM model was set to 100, representing a set of 100 individual instructions. The decision to utilize "phrases", or sets of 100 instructions, was made to conserve resource usage and the time necessary for the experiments to complete. The sequence length for the LSTM model was set to 600 "phrases". The decision to use only the first 60,000 instructions of the APK was also made to conserve resources

and time. Therefore, the 60,000 instructions divided by one "phrase" of 100 "words", or instructions, results in a sequence of 600 elements. More information on how the data is set up for the LSTM model can be found in Section 6.1.

The output dimension for the logistic regression and LSTM models was set to 1. The output value for both models should be a single binary integer that indicates if the APK is malicious.

To optimize the model, the Adam optimizer function will be utilized as this recommended as a good starting point during the Deep Learning Fundamentals Course at Kansas State University[25].

The APKs were split up into different sets for training the model, validating that the performance of the model throughout the training, and testing the final performance of the model after it had been fully trained. The distribution of APKs from the overall set was 70% of the APKs for training, 15% for validation, and 15% for testing.

The batch size specifies how many APKs are processed before the optimizer function is run on the model and the validation set of data is run through the model. A batch size of 50 APKs per batch was used throughout all the experiments.

Epochs refers to the number of times the model is trained using the training dataset. Initially, all experiments were run with 10 epochs being utilized to train the model. This means the model analyzed the training data 10 times and validated those results 10 times before testing the data on the testing dataset. The decision to use 10 epochs was based on the time limitations of this research. Additionally, this number of epochs also being used by other researchers from BGSU inside of the research coalition that these experiments were done in collaboration with.

The LSTM models were run with just the first 60,000 instructions from each APK. Originally, experiments were run with all of the instructions from the APK but those experiments were expected to take over one hundred days to completed. Based on this elongated time and resource requirement, the decision was made to limit the number of instructions in each APK that were considered.

Dynamic Configurations

The model type could be a variation of the logistic regression or LSTM models. Only one variation of the logistic regression model was utilized. Two LSTM model variations were utilized, a model with an encoding layer and a model without an encoding layer.

The learning rate indicates to the model when attempting to determine how much to change the model in response to the error received. Smaller learning rates are assumed to be more accurate in identifying the correct classification because they don't change as much per iteration so the model can be more precise. However, smaller learning rates usually take longer to train than larger learning rates. To explore this relationship for the dataset used in this experiment, three learning rates were used: 0.01, 0.001, and 0.0001. These learning rates were increased or decreased by the power of ten because the results would better indicate which range of learning rates would be more likely to provide better results.

The LSTM experiments had an additional parameter that was set as a dynamic configuration option, the hidden units size. The size of the units in the hidden layer is the number of cells within the hidden layer for the LSTM models. To determine if the number of hidden layer units had an impact on the performance of the model, two different sizes were utilized: 64 and 128.

For the Logisic Regression experiments, the model took less time to train, validate, and test. Based on the decreased time requirement and decreased resource requirement based on the data input format, the model was run considering the first 60,000 instructions as well as all the instructions in the APK.

The dynamic aspects of the experiment configuration can be summarized by Table 4.5.

These experiment numbers will be referenced when the results of this project are discussed.

The majority of these configurations were selected based on suggestions during the Principles of Artificial Intelligence[26] and the Deep Learning Fundamentals[25] courses at Kansas State University.

Table 4.5: *Experiment Definitions*

Experiment Number	Model Type	# of Instructions Per APK	Learning Rate	Unit Size
Experiment #1	Logistic Regression	60,000	0.01	-
Experiment #2	Logistic Regression	60,000	0.001	-
Experiment #3	Logistic Regression	60,000	0.0001	-
Experiment #4	Logistic Regression	All	0.01	-
Experiment #5	Logistic Regression	All	0.001	-
Experiment #6	Logistic Regression	All	0.0001	-
Experiment #7	LSTM	60,000	0.01	64
Experiment #8	LSTM	60,000	0.001	64
Experiment #9	LSTM	60,000	0.0001	64
Experiment #10	LSTM	60,000	0.01	128
Experiment #11	LSTM	60,000	0.001	128
Experiment #12	LSTM	60,000	0.0001	128

4.4 Success Metrics

To evaluate this binary classification problem, the classification metrics in Table 4.6 were considered.

Table 4.6: *Confusion Matrix*

True Positive (TP): A malicious APK was labeled as malicious	False Positive (FP): A benign APK was labeled as malicious
False Negative (FN): A malicious APK was labeled as benign	True Negative (TN): A benign APK was labeled as benign

Five metrics were captured to determine the performance of the models: loss, accuracy, precision, recall, and the F-score. These terms will be defined below.

The loss function determines the distance between the predicted value and the correct value. A model with a lower loss function is preferred, since that indicates a lower incidence of deviation from the expected classification. The loss function is also called the cost function

or criterion function. The loss function utilized for these experiments was the Binary Cross-Entropy (BCE) loss function[27] which determines the loss for binary values.

The accuracy function determines whether the predicted value matches the correct value exactly. A high accuracy is preferred because it indicates that the model is more likely to correctly classify the APK.

$Accuracy = (TP + TN)/(TP + FP + FN + TN)$

The accuracy can be an unreliable metric as it can be heavily dependent on the dataset being utilized for analysis. Since a 3 to 1 ratio of benign to malicious APKs is being utilized, hard coding the response of benign would result in a 75% accuracy. Therefore, additional metrics are captured to ensure that a complete picture of performance is being provided.

The F-score, or F1-score, is another method to determine the accuracy of the model. The F-score is the harmonic mean of the precision and the recall. The precision is the number of malicious APKs that were labeled correctly over the number of APKs that were labeled as malicious. The recall is the number of malicious APKs that are were labeled correctly.

$Precision = TP/(TP + FP)$

$Recall = TP/(TP + FN)$

$F\text{-}score = 2 * (Recall * Precision)/(Recall + Precision)$

The goal is to maximize both precision and recall up to a value of 1. However, optimizing for one of these two metrics can often reduce the accuracy of the other. For example, optimizing for precision can reduce the recall rate and vice versa.

The sci-kit learn f-score function[28] was utilized to calculate the F-score. The goal is to maximize the value of the F-score to it's maximum of one. The worst value is zero.

TowardsDataScience[29] described some different metrics that could be utilized for evaluating the performance of models.

The performance metrics can be summarized as shown in Table 4.7.

For this specific classification task, the recall should be optimized over the precision. The recall will provide a more accurate picture of how accurately the model is classifying malicious APKs.

The official PyTorch documentation for the Binary Cross-Entropy loss function[27] speci-

Table 4.7: *Performance Metrics Expectations*

Name	Possible Range	Indicator of Improvement
Loss	0 or more	Decrease
Accuracy	0 - 1	Increase
Precision	0 - 1	Increase
Recall	0 - 1	Increase
F-Score / F1-Score	0 - 1	Increase

fied that the expected loss value should be between 0 and 1.

The accuracy value is expected to be between 0.75 and 1. The expected accuracy for most binary classification problems would be between 0.5 and 1 given the fact that random chance of a balanced dataset would result in the result being positive at least 50% of the time. The dataset utilized for these experiments have a ratio of 3 benign APKs to 1 malicious APK. Therefore, random chance should result in a benign result 75% of the time.

When considering benign APKs labeled as benign as the TP case, the model could achieve a precision of 0.75 and recall of 1 with solely labeling each APK as benign and not learning. For the case where malicious APKs labeled as malicious are the TP case, a precision of 0.25 and a recall of 1 could be achieved by labeling each APK as malicious. Therefore would expect the precision and recall values to be above the precision-recall curve that this presents.

Chapter 5

Logistic Regression

A logistic regression model was utilized to generate a baseline set of results and performance metrics that will be used to evaluate the more advanced RNN models. A low accuracy rate was expected due to the naive approach of utilizing such a simple method.

5.1 Data Representation

The input provided to the Logistic Regression model was a bag of words format for identifying which of the unique bytecode instructions existed in the APK. As described in Section 3.3.2, a unique set of bytecode instructions throughout all the APKs in the dataset was created. This unique set of bytecode instructions was used as the "vocabulary" for all the APKs. An array was created for each APK that was the same size as the number of instructions in the "vocabulary" that was filled with zeros. A zero in this array indicates that the specific instruction at that position in the "vocabulary" does not exist anywhere in the APK. The indicators in the array are switched to a one value if that instruction exists anywhere in the APK. This format allows the model to correlate whether the presence or absence of specific bytecode instructions in the APK cause malicious behavior.

5.2 Architecture

The linear regression model was developed using the base Module class[30] in the PyTorch Python library. This model only contained one layer, a linear layer[31]. The bag of words data representation described in Section 5.1 is inputted to the linear layer. A one dimensional output value is returned from this layer.

The forward function for the model applies the Sigmoid function[32] to the output of the linear layer. This operation is what differentiates the logistic regression model from a linear regression model. This binary value is then returned from the model as the predicted value.

The code for these models can be found in Appendix A.

5.3 Results

The logistic regression model was utilized to perform the six logistic regression experiments outlined in Table 4.4. The performance of the model ranged widely for most of the experiments, but a configuration was found that provided performance values within the expected ranges. A full list of the results from all the experiments can be found in Appendix D.

For several experiments, the results showed that the model did not learn from certain datasets. To account for this, both an unrefined and refined views of the results are discussed below. The unrefined view of the results includes all five datasets. The refined view of the results removes the datasets that the model did not learn from. Specifically, if the performance remained the same throughout all of the epochs, then the dataset was removed from the refined view of the model performance. Providing the refined view increases the ease of reviewing the results for those specific datasets. For example, Experiment #1 had three datasets with a consistent loss value of 75 across all epochs. Given the expectation of the loss value existing between 0 - 1, as defined in Table 4.6, the assumption may be made that the model did not learn from these datasets and they were removed from the refined view.

5.3.1 Experiments

For each experiment, an overview table of the unrefined performance throughout the training and validation of the model is shown. These numbers were obtained by taking the average and standard deviation of the last epoch across all five datasets. This provides us an overall average and standard deviation of the performance per experiment.

Table 5.1: *Experiment #1 Train / Validation Performance*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	20.153	32.479	20.159	32.474	18.183	30.196
Accuracy	0.742	0.283	0.734	0.281	0.76	0.27
Precision	0.578	0.423	0.585	0.431	0.66	0.477
Recall	0.628	0.372	0.614	0.37	0.67	0.412
F1-Score	0.545	0.346	0.54	0.342	0.613	0.396

Experiment #1 configured the logistic regression model with a learning rate of 0.01, and only the first 60,000 instructions of each APK were considered when creating the input for the model. The results of the training and validation of the model are provided in Table 5.1. The model learned from three of the five datasets, but didn't learn from dataset #3 and #5. The model not learning on a couple of datasets resulted in a higher loss value as well as lower accuracy. The expected loss value is less than one, but the average of the datasets across the epochs was 20. Additionally, the accuracy was 0.73 which is lower than expected accuracy since there were three times as many benign applications than there were malicious. Similarly low values were found for other three performance metrics. These training and validation results indicate that the model could is labeling at least some number of the benign applications incorrectly. However, the precision and the recall values indicate that at least some of the malicious applications were labeled correctly.

The trend in the training and validation data indicates that it is unlikely that the model would have improved its understanding, and therefore performance, of the three datasets that the model was learning from given additional epochs with the Experiment #1 configuration to train on the data.

Table 5.2: *Experiment #2 Train / Validation Performance*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	50.064	35.242	50.064	35.242	48.055	34.109
Accuracy	0.477	0.315	0.478	0.316	0.5	0.308
Precision	0.321	0.318	0.326	0.329	0.38	0.39
Recall	0.73	0.435	0.729	0.436	0.75	0.433
F1-Score	0.383	0.258	0.385	0.26	0.438	0.332

A learning rate of 0.001 was utilized by the model and the first 60,000 instructions of each APK were considered in Experiment #2. The average and standard deviation of the average performance epochs for the training and validation of the model is shown in Table 5.2. The model only learned from one of the five datasets, dataset #4. The average and standard deviation values in the table show the effect of the first three datasets have significantly worse performance, such as a loss value of 75 for the first three datasets. The model had very high performance for dataset #4 with a learning rate of 0.001, but this success was overshadowed by the model not learning on the other datasets.

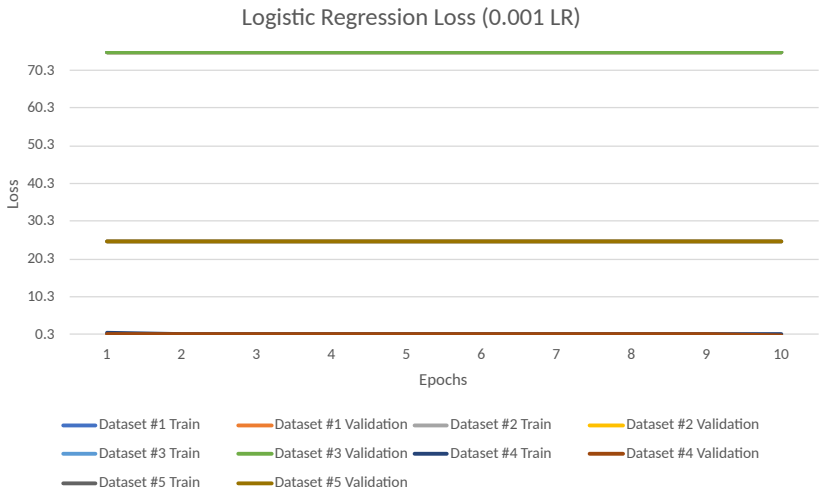

Figure 5.1: *The Loss for Experiment #2 for all five datasets*

This is an interesting observation considering the model learned on datasets #1 and #2 for Experiment #1. One reason this could be the case is that the model was initialized with different coefficients for the Adam optimization function, since the weights for the model are randomized, and these weights went past the point of the learning curve that was optimal

for datasets #1 and #2. Another possible explanation is that the learning rate was smaller so the model did not reach the optimal point in the learning curve for datasets #1 and #2 within the first ten epochs of training. Figure 5.1 provides an unrefined view on how the model learned on all five datasets.

Given more epochs of training, the model may have kept learning and improving it's performance for dataset #4 based on the improvement that was occurring in the later epochs of the training.

Table 5.3: *Experiment #3 Train / Validation Performance*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	5.492	10.905	5.489	10.907	10.198	22.25
Accuracy	0.806	0.035	0.808	0.034	0.86	0.207
Precision	0.531	0.299	0.544	0.305	0.733	0.435
Recall	0.448	0.256	0.443	0.252	0.75	0.433
F1-Score	0.477	0.27	0.478	0.27	0.731	0.418

Experiment #3 specified a model configuration of a learning rate of 0.0001 and the first 60,000 instructions of the APKs under consideration when reviewing each APK. Table 5.3 shows the average and standard deviation of the performance across the epochs. The model learned from every dataset, besides dataset #2. A significant decrease of the average loss and an increase of the other performance metrics resulted from this experiment, compared to Experiments #1 and #2. From the logistic regression experiments that considered the first 60,000 instructions of the APK, this configuration had the best performance by a significant margin.

Based on the later epochs, the model's understanding of the APKs may have improved or deteriorated given more epochs of training. Some datasets appeared as they would have improved. However, particularly in relationship to the accuracy, recall, and the f1-score, the model seemed to be getting worse performance in the later epochs.

All of the instructions in each APK were considered by the model and a learning rate of 0.01 was used in Experiment #4. The average and standard deviation of the performance metrics is shown by 5.4. The model did not learn from datasets #1 and #4, but learned

Table 5.4: *Experiment #4 Train / Validation Performance*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	30.157	40.936	30.158	40.935	28.128	38.873
Accuracy	0.64	0.356	0.641	0.357	0.68	0.363
Precision	0.628	0.346	0.652	0.368	0.72	0.39
Recall	0.823	0.162	0.795	0.188	0.92	0.179
F1-Score	0.621	0.205	0.613	0.198	0.730	0.287

from the remaining three datasets.

There are a couple of interesting points of comparison with the results from Experiment #1. The average of the loss function and accuracy function performed worse when all the instructions from the APK were considered with the same learning rate. Additionally, the average of the precision and recall functions performed better. A possible explanation for the difference in the performance metrics could be that the additional input data confused the model. However, that would indicate that the malicious activities of the APKs were already present in the first 60,000 instructions, and adding the additional instructions further confused the model. Additional research would be required to better determine what the connection is.

The performance of the model would have been unlikely to improve given further epochs of training based on the performance in the later epochs.

Table 5.5: *Experiment #5 Train / Validation Performance*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	0.339	0.011	0.335	0.011	0.283	0.165
Accuracy	0.873	0.008	0.87	0.006	0.86	0.114
Precision	0.811	0.023	0.791	0.02	0.85	0.224
Recall	0.643	0.034	0.657	0.035	0.803	0.187
F1-Score	0.707	0.023	0.707	0.02	0.798	0.14

Experiment #5 had the model configured with a learning rate of 0.001 and all of the instructions in the APK being reviewed by the model. Table 5.5 displays the average and

standard deviation of the performance across the epochs. This experiment was the only experiment where the logistic regression model learned from all five datasets.

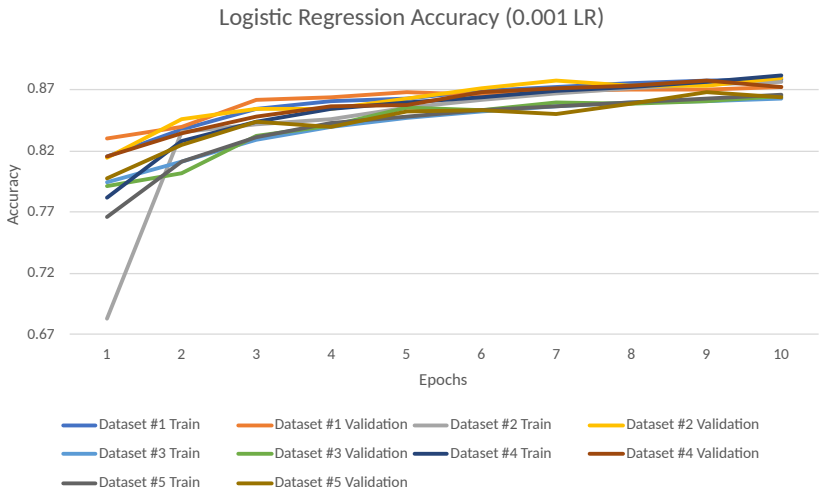

Figure 5.2: *The Accuracy for Experiment #5 considering all five datasets*

Due to the fact that the model learned from all five datasets, the performance of the model was significantly better than the results Experiment #2, where the model was run with the same learning rate but less instructions, and Experiment #4, where the model was run with the same number of instructions but a higher learning rate. One explanation for the improved performance may be that the the additional instructions contained the necessary bytecode instructions to make a more accurate distinction between a malicious and benign application. However, this idea is not supported by Experiment #4 where additional instructions being considered caused the performance to worsen. However, there is possibility of the coupled nature of more instructions considered and a smaller learning rate resulted in better performance. Another possibility is that it was the product of the model initializing at a more optimal point in the learning curve, while that is a less likely explanation. To provide context, the accuracy of the model is shown in Figure 5.2.

The performance of the model may have improved given more epochs of training.

A learning rate of 0.0001 was utilized by the model in Experiment #6 and all of the instructions in each APK were considered. The accuracy and standard deviation of the average performance across epochs is shown in Table 5.6. The model learned from the first three datasets, but not on dataset #4 or #5. The performance of the model significantly

Table 5.6: *Experiment #6 Train / Validation Performance*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	30.334	40.775	30.33	40.778	26.167	35.626
Accuracy	0.601	0.321	0.602	0.322	0.7	0.324
Precision	0.517	0.246	0.522	0.251	0.673	0.327
Recall	0.766	0.218	0.764	0.221	0.96	0.089
F1-Score	0.542	0.139	0.542	0.141	0.744	0.223

decreased between Experiments #5 and #6. This decrease is most likely due to the model did not learn as fast with a lower learning rate. So it may not have found as optimal of an understanding as Experiment #5.

Based on the later epochs, it was identified that for some experiment configurations the model's understanding of the APKs may have improved or deteriorated given more epochs of training. Some datasets appeared as they would have improved. To test this theory, Experiment #3 was re-run with 20 epochs for training instead of 10 epochs. This experiment was selected because it was the most closely correlated to the Experiment #12 which was the LSTM experiment that had the best performance. Experiment #3 was re-run for comparison purposes. The performance of Experiment #3 with an additional 10 epochs of training is shown in Table 5.7.

Table 5.7: *Experiment #3 Train / Validation Performance with Additional Epochs of Training*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	15.486	33.269	15.497	33.263	10.4	22.14
Accuracy	0.707	0.256	0.703	0.254	0.76	0.195
Precision	0.592	0.199	0.578	0.191	0.68	0.295
Recall	0.642	0.203	0.638	0.205	0.817	0.291
F1-Score	0.562	0.099	0.554	0.096	0.699	0.236

The comparison between Table 5.7 and Table 5.7 results in the interesting observation that half of the performance metrics lost efficacy while the other half improved. Specifically, the loss acheived when more epochs of training were utilized was significantly worse with a

value of ten higher than the original value. Additionally, the accuracy decreased about 10%. However, it is interesting to not that the precision, recall, and f1-score improved. In the case of recall by about 0.2, which was a greater than expected increase based on the other logistic regression experiments where only the first 60,000 instructions were considered. Similar to the original version of Experiment #3 where 10 epochs of training were utilized, the version with additional epochs also did not learn from dataset #2.

It is apparent from all of the perforamnce metrics that additional epochs of training would not further improve the performance of Experiment #3. In particular, the accuracy shown in Figure 5.3 displays that the accuracy will not increase further given more training.

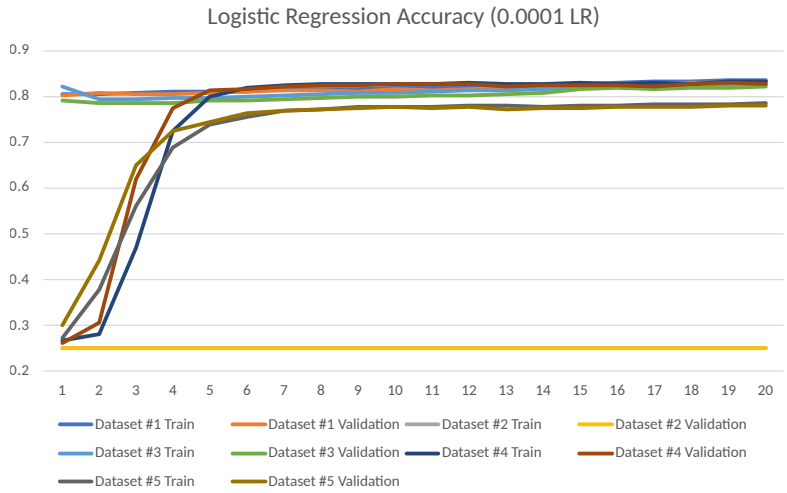

Figure 5.3: *The Accuracy for all five datasets*

5.3.2 Discussion

From all of these results, there are a couple of models that did better than the others from an overall holistic perspective. Experiment #5 did the best due to the balance of the high accuracy of (0.86) along with the lower loss and higher precision, recall, and F1-score, most likely due to the fact that the model was able to learn from all five datasets. Experiment #3 overall had worse performance metrics, but it had a higher recall which is desirable for this classification task. Further research would need to be conducted to determine why the other two experiments where all of the instructions of each APK were considered did considerably

worse than the corresponding experiments with the same learning rate that only considered the first 60,000 instruction from each APK.

Experiment #3 was run with an additional 10 epochs to determine if increasing the number of epochs would improve performance as well as provide comparability to running Experiment #12 with additional epochs. The results from these additional experiment match the understanding that we saw from the testing and validation data where we saw that the performance was not improving across all performance metrics. However, we do see a similar improvement when it comes to the precision, recall, and f1-score performance of the model. A probable reason that Experiment #3 with additional epochs did not perform better than Experiment #5 is that the model was not able to learn from all five datasets.

Chapter 6

LSTM

The LSTM model was the deep learning model utilized to classify the APKs as malicious or not. This model was utilized to answer the question of whether providing the raw bytecode instructions from APKs to a NLP based deep learning model provides an improved performance. The expectation would be to have the performance of the deep learning model surpass the performance of the logistic regression model utilized as a baseline.

6.1 Data Representation

LSTM models expect a sequence of "words" or "phrases" to be passed in as input. For language specific tasks, the LSTM model would then read in these "words" or "phrases" as a sequence and learn based off the differences in the content as well as the sequence of the content. For this project, a "word" is an individual bytecode instruction, and a "phrase" is a group of bytecode instructions. Initially, a sequence of individual "words" were passed to the model, but it was quickly identified that this took additional processing time. So the project changed course to using "phrases" of 100 bytecode instructions as elements of the sequence to pass as input to the model.

A tensor with three layers of abstraction is the expected input for an LSTM model. These tensors can conceptually be thought about as nested arrays.

The last layer of abstraction is a single "phrase". So 100 bytecode instructions were put into a single array. Rather than utilizing the bytecode instructions themselves, an index in the list of unique bytecode instructions from all the APKs was utilized to represent a bytecode instruction. Additionally, if there were not enough bytecode instructions to fill out the "phrase", then zeros were added to pad the array to get to 100 elements.

The second layer of abstraction represents the sequence of all the "phrases" in the APK. Therefore this is an array with multiple "phrases", or array of 100 bytecode instructions, as elements. The number of phrases that are within this array is the number of instructions considered by the model, *60,000*, divided by the number of instructions per "phrase", *100*, which is 600 "phrases". If there were not enough "phrases" within the APK, a zero padded phrase would be added to the second layer array.

The third layer of abstraction represented one APK. This was an additional array that only contained one element, which was the array that contained the sequence of "phrases". This data representation was then passed into the LSTM model as input.

Previously, a one-hot encoding was utilized for each bytecode instruction, rather than the index of where that bytecode instruction existed in the list of unique bytecode instructions. However, the project moved away from this approach given the greater computational and storage requirements that this approach required. The switch greatly increased the processing speed and made it simpler to store the data on a single compute node.

6.2 Architecture

The Python code utilized for the model was built on top of the PyTorch Module class[30].

The LSTM model contains two layers: 1) an LSTM layer[33] and a linear layer[31].

The forward function for the model applies the Sigmoid function[32] to the output of the linear layers. This binary value is then returned from the model as the predicted value.

The source code for this model can be found in Appendix C.

6.3 Results

The LSTM model was utilized to perform the six LSTM experiments outlined in Table 4.4. In many aspects the model performed consistently across all experiments, but there are some major differences between experiments that became apparent upon analysis of the results. All of the LSTM experiments only considered the first 60,000 instructions from each APK. A full list of the results from all the experiments can be found in Appendix E.

Unlike the logistic regression experiments, the model learned from all five datasets throughout all the LSTM experiments. Therefore a refined view was not required for any of the results.

6.3.1 Experiments

For each experiment, an overview table of the unrefined performance throughout the training and validation of the model is shown. These numbers were obtained by taking the average and standard deviation of the last epoch across all five datasets. This provides us an overall average and standard deviation of the performance per experiment.

Table 6.1: *Experiment #7 Train / Validation Performance*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	0.428	0.002	0.429	0.005	0.348	0.174
Accuracy	0.842	0.001	0.841	0.003	0.9	0.071
Precision	0.721	0.003	0.724	0.011	0.933	0.149
Recall	0.6	0.002	0.598	0.013	0.803	0.187
F1-Score	0.644	0.002	0.644	0.005	0.841	0.096

Experiment #7 had the model configured with a learning rate of 0.01 and utilized 64 units in the hidden layer. Table 6.1 shows the average and standard deviation of the average performance across all epochs.

From this first LSTM experiment, the average accuracy is greater than the expected 0.75, which is good since 0.75 was the lower bound from the expected range. The average precision is higher than the recall, which is not bad but we want to prioritize improving the recall.

Overall the standard deviation is significantly lower, throughout all the performance metrics, than what the logistic regression results had which is good because it showed increased consistency. The full results of the accuracy of the model during training training and validation are shown in Figure 6.1.

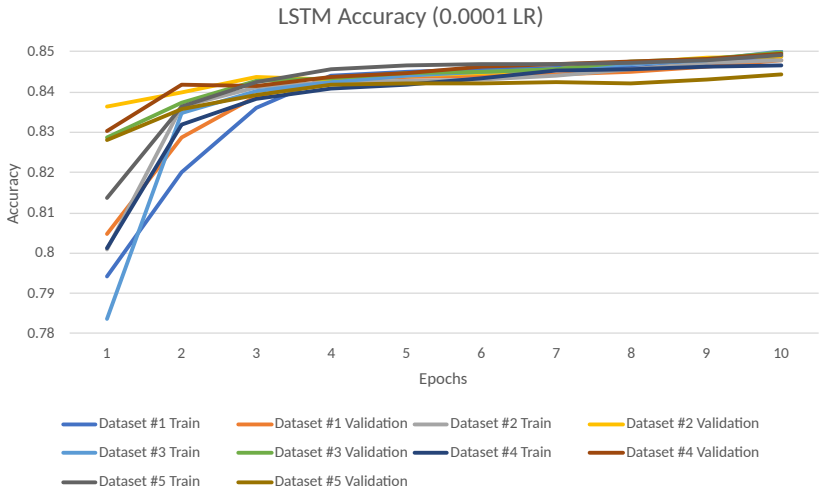

Figure 6.1: *The Accuracy for Experiment #7 across all five datasets*

The performance of the model would have been unlikely to improve given further epochs of training based on the low rate of improvement between epochs.

Table 6.2: *Experiment #8 Train / Validation Performance*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	0.367	0.003	0.374	0.003	0.37	0.209
Accuracy	0.857	0.003	0.857	0.002	0.9	0.071
Precision	0.734	0.005	0.735	0.011	0.933	0.149
Recall	0.672	0.01	0.674	0.019	0.803	0.187
F1-Score	0.693	0.008	0.693	0.008	0.841	0.096

The model utilized a learning rate of 0.001 and 64 units in the hidden layer for Experiment #8. The average and standard deviation of the average performance across all epochs is shown in Table 6.2.

Lowering the learning rate improved all of the performance metrics besides the precision. The standard deviation of the validation performance was lower or equivalent to that in

Experiment #7. However, the standard deviation was in some cases higher when it came to the training performance results, but this is not unexpected since the model is still learning from the dataset.

The performance of the model may have improved with more epochs of training, but the later epochs provided an inconclusive indication.

Table 6.3: *Experiment #9 Train / Validation Performance*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	0.368	0.004	0.379	0.006	0.367	0.18
Accuracy	0.848	0.001	0.848	0.002	0.9	0.071
Precision	0.728	0.003	0.728	0.01	0.933	0.149
Recall	0.631	0.008	0.627	0.006	0.803	0.187
F1-Score	0.666	0.005	0.664	0.004	0.841	0.096

Experiment #9 had the model configured with a learning rate of 0.0001 and utilized 64 units in the hidden layer. Table 6.3 shows the average and standard deviation of the average performance across all epochs.

The average performance of Experiment #9 is slightly worse than the performance of Experiment #8. Since the change in the configuration was that Experiment #9 had a lower learning rate, one possible explanation may be that the model didn't learn to the optimal point in the learning curve where the higher performance achieved in Experiment #8 was found.

In addition, is worth noting, that the performance of the precision function presented a lot of oscillation throughout the epochs. Figure 6.2 displays the oscillation of the precision across all the datasets throughout the training and validation of the model. While there is not a large amount of oscillation, unlike the other performance metric results, the value continues to increase and decrease across all the epochs.

The model performance would have likely improved given further epochs of training, based on the observations of the later epochs.

The model utilized a learning rate of 0.01 and 128 units in the hidden layer for Experiment #10. The average and standard deviation of the average performance across all epochs is

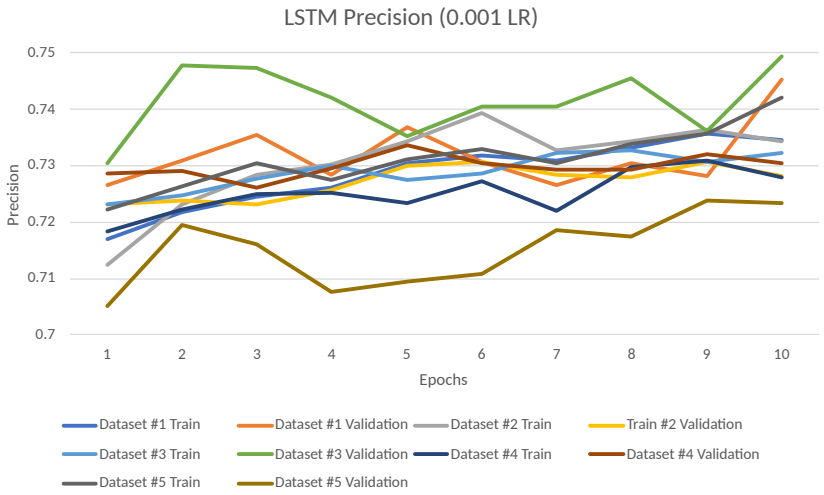

Figure 6.2: *The Precision for Experiment #9 across all five datasets*

Table 6.4: *Experiment #10 Train / Validation Performance*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	0.431	0.001	0.432	0.005	0.354	0.143
Accuracy	0.841	0.001	0.841	0.003	0.9	0.071
Precision	0.721	0.003	0.717	0.013	0.933	0.149
Recall	0.599	0.002	0.592	0.013	0.803	0.187
F1-Score	0.644	0.003	0.639	0.011	0.841	0.096

shown in Table 6.4.

The average performance of this experiment is very similar to the performance of both Experiment #7, which has the same learning rate but a different number of units in the hidden layer. There is a variance of about 0.01 either way between the averages of these two experiments. This may indicate that the increase of the number of units in the hidden layer had a minimal impact on the performance of the model.

The model would have been unlikely to have improved performance from additional epochs of training, based on the low rate of improvement in the later epochs.

Experiment #11 had the model utilize a learning rate of 0.001 and a hidden layer with 128 units. Table 6.5 shows the average and standard deviation of the average performance across epochs.

Similar to Experiment #10, the variance of the performance between Experiment #11

Table 6.5: *Experiment #11 Train / Validation Performance*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	0.366	0.002	0.376	0.006	0.341	0.163
Accuracy	0.858	0.002	0.857	0.002	0.9	0.071
Precision	0.733	0.004	0.737	0.006	0.933	0.149
Recall	0.677	0.008	0.673	0.016	0.803	0.187
F1-Score	0.695	0.005	0.694	0.008	0.841	0.096

and Experiment #8 was 0.01 increase or decrease. Several of the performance metrics were slightly better than Experiment #8, within that 0.01 margin. This data further supports the theory that increasing the number of units in the hidden layer had minimal effect on the learning of the model.

The performance of the model may have improved given further epochs of training, but the results of the later epochs provided inconclusive evidence to make an assertion.

Table 6.6: *Experiment #12 Train / Validation Performance*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	0.329	0.005	0.352	0.001	0.333	0.168
Accuracy	0.866	0.004	0.865	0.005	0.9	0.071
Precision	0.741	0.003	0.739	0.011	0.933	0.149
Recall	0.714	0.017	0.71	0.016	0.803	0.187
F1-Score	0.718	0.01	0.716	0.009	0.841	0.096

The model utilized a learning rate of 0.0001 and 128 units in the hidden layer for Experiment #12. The average and standard deviation of the average performance across all epochs is shown in Table 6.6.

Experiment #12 did have a distinct performance improvement over Experiment #9 that went beyond the 0.01 margin found with Experiments #10 and #11, but the margin did not increase much further. Of particular note, the performance of the recall function had the most increase. The recall attained by Experiment #12 is shown in Figure 6.3. From this graph, the very minimal amount of oscillation and the increase across epochs is apparent.

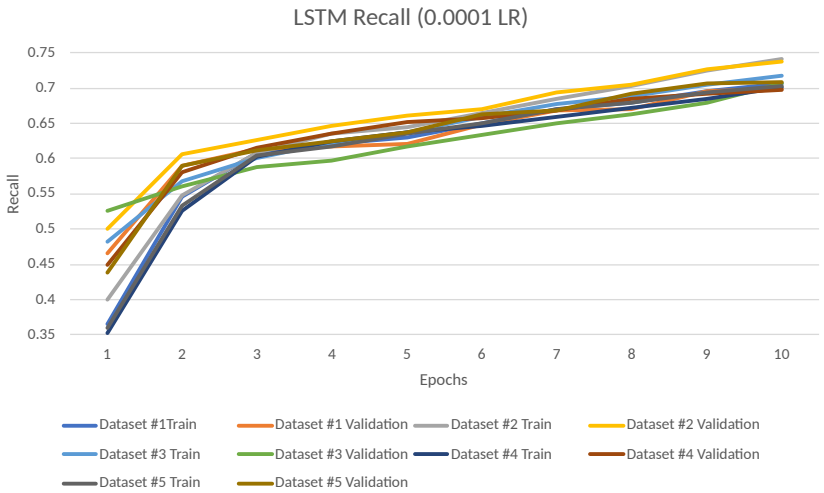

Figure 6.3: *The Recall for Experiment #12 across all five datasets*

Based on the results that were achieved, the performance would have likely improved given additional epochs of training, based on the performance in later epochs. To test this theory, the LSTM configuration that had the best overall performance was re-run with 20 epochs instead of 10 epochs for training. Experiment #12 was selected for additional experimentation because it had the best performance among the LSTM experiments. The results from the experiment with the same configuration and additional epochs are shown in Table 6.7.

Table 6.7: *Experiment #12 Train / Validation Performance with Additional Epochs of Training*

Performance	Train Average	Train St. Dev.	Validation Average	Validation St. Dev.	Test Average	Test St. Dev.
Loss	0.305	0.006	0.344	0.005	0.358	0.169
Accuracy	0.874	0.003	0.871	0.004	0.9	0.071
Precision	0.747	0.003	0.745	0.013	0.933	0.149
Recall	0.749	0.014	0.736	0.014	0.803	0.187
F1-Score	0.744	0.008	0.736	0.009	0.841	0.096

When comparing the results in Table 6.7 to the results from the same experiment with fewer epochs, results in Table 6.6, there is improvement across all the performance metrics. However, the performance increased by 0.01 or 0.02 for all the performance metrics. While an additional increase in performance would have been desired, this does support the idea

that additional epochs of training, even beyond 20 epochs, may continue to improve the performance. Figure 6.4 shows that the performance of the accuracy function showed potential to increase even beyond the 20 epochs.

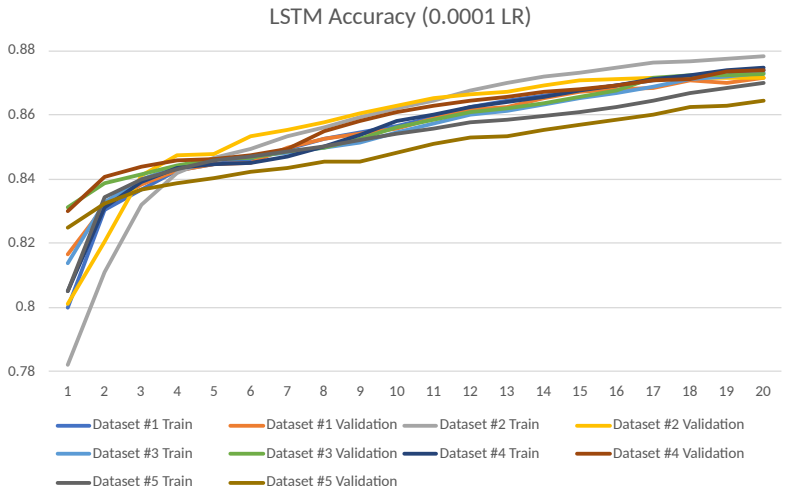

Figure 6.4: *The Accuracy for all five datasets*

6.3.2 Discussion

Based on the loss function values, Experiment #12 could be considered the experiment / configuration that performed the best.

From all of these results, it is apparent that there is not enough data being provided to the model to achieve more accurate results. This result was anticipated when the decision was made to only provide the first 60,000 instructions of the APK when some of the APKs in the dataset had 12 million instructions. The decision was made to run the model on a subset of the instructions in the APK due to the time and cost required to run these experiments using the full set of instructions in each APK would be too high to justify the effort.

However, the very similar results across all configurations that had different learning rates was surprising. This may indicate that with this size of a dataset, the learning rate was less relevant for this binary labeling task.

Additionally, the fact that the model performed worse in some cases with the larger number of 128 units in the hidden layer was interesting and may indicate that attempting a

smaller unit size, 32 for example, would provide better results. Another possible explanation is that the number of units in the hidden layer may have less of an effect on the results of the model. Additional experimentation would be required to make an authoritative determination. Experiment #10 appeared to display a small amount of overfitting to the dataset, while Experiment #6 showed a smaller amount of overfitting. Experiment #7 and Experiment #11 showed a small amount of overfitting toward the earlier epochs, but the concern was primarily resolved in the later epochs. The final epochs training showed more closely correlated results for Experiment #9, but Experiment #12 achieved more desirable results. Based on this relationship, having additional units in the hidden layer was beneficial for Experiment #12 while it may have been less beneficial for Experiments #10 and #11.

These theories are supported by the results of re-running Experiment #12 with additional epochs. When Experiment #12 was re-run with 20 epochs for training instead of 10, the accuracy, precision, recall, and f1-score values remained the same for the average of performance across all five datasets. However, the performance of the loss function actually had worse performance as the loss went from 0.332 for 10 epochs to 0.358 for 20 epochs. These results when testing the data may indicate that it would be better to focus future analysis on providing more data to the model or decreasing the number of units in the hidden layer rather than increasing the number of epochs for training.

Overall, the performance of the model improved throughout the training and validation as the learning rate shrank and the number of units in the hidden layer increased.

Chapter 7

Future Work

7.1 Lessons Learned

There are several lessons that can be learned from the experiments performed for this thesis. These lessons can be summarized as follows.

- Data collection took a significant amount of time beyond what it was expected to take. Getting access to good data should be the first priority when starting a new project as it can take a large amount of time to collect.
- Cleaning of the Android APKs and VirusTotal information also took a significant amount of time. Several planning sessions should be held up front to determine how to format the data because it may take days or weeks to format a large number of APKs or it may be very computationally expensive.
- Properly cleaned data is necessary to get good results from a model.
- Planning the format for the data being provided to the model up front is necessary to ensure that the expected result is returned.

7.2 Future Work

From this thesis, several different areas of future work were identified including improving the current approach or potential new approaches to the same task.

As discussed in the results sections for the two models, providing the models with additional data would probably improve the performance. Especailly for the LSTM model, providing it more instructions from each APK or additional APKs may provide that model increased performance given it was increasing at such a low rate.

It was identified through the LSTM experiments that increasing the number of units in the hidden layer from 64 to 128 did not have a great effect on the performance of the model. A potential future work item would be to run the same experiment with 32 units in the hidden layer and identify if decreasing the complexity increases the performance.

As discussed in the Conclusion 8.3, one potential reason behind why the logistic regression model did not learn from some datasets may have been to the malware family association. It may have been the case that all the malware samples utilized for training, validation, or testing were from the same family so the initial impression of the model did not need to change which is why it did not learn. Additional research would need to be performed on the set of APKs utilized from Androzoo for this research to determine if this is the case.

Another approach to the same problem would be to utilize a Bidirectional Encoder Representations from Transformers (BERT) model, and comparing the results from the BERT model with the LSTM model results. Running both these models would provide a good contrast due to the different approaches of these models that are both utilized for natural language processing. A BERT model is a transformer model that addresses natural language processing tasks. Also, it would be good to see which model is faster. Speed is crucial when determining if an application is malicious, not only within the research community, but also industry.

When reviewing the number of instructions in each APK, many APKs were identified with less than 100 instructions. The fact that some APKs had such a small number of instructions provides some suspicion on if they were being parsed correctly by Androguard.

Therefore, an additional avenue for analysis would be to review other APK parser libraries and compare their performance.

Another question that this research brings up, is what sequences of instructions are getting flagged as malicious. Since the LSTM model is able to identify that an APK is malicious or benign correctly 90% of the time, what sequences does it consider malicious. To be able to identify which sequences it has picked up on, additional research would need to be performed on libraries that would return the sequences that the LSTM model percieces as malicious after training.

This work focused on supervised learning models to reduce the amount of time for manual review of APKs. However, future work could look into semi-supervised techniques using a combination of dynamic and static analysis. However, this approach would require manually labeling a large number of APKs. This would require setting up an environment where manual analysis of the APKs could be performed, which would be resource expensive as well.

A limitation of the dataset that was pulled from the Androzoo repository was the small set of APKs that were labeled malicious according to the thresholds considered within these experiments. One possible solution that was considered was creating synthetic malicious APKs to increase the number of benign APKs that could be utilized from the dataset.

Chapter 8

Conclusion

8.1 Limitations

The results produced by this work should be examined while keeping in mind the below limitations. There were limitations on both the inputs utilized from Androzoo as well as the labels utilized from VirusTotal.

Limitations of the inputs from Androzoo:

- The Androzoo repository does not ensure that there are no duplicates of an APK from multiple Android market places. So it may be possible that an APK was uploaded from the Google Android marketplace as well as another marketplace and the applications would show up as two separate applications in Androzoo. No validation was performed within this research to ensure that APKs were not the same APK from different marketplaces.

- Similarly, the Androzoo repository may upload a different version of the same APK. No validation was performed as part of this research to determine if duplicate versions of an application were included in the dataset.

- The majority of the applications that were analyzed were free applications, which is what was available on Androzoo. The caveat being that some of the applications

Androzoo has in their repository may have been paid applications that were acquired for free by the Androzoo maintainers via BitTorrent.

Limitations of the labels from VirusTotal:

- The labels from VirusTotal are based on the opinions of several different anti-virus solutions to determine what score should be provided for each APK. Replacing a signature-based review of the APKs from anti-virus solutions with a deep learning model that makes assertions based on the opinions from many signature-based solutions may be counter-intuitive. However, the LSTM model is perform a much deeper review of the APKs to determine the correlation between specific sequences of instructions. To replace the VirusTotal scores with a different approach for labels would have been very expensive from a time perspective so we accepted this limitation.

- There was a very small threshold for which APKs were considered malicious according to VirusTotal. Every APK that has a score of two or more when there is a top score of 70 is rather low. However, this decision was made due to the small number of malicious APKs in the dataset if the threshold was set to 10 anti-virus solutions had to agree that the APK was malicious.

- When pulling down the information about an APK from VirusTotal, the most recent scan was utilized. This means that if there was an existing scan, no attempt was made to re-scan the APK. This presented a concern from the perspective that our understanding of malware changes often and the last time that an APK had been scanned may have been over three years ago. This risk was accepted by a decision of the research coalition.

8.2 Similar Works Comparison

The DeepRefiner tool and the work of Prabesh Pathak were very similar to the work conducted by this thesis. A comparison of the results for these approaches to the performance achieved within this work are below.

8.2.1 DeepRefiner

DeepRefiner[12] was similar to the work within this thesis in many ways. First, both works were performing binary classification of APKs on whether the APKs are malicious. One piece of DeepRefiner analyzed the Java bytecode instructions from the APKs with a LSTM model to determine if the APK was malicious. When it comes to the second stage of DeepRefiner where an LSTM was utilized, the big difference between this project and DeepRefiner was an additional abstraction layer applied by DeepRefiner. DeepRefiner collected the raw Java bytecode instructions from the APK, but abstracted each instruction out to one of fifteen classes for types of Java bytecode instructions. However, this project retained each individual instruction as separate, but provided an abbreviated representation to the model to reduce memory size. So instead of providing a string containing the actual bytecode instruction, this work provided the index of where that work existed in the list of unique bytecode instructions across all APKs in the dataset. Based on these similarities, it is worth while to compare the results of the second stage of DeepRefiner and this project.

Only the results of the second stage of DeepRefiner were included here because that stage is the most similar to this thesis. DeepRefiner was able to achieve 96.14% accuracy in labeling APKs as malicious or benign. Additionally, it had a true positive rate of 96.47%. Overall 683 out of 16,644 benign APKs were labeled incorrect, and 448 out of 12,676 malicious APKs were labeled incorrectly.

There are some limitations in terms of how reliably these two works can be compared. For this thesis, 40,000 APKs were analyzed with a ratio of three benign APKs for every malicious APK. The first layer of DeepRefiner handled the majority of the 110,440 APKs utilized to train, validate, and test DeepRefiner. Therefore, the second layer of DeepRefiner, or LSTM layer, utilized only 16,644 benign and 12,676 malicious APKs. These numbers of APKs show that the dataset utilized for the second layer was an unbalanced dataset which causes concern for the comparability of the results. Additionally, the ratio of the APKs utilized by DeepRefiner was closer to one benign APK for every malicious APK which would indicate that we would expect better results than a three benign APKs to one malicious APK ratio.

Additionally, the benign APKs that were utilized by DeepRefiner for review were collected from March 2016 to May 2016, while the malicious APKs were collected before 2015. There may be some differences in the application architecture or other fundamental differences in the APKs that would allow it to more accurately label the APKs based on their age.

Based on the comparison of DeepRefiner and this thesis, we can discern that evaluating a higher level representation of the Java bytecode instructions may lead to better results. More research would need to be conducted given the fact that different datasets and ratios of benign to malicious APKs were used. Additionally, if this thesis moved to using a higher level representation, as suggestion for a maintenance plan would be required similar to the maintenance that would be required for the DeepRefiner tool.

8.2.2 Comparison with BGSU Group

Bowling Green State University (BGSU) was another university in the research coalition who performed similar research[34] to solve the same research question. They performed similar work to the work performed in this thesis from the perspective that they also utilized LSTM models to perform binary classification on APKs specifying if they are malicious. However, this work utilized a standard LSTM while the work performed by BGSU utilized a bi-directional LSTM.

Collaboration with BGSU led to the selection of the original 172,000 APKs utilized for training, validation, and testing of the model. While this work reduced the number from 172,000 to 40,000 APKs based on resource and time concerns, the work of BGSU utilized the full 172,000 APKs originally selected for their research. Additionally, collaboration with BGSU led to the selection of a VirusTotal score of 2 or more being the threshold for an APK being malicious. When the threshold was set to a higher value, there were very few APKs to be utilized as malicious APK inputs for the models. These works are more comparable given the usage of the same inputs, the same labels, the same learning rate, and the same type of model for evaluation. The distinction here is that this thesis used a reduced set of the same inputs and labels. Additionally, BGSU incorporated experiments with the learning

rate of 0.01. Equivalent configurations were run in this thesis, but additional experiments were run as well to determine if there is a configuration that provides better performance. The primary difference between these two works is that BGSU's work considers the API calls within the APK rather than reviewing the raw Java bytecode instructions.

Table 8.1: *Performance Comparison between BGSU and this work*

Model	Precision	Recall	F1-Score
BGSU Bi-LSTM Attn	0.8895	0.8525	0.8706
KSU LSTM	0.933	0.803	0.841

Table 8.1 shows the precision, recall, and f1-score that the bi-directional model BGSU ran was able to achieve. The performance that BGSU's model achieved was more balanced than the performance obtained by the LSTM model within this work. One reason for this may be that a bi-directional LSTM could be more effective for these types or problems. Another potential reason that BGSU's work achieve improved performance is that evaluating the API calls may provide the granularity necessary to determine if an APK is malicious. Given the fact that there are a smaller number of API calls than bytecode instructions, a more holistic view of the API calls in each APK were considered in BGSU's work. However, only the first 60,000 instruction of each APK were considered in this work, so indicators of malicious behavior may have been missed. But the amount of resources necessary may be too much to utilize in a practical environment without abstracting out the individual instructions to a higher level representation.

8.3 Conclusion

This project used both a Logistic Regression Model and an LSTM model to categorize Android APKs into malicious or benign categories. Six different configurations of these models were used to determine the configuration that would provide optimal performance for the dataset. Five-fold validation was performed to provide additional assurance that the results were as accurate as possible and not the result of a favorable split between training

and test datasets.

The logistic regression model had results that spanned the full range of expected values. It was obvious that some datasets did not result in the model learning anything useful for some experiment configurations. However, those same datasets did produce learning in the model under other configurations. The experiment that used a learning rate of 0.001 and only included the first 60,000 instructions of the application only appeared to have one dataset that learned anything beyond the initial value identified. In contrast, the experiment where a learning rate of 0.0001 was utilized and the first 60,000 instructions were considered produced results that showed the model learned across all five datasets, with most measurements of learning quality increasing as the number of epochs increased. The rest of the configurations had a mix of datasets that did or did not learn across the epochs. Since there were so many datasets that did not learn information across the epochs, a refined view of the results was provided to focus on the datasets that appeared to be learning.

From the logistic regression results, a relationship between the number of instructions considered and the performance of the model cannot necessarily be established. This is most likely due to the bag of words data representation, which may prevent the performance of the model from increasing as the number of instructions being considered also increases. On the same topic, the training and validation phase results for some of the experiments indicated that the bag of words data representation for malicious applications may be common across all the malicious applications considered in the testing and validation, given the fact that the model did not learn after the initial value. This could be due to a variety of factors such as the number of malicious applications in the training and validation datasets, or it could be due to all of the malware samples in that dataset being of the same malware type or family. Additional research to determine if there one or two data representations that represented all the malicious APK samples in the training dataset would be interesting to explain why the model did not learn after the initial epoch. However, given the large variety of potentially contributing factors and the limited expected performance of this naïve approach, it may not be worth the time to further analyze this question.

The results for the LSTM model consistently showed that all five datasets were actively

learning across the various configurations. Therefore, no refined view of the data was required. For the training and validation of the model, the performance appeared to generally improve as the learning rate increased and the number of units in the hidden layer grew. However, the results of the testing of the model were surprising, as four of the five performance metrics were the same across all the six different model configurations. The only performance metric that was different was the loss.

The fact that the LSTM experiments across six different configurations produced similar results points to a potential future efficiency. One interpretation of these results is that the models could not perform better with the limited information that was provided to the models. While 60,000 instructions is not necessarily a small amount of input, some APKs had up to 12 million instructions in the APK, which is a significantly higher number. Therefore, more refined results would most likely be achieved from running the model with more instructions. However, this was not done in this project given the vastly larger time and resource commitment required to train the models using such a large dataset.

Bibliography

[1] Brandon Amos, Hamilton Turner, and Jules White. Applying machine learning classifiers to dynamic android malware detection at scale. *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 1666–1671, August 2013. URL doi:10.1109/IWCMC.2013.6583806.

[2] Rudolf J Freund, William J Wilson, and Ping Sa. *Regression analysis*. Elsevier, 2006.

[3] F Graybill. A.(1961): An introduction to linear statistical models.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. The MIT Press, 2016. ISBN 0262035618.

[5] Zhenlong Yuan, Yongqiang Lu, and Yibo Xue. Droiddetector: Android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21.1:114–123, February 2016. URL https://ieeexplore.ieee.org/ielx7/5971803/7399276/07399288.pdf.

[6] Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou. Android malware clustering through malicious payload mining. *Research in Attacks, Intrusions and Defenses Lecture Notes in Computer Science*, pages 192—-214, October 2017. URL arxiv.org/abs/1707.04795.

[7] Average number of new android app releases via google play per month from march 2019 to february 2021. https://www.statista.com/statistics/1020956/android-app-releases-worldwide/. Accessed: 2021-04-18.

[8] Android and google play statistics. https://www.appbrain.com/stats. Accessed: 2021-04-18.

[9] App download and usage statistics (2020). https://www.businessofapps.com/data/app-statistics/, . Accessed: 2021-04-18.

[10] Mobile app download and usage statistics (2021). https://buildfire.com/app-statistics/, . Accessed: 2021-04-18.

[11] Daniel R. Thomas, Alastair R. Beresford, and Thomas Coudray. The lifetime of android api vulnerabilities: Case study on the javascript-to-java interface. *Security Protocols XXIII*, 9379:126–138, November 2015. URL doi:10.1007/978-3-319-26096-9_14.

[12] Ke Xu, Yingjiu Li, Robert H. Deng, and Kai Chen. Deeprefiner: Multi-layer android malware detection system applying deep neural networks. *2018 IEEE European Symposium on Security and Privacy (EuroSP)*, pages 473–487, April 2018.

[13] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, page 45–54, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324885. doi: 10.1145/2517312.2517315. URL https://doi.org/10.1145/2517312.2517315.

[14] Carl Sabottke, E. Tanner, and R. Johnson. Improved malware clustering using virustotal meta data. URL https://www.semanticscholar.org/paper/Improved-Malware-Clustering-Using-VirusTotal-Meta-Sabottke-Tanner/73241b4e2f2f5e6833b7f621987a3c1358da2959.

[15] Yuping Li, Sathya Chandran Sundaramurthy, Alexandru G. Bardas, Xinming Ou, Doina Caragea, Xin Hu, and Jiyong Jang. Experimental study of fuzzy hashing in malware clustering analysis. *8th Workshop on Cyber Security Experimentation and Test - CSET '15*, August 2015.

[16] M. Zubair Rafique, Ping Chen, Christophe Huygens, and Wouter Joosen. Evolutionary algorithms for classification of malware families through different network behaviors.

GECCO '14 Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation, pages 1167–1174, July 2014. URL doi:10.1145/2576768.2598238.

[17] Fengguo Wei, Sankardas Roy, Xinming Ou, and Robby. Amandroid. *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*, pages 1329–1341, November 2014. URL doi:10.1145/2660267.2660357.

[18] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. *2010 5th International Conference on Malicious and Unwanted Software*, pages 55–62, December 2010. URL doi:10.1109/MALWARE.2010.5665792.

[19] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization. *CCS '08 Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62, October 2008. URL doi>10.1145/1455770.1455779.

[20] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: Hindering dynamic analysis of android malware. *EuroSec '14 Proceedings of the Seventh European Workshop on System Security*, April 2014. URL doi:10.1145/2592791.2592796.

[21] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 468–471, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4186-8. doi: 10.1145/2901739.2903508. URL http://doi.acm.org/10.1145/2901739.2903508.

[22] Virus total api. https://developers.virustotal.com/reference, .

[23] Virus total how it works. https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works, .

[24] Beocat. https://support.beocat.ksu.edu/BeocatDocs/index.php?title=Main_Page.

[25] Doina Caragea. Deep learning fundamentals. University Lecture, 2020.

[26] William Hsu. Principles of artificial intelligence. University Lecture, 2019.

[27] Binary cross-entropy (bce) loss function. https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html. Accessed: 2021-01-20.

[28] sklearn.metrics.precision_recall_fscore_support. https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html. Accessed: 2020-03-1.

[29] Accuracy, recall, precision, f-score specificity, which to optimize on? https://towardsdatascience.com/accuracy-recall-precision-f-score-specificity-which-to-optimize-on-867d3f11124. Accessed: 2021-02-10.

[30] Pytorch module class. https://pytorch.org/docs/stable/generated/torch.nn.Module.html, . Accessed: 2020-01-05.

[31] Pytorch linear layer. https://pytorch.org/docs/stable/generated/torch.nn.Linear.html, . Accessed: 2020-01-05.

[32] Pytorch sigmoid function. https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html#torch.nn.Sigmoid, . Accessed: 2020-01-05.

[33] Pytorch lstm layer. https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html, . Accessed: 2020-01-05.

[34] Prabesh Pathak. Leveraging attention-based deep learning neural networks for security vetting of android applications. Master's thesis, Bowling Green State University, 2021.

Appendix A

Logistic Regression Model Code

```
# Class pulled from
   https://medium.com/biaslyai/pytorch-linear-and-logistic-regression-models-5c5f0da2cb9
class LogisticRegression(torch.nn.Module):
   def __init__(self, input_dim, output_dim):
      super(LogisticRegression, self).__init__()
      self.linear = torch.nn.Linear(input_dim, output_dim)

   def forward(self, x):
      y_pred = F.sigmoid(self.linear(x))
      return y_pred
```

Appendix B

LSTM Base Model Code

```
class LSTMModel_Base(nn.Module):
    def __init__(self, input_dim, sequence_length, units, output_size):
        super(LSTMModel, self).__init__()

        # Initialize parameters
        self.input_dim = input_dim
        self.sequence_length = sequence_length
        self.hidden_dim = units

        # Create embedding layer
        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)
        # Create LSTM layer
        self.lstm = nn.LSTM(input_dim, units)
        # Create linear layer
        self.fc = nn.Linear(units, output_size)

    def forward(self, encoding, hidden):
        # Create embedding of encoding
```

```
        embeds = self.word_embeddings(encoding)
        # Run the data through the LSTM layer
        output, hidden = self.lstm(embeds.view(len(encoding), 1, -1))
        # Run the data through the linear layer
        output = self.fc(output)
        # Run a sigmoid operation on the data
        output = torch.sigmoid(output)

        return output, hidden

    def initHidden(self):
        # Initialize the hidden state
        return (torch.randn(1, self.sequence_length, self.hidden_dim),
                torch.randn(1, self.sequence_length, self.hidden_dim))
```

Appendix C

LSTM No Embeddings Model Code

```
class LSTMModel_NoEmbeddings(nn.Module):
    def __init__(self, input_dim, sequence_length, units, output_size):
        super(LSTMModel, self).__init__()

        # Initialize parameters
        self.input_dim = input_dim
        self.sequence_length = sequence_length
        self.hidden_dim = units

        # Create LSTM layer
        self.lstm = nn.LSTM(input_dim, units)
        # Create linear layer
        self.fc = nn.Linear(units, output_size)

    def forward(self, encoding, hidden):
        # Run the data through the LSTM layer
        output, hidden = self.lstm(encoding, hidden)
        # Run the data through the linear layer
```

```
        output = self.fc(output)
        # Run a sigmoid operation on the data
        output = torch.sigmoid(output)

        return output, hidden

    def initHidden(self):
        # Initialize the hidden state
        return (torch.randn(1, self.sequence_length, self.hidden_dim),
                torch.randn(1, self.sequence_length, self.hidden_dim))
```

Appendix D

All Logistic Regression Experiment Results

D.1 Learning Rate of 0.01

D.1.1 First 60,000 Instructions in Each APK With 40,000 APKs

Loss:

Figure D.1 shows the loss for all five datasets.

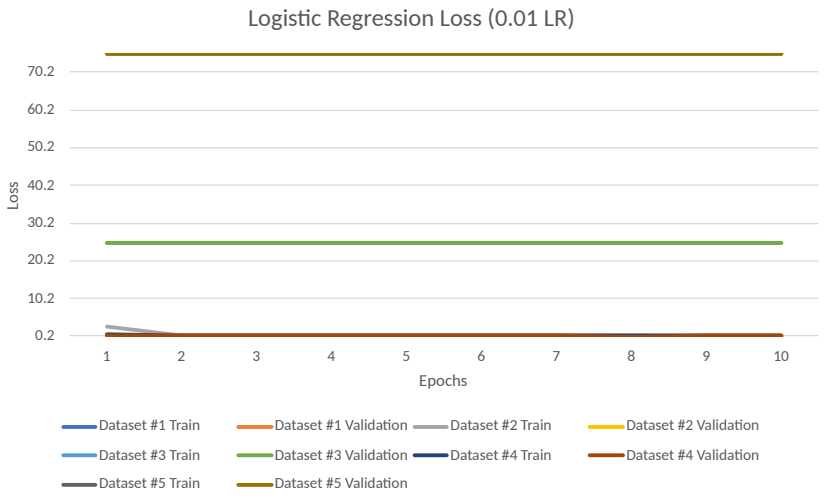

Figure D.1: *The Loss for all five datasets*

Figure D.2 shows a more refined view of the datasets that were in the expected range that showed variance.

Figure D.2: *A refined view of the Loss results that focused on the expected results*

Accuracy:

Figure D.3 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

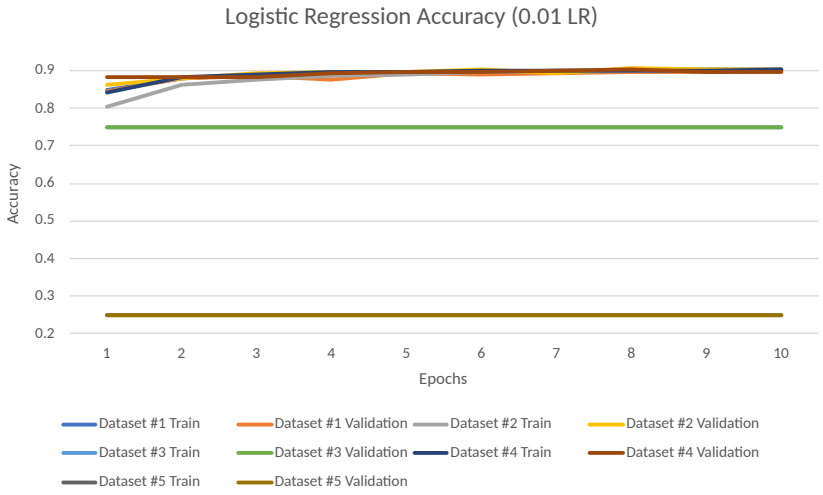

Figure D.3: *The Accuracy for all five datasets*

Figure D.4 shows a more refined view of the datasets that were in the expected range that showed variance.

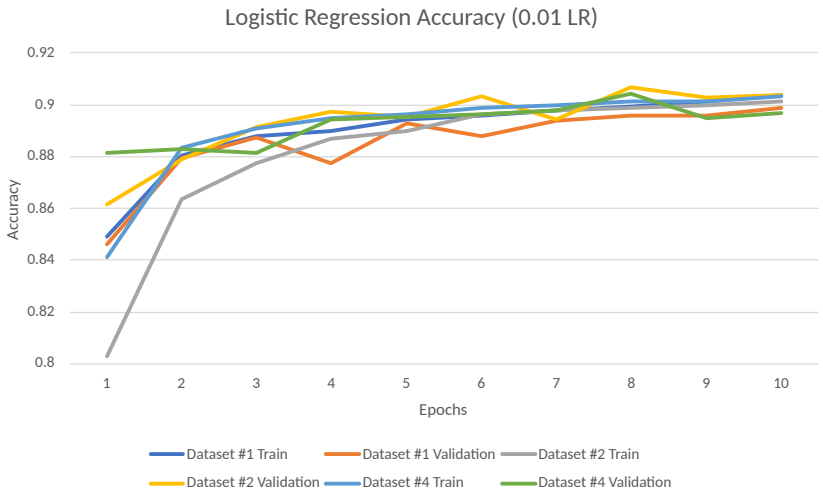

Figure D.4: *A refined view of the Accuracy results that focused on the expected results*

Precision:

Figure D.5 shows the precision for all five datasets.

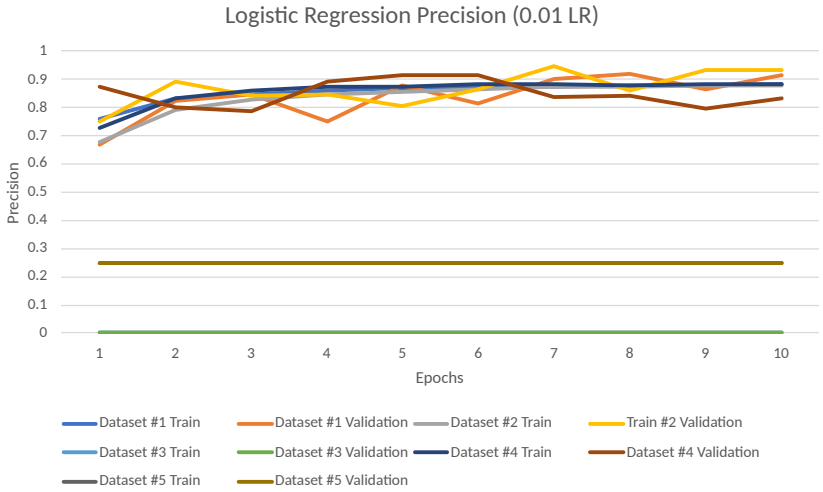

Figure D.5: *The Precision for all five datasets*

Figure D.6 shows a more refined view of the datasets that were in the expected range that showed variance.

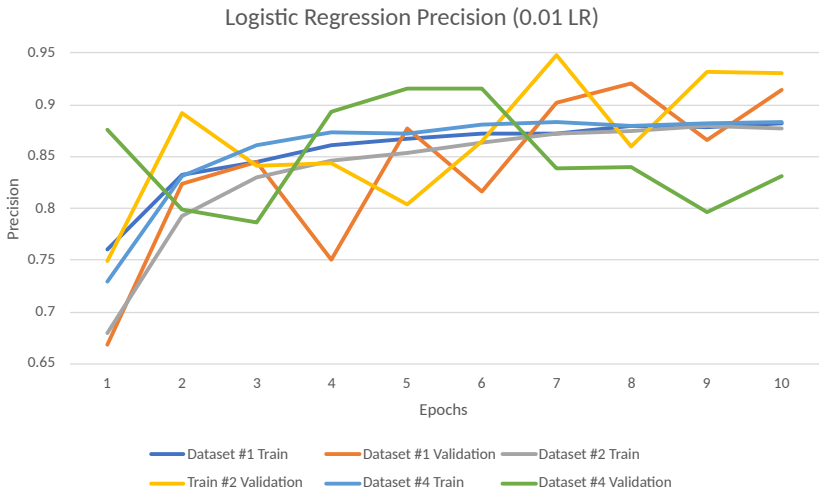

Figure D.6: *A refined view of the Precision results that focused on the expected results*

Recall:

Figure D.7 shows the recall for all five datasets.

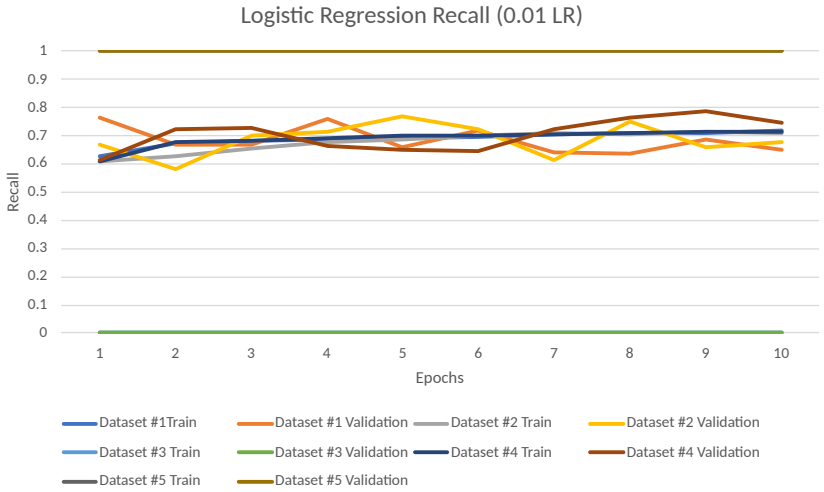

Figure D.7: *The Recall for all five datasets*

Figure D.8 shows a more refined view of the datasets that were in the expected range that showed variance.

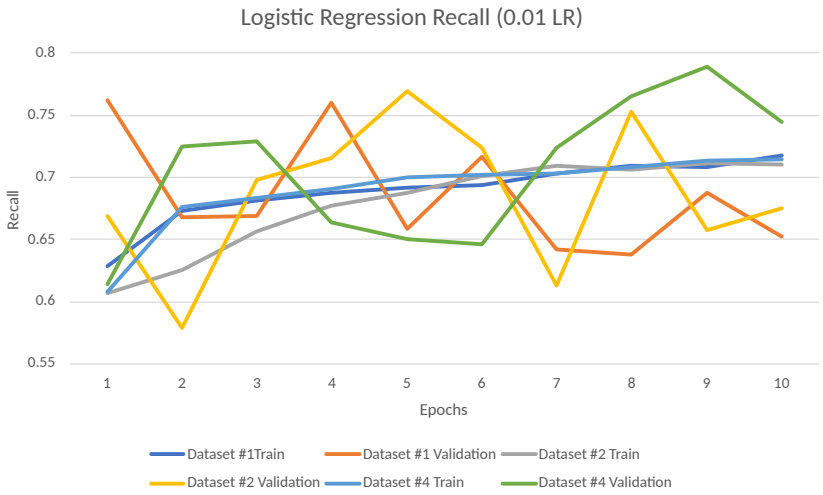

Figure D.8: *A refined view of the Recall results that focused on the expected results*

F1-Score:

Figure D.9 shows the f1-score for all five datasets.

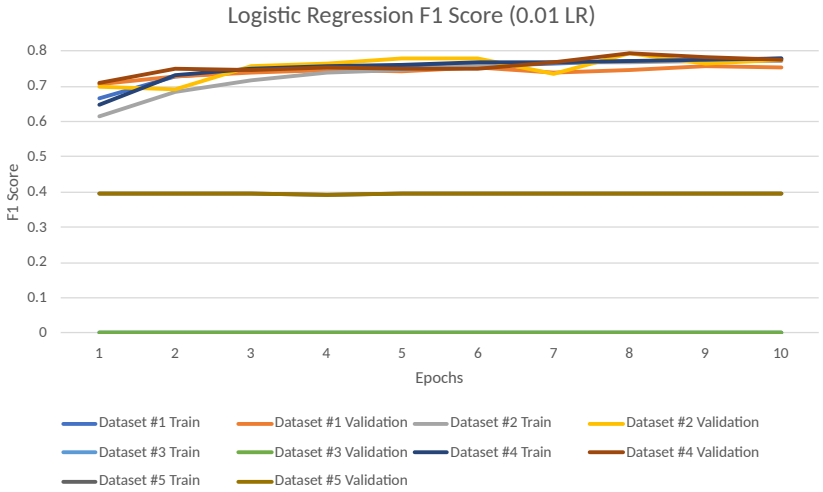

Figure D.9: *The F1-Score for all five datasets*

Figure D.10 shows a more refined view of the datasets that were in the expected range that showed variance.

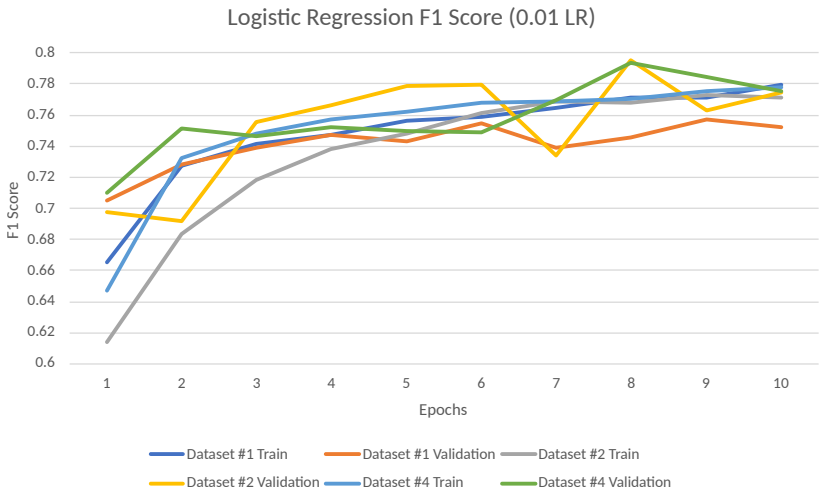

Figure D.10: *A refined view of the F1-Score results that focused on the expected results*

Test:

Figure D.11 shows the test results for all five datasets for all five metrics.

Figure D.11: *The test results for all five datasets*

Figure D.12 shows a more refined view of the datasets that were in the expected range that showed variance.

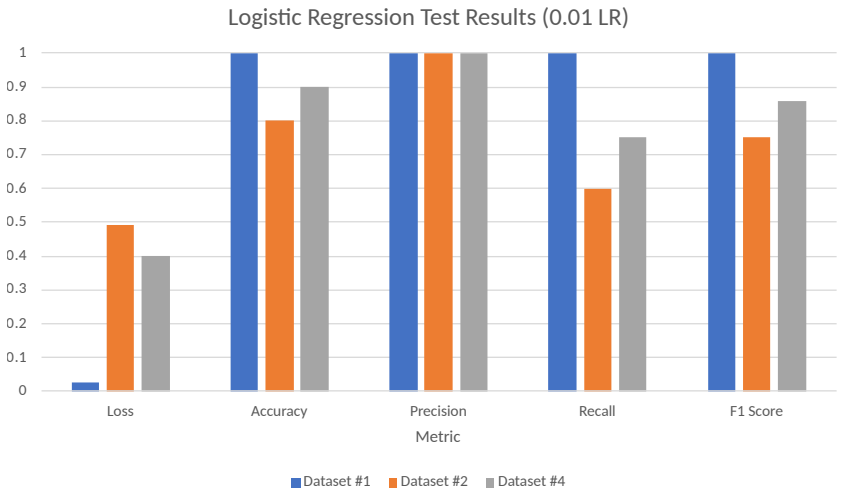

Figure D.12: *A refined view of the test results that focused on the expected results*

D.1.2 All Instructions in Each APK With 40,000 APKs

Loss:

Figure D.13 shows the loss for all five datasets.

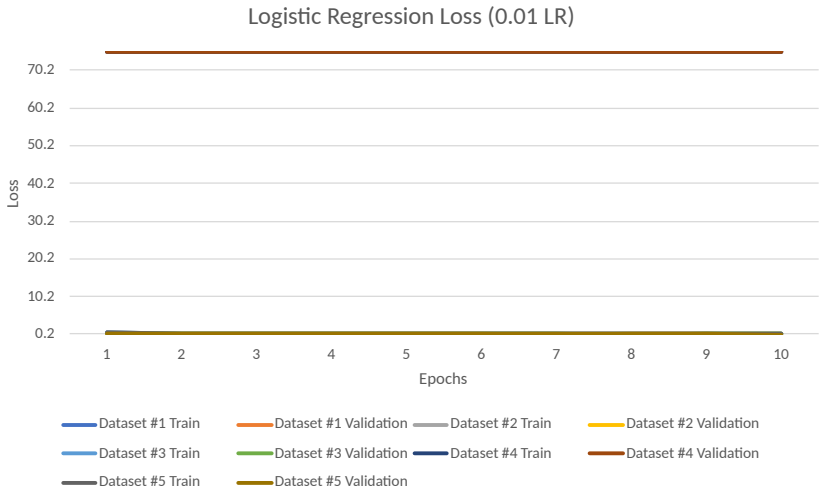

Figure D.13: *The Loss for all five datasets*

Figure D.14 shows a more refined view of the datasets that were in the expected range that showed variance.

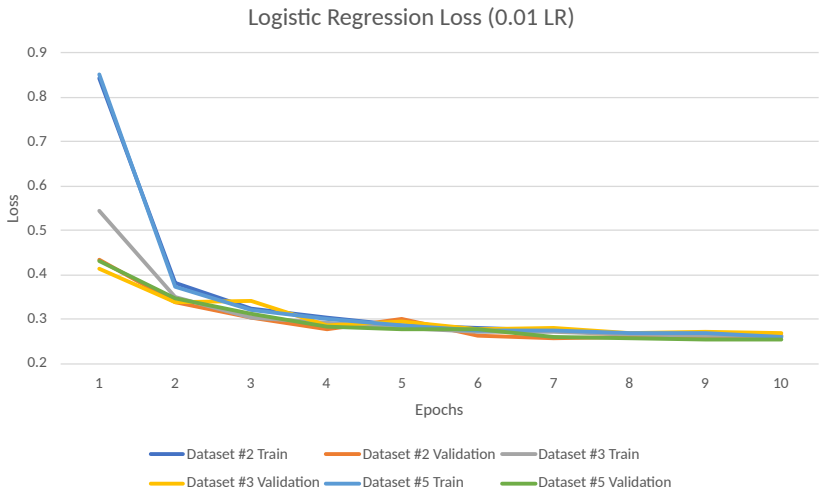

Figure D.14: *A refined view of the Loss results that focused on the expected results*

Accuracy:

Figure D.15 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

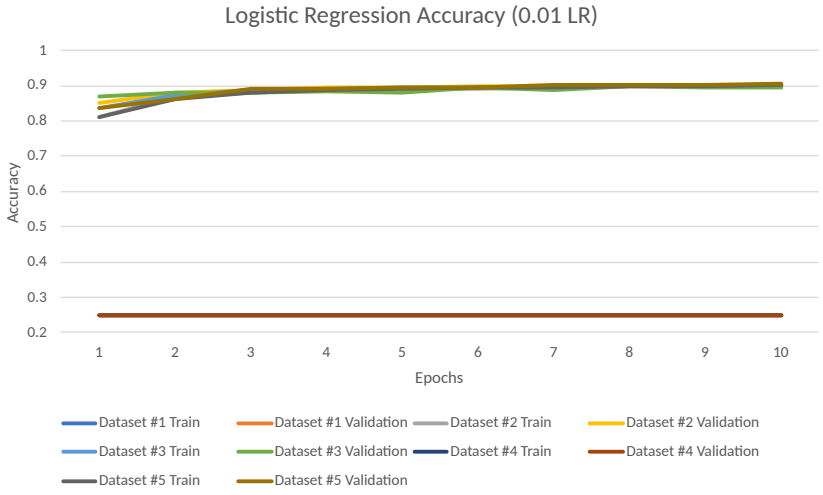

Figure D.15: *The Accuracy for all five datasets*

Figure D.16 shows a more refined view of the datasets that were in the expected range that showed variance.

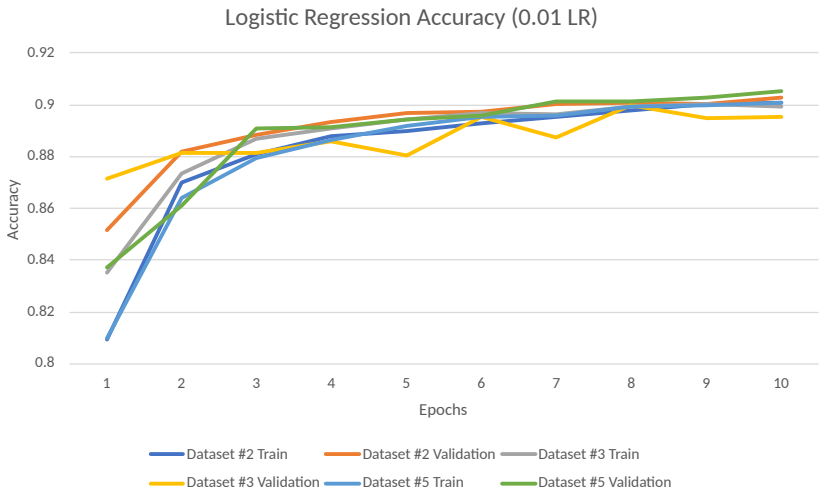

Figure D.16: *A refined view of the Accuracy results that focused on the expected results*

Precision:

Figure D.17 shows the precision for all five datasets.

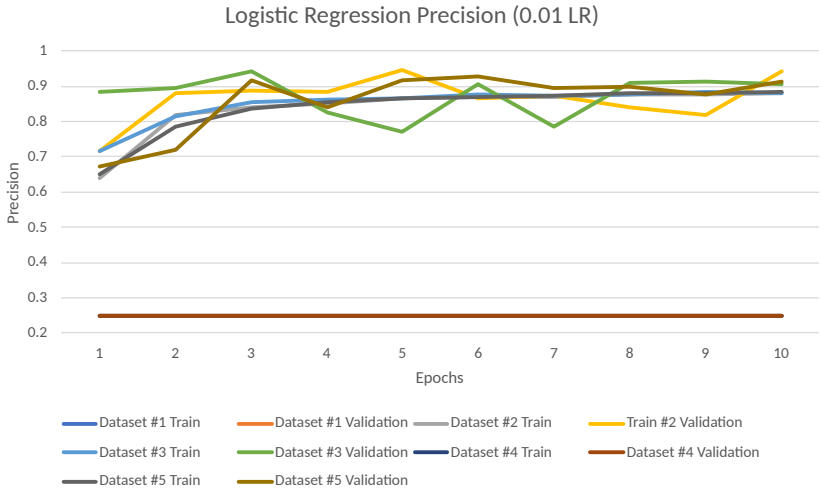

Figure D.17: *The Precision for all five datasets*

Figure D.18 shows a more refined view of the datasets that were in the expected range that showed variance.

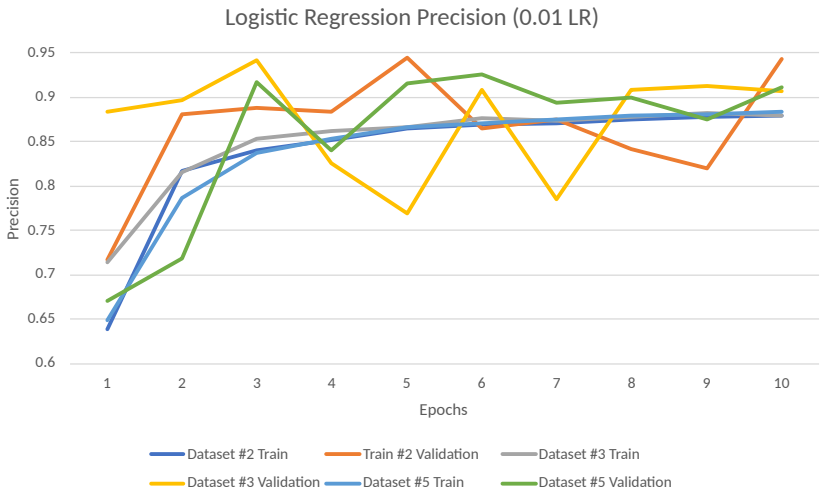

Figure D.18: *A refined view of the Precision results that focused on the expected results*

Recall:

Figure D.19 shows the recall for all five datasets.

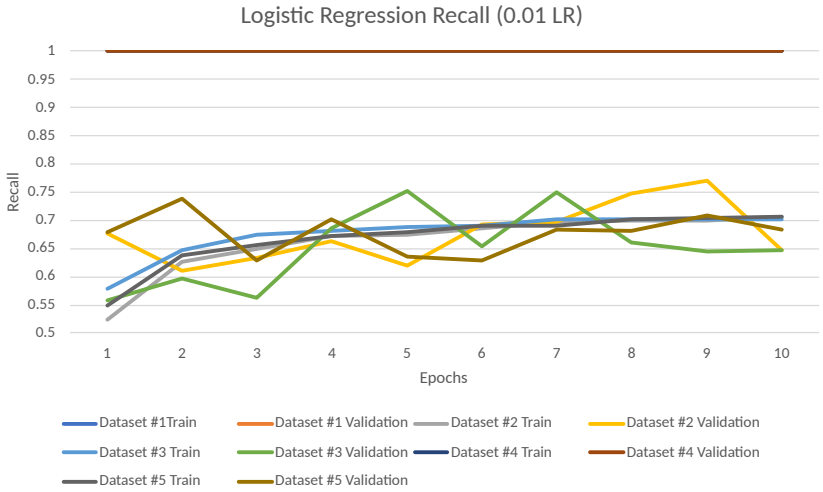

Figure D.19: *The Recall for all five datasets*

Figure D.20 shows a more refined view of the datasets that were in the expected range that showed variance.

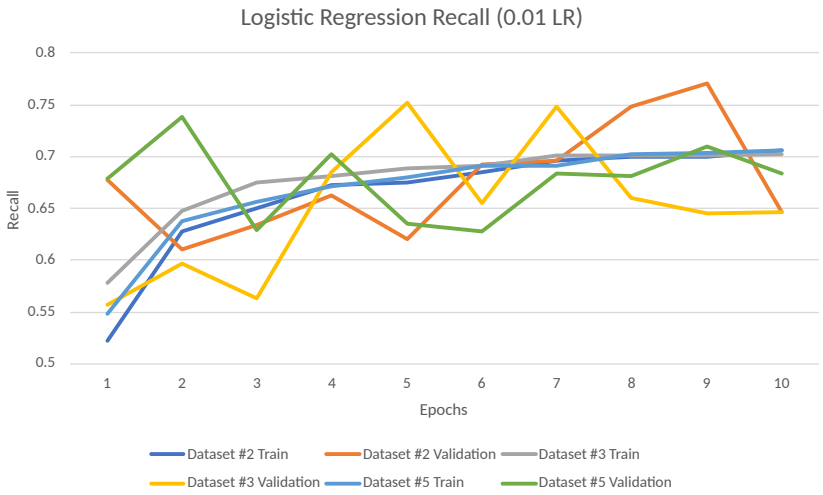

Figure D.20: *A refined view of the Recall results that focused on the expected results*

F1-Score:

Figure D.21 shows the f1-score for all five datasets.

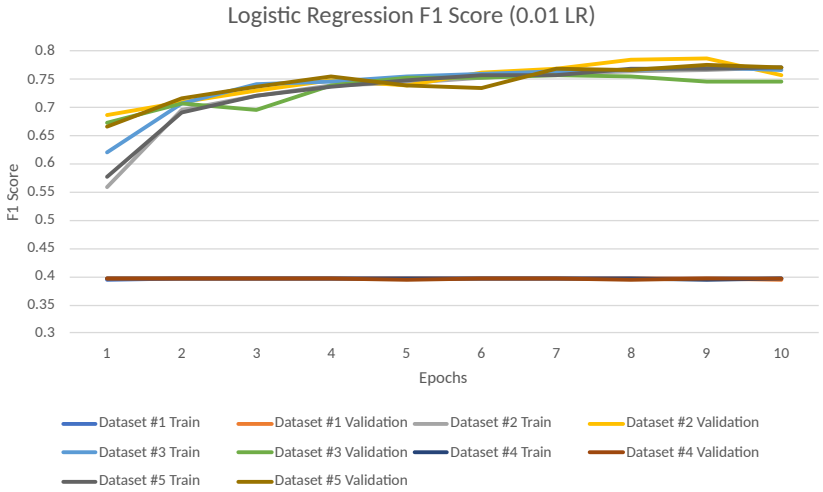

Figure D.21: *The F1-Score for all five datasets*

Figure D.22 shows a more refined view of the datasets that were in the expected range that showed variance.

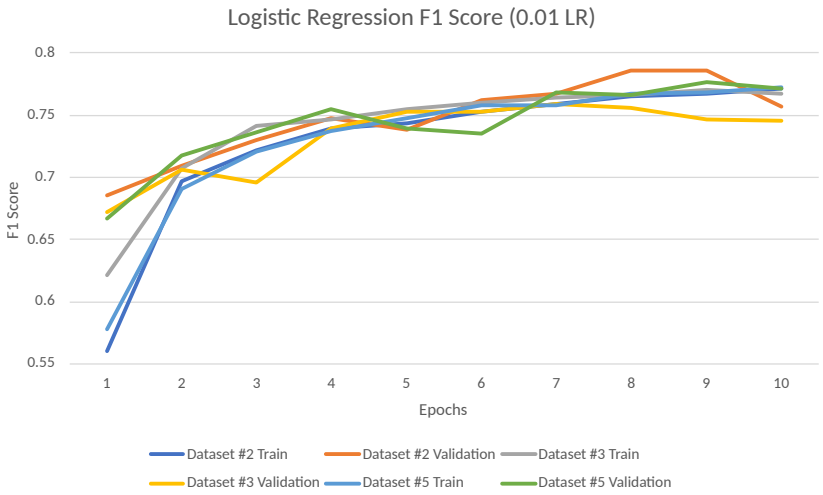

Figure D.22: *A refined view of the F1-Score results that focused on the expected results*

Test:

Figure D.23 shows the test results for all five datasets for all five metrics.

Figure D.23: *The test results for all five datasets*

Figure D.24 shows a more refined view of the datasets that were in the expected range that showed variance.

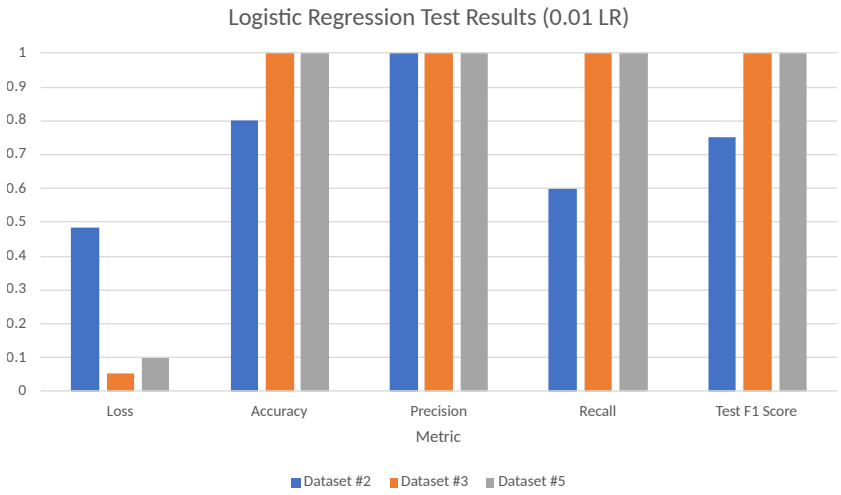

Figure D.24: *A refined view of the test results that focused on the expected results*

D.2 Learning Rate of 0.001

D.2.1 First 60,000 Instructions in Each APK With 40,000 APKs

Loss:

Figure D.25 shows the loss for all five datasets.

Figure D.25: *The Loss for all five datasets*

Figure D.26 shows a more refined view of the datasets that were in the expected range that showed variance.

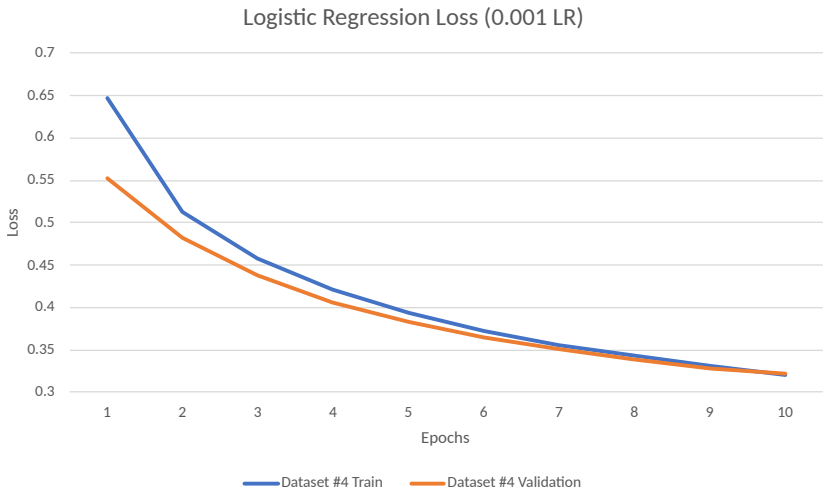

Figure D.26: *A refined view of the Loss results that focused on the expected results*

Accuracy:

Figure D.27 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

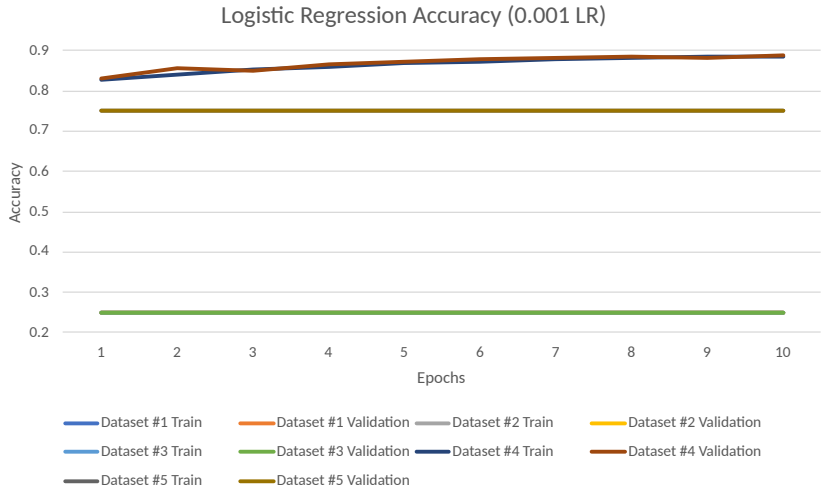

Figure D.27: *The Accuracy for all five datasets*

Figure D.28 shows a more refined view of the datasets that were in the expected range that showed variance.

Figure D.28: *A refined view of the Accuracy results that focused on the expected results*

Precision:

Figure D.29 shows the precision for all five datasets.

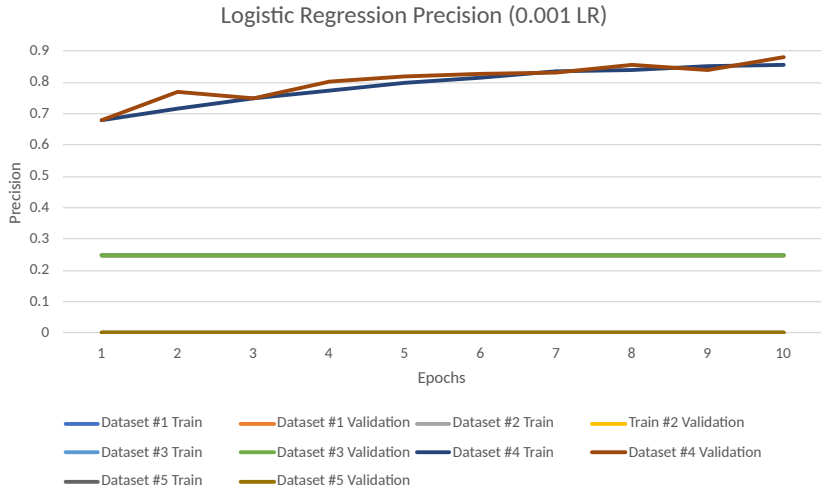

Figure D.29: *The Precision for all five datasets*

Figure D.30 shows a more refined view of the datasets that were in the expected range that showed variance.

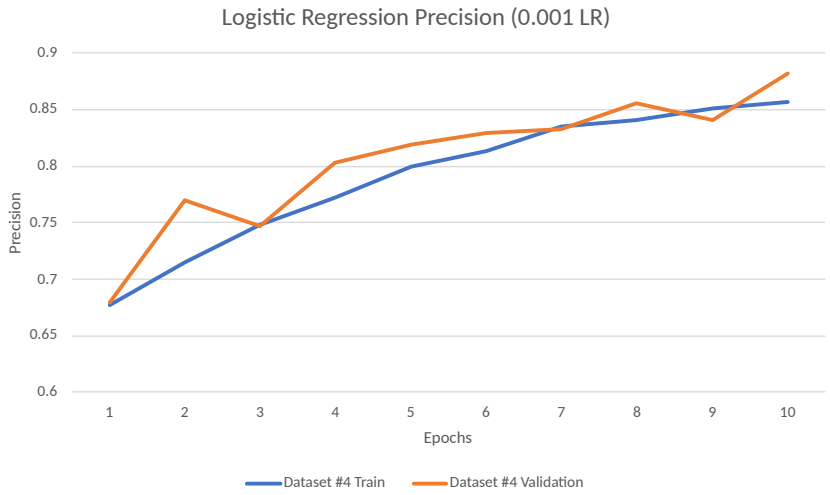

Figure D.30: *A refined view of the Precision results that focused on the expected results*

Recall:

Figure D.31 shows the recall for all five datasets.

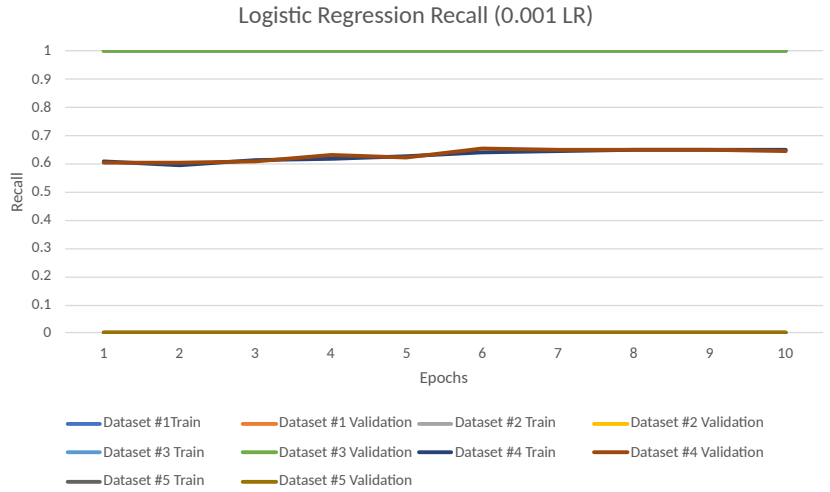

Figure D.31: *The Recall for all five datasets*

Figure D.32 shows a more refined view of the datasets that were in the expected range that showed variance.

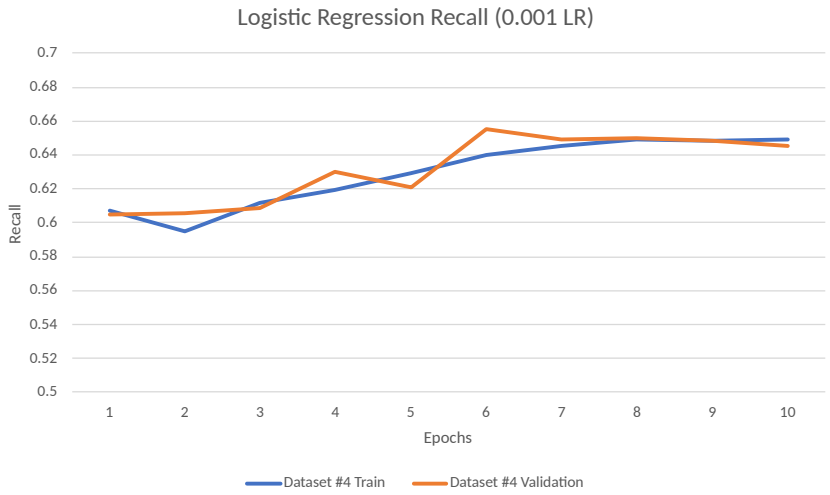

Figure D.32: *A refined view of the Recall results that focused on the expected results*

F1-Score:

Figure D.33 shows the f1-score for all five datasets.

Figure D.33: *The F1-Score for all five datasets*

Figure D.34 shows a more refined view of the datasets that were in the expected range that showed variance.

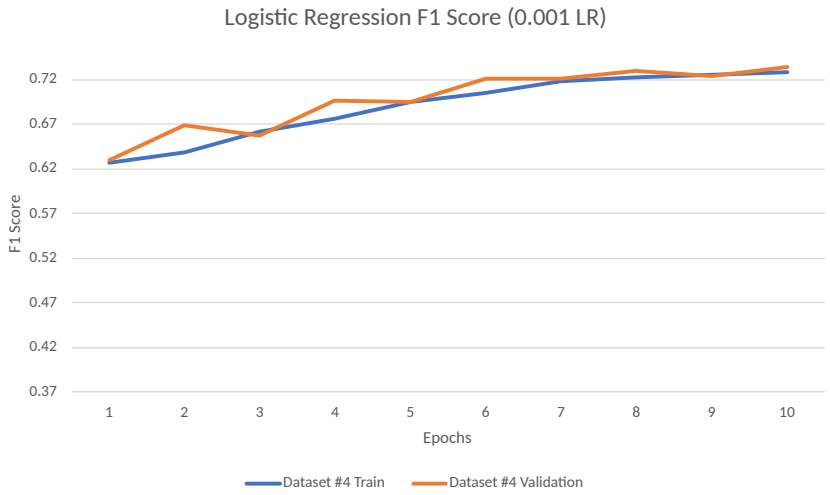

Figure D.34: *A refined view of the F1-Score results that focused on the expected results*

Test:

Figure D.35 shows the test results for all five datasets for all five metrics.

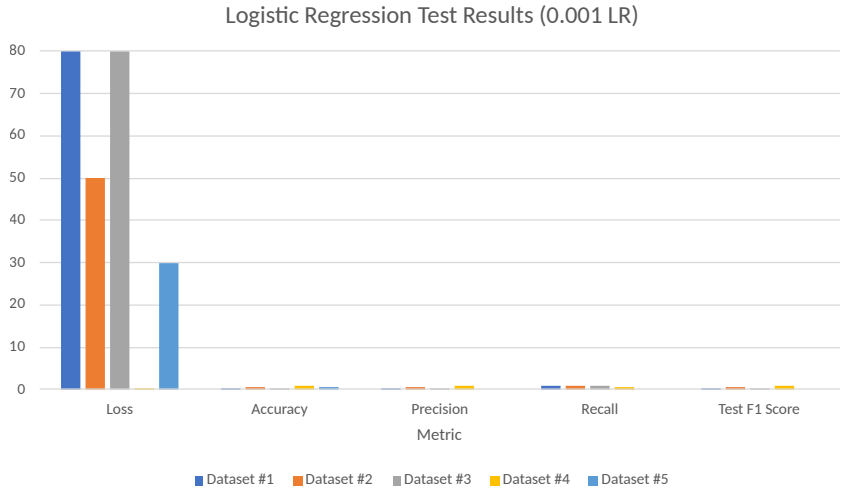

Figure D.35: *The test results for all five datasets*

Figure D.36 shows a more refined view of the datasets that were in the expected range that showed variance.

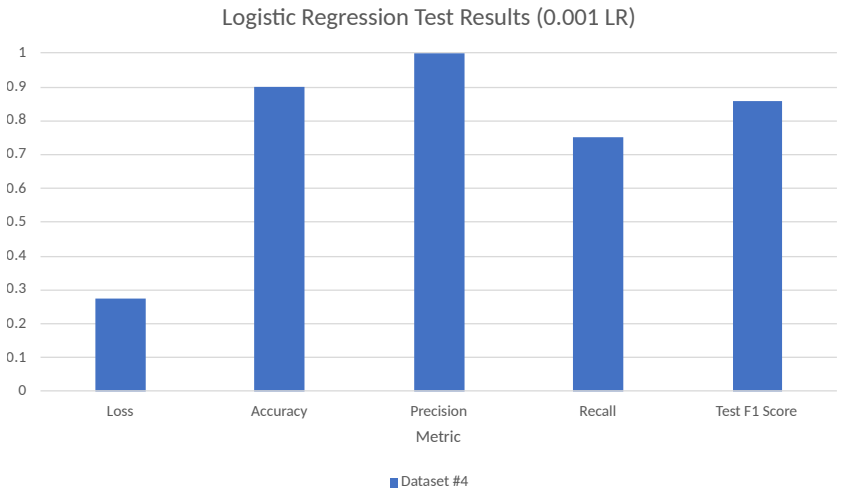

Figure D.36: *A refined view of the test results that focused on the expected results*

D.2.2 All Instructions in Each APK With 40,000 APKs

Loss:

Figure D.37 shows the loss for all five datasets.

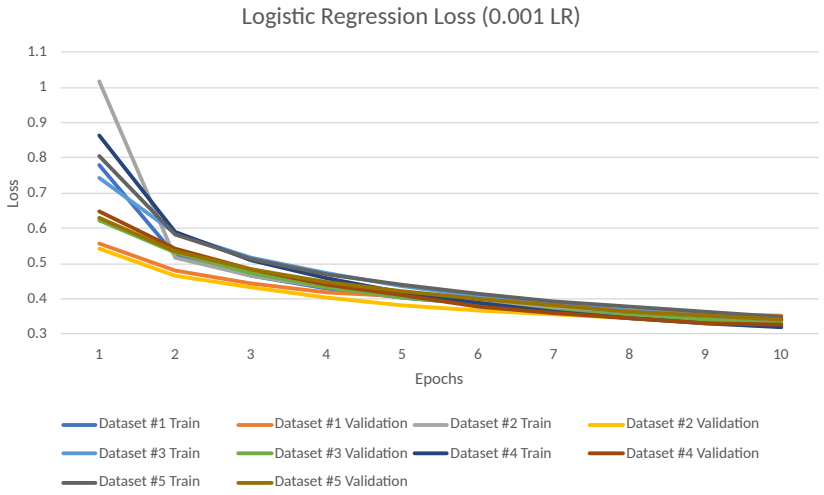

Figure D.37: *The Loss for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Accuracy:

Figure D.38 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

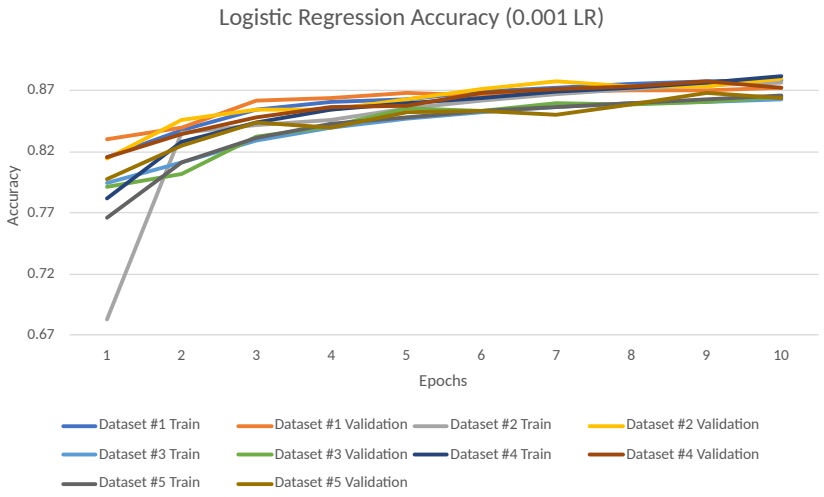

Figure D.38: *The Accuracy for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Precision:

Figure D.39 shows the precision for all five datasets.

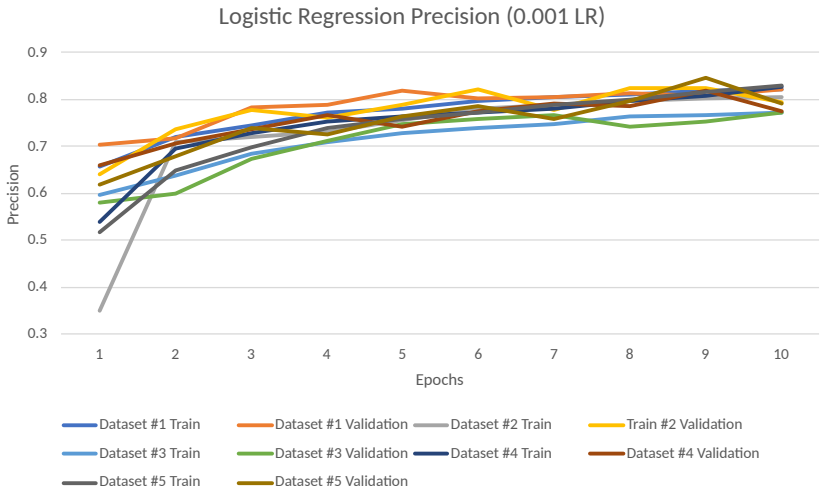

Figure D.39: *The Precision for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Recall:

Figure D.40 shows the recall for all five datasets.

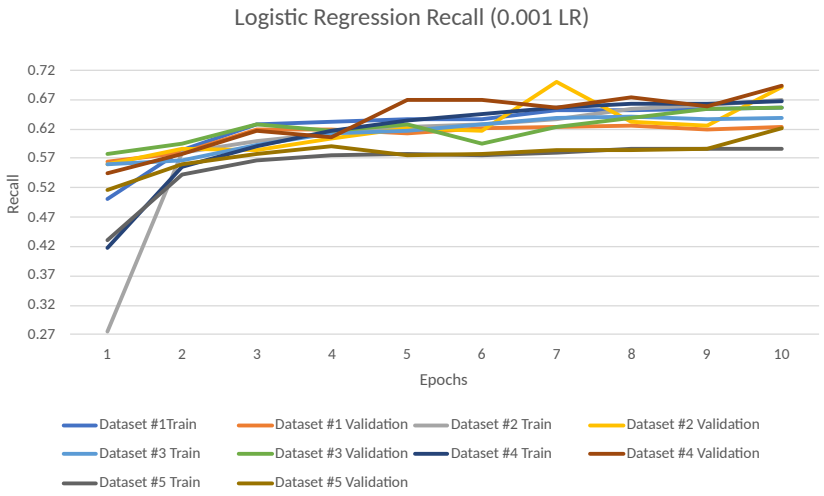

Figure D.40: *The Recall for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

F1-Score:

Figure D.41 shows the f1-score for all five datasets.

Figure D.41: *The F1-Score for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Test:

Figure D.42 shows the test results for all five datasets for all five metrics.

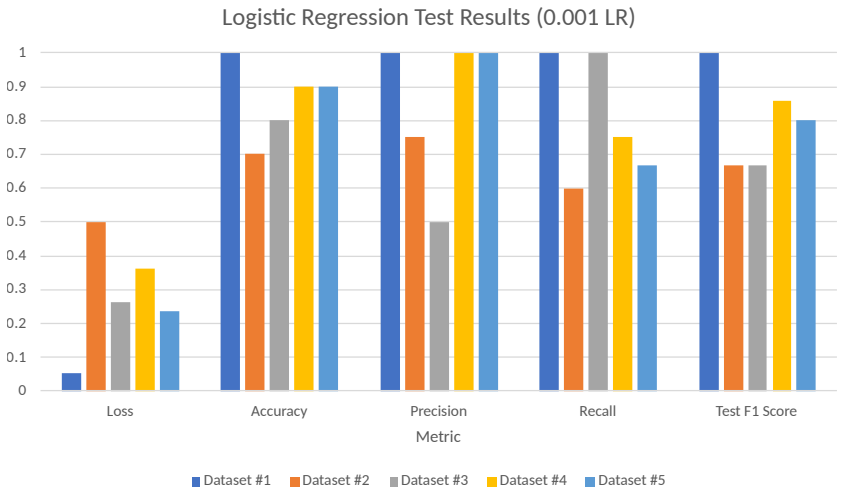

Figure D.42: *The test results for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

D.2.3 All Instructions in Each APK With 172,000 APKs

Loss:

Figure D.43 shows the loss for all five datasets.

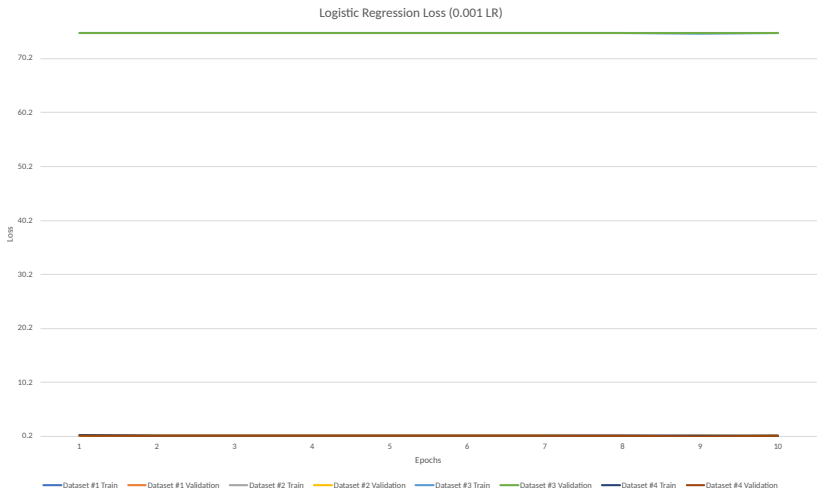

Figure D.43: *The Loss for all five datasets*

Figure D.44 shows a more refined view of the datasets that were in the expected range that showed variance.

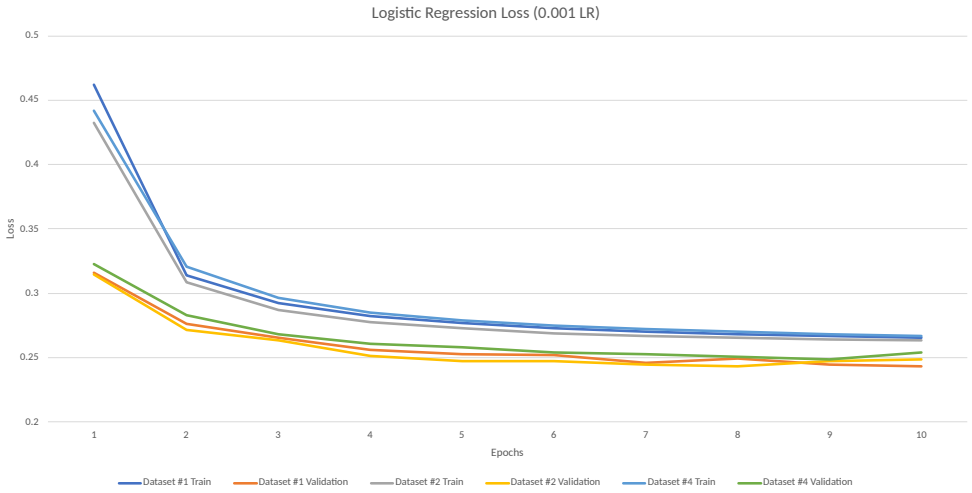

Figure D.44: *A refined view of the Loss results that focused on the expected results*

Accuracy:

Figure D.45 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

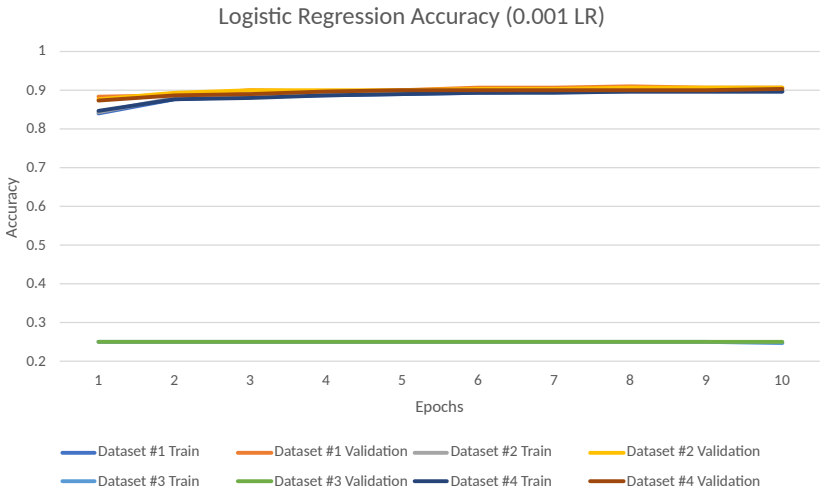

Figure D.45: *The Accuracy for all five datasets*

Figure D.46 shows a more refined view of the datasets that were in the expected range that showed variance.

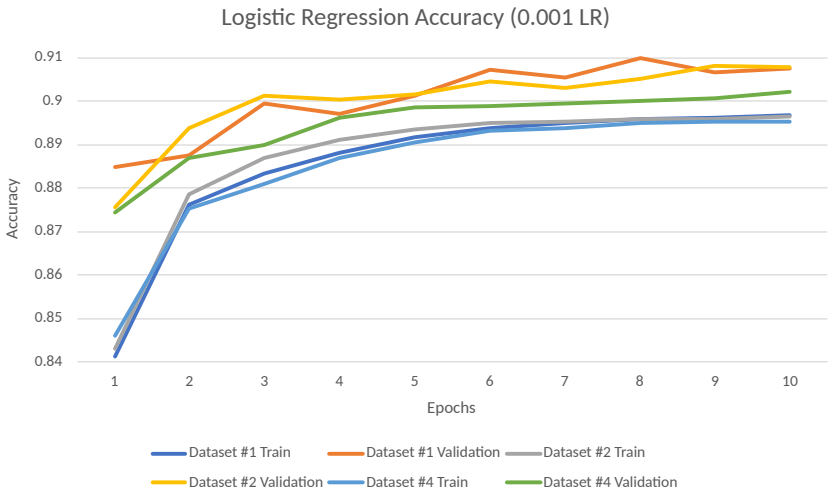

Figure D.46: *A refined view of the Accuracy results that focused on the expected results*

Precision:

Figure D.47 shows the precision for all five datasets.

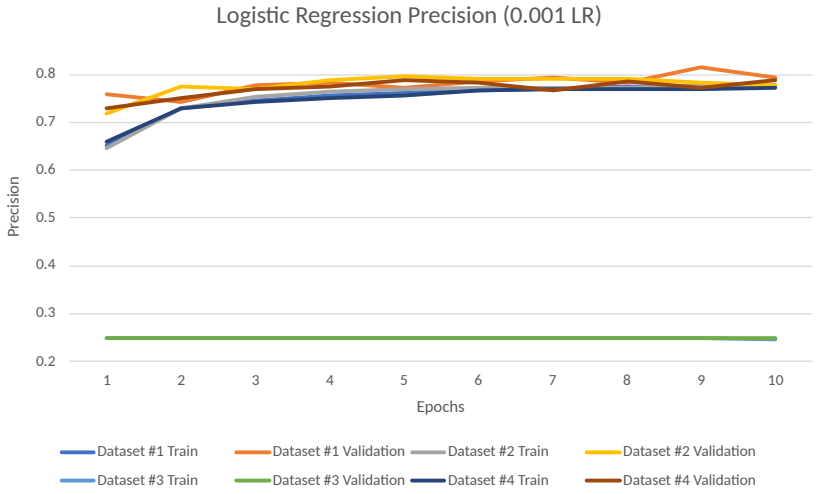

Figure D.47: *The Precision for all five datasets*

Figure D.48 shows a more refined view of the datasets that were in the expected range that showed variance.

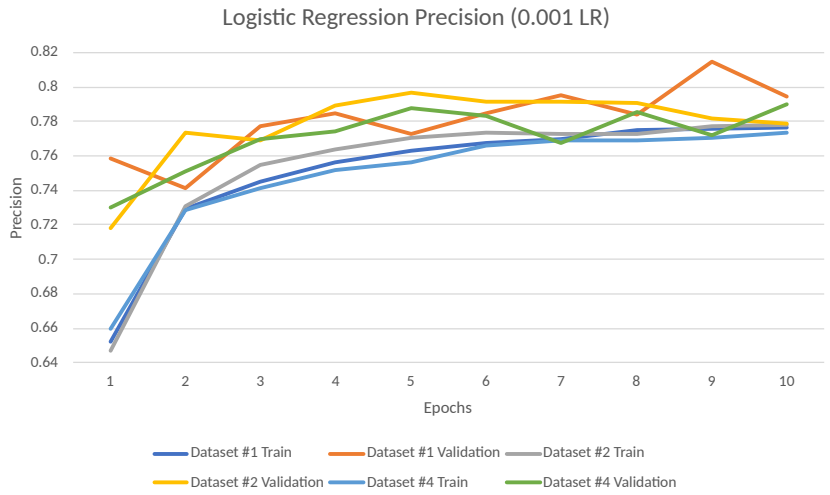

Figure D.48: *A refined view of the Precision results that focused on the expected results*

Recall:

Figure D.49 shows the recall for all five datasets.

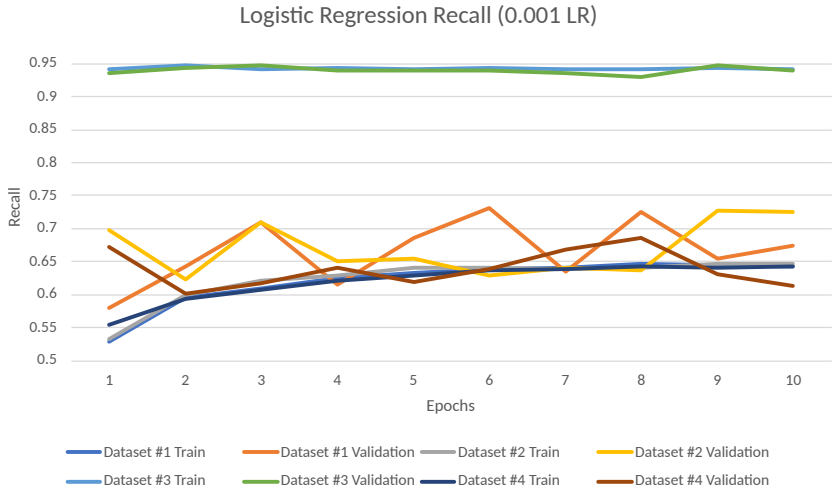

Figure D.49: *The Recall for all five datasets*

Figure D.50 shows a more refined view of the datasets that were in the expected range that showed variance.

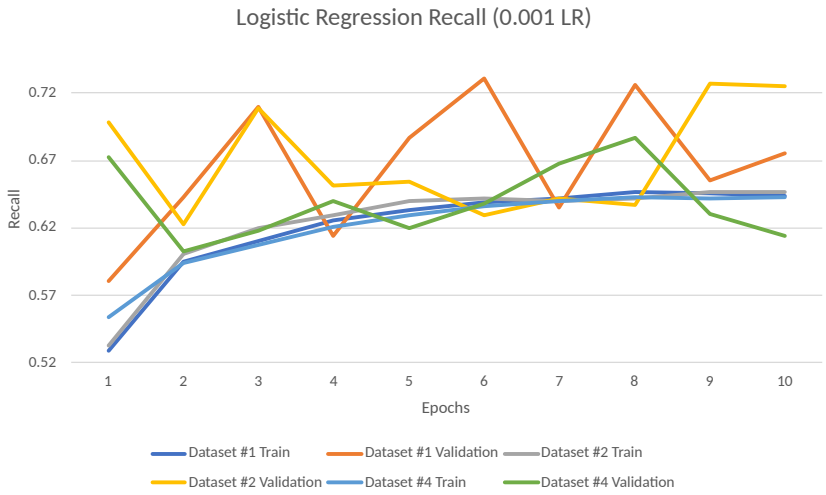

Figure D.50: *A refined view of the Recall results that focused on the expected results*

F1-Score:

Figure D.51 shows the f1-score for all five datasets.

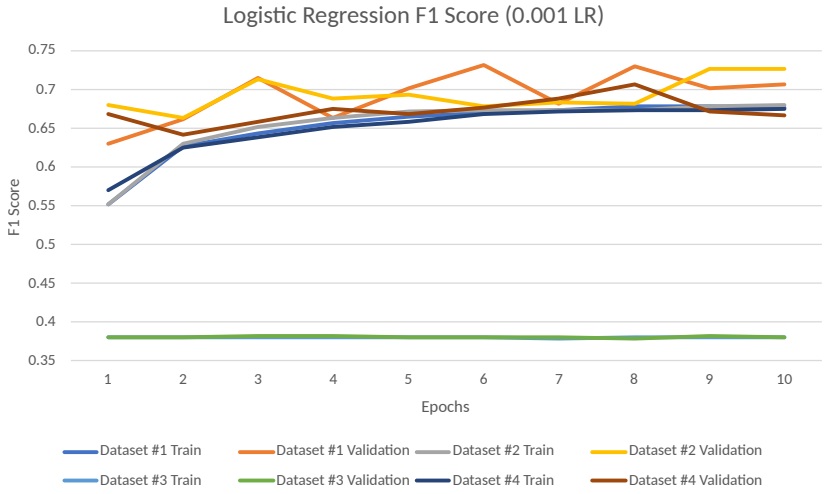

Figure D.51: *The F1-Score for all five datasets*

Figure D.52 shows a more refined view of the datasets that were in the expected range that showed variance.

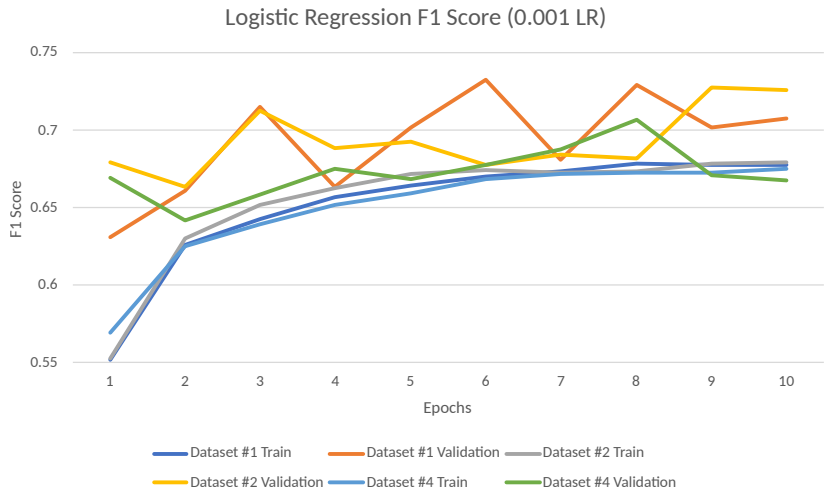

Figure D.52: *A refined view of the F1-Score results that focused on the expected results*

Test:

Figure D.53 shows the test results for all five datasets for all five metrics.

Figure D.53: *The test results for all five datasets*

Figure D.54 shows a more refined view of the datasets that were in the expected range that showed variance.

Figure D.54: *A refined view of the test results that focused on the expected results*

D.3 Learning Rate of 0.0001

D.3.1 First 60,000 Instructions in Each APK With 40,000 APKs

Loss:

Figure D.55 shows the loss for all five datasets.

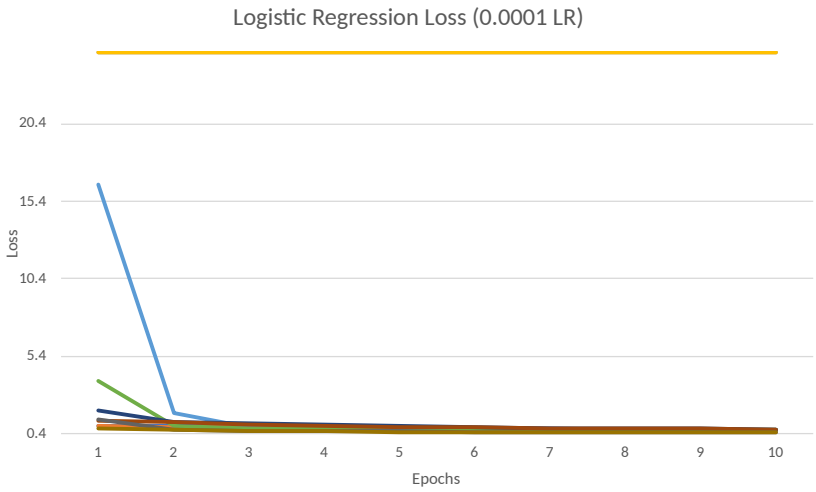

Figure D.55: *The Loss for all five datasets*

Figure D.56 shows a more refined view of the datasets that were in the expected range that showed variance.

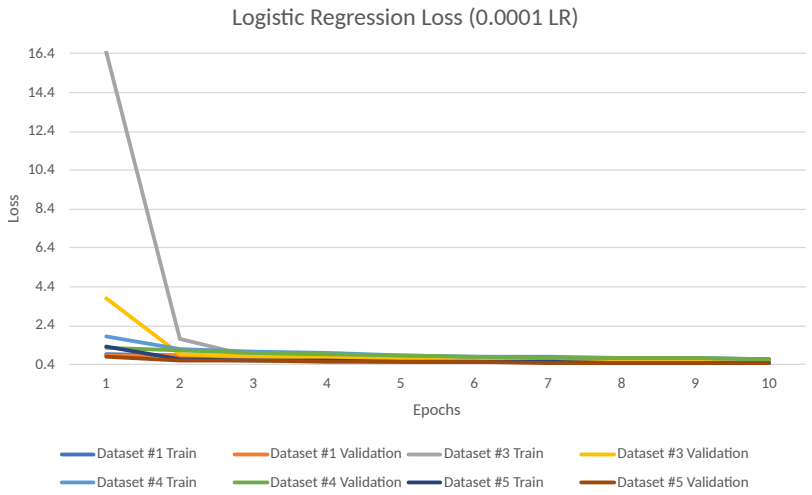

Figure D.56: *A refined view of the Loss results that focused on the expected results*

Accuracy:

Figure D.57 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

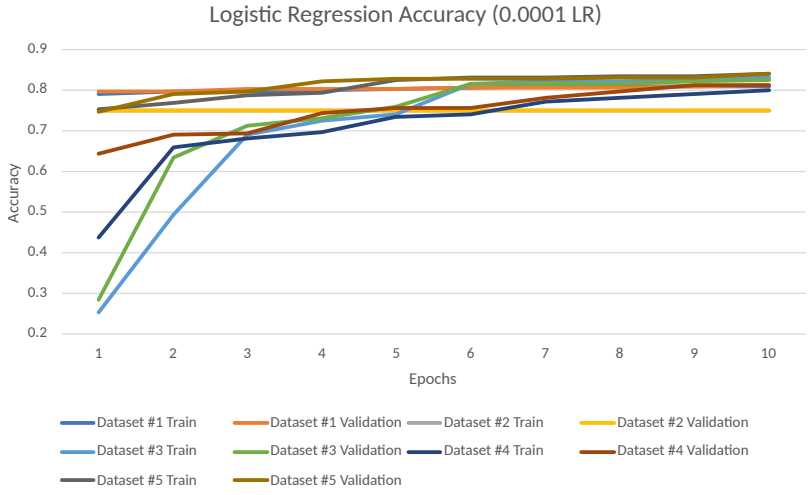

Figure D.57: *The Accuracy for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Precision:

Figure D.58 shows the precision for all five datasets.

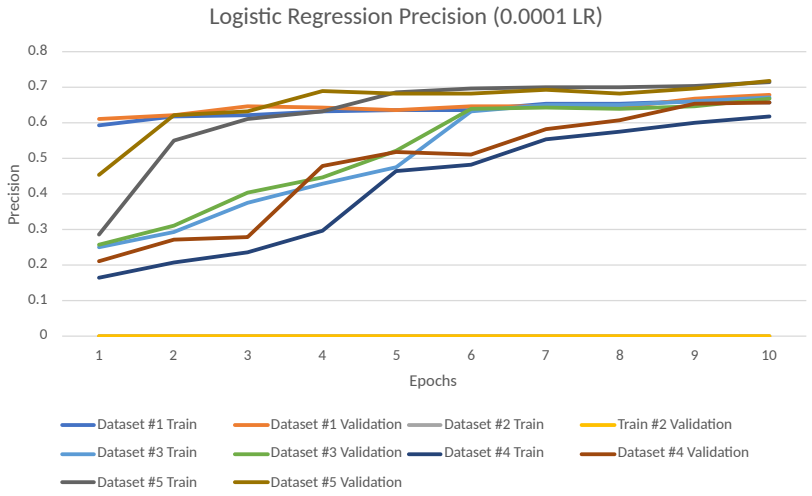

Figure D.58: *The Precision for all five datasets*

Figure D.59 shows a more refined view of the datasets that were in the expected range that showed variance.

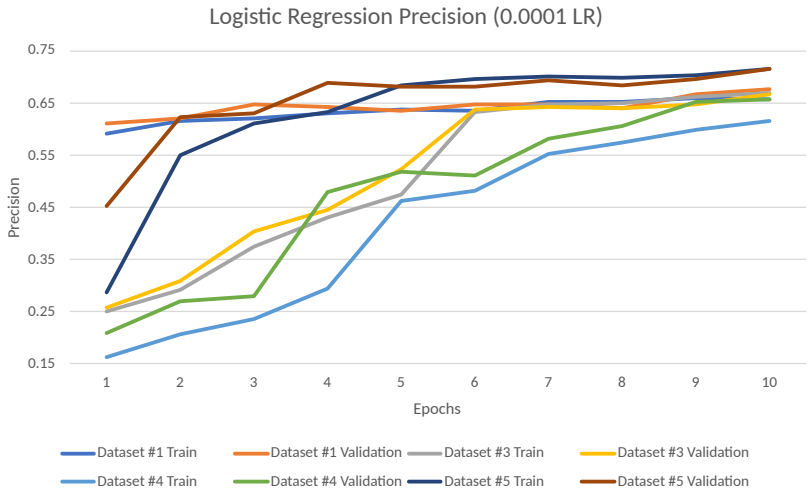

Figure D.59: *A refined view of the Precision results that focused on the expected results*

Recall:

Figure D.60 shows the recall for all five datasets.

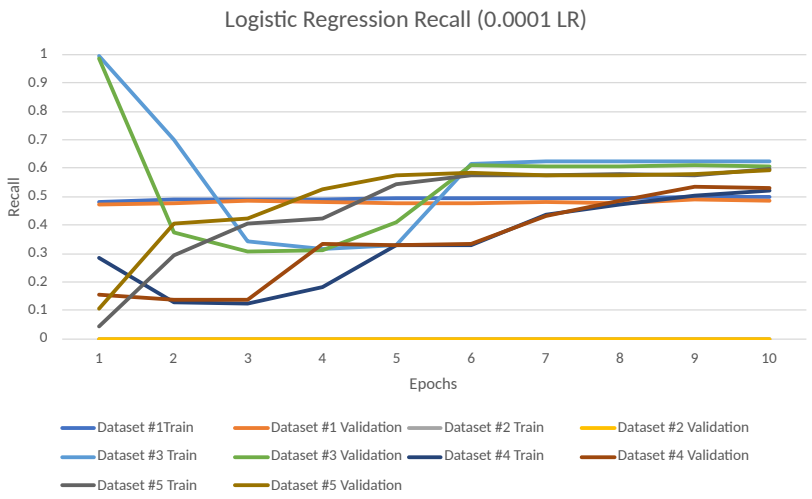

Figure D.60: *The Recall for all five datasets*

Due to the wide range of values displayed by the datasets that were learning from the data set, a refined graph did not make sense for in this case.

F1-Score:

Figure D.61 shows the f1-score for all five datasets.

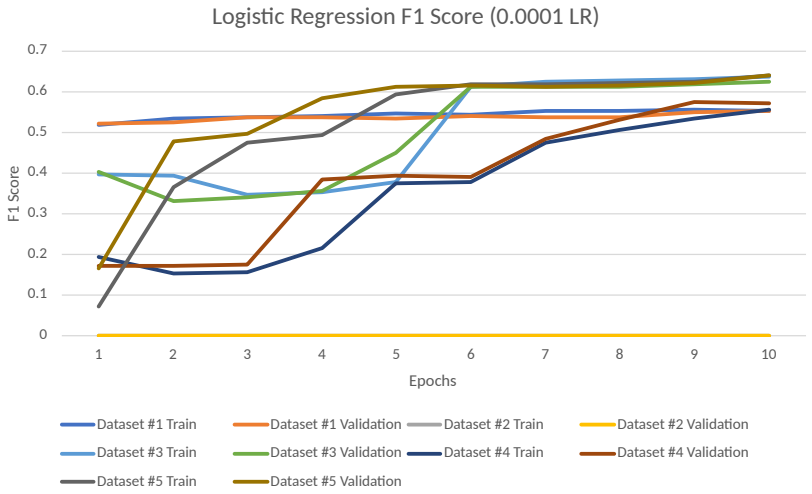

Figure D.61: *The F1-Score for all five datasets*

Figure D.62 shows a more refined view of the datasets that were in the expected range that showed variance.

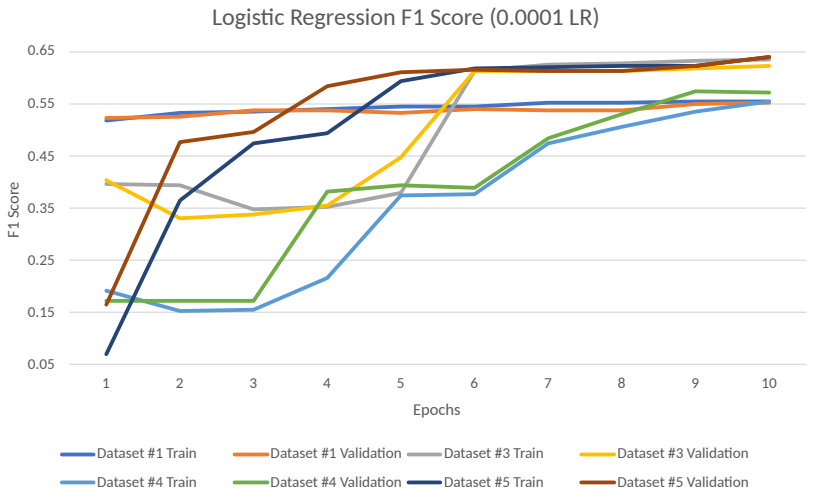

Figure D.62: *A refined view of the F1-Score results that focused on the expected results*

Test:

Figure D.63 shows the test results for all five datasets for all five metrics.

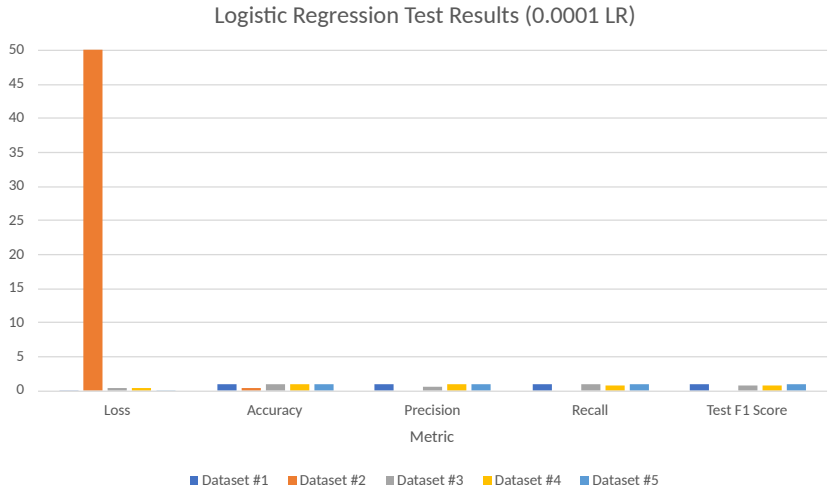

Figure D.63: *The test results for all five datasets*

Figure D.64 shows a more refined view of the datasets that were in the expected range that showed variance.

Figure D.64: *A refined view of the test results that focused on the expected results*

D.3.2 All Instructions in Each APK With 40,000 APKs

Loss:

Figure D.65 shows the loss for all five datasets.

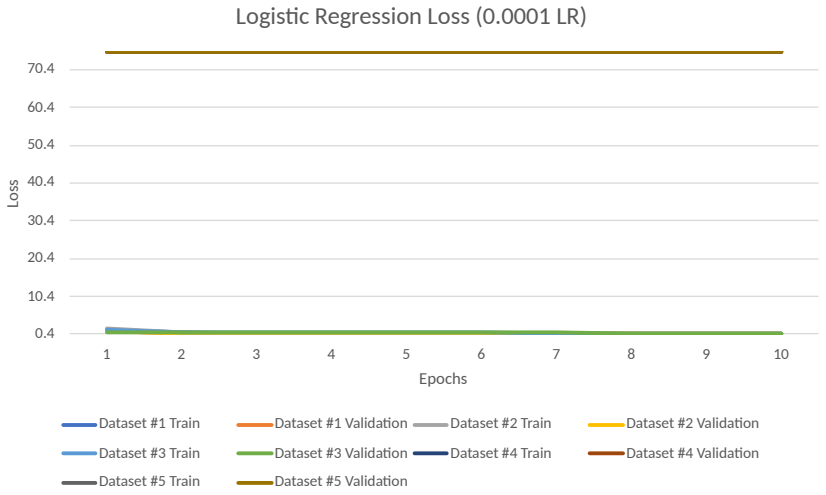

Figure D.65: *The Loss for all five datasets*

Figure D.66 shows a more refined view of the datasets that were in the expected range that showed variance.

Figure D.66: *A refined view of the Loss results that focused on the expected results*

Accuracy:

Figure D.67 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

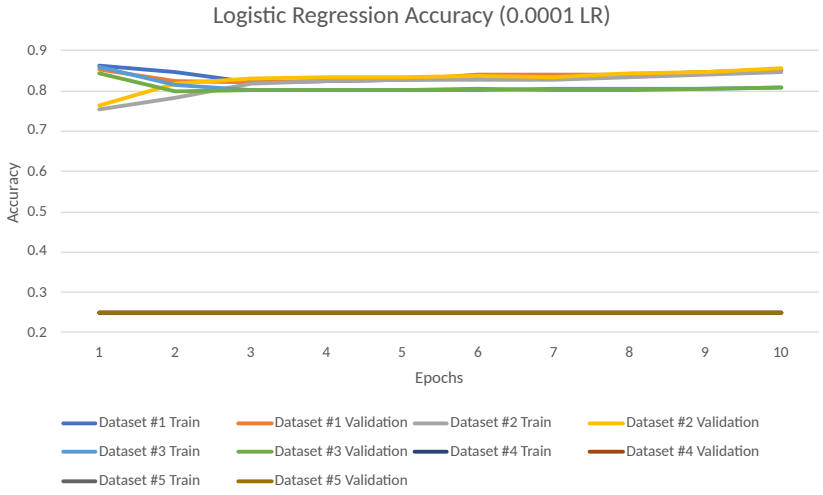

Figure D.67: *The Accuracy for all five datasets*

Figure D.68 shows a more refined view of the datasets that were in the expected range that showed variance.

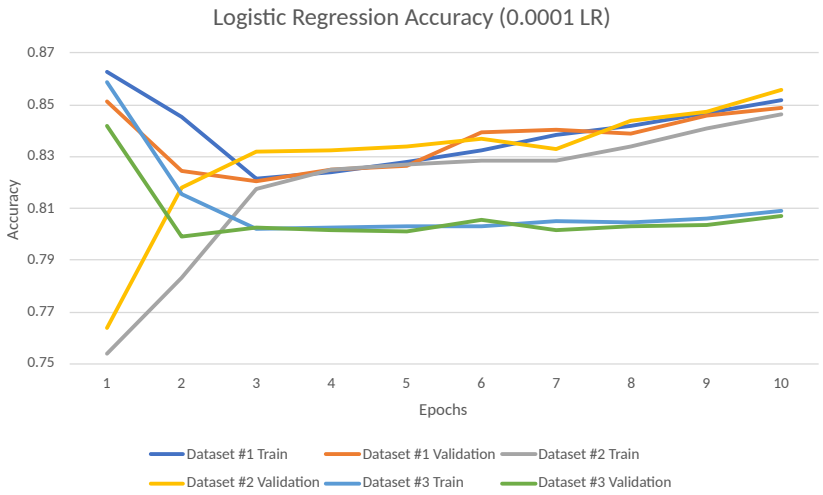

Figure D.68: *A refined view of the Accuracy results that focused on the expected results*

Precision:

Figure D.69 shows the precision for all five datasets.

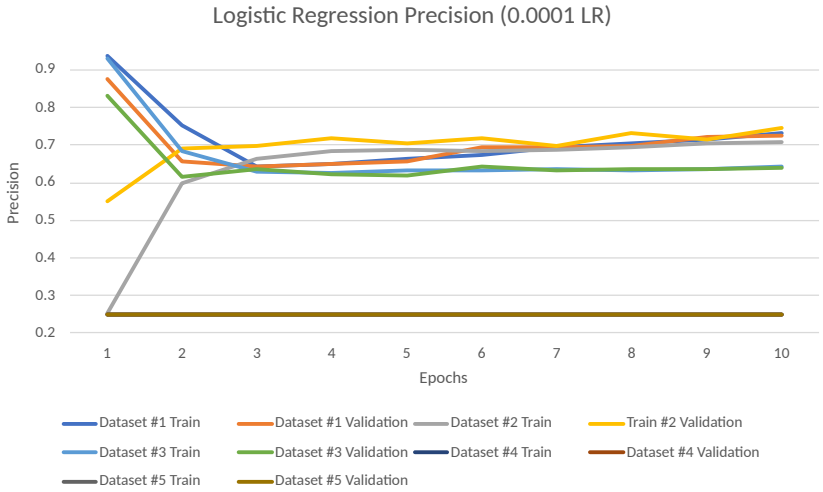

Figure D.69: *The Precision for all five datasets*

Figure D.70 shows a more refined view of the datasets that were in the expected range that showed variance.

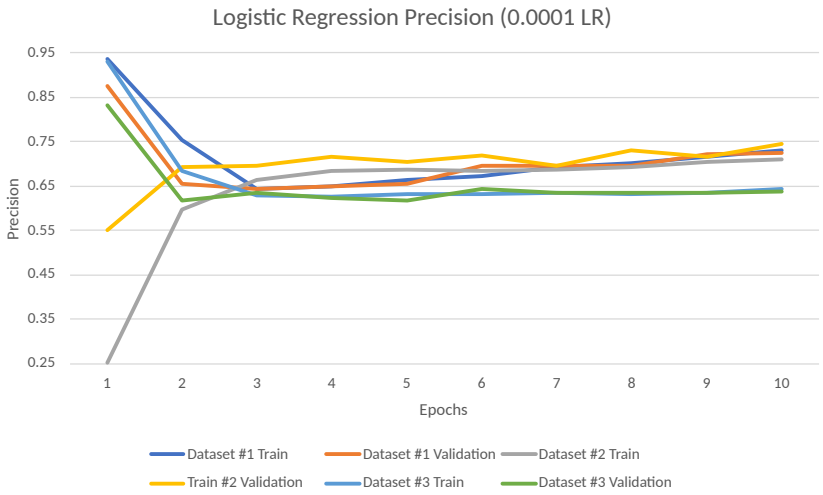

Figure D.70: *A refined view of the Precision results that focused on the expected results*

Recall:

Figure D.71 shows the recall for all five datasets.

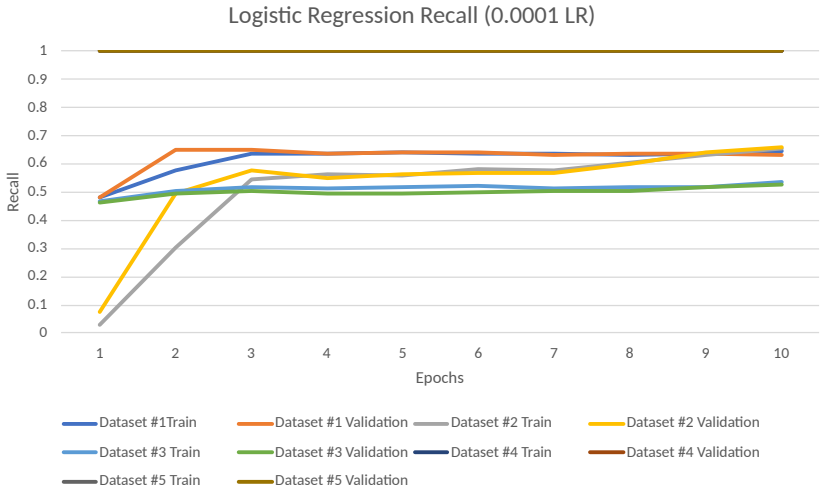

Figure D.71: *The Recall for all five datasets*

Figure D.72 shows a more refined view of the datasets that were in the expected range that showed variance.

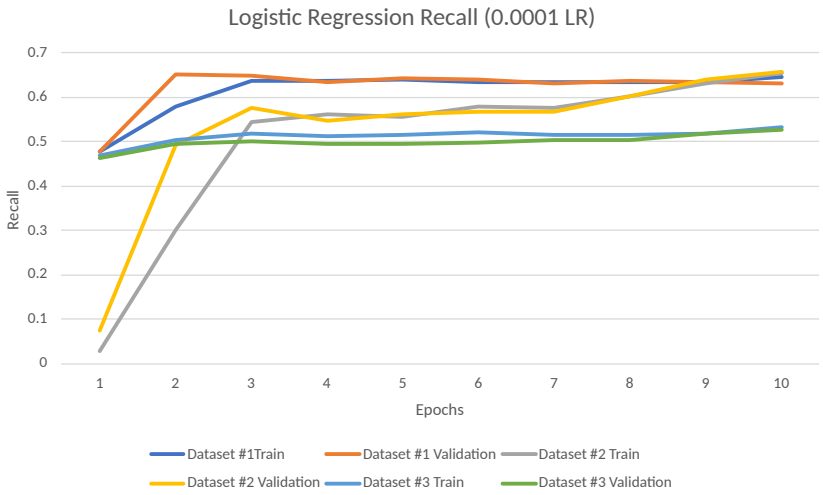

Figure D.72: *A refined view of the Recall results that focused on the expected results*

F1-Score:

Figure D.73 shows the f1-score for all five datasets.

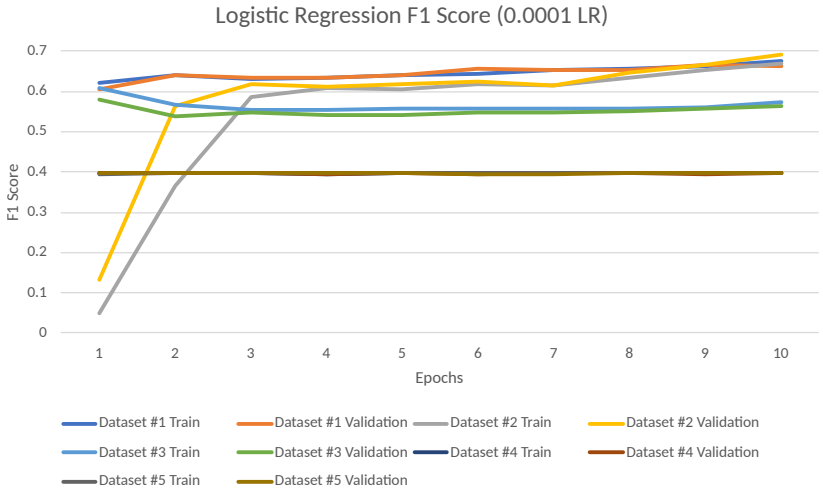

Figure D.73: *The F1-Score for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Test:

Figure D.74 shows the test results for all five datasets for all five metrics.

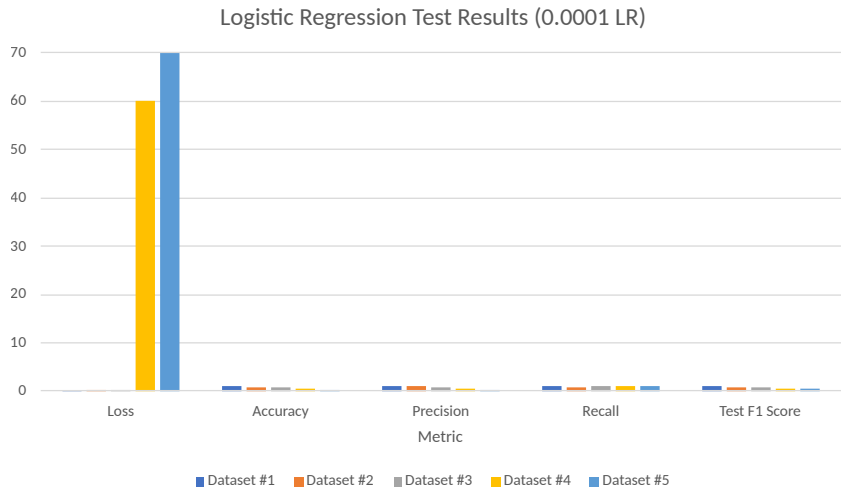

Figure D.74: *The test results for all five datasets*

Figure D.75 shows a more refined view of the datasets that were in the expected range that showed variance.

Figure D.75: *A refined view of the test results that focused on the expected results*

D.3.3 First 60,000 Instructions In Each APK With 40,000 APKs and 20 Epochs of Training

To provide further support for whether increasing the number of training epochs from 10 to 20 would improve the performance of the model, this additional set of experiments was run.

Loss:

Figure D.76 shows the loss for all five datasets.

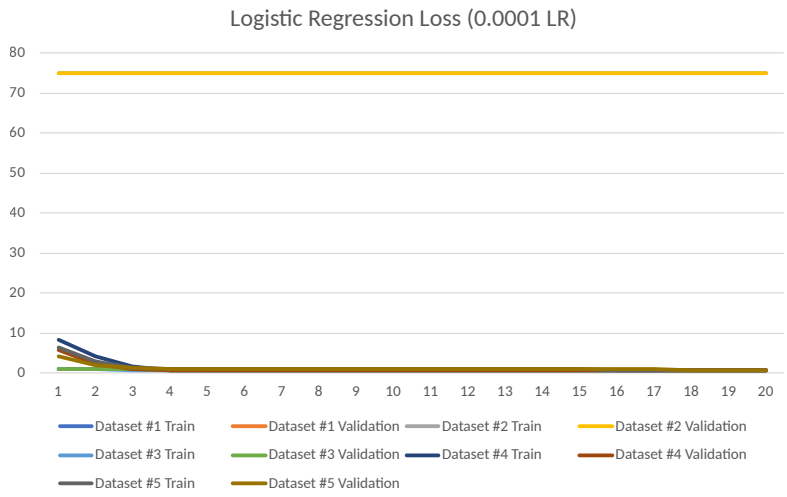

Figure D.76: *The Loss for all five datasets*

Figure D.77 shows a more refined view of the datasets that were in the expected range that showed variance.

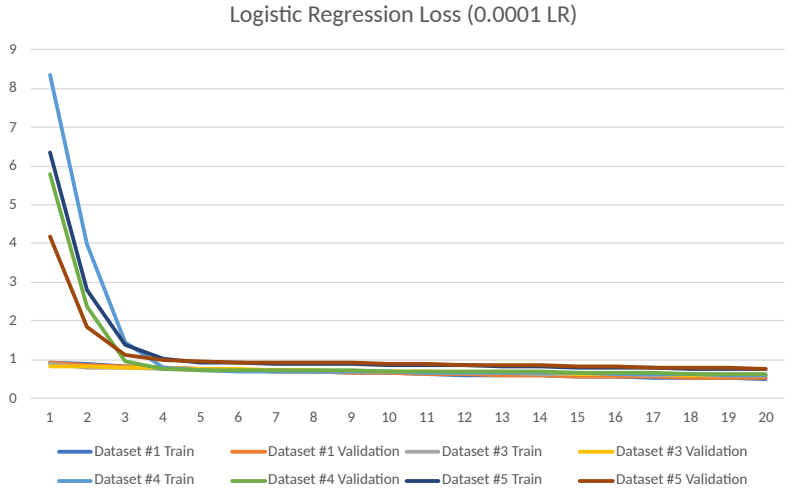

Figure D.77: *A refined view of the Loss results that focused on the expected results*

c

Accuracy:

Figure D.78 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

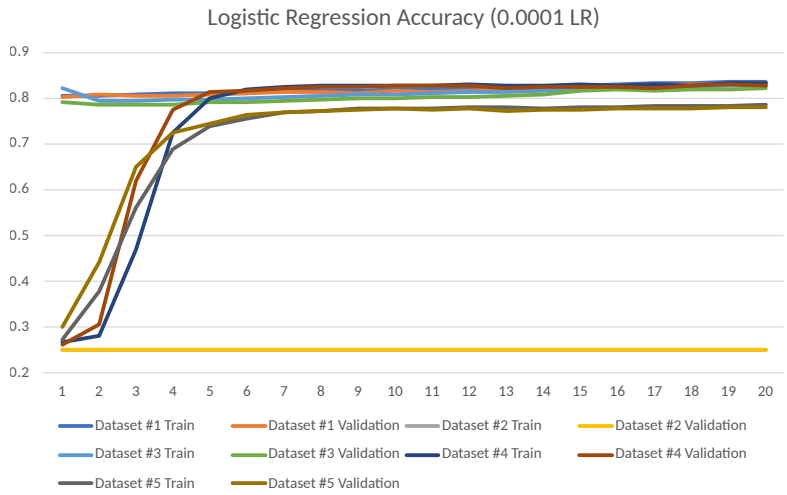

Figure D.78: *The Accuracy for all five datasets*

Figure D.79 shows a more refined view of the datasets that were in the expected range that showed variance.

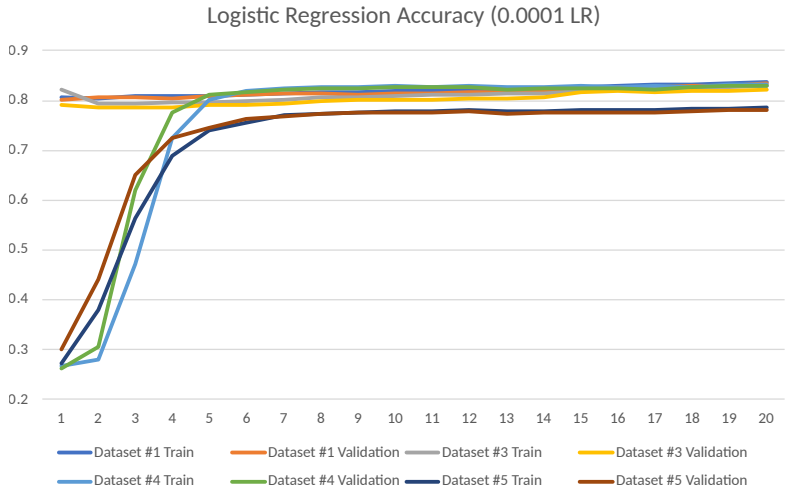

Figure D.79: *A refined view of the Accuracy results that focused on the expected results*

Precision:

Figure D.80 shows the precision for all five datasets.

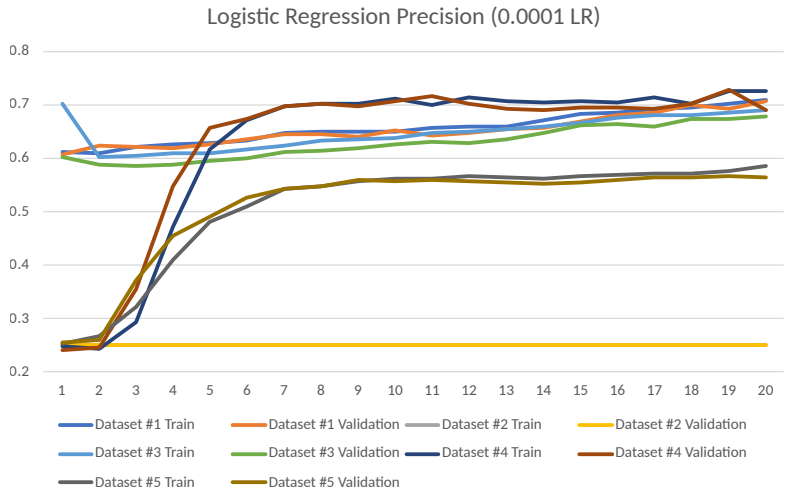

Figure D.80: *The Precision for all five datasets*

Figure D.81 shows a more refined view of the datasets that were in the expected range that showed variance.

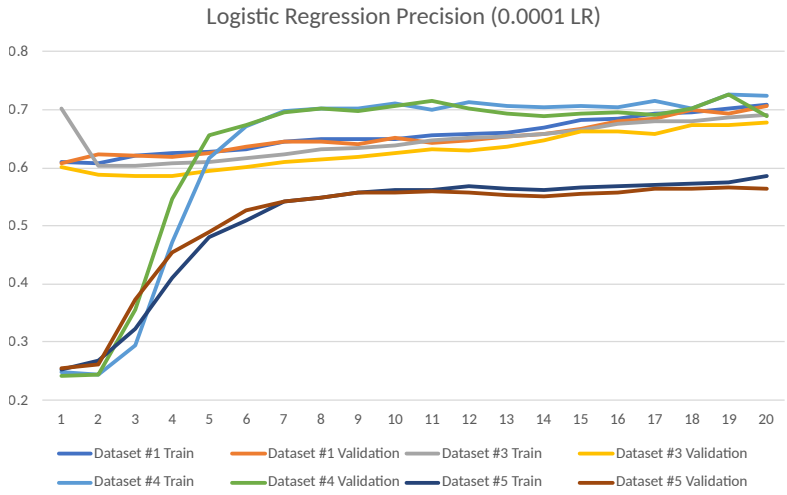

Figure D.81: *A refined view of the Precision results that focused on the expected results*

Recall:

Figure D.82 shows the recall for all five datasets.

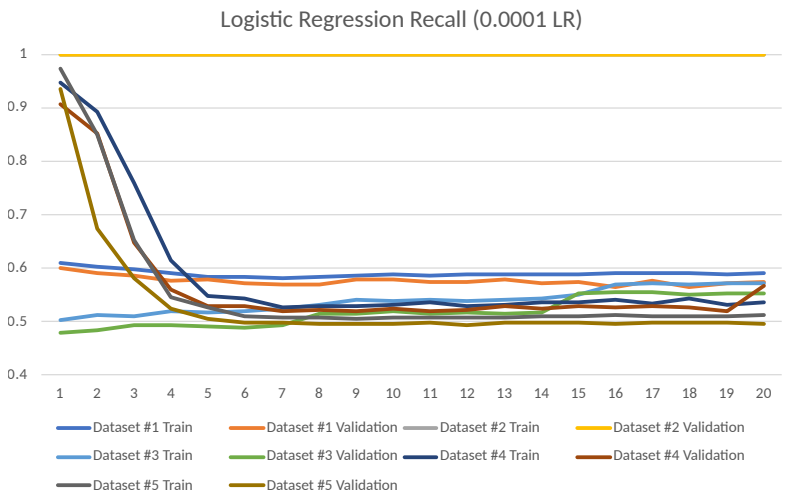

Figure D.82: *The Recall for all five datasets*

Figure D.83 shows a more refined view of the datasets that were in the expected range that showed variance.

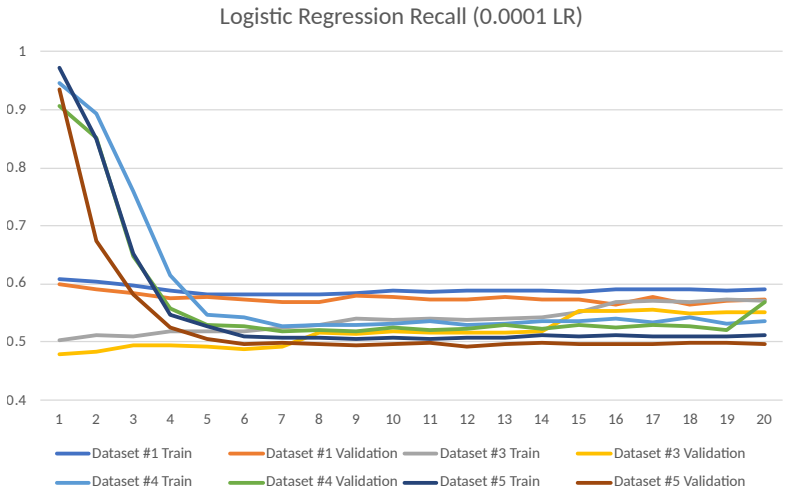

Figure D.83: *A refined view of the Recall results that focused on the expected results*

F1-Score:

Figure D.84 shows the f1-score for all five datasets.

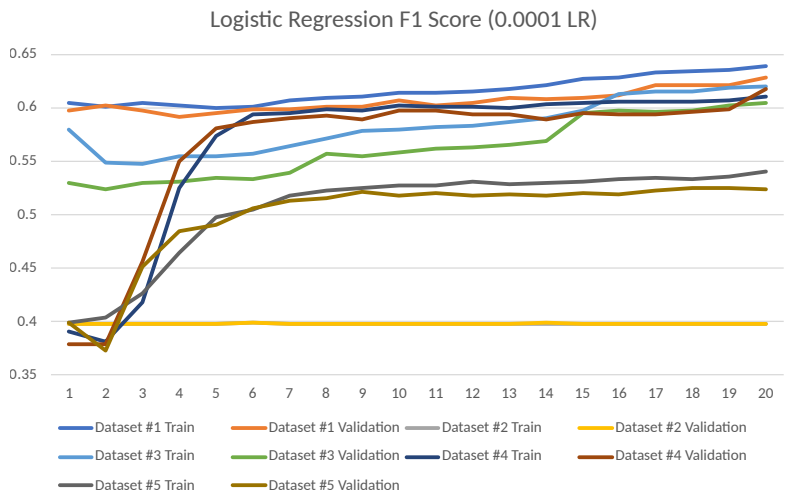

Figure D.84: *The F1-Score for all five datasets*

Figure D.85 shows a more refined view of the datasets that were in the expected range that showed variance.

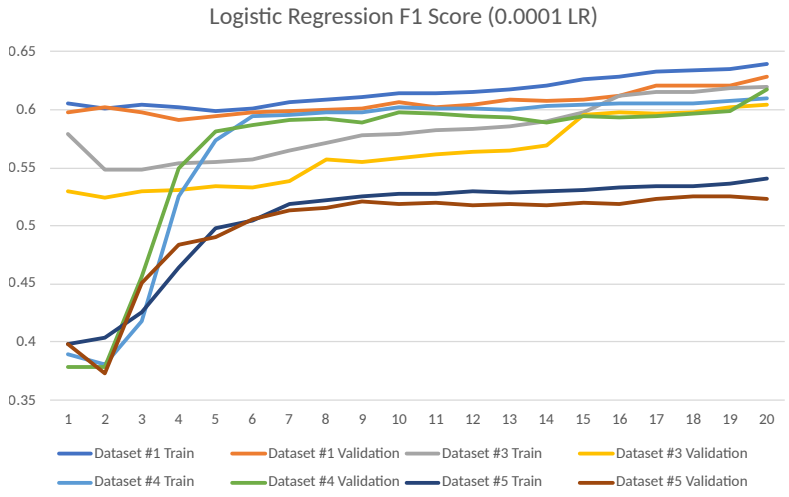

Figure D.85: *A refined view of the F1-Score results that focused on the expected results*

Test:

Figure D.86 shows the test results for all five datasets for all five metrics.

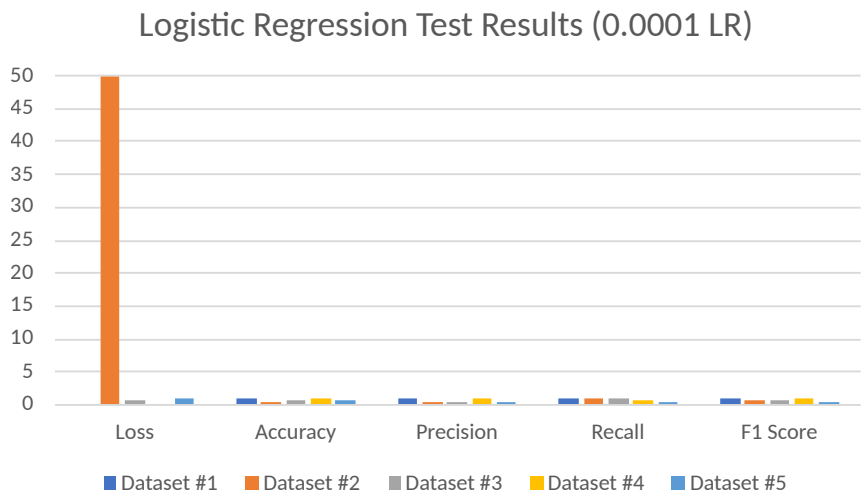

Figure D.86: *The test results for all five datasets*

Figure D.87 shows a more refined view of the datasets that were in the expected range that showed variance.

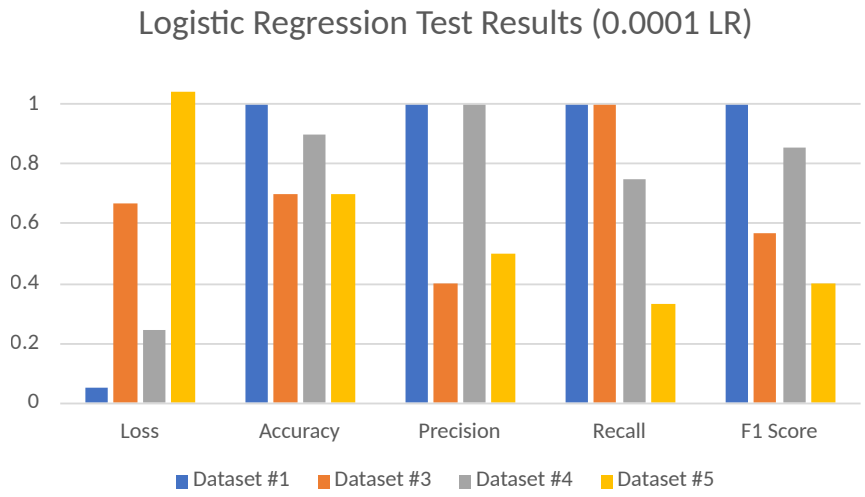

Figure D.87: *A refined view of the test results that focused on the expected results*

Appendix E

All LSTM Experiment Results

E.1 Learning Rate of 0.01

E.1.1 First 60,000 Instructions in Each APK With 64 Units in the Hidden Layer

Loss:

Figure E.1 shows the loss for all five datasets.

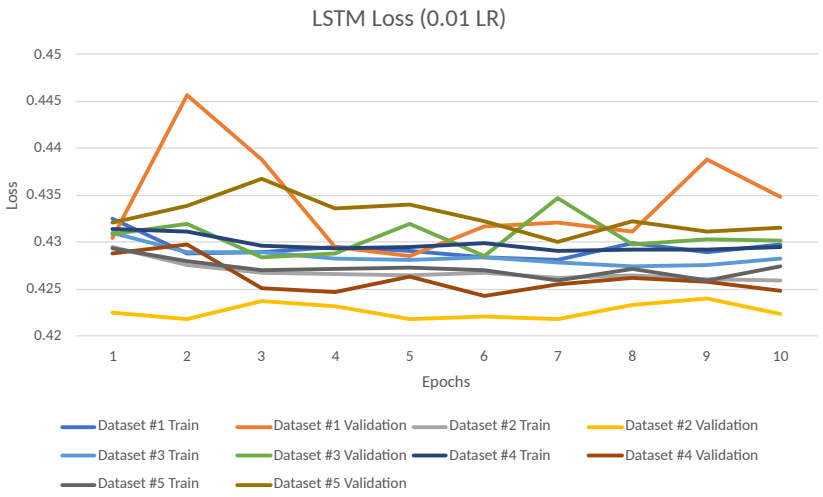

Figure E.1: *The Loss for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the

datasets were very similar.

Accuracy:

Figure E.2 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

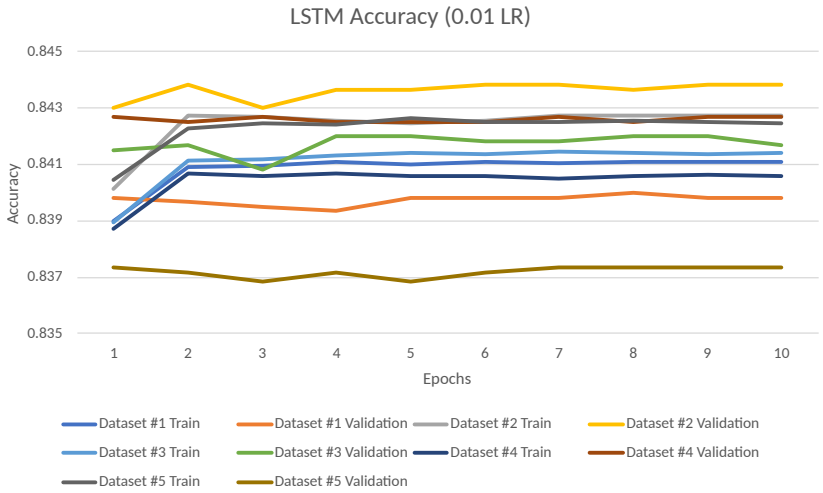

Figure E.2: *The Accuracy for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Precision:

Figure E.3 shows the precision for all five datasets.

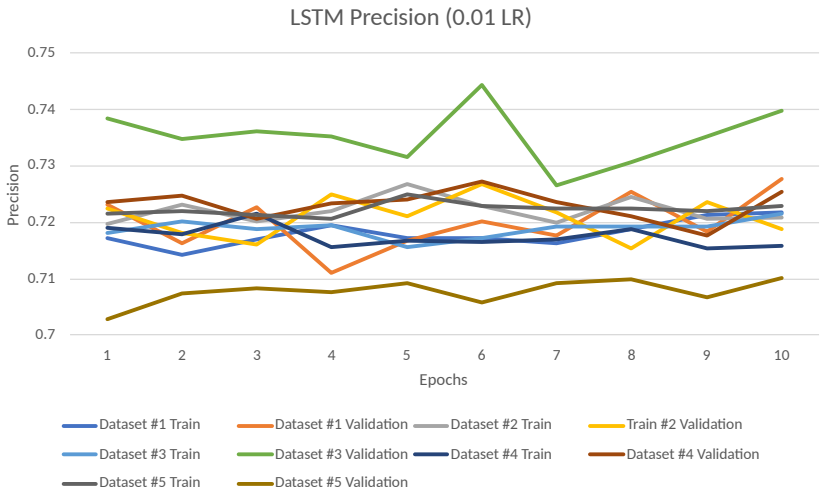

Figure E.3: *The Precision for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Recall:

Figure E.4 shows the recall for all five datasets.

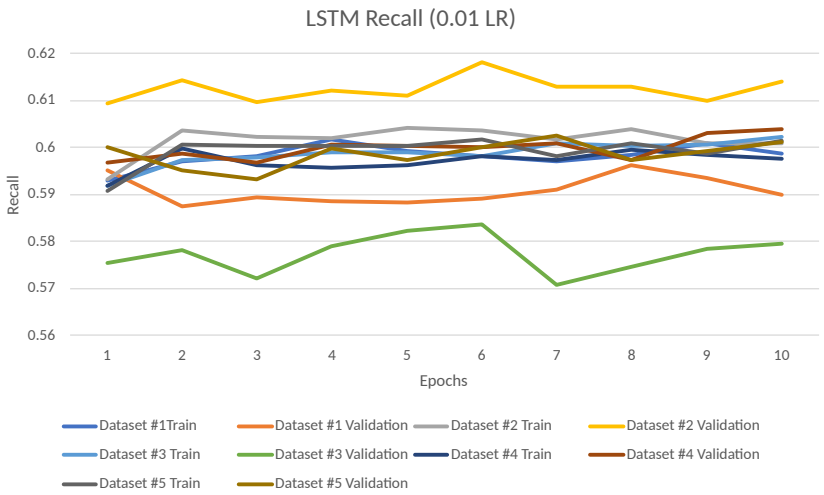

Figure E.4: *The Recall for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

F1-Score:

Figure E.5 shows the f1-score for all five datasets.

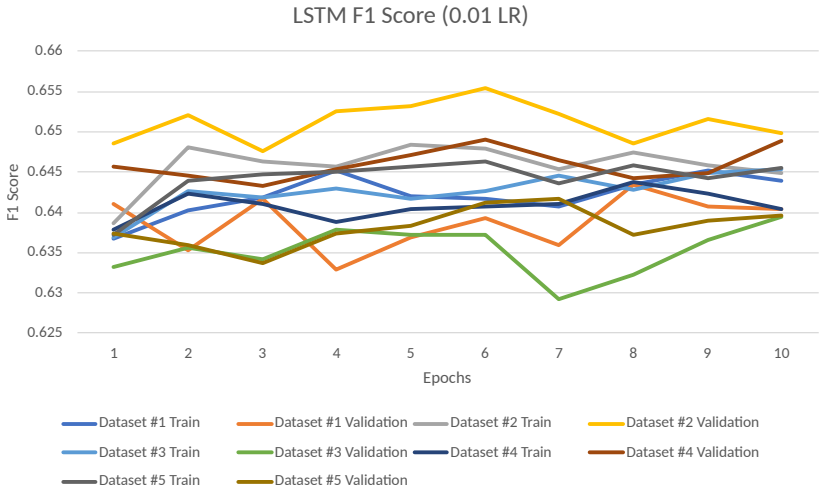

Figure E.5: *The F1-Score for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Test:

Figure E.6 shows the test results for all five datasets for all five metrics.

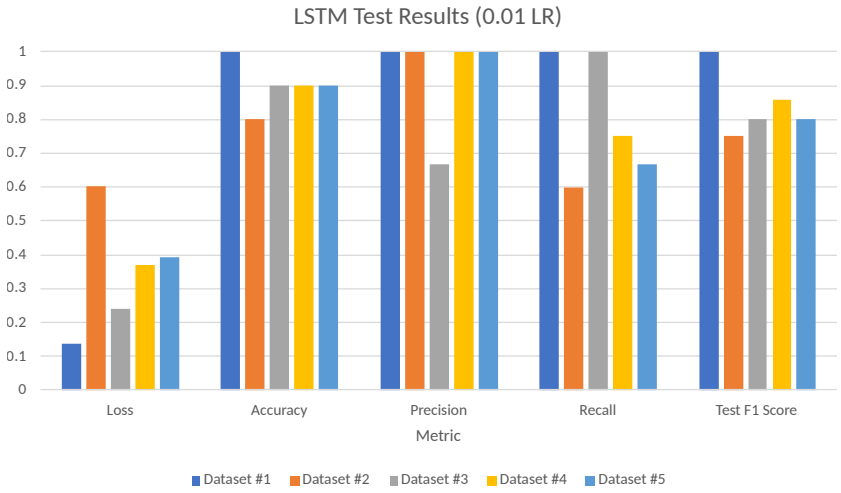

Figure E.6: *The test results for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

E.1.2 First 60,000 Instructions in Each APK With 128 Units in the Hidden Layer

Loss:

Figure E.7 shows the loss for all five datasets.

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

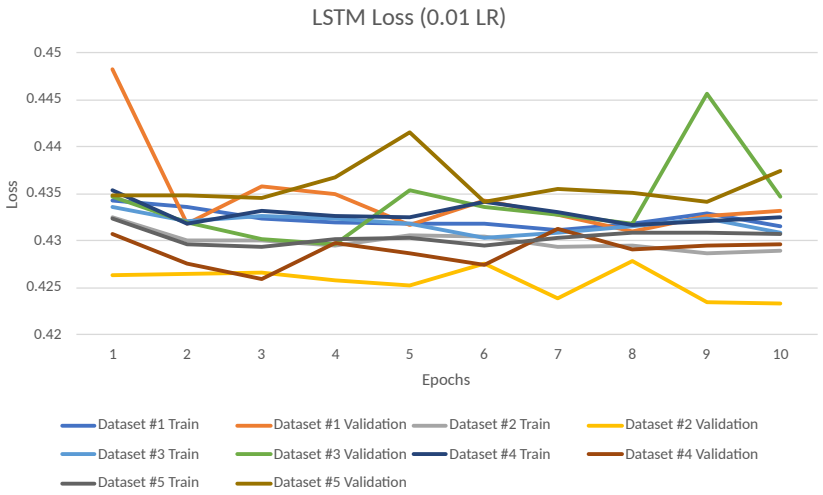

Figure E.7: *The Loss for all five datasets*

Accuracy:

Figure E.8 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

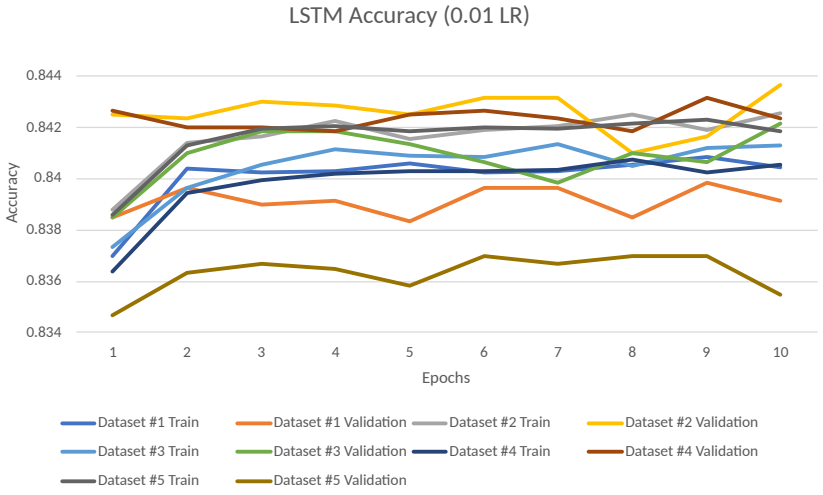

Figure E.8: *The Accuracy for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Precision:

Figure E.9 shows the precision for all five datasets.

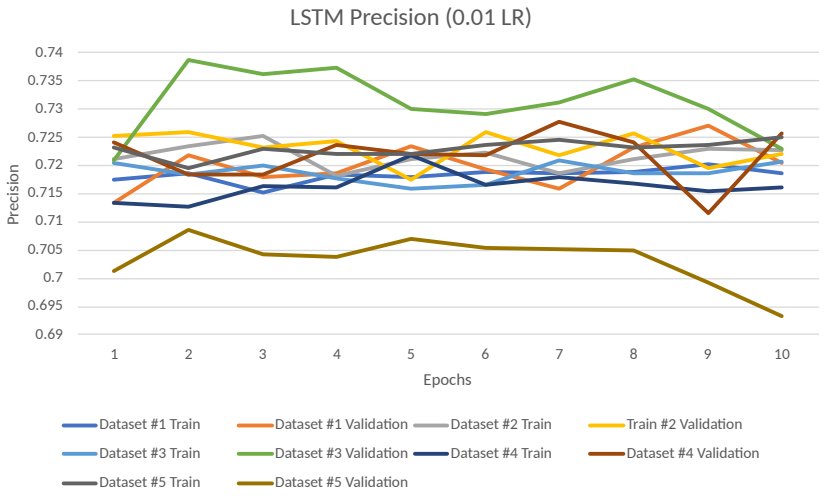

Figure E.9: *The Precision for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Recall:

Figure E.10 shows the recall for all five datasets.

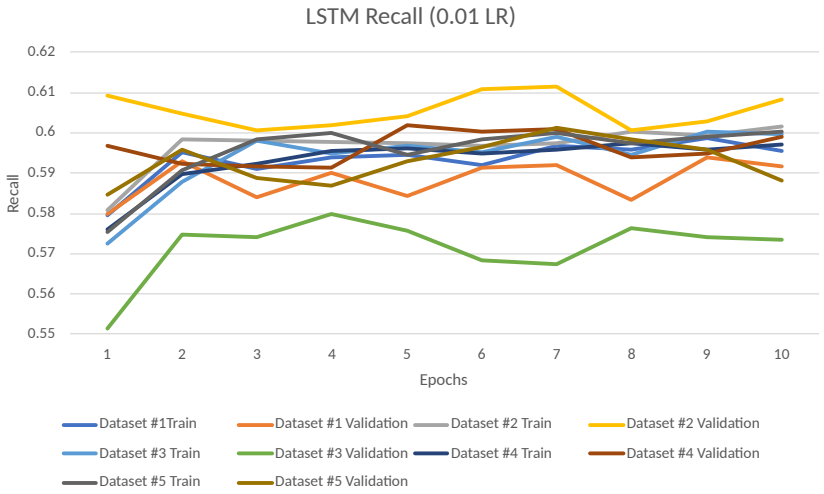

Figure E.10: *The Recall for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

F1-Score:

Figure E.11 shows the f1-score for all five datasets.

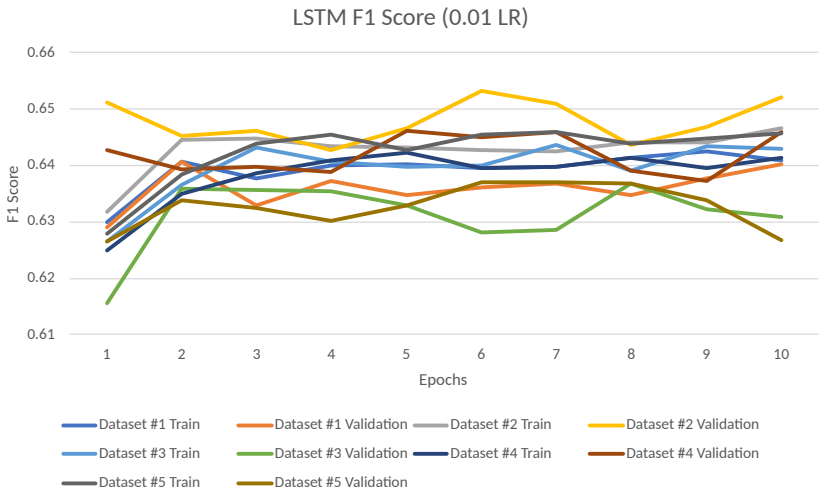

Figure E.11: *The F1-Score for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Test:

Figure E.12 shows the test results for all five datasets for all five metrics.

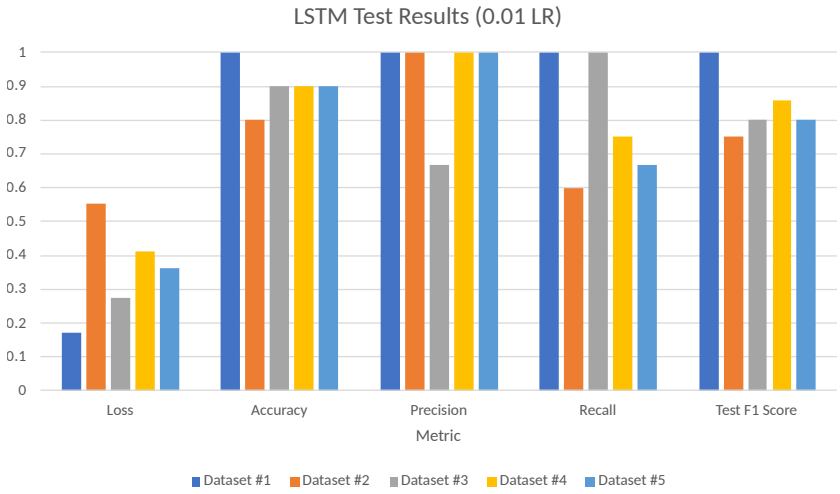

Figure E.12: *The test results for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

E.2 Learning Rate of 0.001

E.2.1 First 60,000 Instructions in Each APK With 64 Units in the Hidden Layer

Loss:

Figure E.13 shows the loss for all five datasets.

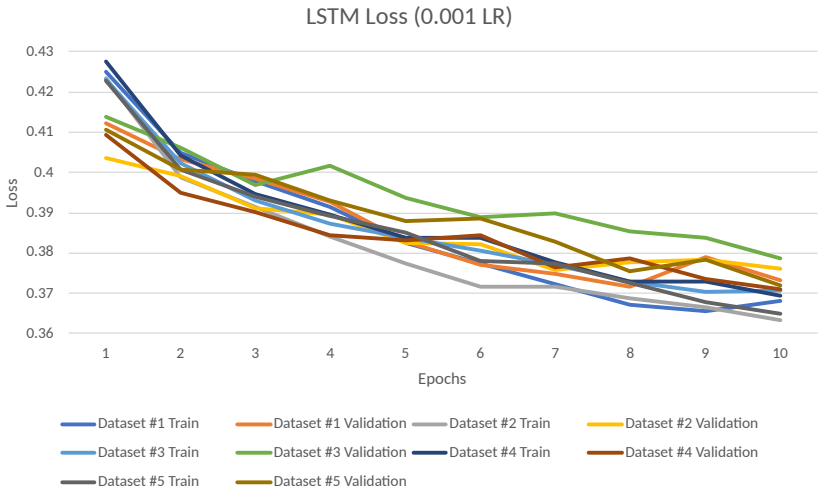

Figure E.13: *The Loss for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Accuracy:

Figure E.14 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

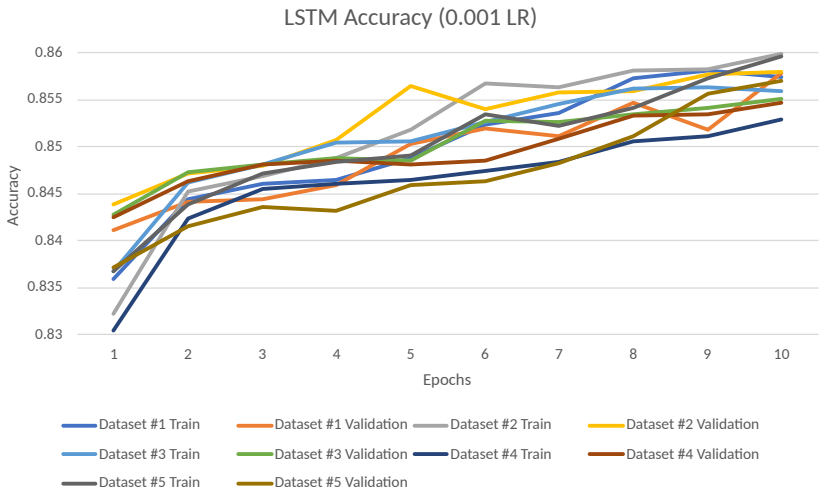

Figure E.14: *The Accuracy for all five datasets*

Precision:

Figure E.15 shows the precision for all five datasets.

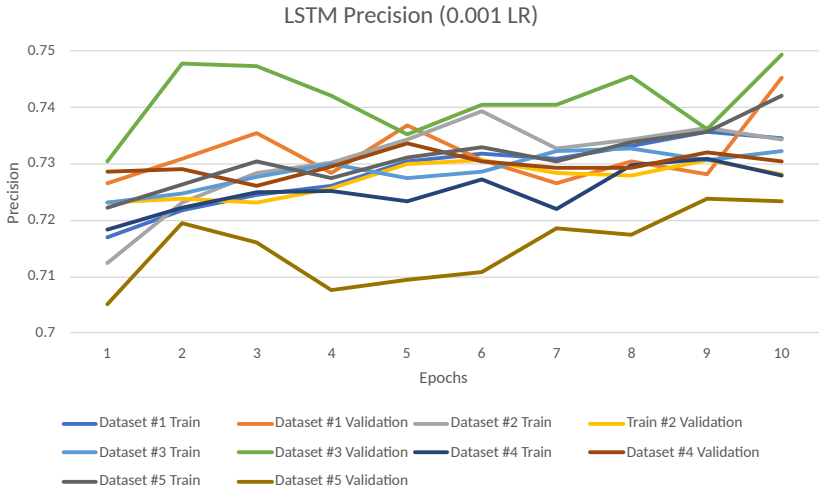

Figure E.15: *The Precision for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Recall:

Figure E.16 shows the recall for all five datasets.

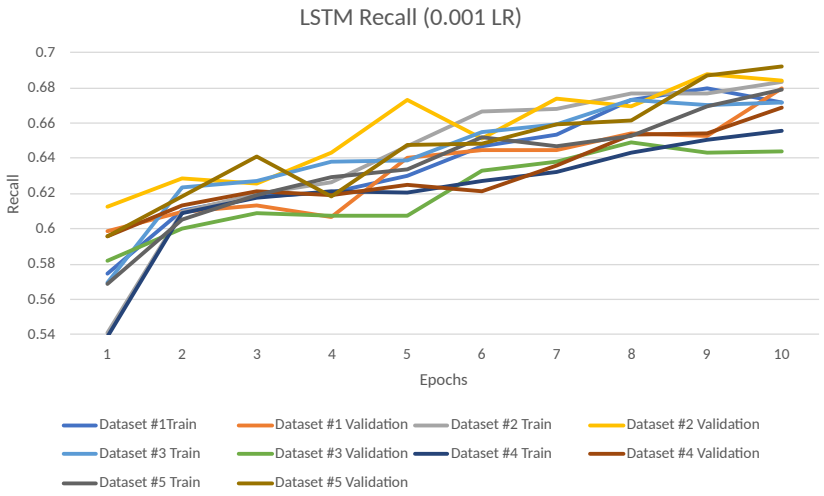

Figure E.16: *The Recall for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

F1-Score:

Figure E.17 shows the f1-score for all five datasets.

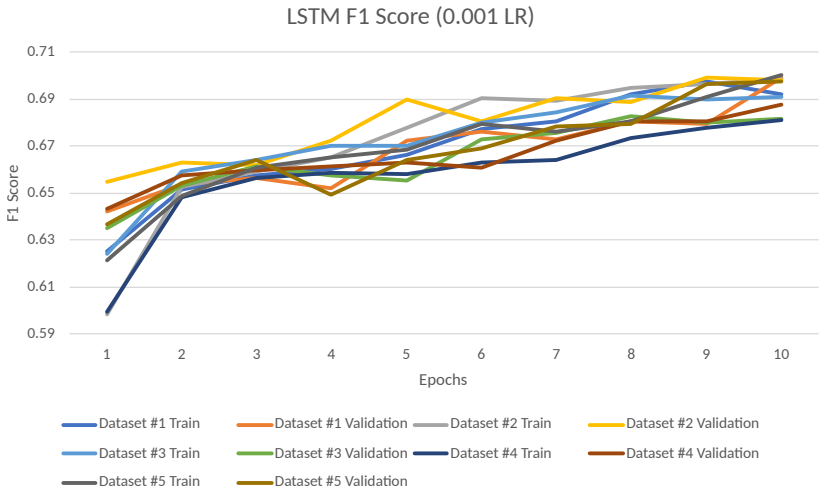

Figure E.17: *The F1-Score for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Test:

Figure E.18 shows the test results for all five datasets for all five metrics.

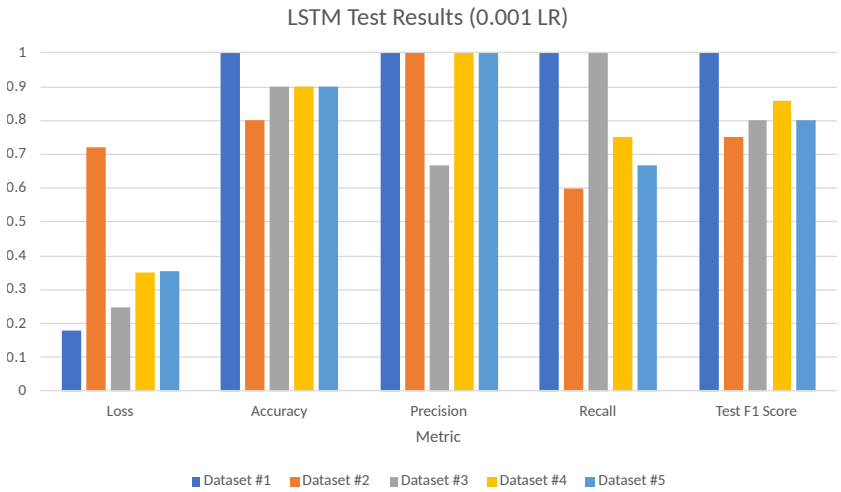

Figure E.18: *The test results for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

E.2.2 First 60,000 Instructions in Each APK With 128 Units in the Hidden Layer

Loss:

Figure E.19 shows the loss for all five datasets.

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

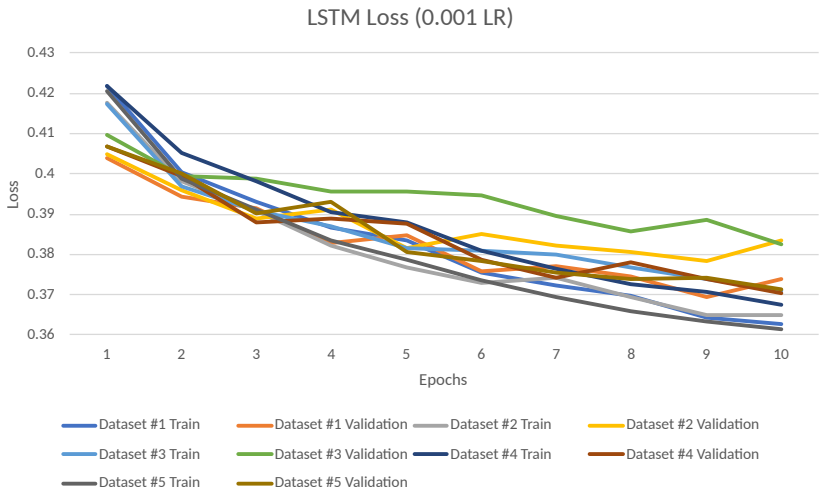

Figure E.19: *The Loss for all five datasets*

Accuracy:

Figure E.20 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

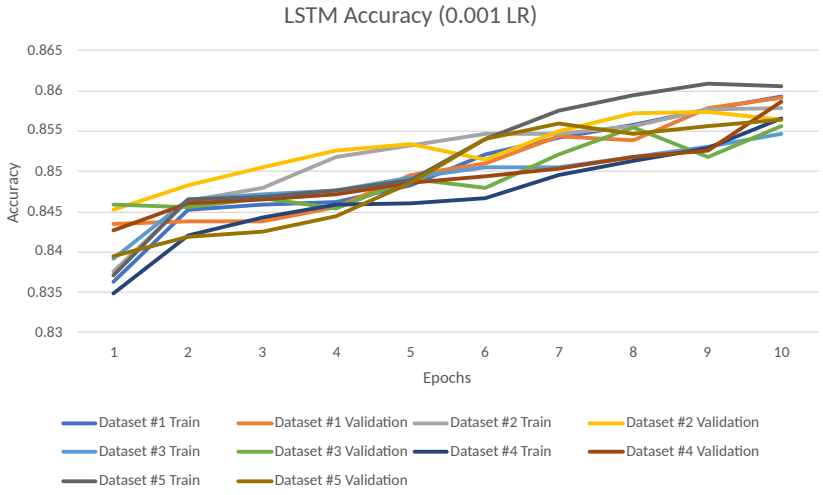

Figure E.20: *The Accuracy for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Precision:

Figure E.21 shows the precision for all five datasets.

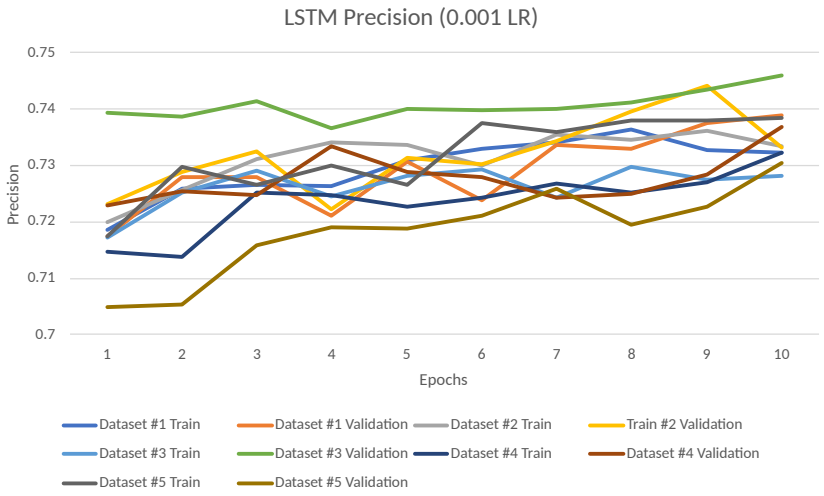

Figure E.21: *The Precision for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Recall:

Figure E.22 shows the recall for all five datasets.

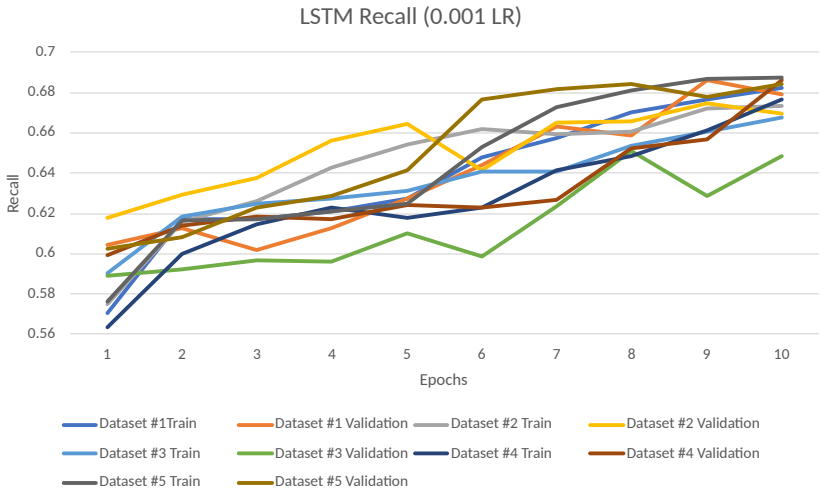

Figure E.22: *The Recall for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

F1-Score:

Figure E.23 shows the f1-score for all five datasets.

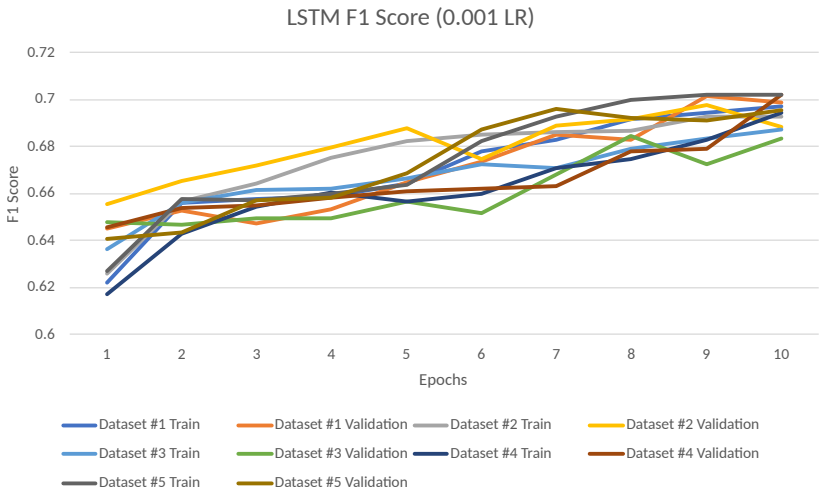

Figure E.23: *The F1-Score for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Test:

Figure E.24 shows the test results for all five datasets for all five metrics.

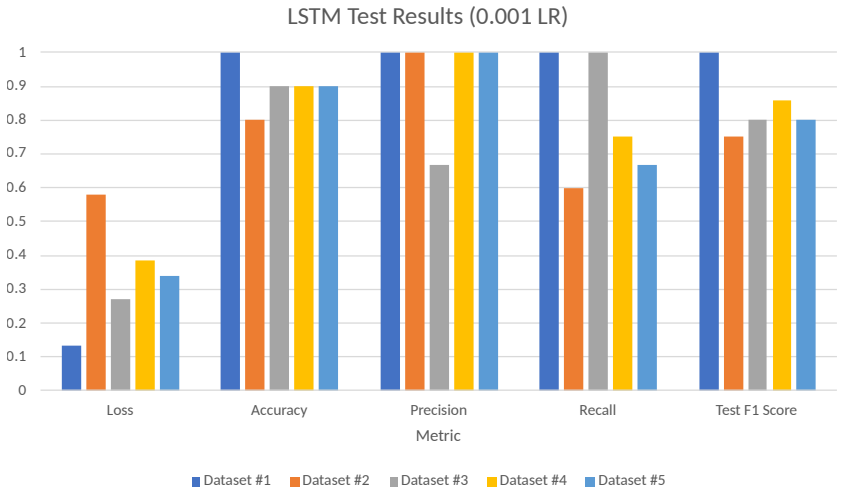

Figure E.24: *The test results for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

E.3 Learning Rate of 0.0001

E.3.1 First 60,000 Instructions in Each APK With 64 Units in the Hidden Layer

Loss:

Figure E.25 shows the loss for all five datasets.

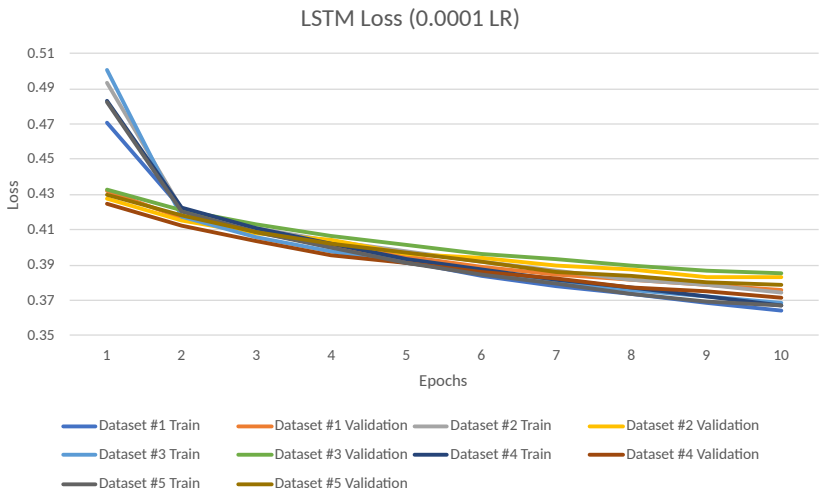

Figure E.25: *The Loss for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Accuracy:

Figure E.26 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

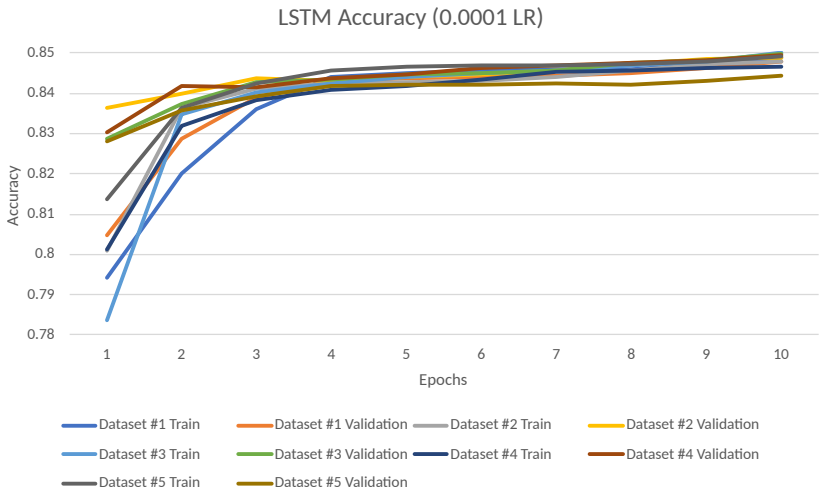

Figure E.26: *The Accuracy for all five datasets*

Precision:

Figure E.27 shows the precision for all five datasets.

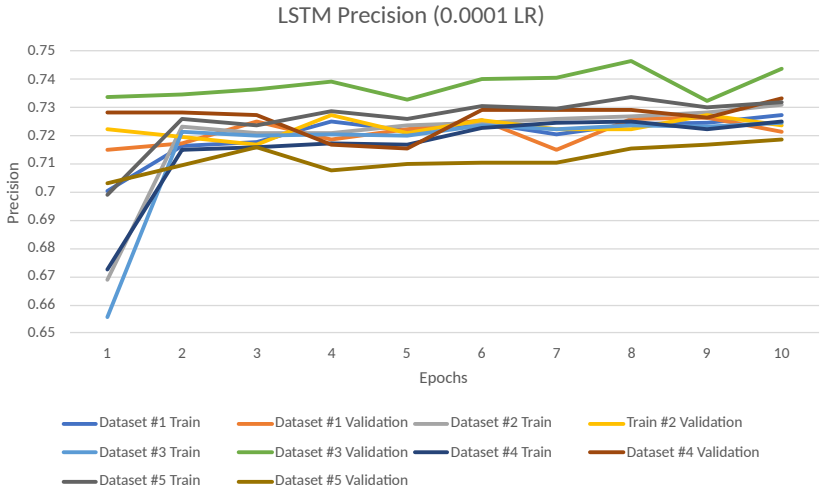

Figure E.27: *The Precision for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Recall:

Figure E.28 shows the recall for all five datasets.

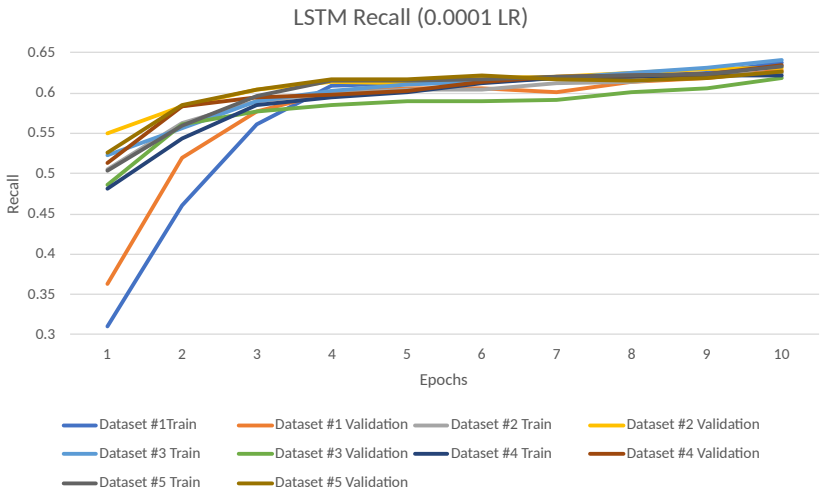

Figure E.28: *The Recall for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

F1-Score:

Figure E.29 shows the f1-score for all five datasets.

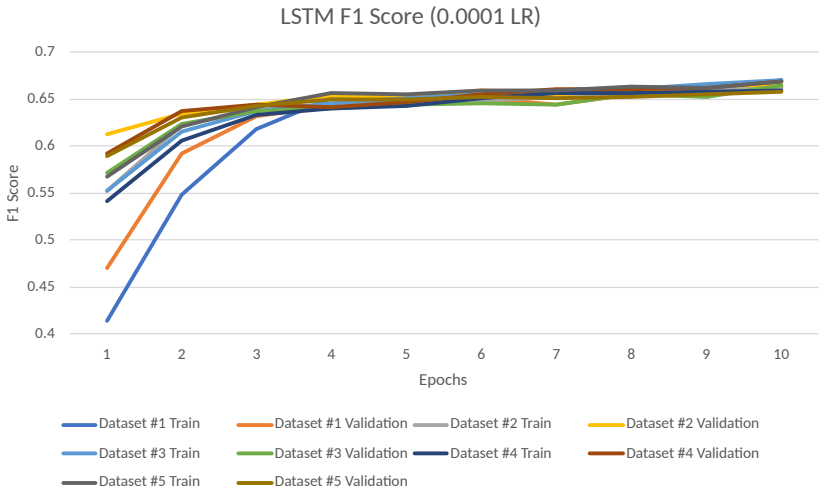

Figure E.29: *The F1-Score for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Test:

Figure E.30 shows the test results for all five datasets for all five metrics.

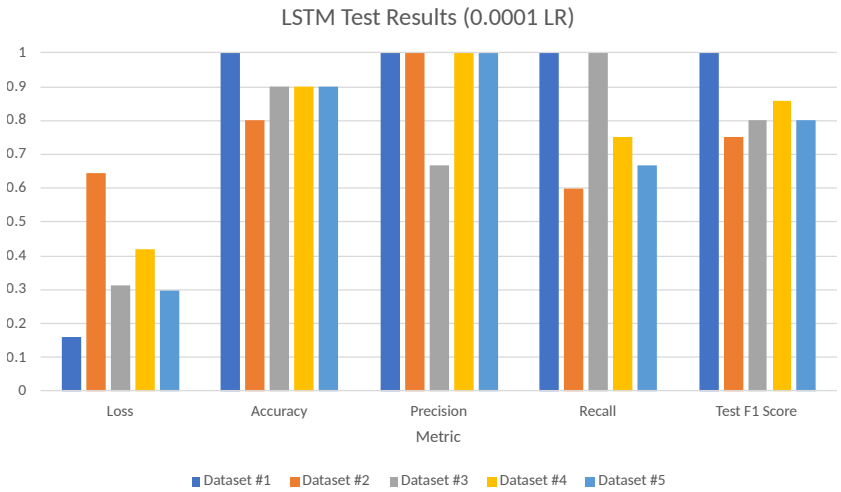

Figure E.30: *The test results for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

E.3.2 First 60,000 Instructions in Each APK With 128 Units in the Hidden Layer

Loss:

Figure E.31 shows the loss for all five datasets.

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

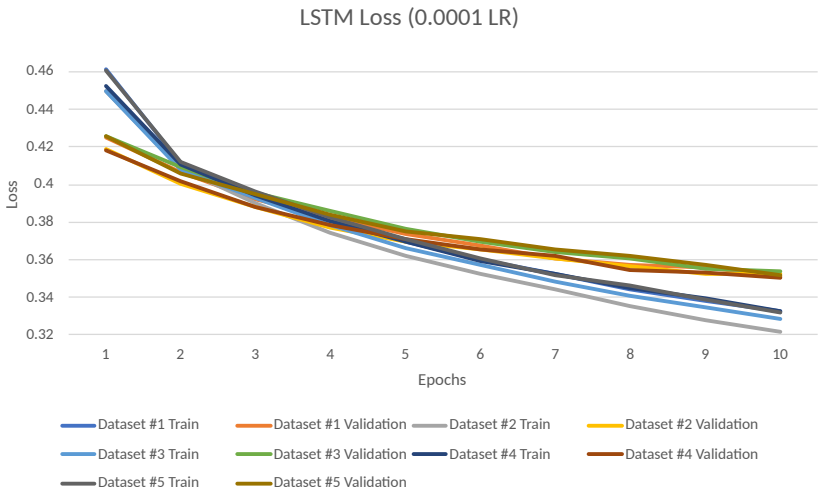

Figure E.31: *The Loss for all five datasets*

Accuracy:

Figure E.32 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

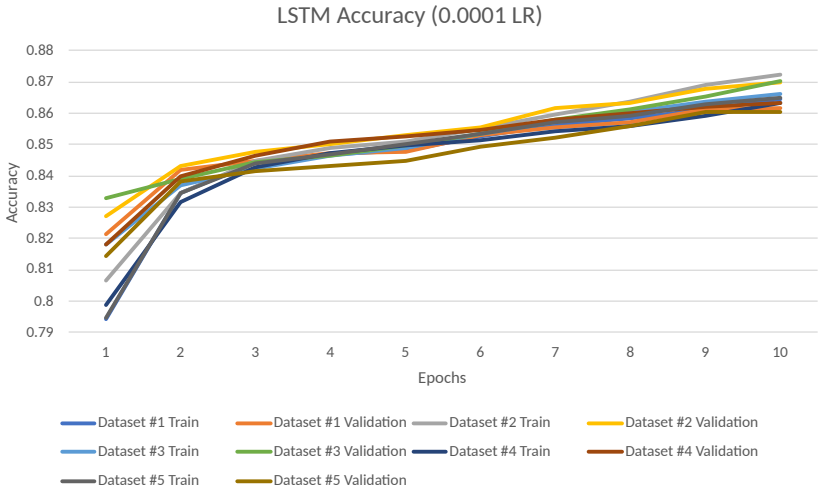

Figure E.32: *The Accuracy for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Precision:

Figure E.33 shows the precision for all five datasets.

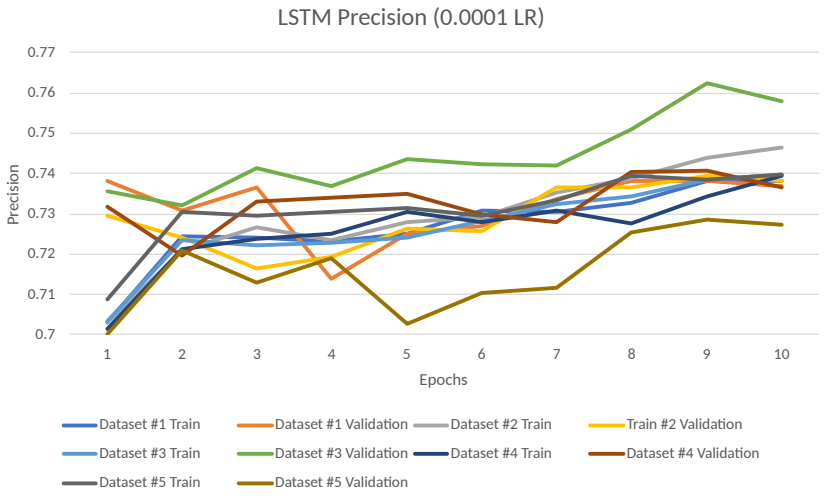

Figure E.33: *The Precision for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Recall:

Figure E.34 shows the recall for all five datasets.

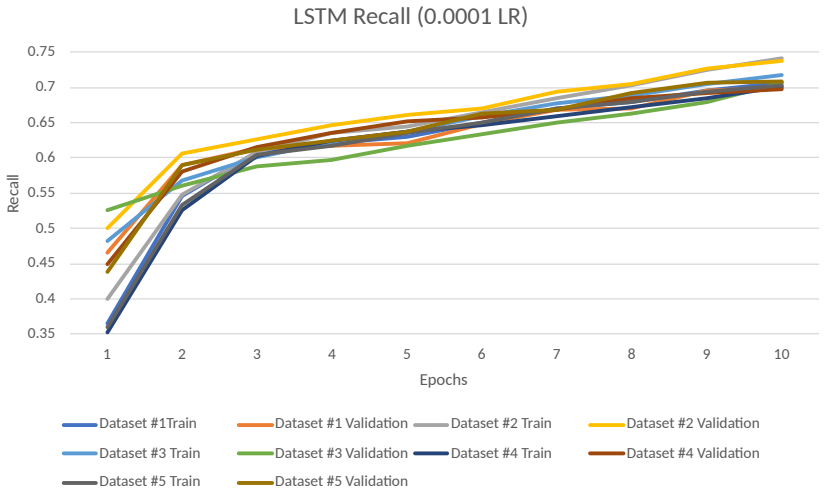

Figure E.34: *The Recall for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

F1-Score:

Figure E.35 shows the f1-score for all five datasets.

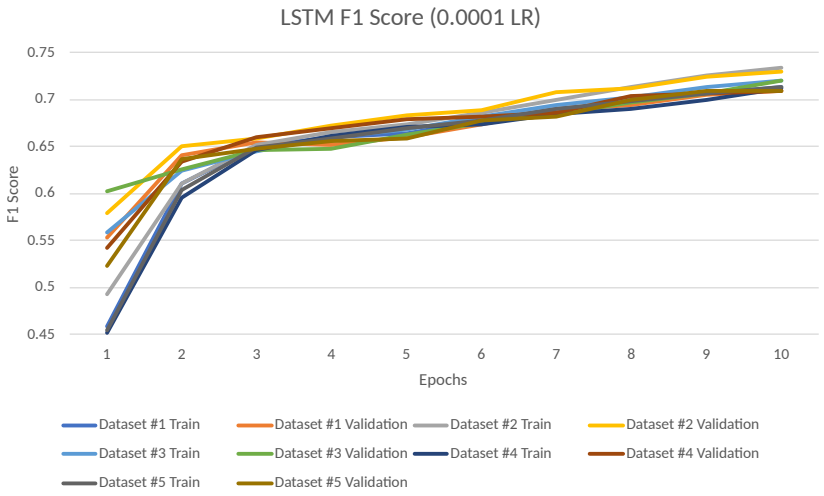

Figure E.35: *The F1-Score for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Test:

Figure E.36 shows the test results for all five datasets for all five metrics.

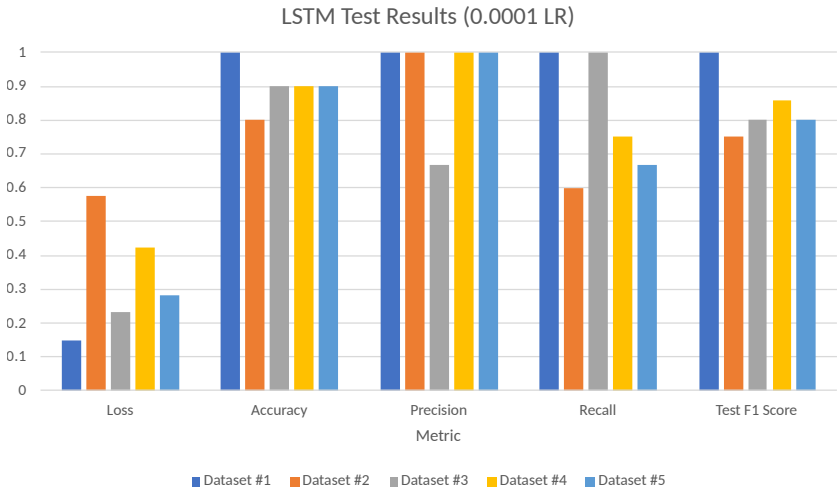

Figure E.36: *The test results for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

E.3.3 First 60,000 Instructions in Each APK With 128 Units in the Hidden Layer Run on 20 Training Epochs

To provide further support for whether increasing the number of training epochs from 10 to 20 would improve the performance of the model, this additional set of experiments was run.

Loss:

Figure E.37 shows the loss for all five datasets.

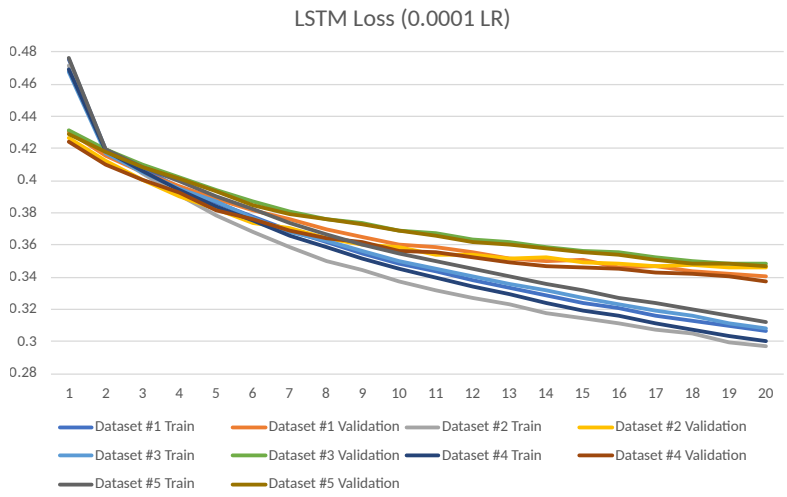

Figure E.37: *The Loss for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Accuracy:

Figure E.38 shows the accuracy for all five datasets. Note that the values in this graph are out of one, but these values map directly to percentage values.

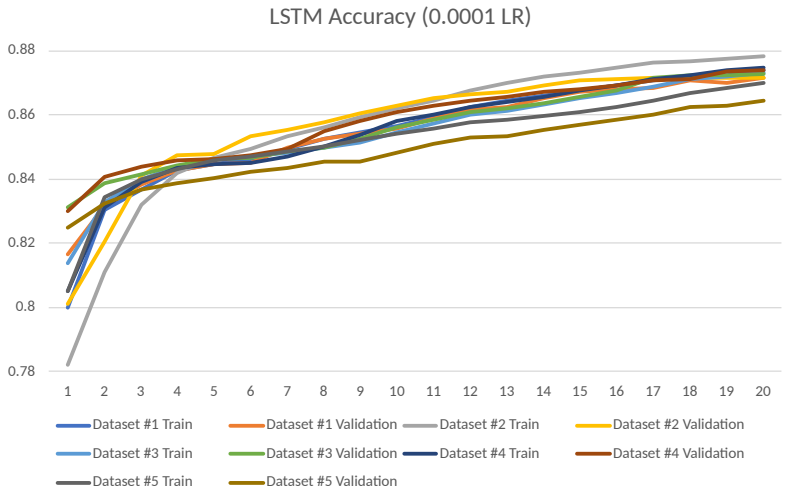

Figure E.38: *The Accuracy for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Precision:

Figure E.39 shows the precision for all five datasets.

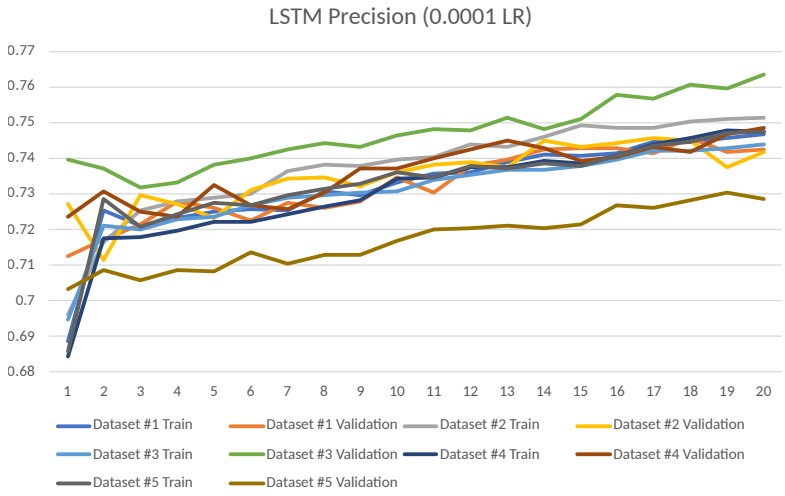

Figure E.39: *The Precision for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Recall:

Figure E.40 shows the recall for all five datasets.

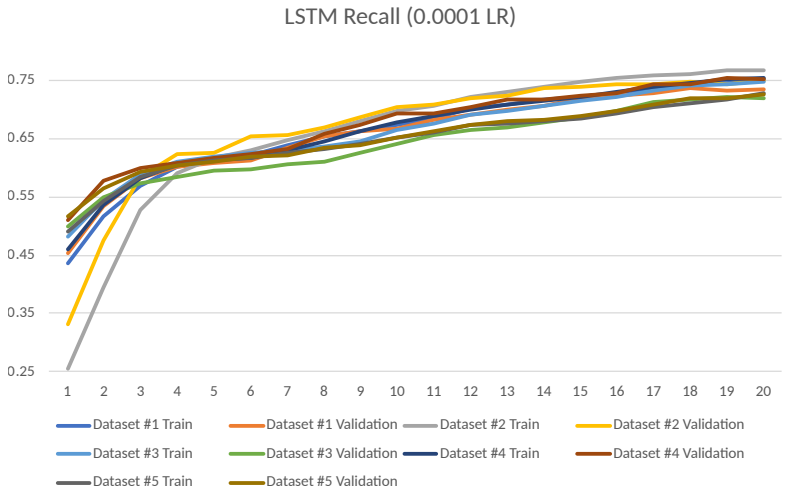

Figure E.40: *The Recall for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

F1-Score:

Figure E.41 shows the f1-score for all five datasets.

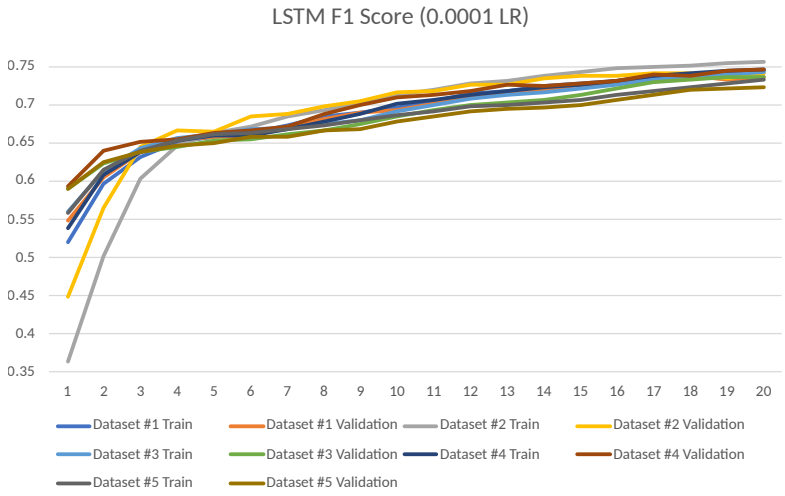

Figure E.41: *The F1-Score for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.

Test:

Figure E.42 shows the test results for all five datasets for all five metrics.

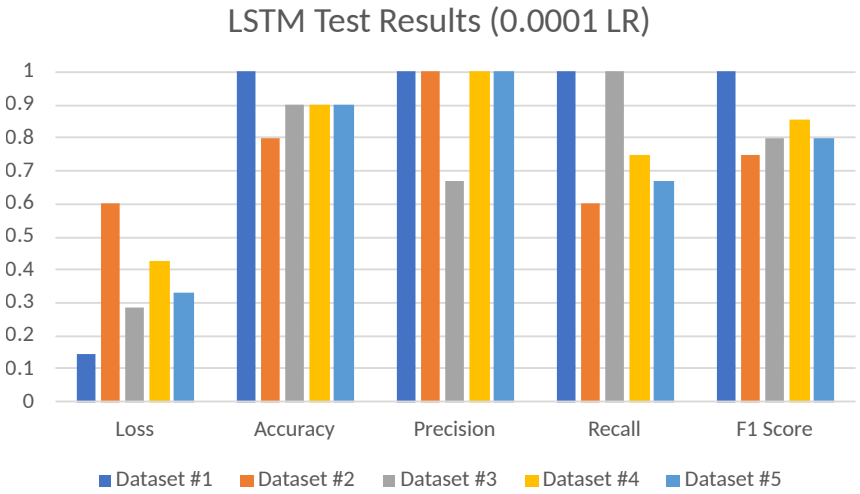

Figure E.42: *The test results for all five datasets*

A refined view of this information was not necessary since the learning rate for all of the datasets were very similar.