A Machine Learning Approach for Identifying Malicious Android Applications Based on Application Permissions

Dingyi Duan

Roberto Cancel

Lane Whitmore

Shiley-Marcos Engineering Department, University of San Diego

M.S. Applied Data Science

ADS-504: Machine Learning

Professor Siamak Aram

August 15, 2022

Abstract

Android owns such a large market share due to its versatility and open-source framework, allowing many companies to implement it into their own devices. However, this has made the Android operating system vulnerable to malware attacks. The high propensity of malware applications has led to the need for an automated malware detection system. The NATICUSdroid Dataset was created by Mathur, Podila, Kulkarni, Niyaz, and Javaid of the University of Toledo to use application permissions on Android operating systems to identify whether or not an application is malicious. We evaluated using Multi-Layer Perceptron (MLP), Support Vector Machines (SVM), Decision Tree (DT), Extra Tree (ET), and Logistic Regression (LR).

Since the cost of false negatives outweighs the cost of false positives, models were optimized for precision with a minimum accuracy threshold of 93%. The RF yielded an accuracy of 0.969 and a precision of 0.973. Our findings indicated that the model correctly classified 4304 malware apps, misclassified 115 malware as non-malware, and misclassified 159 non-malware as malware. While the classes were balanced, we attempted further to optimize our precision scores with a weighted approach. The weighted ET reaches the highest precision of 99.81%, with an accuracy of 92.74%. We, therefore, chose the weighted RF with a precision of 99.47% and an accuracy of 94.88% as the optimal model. The weighted RF yielded only 21 false negatives, but the cost of increased false positives - 430 false positives. Again, comparing the results of the weighted RF to the remaining weighted models performs best when considering all scenarios.

Keywords: Android, malware, machine learning, classification, application security

Table of Contents

List of Tables

List of Figures

Introduction

The evolution of cellular phones has led to the rapid increase of the Android operating systems in smartphones; in fact, the Android operating system controls an estimated 81% share of the global smartphone market (Statista Research Department, 2022). Android owns such a large market share due to its versatility and open-source framework, allowing many companies to implement it into their own devices. However, this has made the Android operating system particularly susceptible to malware attacks. For instance, as recently as 2014, Forbes has reported that 97% of the total malware for mobile devices occurs on devices using Android operating systems (Kelly, 2014). Additionally, Android has dominated the technology news sector with headlines such as "New Android malware apps installed 10 million times from Google Play," published by Bill Toulas of BleepingComputer LLC. The high propensity of malware applications has led to the need for an automated malware detection system. As a result, the NATICUSdroid Dataset was created by Mathur, Podila, Kulkarni, Niyaz, and Javaid of the University of Toledo to use application permissions on Android operating systems to identify whether or not an application is malicious (Mathur et al., 2021).

Objective

This project aimed to correctly classify malware applications to be flagged. An emphasis was placed on finding a statistical relationship between the application permissions and malware application class to most effectively identify and flag applications that may be malware. Due to this, less focus will be placed on accurately classifying non-malicious applications.

Methodology

Characteristics

The NATICUSdroid dataset contains 29,332 samples with 86 binary attributes and one target variable. All of the binary attributes describe the types of permissions gained by the app. These are separated into two main groups: custom permission and native, or Android-created, permission and indicated by the value of "0" and "1". The target variable "Result" simply identifies whether the app is identified as malware. The data was collected between 2010 and 2019. Through EDA below, we will explore the distribution of each feature for all the samples and aim to discover the inner relationship between them as well as with the target variable. We expect to disclose that malware apps follow a specific pattern to cause harm by acquiring certain permission within the android system.

Exploratory Data Analysis

To prevent data leakage, the dataset was immediately split into 70 % training set (n=20,532) and 30% test set (n=8,800) – as suggested by Mathur et al.'s (2021) documentation in the UCI repository. EDA was conducted on only the training set to ensure our test set did not inform our inferences and remained unseen. Examining the means of our binary attributes showed that only ten of the 86 attributes were present for more than 25% of observations - these were strictly native permissions and included: Internet, Access_network_state, write_external_storage, read-phone_state, access_wifi_state, wake_lock, access_coarse_location, receive_boot_completed, access_fine_location, and vibrate. All of these native permissions represent basic app functionality requirements.

Custom and native apps were segregated and explored to further analyze the data. Once segregated, their propensity for identified malware applications was examined. The analysis of apps identified as malware showed strong prevalences for certain custom permissions identified

as malware, while other permissions were not present in malware. Conversely, all custom permissions were present in benign applications. The analysis of our native permissions showed that only three native permissions have no occurrences in malware apps. These include FOREGROUND_SERVICE, REQUEST_INSTALL_PACKAGES, READ_APP_BADGE, AND USE_FINGERPRINT. FOREGROUND_SERVICE requires user notification of performed app operations - notifications of operations would be detrimental to a malicious application. REQUEST_INSTALL_PACKAGES allows an application to request installing packages and is highly restricted by Android. READ_APP_BADGE is, again, a notification-oriented permission. Finally, USE_FINGERPRINT allows an app to use fingerprint hardware, allowing additional security to access or open the app. Only one permission is minimally represented, PROCESS_INCOMING_CALLS, since it was depreciated while the data was collected.Lastly, android.permission.android.permission.READ_PHONE_STATE is an error in the data since android.permission.READ_PHONE_STATE is a highly occurring permission for malware apps.

Data Pre-Processing

The initial data set provided by the University of Toledo is absent of the class issues that a data set would ordinarily contain as both classes are essentially a 50/50 split. Additionally, scaling or normalization processes were not required as all the features are binary. The dataset also contained no missing or null values. Due to this, the primary focus of the pre-processing done for this project was committed to the feature selection for model construction.

Feature Selection

While the data set did not contain any null variables or class imbalance, there were issues with highly correlated and near-zero variance features. First, features with a variance of one

percent or less were removed from the data set. In total, this process removed 11 features from the original 87. Removing zero variance and near-zero variance features will improve model performance later. These 11 features offered little to no variance for the model to learn a pattern and therefore are irrelevant. The remaining 76 features were then used to build a correlation matrix. Features with greater than 0.80 correlation were then removed from the data set to reduce multicollinearity concerns during modeling. This process removed 20 more features and left the final data set with 56 features. By removing these features, models that do not perform feature selection will only be presented with relevant features, and the multicollinearity of the models should be reduced. Reducing the multicollinearity of the models will benefit by reducing the potential of overfitting and ensuring the model results are not inflated by correlated features.

Results

Modeling

Multi Layer Perceptron (MLP)

Multi Layer Perceptron (MLP) is a feed forward neural network consisting of three layers: the input, output layer, and hidden layer(s). We initially trained the MLP by optimizing the learning rate and comparing the Adam and SGD optimizer for precision performance. As we iterated through a list of learning rates from 0.00001 to 4.0, we noted the Adam optimizer offered the best performance with an accuracy of over 96% at a learning rate of 0.005. Once we chose the Adam optimizer, we further tuned through two additional hyperparameters, epsilon which is the stability for Adam, and epochs, to finalize our MLP model. With the final decision for learning rate of 0.005, epsilon of 0.001 and epochs of 8, the MLP model provided impressive

performance: an accuracy score of 0.965, a precision score of 0.973, and an ROC score of 0.965. With nearly 9000 samples, MLP only has 117 false negatives and 195 false positives.

Since we are more concerned with identifying malware rather than "escapes", we decided to turn up a notch by adding a weighted version of the model to optimize the precision score even further. By gradually shifting the weight from the "non-malware" side, we achieved a precision score of a whopping 0.995 with only 18 false negatives. Even though our accuracy dropped, this may be more suitable from a business perspective.

Support Vector Machines

Support Vector Machines (SVM) are a supervised learning approach that takes the data points within a dataset and generates a hyperplane that best separates them by two classes. One of their primary advantages is their effectiveness in high-dimensional settings. Initially, the SVM was trained with varying cost, or regularization, values for the four kernel types: linear, polynomial, radial basis function (RBF), and sigmoid. We note that the radial basis function produced the highest precision of 0.969 at a cost value of 15 – indicating that a non-linear classifier performs best with our data. Then, using the RBF kernel, we again trained our model to identify a more precise cost – identified as 17 with a precision of 0.970. Gamma is a parameter of the RBF kernel and identifies the decision region. After training the SVM with a range of Gamma values, the final optimal SVM was identified using the RBF kernel, cost of 17, and Gamma of 0.35, resulting in a 0.967 accuracy and 0.971 precision. Our findings indicate that the final SVM yielded 124 false negatives and 161 false positives.

In keeping with the weighted approach in our MLP, the SVM was also weighted to optimize the precision score further. Interestingly, the weighted approach of {0: 0.95, 1: 0.05}

yielded the best precision at 0.992; however, our accuracy was sacrificed and reduced to 0.950. Our findings indicate that the weighted SVM yielded only 30 false negatives but 414 false positives.

Decision Tree

Decision Tree (DT), as a fast, easy-to-implement classification model that can handle both categorical and numeric values, is another strong contender for this task. While we expected it to perform well, we must also be aware of the overfitting issue. For decision tree classification we start by iterating through the tree depth for both Gini and Entropy impurity with 10-fold cross validation. Using a base model, both impurities achieved very similar precision scores of 0.963. Since decision trees are known for their overfitting issue, we then proceeded to tune the cost complexity hyperparameter "CCP_alpha" for pruning. The results showed that after pruning, "Gini" impurity outperformed "Entropy" with a precision score of 0.966 at "CCP_alpha" = 0.00005. Furthermore, the confusion matrix showed that the decision tree using "Gini" impurity with "tree_depth = 27" and "CCP_alpha" = 0.00005 only misclassified 132 non-malwares as malwares with an ROC score of 0.965, which is a decent result.

Once again, we added class_weight to our decision tree model and focused on catching escapes rather than false calls. With the optimal weight ratio of 0: 0.95 and 1: 0.05, the weighted decision tree model achieved a precision score of 0.992, a fairly high accuracy score of 0.935, and an ROC score of 0.934. The final confusion matrix gave a false positive of 30 and a false negative of 542.

Random Forest

Random forest (RF) builds multiple decision trees on different samples and takes their majority vote for classification – an ensemble of decision tree algorithms. RF uses bagging and random features when building each tree to create an uncorrelated "forest" of trees to create more accurate voted predictions than any individual tree. Furthermore, since Mathur et. al (2021) identified the RF as the best performing model in their study, it was selected as an approach in our study as well. It, like ET, has three hyperparameters: criterion, max depth, and CCP_alpha. The criterion represents the function used to measure split quality; Max_depth can be constrained to avoid overfitting; CCP_alpha is a complexity parameter used for pruning. Initially a range of max_depth values were used to train the model; however, CCP_alpha values attained a higher precision score, so it was used in place of max_depth for our final model using the gini criterion and ccp_alpha of 0.0005. This yielded an accuracy of 0.969 and precision of 0.973. Our findings indicate that the model correctly classified 4304 malware apps, misclassified 115 malware as non-malware, and misclassified 159 non-malware as malware. These are the best results for our models.

Again, a weighted approach to optimize the precision score further was evaluated. A point of difference is that the weighted approach of {0: 0.90, 1: 0.10} yielded the best precision at 0.995; however, our accuracy was sacrificed and reduced to 0.949. The weighted RF yielded only 21 false negatives, but an astonishing 430 false positives. Again, when comparing the results of the weighted RF to the remaining weighted models, it performs best when considering all scenarios.

Extra Tree Classifier

Extremely Randomized Trees Classifier (ET) is an ensemble learning approach that aggregates the results of multiple decision trees collected in a "forest" to produce its

classification result. The primary difference between RF and ET is that ET chooses the optimum split randomly - indicating ET adds randomization but still has optimization. It, like RF, has three hyperparameters: criterion, max depth, and CCP_alpha. The criterion represents the function used to measure split quality; Max_depth can be constrained to avoid overfitting; CCP_alpha is a complexity parameter used for pruning. First, we trained our ET with a max_depth range between two and 52 for the Gini and entropy criterion, resulting in similar precision scores. Since tuning to ccp_alphas yielded better precision results, we tuned our ET for ccp_alpha values for the Gini and entropy criterion, resulting in our final ET model using the Gini criterion with a ccp_alpha value of 0.00005, producing an accuracy of 0.967 and precision of 0.973. Our findings indicate that the model correctly classified 4303 malware apps, misclassified 116 malware as non-malware, and misclassified 172 non-malware as malware.

Again, a weighted approach to optimize the precision score further was evaluated. Again, the weighted approach of {0: 0.95, 1: 0.05} yielded the best precision at 0.998; however, our accuracy was sacrificed and reduced to .928. The weighted ET yielded only seven false negatives, but an astonishing 632 false positives.

Logistic Regression

Logistic Regression (LR) is an easy-to-implement classification approach that estimates the probability of an event occurring based on a given dataset of independent variables. The saga solver was chosen since it is the solver of choice for sparse multinomial LR and our dataset consisted of sparse binary attributes. L1 regularization produced a higher precision of .953 compared to L2 regularization's precision of 0.951 – resulting in the final LR model with an

accuracy of .959 and precision of .958. Unfortunately, LR was the worst performing model; therefore, its results will not be discussed further.

Discussion

Model Selection

The performance metrics for all models is available in Table 1. While our primary goal was correctly classifying malware, shareholders dictated an Accuracy threshold score of 0.930 to limit false positives. False positives indicate benign apps that are flagged as malware and the threshold was set to avoid discouraging developers see figure in the Android system and spamming users with potential false positives. Given this threshold and our stated goal, the weighted RF is the final selected model. We note that the weighted MLP outperforms the weighted RF in precision with a 0.998 compared to 0.995, respectively; however, the weighted MLP sacrificed Accuracy at 0.920 compared to the weighted RF at 0.949.

Table 1

Final Model Metrics

Model	Accuracy	Precision	Recall	F1	Auc
MLP	0.9648	0.9680	0.9609	0.9645	0.9648
Weighted MLP	0.9200	0.9951	0.8434	0.9130	0.9197
SVM	0.9676	0.9715	0.9633	0.9673	0.9676
Weighted SVM	0.9676	0.9925	0.9055	0.9470	0.9494
Decision Tree	0.9650	0.9696	0.9598	0.9647	0.9650
Weighted DT	0.9350	0.9922	0.8763	0.9307	0.9347
Random Forest	0.9688	0.9733	0.9637	0.9685	0.9687
Weighted RF	0.9488	0.9947	0.9018	0.9460	0.9485

Extra Trees	0.9673	0.9732	0.9607	0.9669	0.9672
Weighted ET	0.9274	0.9981	0.8557	0.9215	0.9271
Logistic Regression	0.9593	0.9581	0.9603	0.9592	0.9593
Weighted LR	0.8856	0.9939	0.7749	0.8708	0.8851

When reviewing the weighted RF confusion matrix (see Figure 1), we note only 21 false negatives and 430 false positives. To provide context for the false positives, flagged apps would be identified as potential malware to users before download. The Android technical team will investigate them and remove them if they are determined to be malware. In reviewing the ROC Curve (see Figure 2), we note an AUC of 0.949. While the AUC for the unweighted RF was higher at 0.969, the number of false negatives was incrementally higher at 115. Again, the cost of false negatives far outweighs false positives in this scenario.

Figure 1

Confusion Matrix of Weighted Random Forest

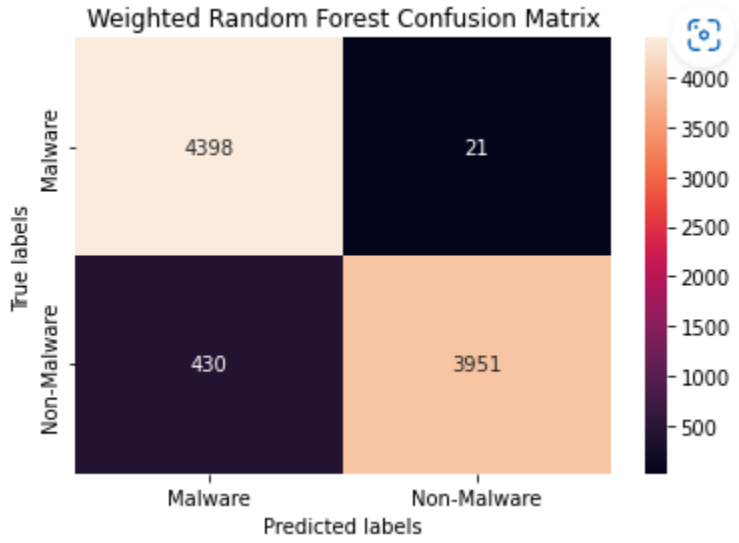
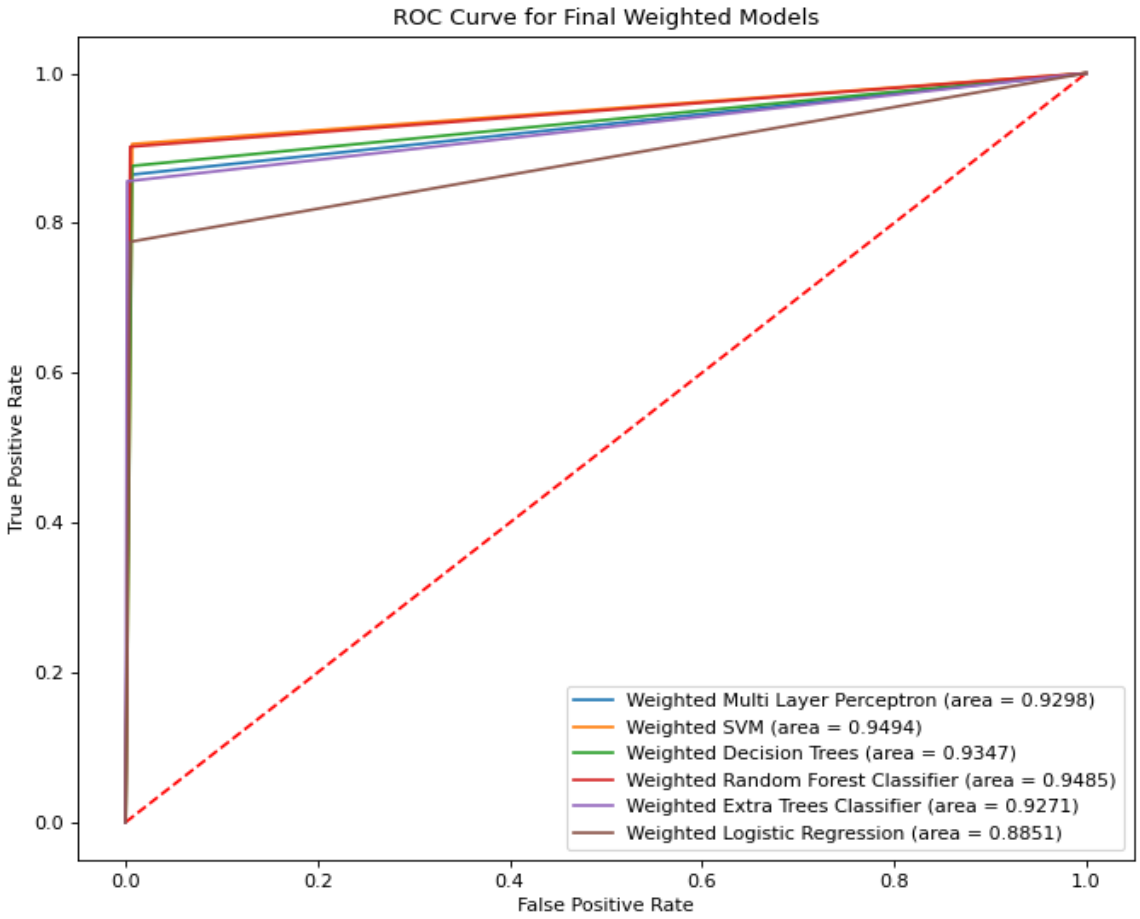


Figure 2

ROC Curve of Weighted Models



Finally, the weighted RF feature importance was calculated and plotted. We note that the two most important features, READ_PHONE_STATE and INSTALL_SHORTCUT, occurred frequently in malware and less frequently in non-malware apps. Conversely, com.google.android.c2dm.permission.RECEIVE predominantly occurred in non-malware apps. RECEIVE_BOOT_COMPLETE occurred more frequently in malware, but also occurred frequently in non-malware. From this preliminary analysis, we can see that the RF-based decision making approach assisted in correctly identifying each class since permissions existed for both classes and in varying combinations.

Strengths and Limitations

One major strength of this study is that all preprocessing and modeling steps are iterable, and results can be easily updated with a new data set of applications and permissions. Unfortunately, the data used in this study is dated since it was collected from 2010-2019 and limited in scope since it examined only 29,332 applications. As of June 2022, the Android ecosystem has more than 2.6 million applications (Ceci, 2022). As malware developers become increasingly sophisticated, trends in permission for malware will continue to change. Additionally, production-ready code for implementation is not currently available but could be a part of future work. Once in production, the model would need to be constantly updated and recalibrated to capture malware shifts and trends.

References

Ceci, L. (2022, July 27). *Google play store: Number of apps 2022*. Statista. Retrieved August 12, 2022, from https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

Kelly, G. (2014, May 24). *Report: 97% of Mobile Malware is on Android. This Is the Easy Way You Stay Safe*. Forbes. Retrieved August 11, 2022, from https://www.forbes.com/sites/gordonkelly/2014/03/24/report-97-of-mobile-malware-is-on-android-this-is-the-easy-way-you-stay-safe/?sh=159e7dba2d4f

Mathur, A., Podila, L. M., Kulkarni, K., Niyaz, Q., & Javaid, A. Y. (2021). *NATICUSdroid: A Malware Detection Framework for Android Using Native and Custom Permissions.* Journal of Information Security and Applications, 58, 102696. Retrieved From https://archive-beta.ics.uci.edu/ml/datasets/naticusdroid+android+permissions+dataset

Statista Research Department. (2022, August 11). *Global Smartphone Market Share From 4th Quarter 2009 to 1st Quarter 2022.* Statista. Retrieved August 11, 2022 from https://www.statista.com/statistics/271496/global-market-share-held-by-smartphone-vendors-since-4th-quarter-2009/

Toulas, B. (2022, July 16). *New Android Malware Apps Installed 10 Million Times From Google Play.* Bleeping Computer. Retrieved August 11, 2022 from https://www.bleepingcomputer.com/news/security/new-android-malware-apps-installed-10-million-times-from-google-play/

Appendix

```
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score,␣
↪RepeatedKFold
```

[ ]:
```
import numpy as np
import os
import matplotlib.pyplot as plt
import matplotlib
import seaborn as sns
import tarfile
from sklearn import metrics
from sklearn.datasets import make_classification
from sklearn.pipeline import make_pipeline, Pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder,␣
↪LabelEncoder,StandardScaler, Normalizer
from sklearn.metrics import confusion_matrix,␣
↪accuracy_score,plot_confusion_matrix, classification_report
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import Perceptron
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.linear_model import SGDClassifier
from mlxtend.plotting import plot_decision_regions
from mlxtend.evaluate import bootstrap_point632_score
import tensorflow as tf
from tensorflow import keras
!pip install scikeras
from scikeras.wrappers import KerasClassifier
import warnings
#### MLXTEND install & import ####
!pip install mlxtend
import mlxtend as ml
warnings.simplefilter(action='ignore', category=FutureWarning)
warnings.simplefilter(action='ignore', category=matplotlib.cbook.mplDeprecation)
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: scikeras in /usr/local/lib/python3.7/dist-
packages (0.9.0)
```

```
Requirement already satisfied: scikit-learn>=1.0.0 in
/usr/local/lib/python3.7/dist-packages (from scikeras) (1.0.2)
Requirement already satisfied: importlib-metadata>=3 in
/usr/local/lib/python3.7/dist-packages (from scikeras) (4.12.0)
Requirement already satisfied: packaging>=0.21 in /usr/local/lib/python3.7/dist-
packages (from scikeras) (21.3)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-
packages (from importlib-metadata>=3->scikeras) (3.8.1)
Requirement already satisfied: typing-extensions>=3.6.4 in
/usr/local/lib/python3.7/dist-packages (from importlib-metadata>=3->scikeras)
(4.1.1)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in
/usr/local/lib/python3.7/dist-packages (from packaging>=0.21->scikeras) (3.0.9)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-
packages (from scikit-learn>=1.0.0->scikeras) (1.1.0)
Requirement already satisfied: numpy>=1.14.6 in /usr/local/lib/python3.7/dist-
packages (from scikit-learn>=1.0.0->scikeras) (1.21.6)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.7/dist-packages (from scikit-learn>=1.0.0->scikeras)
(3.1.0)
Requirement already satisfied: scipy>=1.1.0 in /usr/local/lib/python3.7/dist-
packages (from scikit-learn>=1.0.0->scikeras) (1.7.3)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: mlxtend in /usr/local/lib/python3.7/dist-packages
(0.14.0)
Requirement already satisfied: scikit-learn>=0.18 in
/usr/local/lib/python3.7/dist-packages (from mlxtend) (1.0.2)
Requirement already satisfied: setuptools in /usr/local/lib/python3.7/dist-
packages (from mlxtend) (57.4.0)
Requirement already satisfied: scipy>=0.17 in /usr/local/lib/python3.7/dist-
packages (from mlxtend) (1.7.3)
Requirement already satisfied: matplotlib>=1.5.1 in
/usr/local/lib/python3.7/dist-packages (from mlxtend) (3.2.2)
Requirement already satisfied: pandas>=0.17.1 in /usr/local/lib/python3.7/dist-
packages (from mlxtend) (1.3.5)
Requirement already satisfied: numpy>=1.10.4 in /usr/local/lib/python3.7/dist-
packages (from mlxtend) (1.21.6)
Requirement already satisfied: python-dateutil>=2.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib>=1.5.1->mlxtend) (2.8.2)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib>=1.5.1->mlxtend) (3.0.9)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-
packages (from matplotlib>=1.5.1->mlxtend) (0.11.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.7/dist-packages (from matplotlib>=1.5.1->mlxtend) (1.4.4)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.7/dist-packages (from
```

```
kiwisolver>=1.0.1->matplotlib>=1.5.1->mlxtend) (4.1.1)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.7/dist-
packages (from pandas>=0.17.1->mlxtend) (2022.1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-
packages (from python-dateutil>=2.1->matplotlib>=1.5.1->mlxtend) (1.15.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/usr/local/lib/python3.7/dist-packages (from scikit-learn>=0.18->mlxtend)
(3.1.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist-
packages (from scikit-learn>=0.18->mlxtend) (1.1.0)
```

0.1 Pre-Processing

[ ]:
```
# import the android data
android_data =pd.read_csv('data (1).csv')
android_data.head()
```

[ ]:
```
   android.permission.GET_ACCOUNTS  \
0                                 0
1                                 0
2                                 0
3                                 0
4                                 0

   com.sonyericsson.home.permission.BROADCAST_BADGE  \
0                                                 0
1                                                 0
2                                                 0
3                                                 0
4                                                 0

   android.permission.READ_PROFILE  android.permission.MANAGE_ACCOUNTS  \
0                                 0                                   0
1                                 0                                   0
2                                 0                                   0
3                                 0                                   0
4                                 0                                   0

   android.permission.WRITE_SYNC_SETTINGS  \
0                                       0
1                                       0
2                                       0
3                                       0
4                                       0

   android.permission.READ_EXTERNAL_STORAGE  android.permission.RECEIVE_SMS  \
```

```
0                                          0                                  0
1                                          1                                  0
2                                          0                                  0
3                                          0                                  0
4                                          0                                  0

   com.android.launcher.permission.READ_SETTINGS  \
0                                               0
1                                               0
2                                               0
3                                               0
4                                               0

   android.permission.WRITE_SETTINGS  \
0                                  0
1                                  0
2                                  0
3                                  0
4                                  0

   com.google.android.providers.gsf.permission.READ_GSERVICES  …  \
0                                                  0           …
1                                                  0           …
2                                                  0           …
3                                                  0           …
4                                                  0           …

   com.android.launcher.permission.UNINSTALL_SHORTCUT  \
0                                                  0
1                                                  0
2                                                  0
3                                                  0
4                                                  0

   com.sec.android.iap.permission.BILLING  \
0                                       0
1                                       0
2                                       0
3                                       0
4                                       0

   com.htc.launcher.permission.UPDATE_SHORTCUT  \
0                                            0
1                                            0
2                                            0
3                                            0
4                                            0
```

```
   com.sec.android.provider.badge.permission.WRITE  \
0                                                 0
1                                                 0
2                                                 0
3                                                 0
4                                                 0

   android.permission.ACCESS_NETWORK_STATE  \
0                                        0
1                                        1
2                                        1
3                                        1
4                                        1

   com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE  \
0                                                  0
1                                                  0
2                                                  0
3                                                  0
4                                                  1

   com.huawei.android.launcher.permission.READ_SETTINGS  \
0                                                  0
1                                                  0
2                                                  0
3                                                  0
4                                                  0

   android.permission.READ_SMS  android.permission.PROCESS_INCOMING_CALLS  \
0                            0                                          0
1                            0                                          0
2                            0                                          0
3                            0                                          0
4                            0                                          0

   Result
0       0
1       0
2       0
3       0
4       0

[5 rows x 87 columns]
```

[ ]:
```
android_data.shape
```

[ ]:
```
(29332, 87)
```

[ ]:
```
#Split train/test split
train, test = train_test_split(android_data, test_size=0.30, random_state=42)
```

[ ]:
```
train.describe()
```

[ ]:
```
       android.permission.GET_ACCOUNTS  \
count                     20532.000000
mean                          0.227791
std                           0.419417
min                           0.000000
25%                           0.000000
50%                           0.000000
75%                           0.000000
max                           1.000000

       com.sonyericsson.home.permission.BROADCAST_BADGE  \
count                                      20532.000000
mean                                           0.035408
std                                            0.184814
min                                            0.000000
25%                                            0.000000
50%                                            0.000000
75%                                            0.000000
max                                            1.000000

       android.permission.READ_PROFILE  android.permission.MANAGE_ACCOUNTS  \
count                     20532.000000                        20532.000000
mean                          0.048120                            0.014709
std                           0.214025                            0.120387
min                           0.000000                            0.000000
25%                           0.000000                            0.000000
50%                           0.000000                            0.000000
75%                           0.000000                            0.000000
max                           1.000000                            1.000000

       android.permission.WRITE_SYNC_SETTINGS  \
count                            20532.000000
mean                                 0.005796
std                                  0.075911
min                                  0.000000
25%                                  0.000000
50%                                  0.000000
75%                                  0.000000
max                                  1.000000
```

```
       android.permission.READ_EXTERNAL_STORAGE  \
count                              20532.000000
mean                                   0.171099
std                                    0.376604
min                                    0.000000
25%                                    0.000000
50%                                    0.000000
75%                                    0.000000
max                                    1.000000

       android.permission.RECEIVE_SMS  \
count                    20532.000000
mean                         0.059468
std                          0.236505
min                          0.000000
25%                          0.000000
50%                          0.000000
75%                          0.000000
max                          1.000000

       com.android.launcher.permission.READ_SETTINGS  \
count                                    20532.000000
mean                                         0.014514
std                                          0.119599
min                                          0.000000
25%                                          0.000000
50%                                          0.000000
75%                                          0.000000
max                                          1.000000

       android.permission.WRITE_SETTINGS  \
count                        20532.000000
mean                             0.112800
std                              0.316355
min                              0.000000
25%                              0.000000
50%                              0.000000
75%                              0.000000
max                              1.000000

       com.google.android.providers.gsf.permission.READ_GSERVICES  …  \
count                                         20532.000000           …
mean                                              0.043006           …
std                                               0.202876           …
min                                               0.000000           …
25%                                               0.000000           …
50%                                               0.000000           …
```

```
75%                                                0.000000           …
max                                                1.000000           …

       com.android.launcher.permission.UNINSTALL_SHORTCUT  \
count                                       20532.000000
mean                                            0.030051
std                                             0.170731
min                                             0.000000
25%                                             0.000000
50%                                             0.000000
75%                                             0.000000
max                                             1.000000

       com.sec.android.iap.permission.BILLING  \
count                              20532.000000
mean                                   0.004383
std                                    0.066064
min                                    0.000000
25%                                    0.000000
50%                                    0.000000
75%                                    0.000000
max                                    1.000000

       com.htc.launcher.permission.UPDATE_SHORTCUT  \
count                                    20532.000000
mean                                         0.034678
std                                          0.182966
min                                          0.000000
25%                                          0.000000
50%                                          0.000000
75%                                          0.000000
max                                          1.000000

       com.sec.android.provider.badge.permission.WRITE  \
count                                        20532.000000
mean                                             0.035506
std                                              0.185058
min                                              0.000000
25%                                              0.000000
50%                                              0.000000
75%                                              0.000000
max                                              1.000000

       android.permission.ACCESS_NETWORK_STATE  \
count                               20532.000000
mean                                    0.948422
std                                     0.221179
```

```
min                                    0.000000
25%                                    1.000000
50%                                    1.000000
75%                                    1.000000
max                                    1.000000

       com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE  \
count                                       20532.000000
mean                                            0.024109
std                                             0.153390
min                                             0.000000
25%                                             0.000000
50%                                             0.000000
75%                                             0.000000
max                                             1.000000

       com.huawei.android.launcher.permission.READ_SETTINGS  \
count                                       20532.000000
mean                                            0.024255
std                                             0.153843
min                                             0.000000
25%                                             0.000000
50%                                             0.000000
75%                                             0.000000
max                                             1.000000

       android.permission.READ_SMS  android.permission.PROCESS_INCOMING_CALLS  \
count                 20532.000000                                20532.000000
mean                      0.051919                                    0.004042
std                       0.221869                                    0.063453
min                       0.000000                                    0.000000
25%                       0.000000                                    0.000000
50%                       0.000000                                    0.000000
75%                       0.000000                                    0.000000
max                       1.000000                                    1.000000

             Result
count  20532.000000
mean       0.502581
std        0.500006
min        0.000000
25%        0.000000
50%        1.000000
75%        1.000000
max        1.000000

[8 rows x 87 columns]
```

[ ]:
```
# Evaluate means of binary features to explore frequency
means = train.mean().sort_values(ascending=False)
means.head(15)
```

[ ]:
```
custom = train.filter(regex = "^com|Result|^me").columns
custom
```

[ ]:
```
Index(['com.sonyericsson.home.permission.BROADCAST_BADGE',
       'com.android.launcher.permission.READ_SETTINGS',
       'com.google.android.providers.gsf.permission.READ_GSERVICES',
       'com.huawei.android.launcher.permission.CHANGE_BADGE',
       'com.oppo.launcher.permission.READ_SETTINGS',
       'com.android.launcher.permission.INSTALL_SHORTCUT',
'com.samsung.android.providers.context.permission.WRITE_USE_APP_FEATURE_SURVEY',
       'com.majeur.launcher.permission.UPDATE_BADGE',
       'com.htc.launcher.permission.READ_SETTINGS',
       'com.anddoes.launcher.permission.UPDATE_COUNT',
       'com.android.alarm.permission.SET_ALARM',
       'com.google.android.c2dm.permission.RECEIVE',
       'com.sonymobile.home.permission.PROVIDER_INSERT_BADGE',
       'com.sec.android.provider.badge.permission.READ',
       'com.huawei.android.launcher.permission.WRITE_SETTINGS',
       'com.oppo.launcher.permission.WRITE_SETTINGS',
       'me.everything.badger.permission.BADGE_COUNT_WRITE',
       'com.google.android.gms.permission.ACTIVITY_RECOGNITION',
       'com.amazon.device.messaging.permission.RECEIVE',
       'me.everything.badger.permission.BADGE_COUNT_READ',
       'com.android.vending.BILLING', 'com.android.vending.CHECK_LICENSE',
       'com.android.launcher.permission.UNINSTALL_SHORTCUT',
       'com.sec.android.iap.permission.BILLING',
       'com.htc.launcher.permission.UPDATE_SHORTCUT',
       'com.sec.android.provider.badge.permission.WRITE',
       'com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE',
       'com.huawei.android.launcher.permission.READ_SETTINGS', 'Result'],
      dtype='object')
```

[ ]:
```
# Create a numeric dataframe
custom_df = pd.DataFrame()

# Import custom permission features from train (contain com. in name)
custom_df[custom] = train[custom]
```

[ ]:
```
# Segregate observations identified as malware
malware_custom = custom_df.loc[custom_df['Result'] == 1]
# Find the sum/frequency of each permissions occurence for malware␣
 ↪(0=permission not used in malware observations)
malware_custom.sum().sort_values(ascending=False)
```

[ ]:
```
Result
10319
com.android.launcher.permission.INSTALL_SHORTCUT
4090
com.android.launcher.permission.UNINSTALL_SHORTCUT
484
com.android.vending.BILLING
469
com.android.vending.CHECK_LICENSE
248
com.android.launcher.permission.READ_SETTINGS
219
com.htc.launcher.permission.READ_SETTINGS
157
com.google.android.c2dm.permission.RECEIVE
146
com.google.android.providers.gsf.permission.READ_GSERVICES
40
com.google.android.gms.permission.ACTIVITY_RECOGNITION
21
com.android.alarm.permission.SET_ALARM
8
com.huawei.android.launcher.permission.READ_SETTINGS
4
com.huawei.android.launcher.permission.WRITE_SETTINGS
4
com.sec.android.provider.badge.permission.READ
1
com.amazon.device.messaging.permission.RECEIVE
1
com.sec.android.iap.permission.BILLING
1
com.htc.launcher.permission.UPDATE_SHORTCUT
1
com.sec.android.provider.badge.permission.WRITE
1
com.sonyericsson.home.permission.BROADCAST_BADGE
1
com.sonymobile.home.permission.PROVIDER_INSERT_BADGE
0
com.oppo.launcher.permission.WRITE_SETTINGS
0
me.everything.badger.permission.BADGE_COUNT_WRITE
0
com.anddoes.launcher.permission.UPDATE_COUNT
0
me.everything.badger.permission.BADGE_COUNT_READ
```

```
0
com.majeur.launcher.permission.UPDATE_BADGE
0
com.samsung.android.providers.context.permission.WRITE_USE_APP_FEATURE_SURVEY
0
com.oppo.launcher.permission.READ_SETTINGS
0
com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE
0
com.huawei.android.launcher.permission.CHANGE_BADGE
0
dtype: int64
```

[ ]:
```
# Segregate observations identified as benign
benign_custom = custom_df.loc[custom_df['Result'] == 0]
# Find the sum/frequency of each permissions occurence for benign (0=permission␣
 ↪not used in benign observations)
benign_custom.sum().sort_values(ascending=False)
```

[ ]:
```
com.google.android.c2dm.permission.RECEIVE
4170
com.android.vending.BILLING
2252
com.google.android.providers.gsf.permission.READ_GSERVICES
843
com.sec.android.provider.badge.permission.WRITE
728
com.sec.android.provider.badge.permission.READ
727
com.sonyericsson.home.permission.BROADCAST_BADGE
726
com.htc.launcher.permission.READ_SETTINGS
724
com.htc.launcher.permission.UPDATE_SHORTCUT
711
com.anddoes.launcher.permission.UPDATE_COUNT
699
com.majeur.launcher.permission.UPDATE_BADGE
693
com.google.android.finsky.permission.BIND_GET_INSTALL_REFERRER_SERVICE
495
com.huawei.android.launcher.permission.READ_SETTINGS
494
com.huawei.android.launcher.permission.CHANGE_BADGE
488
com.huawei.android.launcher.permission.WRITE_SETTINGS
487
```

```
com.sonymobile.home.permission.PROVIDER_INSERT_BADGE
482
com.oppo.launcher.permission.READ_SETTINGS
379
com.oppo.launcher.permission.WRITE_SETTINGS
374
com.android.vending.CHECK_LICENSE
317
com.android.launcher.permission.INSTALL_SHORTCUT
298
me.everything.badger.permission.BADGE_COUNT_WRITE
295
me.everything.badger.permission.BADGE_COUNT_READ
295
com.android.launcher.permission.UNINSTALL_SHORTCUT
133
com.google.android.gms.permission.ACTIVITY_RECOGNITION
98
com.android.alarm.permission.SET_ALARM
97
com.samsung.android.providers.context.permission.WRITE_USE_APP_FEATURE_SURVEY
92
com.sec.android.iap.permission.BILLING
89
com.android.launcher.permission.READ_SETTINGS
79
com.amazon.device.messaging.permission.RECEIVE
71
Result
0
dtype: int64
```

[ ]:
```
#Examine distributions of mostly custom permissions
histlist = custom_df.hist(figsize = (40,50))
```

[ ]:
```
# Segregate native filers based on names that start with "android"
native = train.filter(regex = "^android|Result").columns
native
```

[ ]:
```
Index(['android.permission.GET_ACCOUNTS', 'android.permission.READ_PROFILE',
       'android.permission.MANAGE_ACCOUNTS',
```

```
'android.permission.WRITE_SYNC_SETTINGS',
'android.permission.READ_EXTERNAL_STORAGE',
'android.permission.RECEIVE_SMS', 'android.permission.WRITE_SETTINGS',
'android.permission.DOWNLOAD_WITHOUT_NOTIFICATION',
'android.permission.GET_TASKS',
'android.permission.WRITE_EXTERNAL_STORAGE',
'android.permission.RECORD_AUDIO',
'android.permission.CHANGE_NETWORK_STATE',
'android.permission.android.permission.READ_PHONE_STATE',
'android.permission.CALL_PHONE', 'android.permission.WRITE_CONTACTS',
'android.permission.READ_PHONE_STATE',
'android.permission.MODIFY_AUDIO_SETTINGS',
'android.permission.ACCESS_LOCATION_EXTRA_COMMANDS',
'android.permission.INTERNET',
'android.permission.MOUNT_UNMOUNT_FILESYSTEMS',
'android.permission.AUTHENTICATE_ACCOUNTS',
'android.permission.ACCESS_WIFI_STATE', 'android.permission.FLASHLIGHT',
'android.permission.READ_APP_BADGE',
'android.permission.USE_CREDENTIALS',
'android.permission.CHANGE_CONFIGURATION',
'android.permission.READ_SYNC_SETTINGS',
'android.permission.BROADCAST_STICKY',
'android.permission.KILL_BACKGROUND_PROCESSES',
'android.permission.WRITE_CALENDAR', 'android.permission.SEND_SMS',
'android.permission.REQUEST_INSTALL_PACKAGES',
'android.permission.SET_WALLPAPER_HINTS',
'android.permission.SET_WALLPAPER',
'android.permission.RESTART_PACKAGES',
'android.permission.ACCESS_MOCK_LOCATION',
'android.permission.ACCESS_COARSE_LOCATION',
'android.permission.READ_LOGS',
'android.permission.SYSTEM_ALERT_WINDOW',
'android.permission.DISABLE_KEYGUARD',
'android.permission.USE_FINGERPRINT',
'android.permission.CHANGE_WIFI_STATE',
'android.permission.READ_CONTACTS', 'android.permission.READ_CALENDAR',
'android.permission.RECEIVE_BOOT_COMPLETED',
'android.permission.WAKE_LOCK',
'android.permission.ACCESS_FINE_LOCATION',
'android.permission.BLUETOOTH', 'android.permission.CAMERA',
'android.permission.FOREGROUND_SERVICE',
'android.permission.BLUETOOTH_ADMIN', 'android.permission.VIBRATE',
'android.permission.NFC', 'android.permission.RECEIVE_USER_PRESENT',
'android.permission.CLEAR_APP_CACHE',
'android.permission.ACCESS_NETWORK_STATE',
'android.permission.READ_SMS',
'android.permission.PROCESS_INCOMING_CALLS', 'Result'],
```

```
      dtype='object')
```

[ ]:
```
# Create a numeric dataframe
native_df = pd.DataFrame()

# Import customer permission features from X_train (contain com. in name)
native_df[native] = train[native]
```

[ ]:
```
# Segregate observations identified as malware
malware_native = native_df.loc[native_df['Result'] == 1]
# Find the sum/frequency of each permission's ocurrence for malware
 ↪(0=permission not used in malware observations)
malware_native.sum().sort_values(ascending=False)
```

[ ]:
```
Result                                          10319
android.permission.INTERNET                      9996
android.permission.ACCESS_NETWORK_STATE          9882
android.permission.READ_PHONE_STATE              9721
android.permission.WRITE_EXTERNAL_STORAGE        7593
android.permission.RECEIVE_BOOT_COMPLETED        7242
android.permission.ACCESS_COARSE_LOCATION        7018
android.permission.ACCESS_FINE_LOCATION          6709
android.permission.ACCESS_WIFI_STATE             6460
android.permission.WAKE_LOCK                     4750
android.permission.GET_TASKS                     4140
android.permission.VIBRATE                       3898
android.permission.SYSTEM_ALERT_WINDOW           3275
android.permission.GET_ACCOUNTS                  2936
android.permission.READ_CONTACTS                 2067
android.permission.WRITE_SETTINGS                1879
android.permission.CAMERA                        1380
android.permission.CHANGE_WIFI_STATE             1345
android.permission.SEND_SMS                      1195
android.permission.WRITE_CONTACTS                1125
android.permission.RECEIVE_SMS                   1054
android.permission.CALL_PHONE                    1051
android.permission.MOUNT_UNMOUNT_FILESYSTEMS      992
android.permission.READ_SMS                       953
android.permission.READ_LOGS                      950
android.permission.READ_PROFILE                   913
android.permission.READ_EXTERNAL_STORAGE          909
android.permission.RESTART_PACKAGES               907
android.permission.SET_WALLPAPER                  716
android.permission.ACCESS_LOCATION_EXTRA_COMMANDS 669
android.permission.KILL_BACKGROUND_PROCESSES      583
android.permission.CHANGE_NETWORK_STATE           494
android.permission.RECORD_AUDIO                   405
```

```
android.permission.FLASHLIGHT                                  378
android.permission.CHANGE_CONFIGURATION                        353
android.permission.DISABLE_KEYGUARD                            342
android.permission.MODIFY_AUDIO_SETTINGS                       304
android.permission.BLUETOOTH                                   255
android.permission.BLUETOOTH_ADMIN                             209
android.permission.DOWNLOAD_WITHOUT_NOTIFICATION               137
android.permission.ACCESS_MOCK_LOCATION                        131
android.permission.RECEIVE_USER_PRESENT                        130
android.permission.MANAGE_ACCOUNTS                             116
android.permission.USE_CREDENTIALS                             101
android.permission.AUTHENTICATE_ACCOUNTS                        94
android.permission.PROCESS_INCOMING_CALLS                       80
android.permission.BROADCAST_STICKY                             73
android.permission.SET_WALLPAPER_HINTS                          64
android.permission.READ_CALENDAR                                56
android.permission.WRITE_CALENDAR                               52
android.permission.CLEAR_APP_CACHE                              28
android.permission.READ_SYNC_SETTINGS                           15
android.permission.WRITE_SYNC_SETTINGS                          14
android.permission.NFC                                           3
android.permission.android.permission.READ_PHONE_STATE           0
android.permission.FOREGROUND_SERVICE                            0
android.permission.REQUEST_INSTALL_PACKAGES                      0
android.permission.READ_APP_BADGE                                0
android.permission.USE_FINGERPRINT                               0
dtype: int64
```

[ ]:
```
# Segregate observations identified as benign
benign_native = native_df.loc[native_df['Result'] == 0]
# Find the sum/frequency of each permissions occurence for benign (0=permission␣
 ↪not used in benign observations)
benign_native.sum().sort_values(ascending=False)
```

[ ]:
```
android.permission.INTERNET                                  10014
android.permission.ACCESS_NETWORK_STATE                       9591
android.permission.WRITE_EXTERNAL_STORAGE                     6210
android.permission.WAKE_LOCK                                  5154
android.permission.ACCESS_WIFI_STATE                          4303
android.permission.VIBRATE                                    3336
android.permission.READ_EXTERNAL_STORAGE                      2604
android.permission.ACCESS_COARSE_LOCATION                     2513
android.permission.ACCESS_FINE_LOCATION                       2455
android.permission.READ_PHONE_STATE                           2366
android.permission.RECEIVE_BOOT_COMPLETED                     2131
android.permission.CAMERA                                     1753
android.permission.GET_ACCOUNTS                               1741
```

```
android.permission.RECORD_AUDIO                          816
android.permission.GET_TASKS                             711
android.permission.READ_CONTACTS                         617
android.permission.SYSTEM_ALERT_WINDOW                   613
android.permission.BLUETOOTH                             604
android.permission.CALL_PHONE                            525
android.permission.USE_CREDENTIALS                       476
android.permission.BLUETOOTH_ADMIN                       443
android.permission.WRITE_SETTINGS                        437
android.permission.CHANGE_WIFI_STATE                     425
android.permission.FLASHLIGHT                            417
android.permission.READ_APP_BADGE                        369
android.permission.MODIFY_AUDIO_SETTINGS                 344
android.permission.SET_WALLPAPER                         290
android.permission.CHANGE_NETWORK_STATE                  285
android.permission.WRITE_CONTACTS                        284
android.permission.WRITE_CALENDAR                        230
android.permission.READ_CALENDAR                         229
android.permission.DISABLE_KEYGUARD                      225
android.permission.MOUNT_UNMOUNT_FILESYSTEMS             215
android.permission.READ_LOGS                             212
android.permission.SEND_SMS                              198
android.permission.USE_FINGERPRINT                       197
android.permission.MANAGE_ACCOUNTS                       186
android.permission.ACCESS_LOCATION_EXTRA_COMMANDS        174
android.permission.RECEIVE_SMS                           167
android.permission.RECEIVE_USER_PRESENT                  155
android.permission.REQUEST_INSTALL_PACKAGES              149
android.permission.NFC                                   135
android.permission.KILL_BACKGROUND_PROCESSES             134
android.permission.AUTHENTICATE_ACCOUNTS                 131
android.permission.BROADCAST_STICKY                      117
android.permission.READ_SMS                              113
android.permission.ACCESS_MOCK_LOCATION                  108
android.permission.WRITE_SYNC_SETTINGS                   105
android.permission.SET_WALLPAPER_HINTS                    91
android.permission.READ_SYNC_SETTINGS                     80
android.permission.DOWNLOAD_WITHOUT_NOTIFICATION          78
android.permission.RESTART_PACKAGES                       76
android.permission.READ_PROFILE                           75
android.permission.CHANGE_CONFIGURATION                   74
android.permission.android.permission.READ_PHONE_STATE    73
android.permission.FOREGROUND_SERVICE                     70
android.permission.CLEAR_APP_CACHE                        67
android.permission.PROCESS_INCOMING_CALLS                  3
Result                                                     0
dtype: int64
```

[ ]:
```
#Examine distributions of native permissions
histlist = native_df.hist(figsize = (40,50))
```

[ ]:
```
#storing the variance and name of variables

variance = train.var()
columns = train.columns
```

```
#saving the names of variables having variance more than a threshold value

variable = [ ]

for i in range(0,len(variance)):
    if variance[i]>=0.006: #setting the threshold as 1%
        variable.append(columns[i])
```

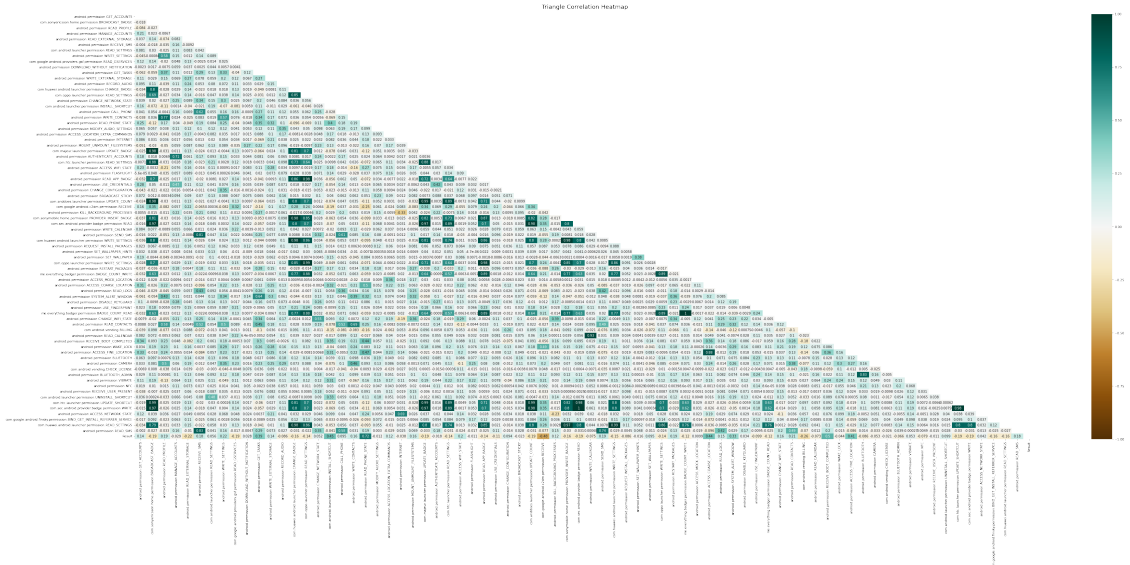
[ ]:
```
len(variable)
```

[ ]: 76

[ ]:
```
# Retain only the non-near zero variance features in train
train_red = train[variable]
# Retain only the non-near zero variance features in test
test_red = test[variable]
train_red.shape
```

[ ]: (20532, 76)

[ ]:
```
plt.figure(figsize=(62, 24))
# define the mask to set the values in the upper triangle to True
mask = np.triu(np.ones_like(train_red.corr(), dtype=bool))
heatmap = sns.heatmap(train_red.corr(), mask=mask, vmin=-1, vmax=1, annot=True,␣
 ↪cmap='BrBG')
heatmap.set_title('Triangle Correlation Heatmap', fontdict={'fontsize':18},␣
 ↪pad=16);
```



[ ]:
```
# Create correlation matrix
corr_matrix = train_red.corr().abs()

# Select upper triangle of correlation matrix
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))

# Find features with correlation greater than 0.95
to_drop = [column for column in upper.columns if any(upper[column] > 0.80)]

# Display features to drop
print(to_drop)
```

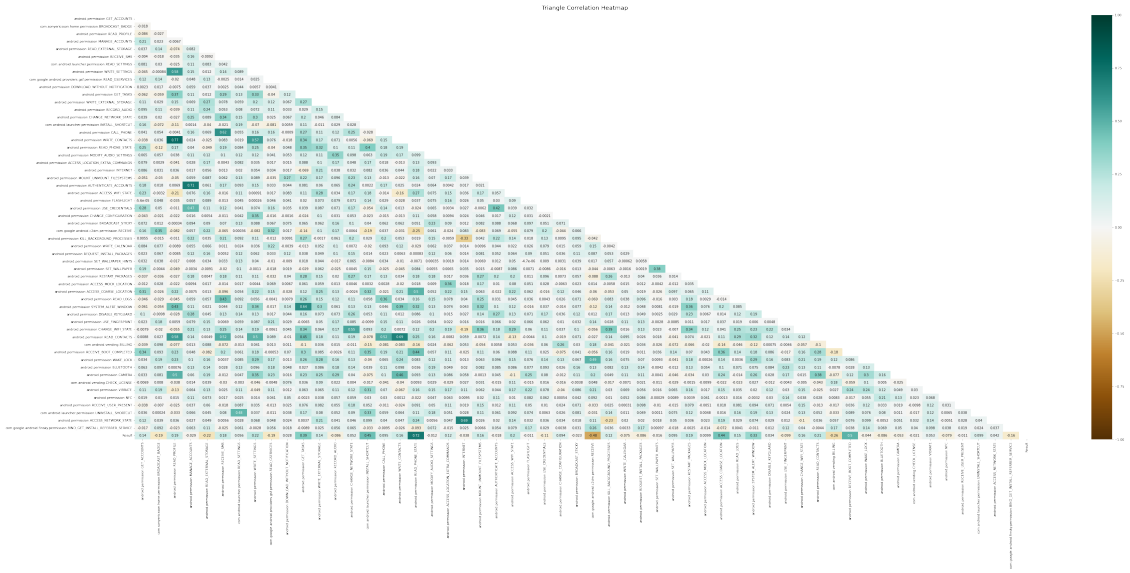
```
['com.huawei.android.launcher.permission.CHANGE_BADGE',
'com.oppo.launcher.permission.READ_SETTINGS',
'com.majeur.launcher.permission.UPDATE_BADGE',
'com.htc.launcher.permission.READ_SETTINGS',
'android.permission.READ_APP_BADGE',
'com.anddoes.launcher.permission.UPDATE_COUNT',
'com.sonymobile.home.permission.PROVIDER_INSERT_BADGE',
'com.sec.android.provider.badge.permission.READ', 'android.permission.SEND_SMS',
'com.huawei.android.launcher.permission.WRITE_SETTINGS',
'com.oppo.launcher.permission.WRITE_SETTINGS',
'me.everything.badger.permission.BADGE_COUNT_WRITE',
'me.everything.badger.permission.BADGE_COUNT_READ',
'android.permission.READ_CALENDAR', 'android.permission.ACCESS_FINE_LOCATION',
'android.permission.BLUETOOTH_ADMIN',
'com.htc.launcher.permission.UPDATE_SHORTCUT',
'com.sec.android.provider.badge.permission.WRITE',
'com.huawei.android.launcher.permission.READ_SETTINGS',
'android.permission.READ_SMS']
```

[ ]:
```
# Drop highly correlated features
train_red.drop(to_drop, axis=1, inplace=True)
test_red.drop(to_drop, axis=1, inplace=True)
```

```
/usr/local/lib/python3.7/dist-packages/pandas/core/frame.py:4913:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  errors=errors,
```

[ ]:
```
# Add a second correlation heat map after features have been reduced to show␣
 ↪that issues have been reconciled.
plt.figure(figsize=(62, 24))
# define the mask to set the values in the upper triangle to True
```

```
mask = np.triu(np.ones_like(train_red.corr(), dtype=bool))
heatmap = sns.heatmap(train_red.corr(), mask=mask, vmin=-1, vmax=1, annot=True,␣
↪cmap='BrBG')
heatmap.set_title('Triangle Correlation Heatmap', fontdict={'fontsize':18},␣
↪pad=16);
```



[ ]:
```
#Split our Train / test sets for X and y

#Define X
select = [x for x in train_red.columns if x != "Result"]
X_train = train_red.loc[:, select]
X_test = test_red.loc[:,select]
#Define y
y_train = train_red['Result']
y_test = test_red['Result']
```

[ ]:
```
k_vals = range(1,56)
results = []
for k in k_vals:
    pipe = make_pipeline(SelectKBest(chi2, k=k), Perceptron(class_weight =␣
↪'balanced'))
    model = pipe.fit(X_train, y_train)
    kbest = SelectKBest(chi2, k=k)
    kbest.fit_transform(X_train, y_train)

    train_pred = model.predict(X_train)
    train_acc = accuracy_score(y_train, train_pred)
    test_pred = model.predict(X_test)
```

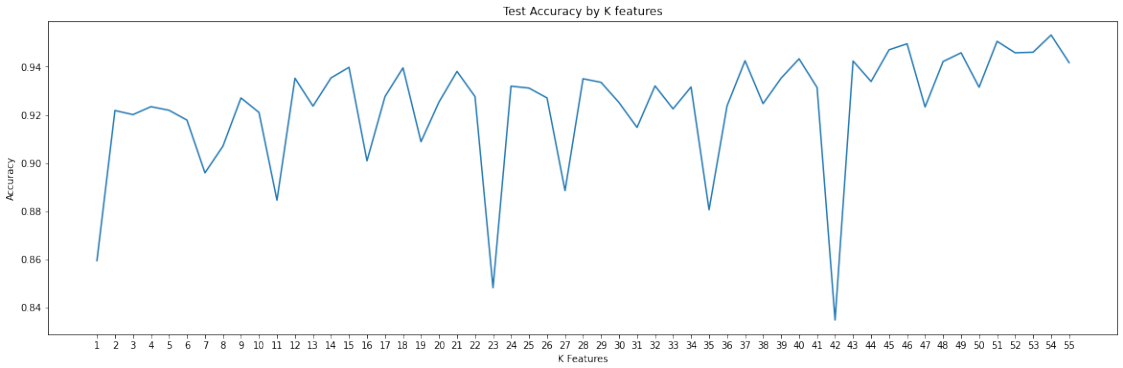
```
    test_acc = accuracy_score(y_test, test_pred)

    results.append({'k-value': k, 'Training Accuracy': train_acc, 'Test␣
↪Accuracy':test_acc})
```

[ ]:
```
results_df = pd.DataFrame(results)
#print(results_df.to_string())
print(results_df.sort_values('Test Accuracy', ascending = False).to_string())
```

```
    k-value  Training Accuracy  Test Accuracy
53       54           0.950468       0.953182
50       51           0.946961       0.950568
45       46           0.945110       0.949545
44       45           0.942383       0.947045
52       53           0.943698       0.946023
51       52           0.942042       0.945795
48       49           0.942383       0.945795
39       40           0.936100       0.943295
36       37           0.938535       0.942500
42       43           0.938243       0.942386
47       48           0.940873       0.942159
54       55           0.936782       0.941705
14       15           0.934249       0.939773
17       18           0.931960       0.939545
20       21           0.931570       0.938068
13       14           0.930060       0.935341
11       12           0.930012       0.935227
38       39           0.930840       0.935227
27       28           0.930401       0.935000
43       44           0.928989       0.933864
28       29           0.929671       0.933523
31       32           0.927138       0.932045
23       24           0.927041       0.931932
33       34           0.926651       0.931591
49       50           0.926456       0.931477
40       41           0.928112       0.931364
24       25           0.926359       0.931136
21       22           0.919930       0.927614
16       17           0.920368       0.927614
25       26           0.921635       0.927045
8         9           0.920660       0.927045
19       20           0.920466       0.925341
29       30           0.920612       0.925114
37       38           0.923826       0.924659
35       36           0.920758       0.923750
12       13           0.915303       0.923636
3         4           0.915936       0.923409
```

46	47	0.919930	0.923295
32	33	0.916861	0.922500
4	5	0.914085	0.921932
1	2	0.916326	0.921818
9	10	0.914475	0.921023
2	3	0.914231	0.920114
5	6	0.911358	0.917841
30	31	0.910871	0.914773
18	19	0.900692	0.908864
7	8	0.902396	0.907045
15	16	0.894117	0.900909
6	7	0.891146	0.895909
26	27	0.884619	0.888523
10	11	0.878385	0.884545
34	35	0.878141	0.880568
0	1	0.855640	0.859432
22	23	0.841954	0.848182
41	42	0.830996	0.834773

[ ]:
```
#Plot it here
x= results_df['k-value']
plt.figure(figsize=(20,6))
plt.plot(results_df['Test Accuracy'])
default_x_ticks = range(len(x))

plt.xticks(default_x_ticks, x)

plt.xlabel('K Features')
plt.ylabel('Accuracy')

plt.title('Test Accuracy by K features')
plt.show()
```



0.2 Multi Layer Perceptron

0.2.1 Optimizing Learning Rate | Comparing Adam & SGD Optimizers

[ ]:
```
nn_results = []

learn_rate = [0.00001,0.0001,0.0005,0.001,0.005,0.01,0.05,0.1,0.5,1,4]

mod = keras.models.Sequential([
    keras.Input(shape=55),
    keras.layers.Dense(50, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(200, activation='relu'),
    keras.layers.Dense(1, activation = 'sigmoid')
])

mod_adam = keras.models.Sequential([
    keras.Input(shape=55),
    keras.layers.Dense(50, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(200, activation='relu'),
    keras.layers.Dense(1, activation = 'sigmoid')
])

for l in learn_rate:
  # compiling the model using SGD optimizer
  mod.compile(loss = 'binary_crossentropy', optimizer=keras.optimizers.
 ↪SGD(learning_rate= l),
              metrics=['accuracy'])
  # compiling the model using Adam optimizer
  mod_adam.compile(loss = 'binary_crossentropy', optimizer=keras.optimizers.
 ↪Adam(learning_rate= l),
              metrics=['accuracy'])

  # To use Sklearn's Cross_val_score, the model must rewrapped in sklearn format
  estimator_1 = KerasClassifier(build_fn = mod, epochs = 10)
  estimator_2 = KerasClassifier(build_fn = mod_adam, epochs = 10)

  # Now that the model has been wrapped, the estimator created can be used
  cv1 = cross_val_score(estimator_1, X_train, y_train, cv=10, n_jobs=2, scoring␣
 ↪= 'precision')
  cv2 = cross_val_score(estimator_2, X_train, y_train, cv=10, n_jobs=2, scoring␣
 ↪= 'precision')


  Train_acc_a = accuracy_score(y_train, np.round(mod_adam.predict(X_train)))
  Test_acc_a = accuracy_score(y_test, np.round(mod_adam.predict(X_test)))
```

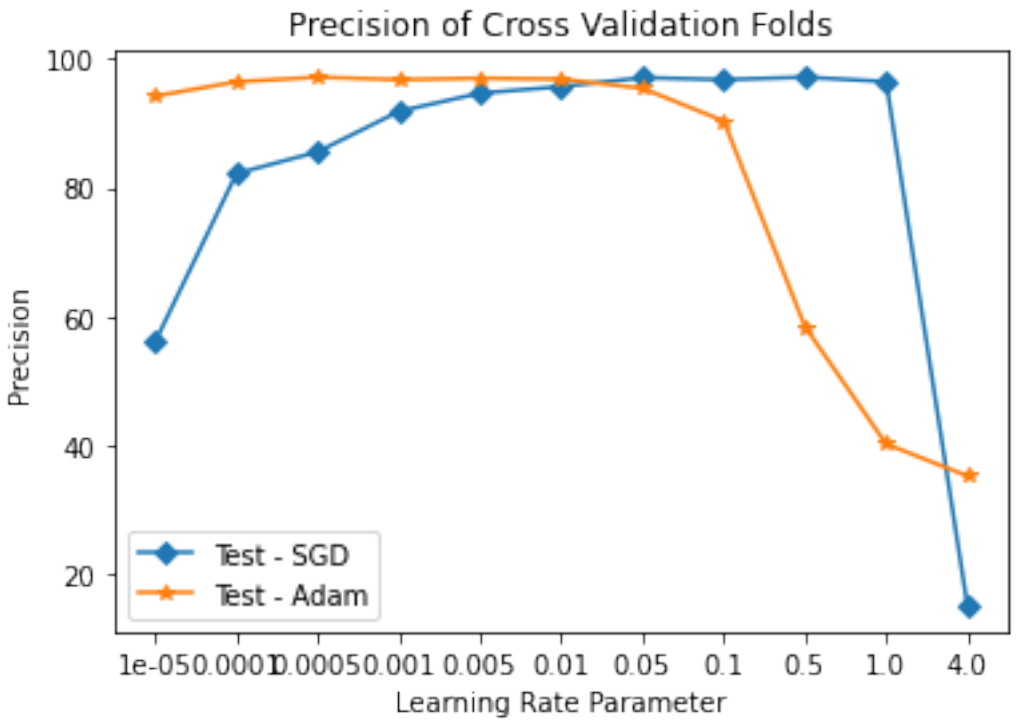
```
    nn_results.append({'Learning Rate': l, 'SGD Precision': cv1.mean(),
                       'Adam Precision': cv2.mean()})
```

[ ]:
```
nn_results_df = pd.DataFrame(nn_results)
nn_results_df
```

[ ]:

	Learning Rate	SGD Precision	Adam Precision
0	0.00001	0.560733	0.942411
1	0.00010	0.821958	0.964665
2	0.00050	0.855783	0.971779
3	0.00100	0.918621	0.967966
4	0.00500	0.947014	0.970184
5	0.01000	0.957099	0.968808
6	0.05000	0.971048	0.954459
7	0.10000	0.967979	0.903718
8	0.50000	0.971790	0.584105
9	1.00000	0.964944	0.401755
10	4.00000	0.150779	0.352047

[ ]:
```
tick = range(len(nn_results_df['Learning Rate']))
plt.plot(nn_results_df['SGD Precision']*100, marker = 'D', label = 'Test - SGD')
plt.plot(nn_results_df['Adam Precision']*100, marker = '*', label = 'Test -␣
 ↪Adam')
plt.xlabel('Learning Rate Parameter')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.legend(loc = 'top left')
plt.xticks(ticks=tick, labels=nn_results_df['Learning Rate'])
plt.show()
```



The Adaptive Movement Estimation Algorithm (Adam) offered the best performance with an accuracy of over 96% at a learning rate of 0.005.

Adam is an extension of gradient descent that builds on the ideas of AdaGrad. Adam automatically updates the learning rate by exponentially decreasing the moving average.

0.2.2 Tuning Epsilon for Adam Classifier

[ ]:
```
epsilons = [0.000005, 0.00001, 0.00005, 0.0001, 0.0005,0.001,0.01,0.1,1]
eps_results = []

modad = keras.models.Sequential([
    keras.Input(shape=55),
    keras.layers.Dense(50, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(200, activation='relu'),
    keras.layers.Dense(1, activation = 'sigmoid')
])
for e in epsilons:
  modad.compile(loss = 'binary_crossentropy', optimizer=keras.optimizers.
 ↪Adam(learning_rate = 0.0005, epsilon = e),metrics=['accuracy'])
  estimator_a = KerasClassifier(build_fn = modad, epochs = 10)
```

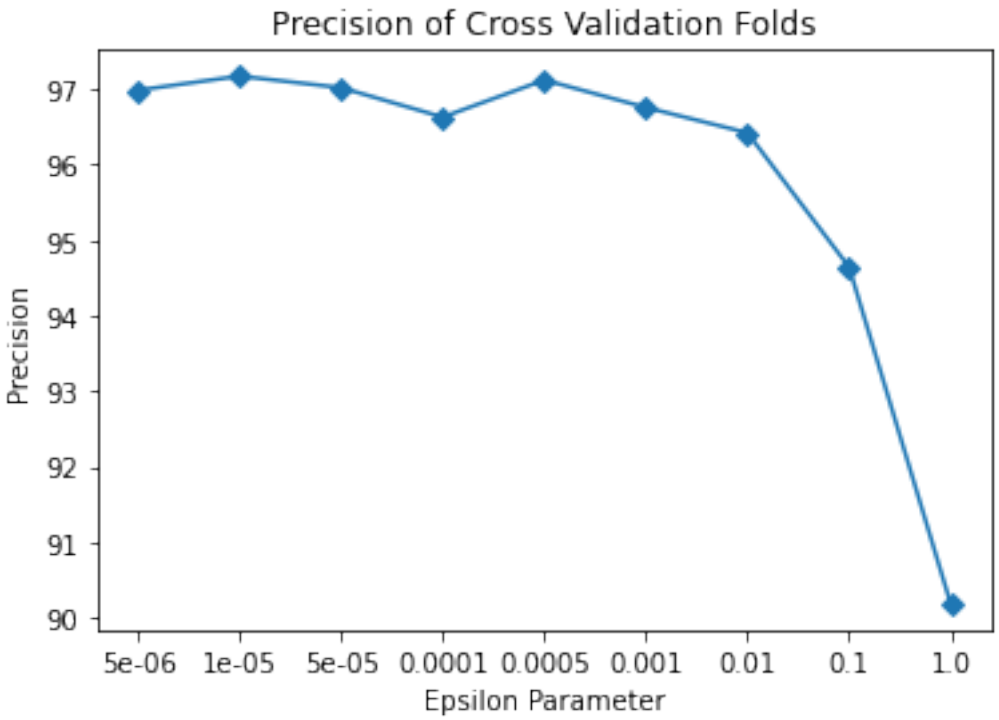
```
 cva = cross_val_score(estimator_a, X_train, y_train, cv=10, n_jobs=2, scoring␣
↪= 'precision')
 eps_results.append({'Epsilon': e, 'Precision': cva.mean()})
```

[ ]:
```
eps_results_df = pd.DataFrame(eps_results)
eps_results_df
```

[ ]:

	Epsilon	Precision
0	0.000005	0.969864
1	0.000010	0.971741
2	0.000050	0.970198
3	0.000100	0.966301
4	0.000500	0.971218
5	0.001000	0.967578
6	0.010000	0.964222
7	0.100000	0.946426
8	1.000000	0.901827

[ ]:
```
tick = range(len(eps_results_df['Epsilon']))
plt.plot(eps_results_df['Precision']*100, marker = 'D')
plt.xlabel('Epsilon Parameter')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.xticks(ticks=tick, labels=eps_results_df['Epsilon'])
plt.show()
```



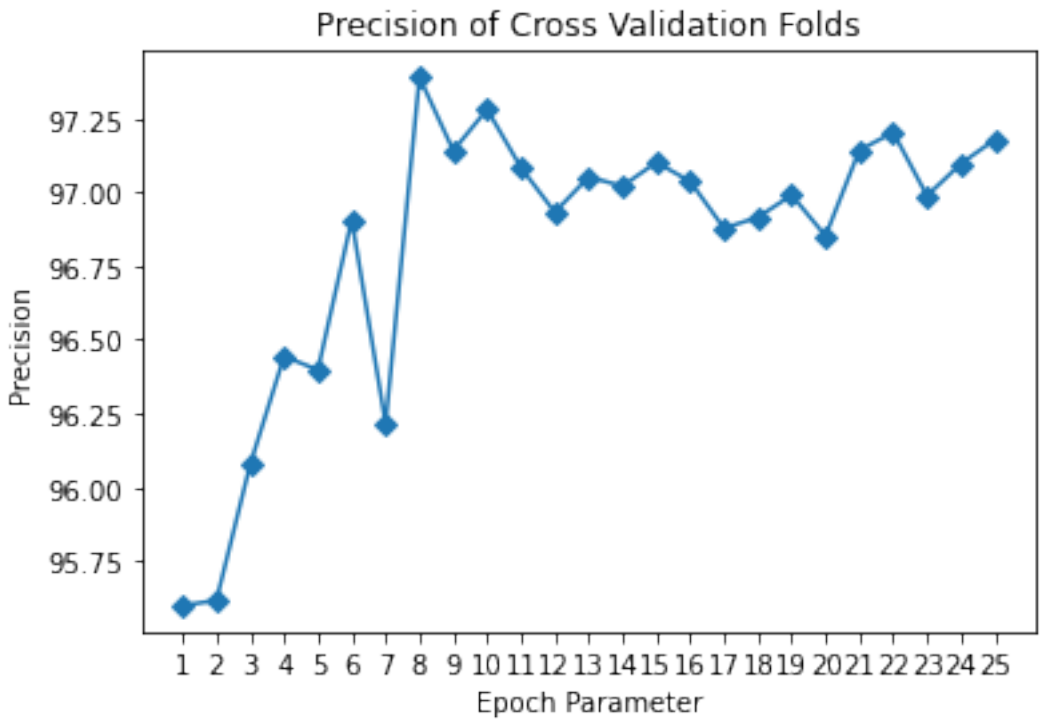
The epsilon parameter for adam of 0.0005 offered the best accuracy over the cross validation. Keras cites epsilon as being a hyperparameter for optimizing numerical stability.

0.2.3 Tuning the Epochs

[ ]:
```
epoch = range(1,26)

epoch_results = []

epochMLP = keras.models.Sequential([
    keras.Input(shape=55),
    keras.layers.Dense(50, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(200, activation='relu'),
    keras.layers.Dense(1, activation = 'sigmoid')
])


for e in epoch:
  epochMLP.compile(loss = 'binary_crossentropy', optimizer=keras.optimizers.
 ↪Adam(learning_rate = 0.0005, epsilon = 0.001),metrics=['accuracy'])
  estimator_ep = KerasClassifier(build_fn = epochMLP, epochs = e)

  cva = cross_val_score(estimator_ep, X_train, y_train, cv=10, n_jobs=2,␣
 ↪scoring = 'precision')
  epoch_results.append({'Epochs': e, 'Precision': cva.mean()})
```

[ ]:
```
epoch_results_df = pd.DataFrame(epoch_results)
```

[ ]:
```
tick = range(len(epoch_results_df['Epochs']))
plt.plot(epoch_results_df['Precision']*100, marker = 'D')
plt.xlabel('Epoch Parameter')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.xticks(ticks=tick, labels=epoch_results_df['Epochs'])
plt.show()
```



The optimal epoch was found to be 8 epochs. Epochs essentially is manually deciding how many times the model should iterate through the data. Too few of epochs and the neural network's weights will be under tuned causing underfit, too many epochs and the model will be extremely overfit.

0.2.4 Final MLP Model & Metrics

After evaluating the MLP parameters using 10 fold cross validation to fine tune the model based on precision to accurately predict as many Malware applications as possible.

[ ]:
```
finalMLP = keras.models.Sequential([
    keras.Input(shape=55),
    keras.layers.Dense(50, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(200, activation='relu'),
    keras.layers.Dense(1, activation = 'sigmoid')
])

finalMLP.compile(loss = 'binary_crossentropy', optimizer=keras.optimizers.
 ↪Adam(learning_rate = 0.0005,epsilon = 0.001),metrics=['accuracy'])

finalMLP.fit(X_train, y_train, epochs = 8)
```

```
pred_finalMLP = np.round(finalMLP.predict(X_test))
```

```
Epoch 1/8
642/642 [==============================] - 3s 3ms/step - loss: 0.2233 -
accuracy: 0.9192
Epoch 2/8
642/642 [==============================] - 2s 3ms/step - loss: 0.1245 -
accuracy: 0.9581
Epoch 3/8
642/642 [==============================] - 2s 3ms/step - loss: 0.1141 -
accuracy: 0.9623
Epoch 4/8
642/642 [==============================] - 2s 3ms/step - loss: 0.1078 -
accuracy: 0.9642
Epoch 5/8
642/642 [==============================] - 2s 3ms/step - loss: 0.1029 -
accuracy: 0.9647
Epoch 6/8
642/642 [==============================] - 2s 3ms/step - loss: 0.0991 -
accuracy: 0.9663
Epoch 7/8
642/642 [==============================] - 2s 3ms/step - loss: 0.0955 -
accuracy: 0.9672
Epoch 8/8
642/642 [==============================] - 2s 3ms/step - loss: 0.0938 -
accuracy: 0.9686
```

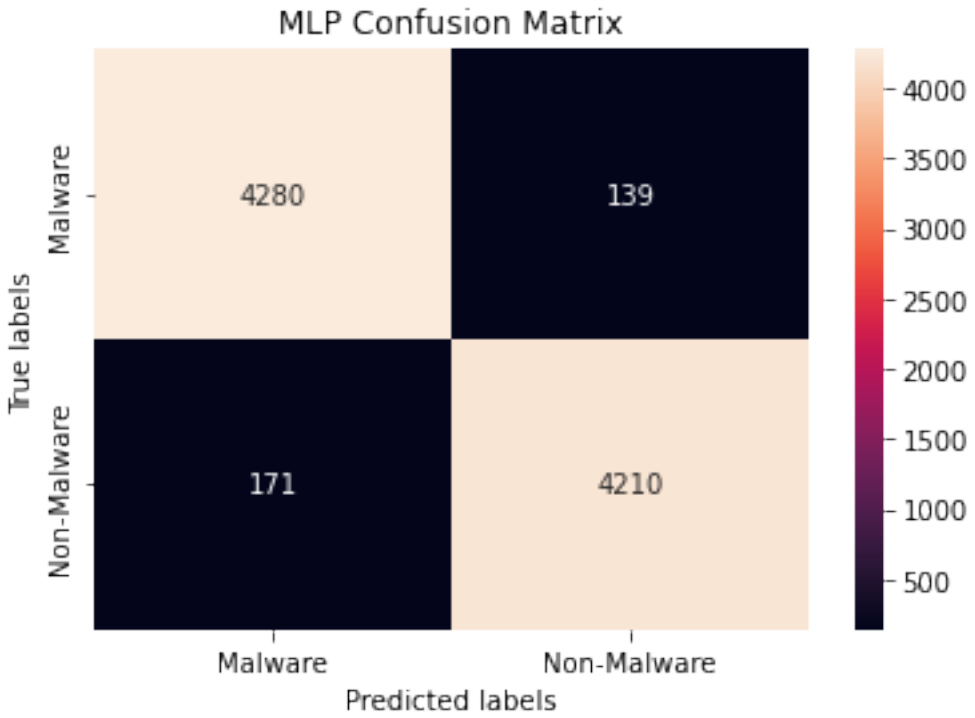
[ ]:
```
mlp_acc = accuracy_score(y_test, pred_finalMLP)
mlp_prec = metrics.precision_score(y_test, pred_finalMLP)
mlp_rec = metrics.recall_score(y_test, pred_finalMLP)
mlp_f1 = metrics.f1_score(y_test, pred_finalMLP)


print(accuracy_score(y_test, pred_finalMLP))
print(metrics.precision_score(y_test, pred_finalMLP))
```

```
0.9647727272727272
0.9680386295700161
```

MLP Confusion Matrix

[ ]:
```
mlp_cm = confusion_matrix(y_test, pred_finalMLP, labels = [0,1])


ax = plt.subplot()
sns.heatmap(mlp_cm, annot=True, fmt='g', ax=ax)
ax.set_xlabel('Predicted labels')
```

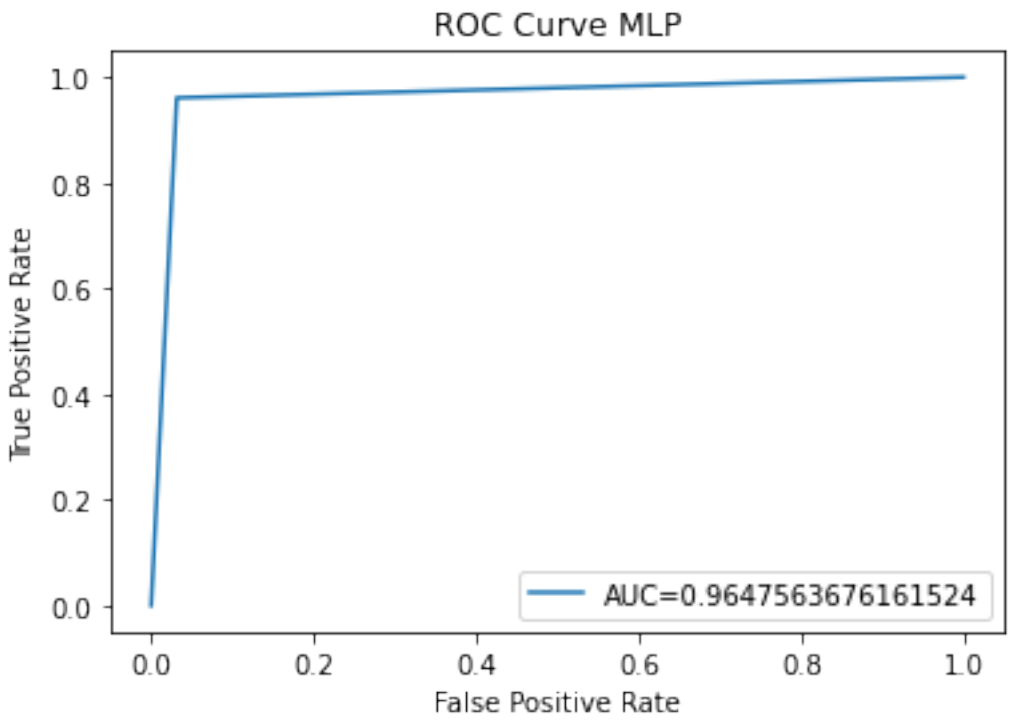
```
ax.set_ylabel('True labels')
ax.set_title('MLP Confusion Matrix')
ax.xaxis.set_ticklabels(['Malware', 'Non-Malware'])
ax.yaxis.set_ticklabels(['Malware', 'Non-Malware'])
plt.show()
```



The model is still giving pretty good overall accuracy performance. Considering the size of the test set, nearly 9,000 records, 117 missed classes for the Malware class is fairly decent, but using class weights could improve the precision even further.

MLP ROC Curve

[ ]:
```
mlpfalsepr, mlptruepr, _ = metrics.roc_curve(y_test,  pred_finalMLP)
mlpauc = metrics.roc_auc_score(y_test, pred_finalMLP)

plt.plot(mlpfalsepr,mlptruepr,label="AUC="+str(mlpauc))
plt.title('ROC Curve MLP')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```



0.2.5 Weighted MLP

In order to optimize precision even further, although our classes are balanced, class weights can be used when one is more important to solving a business problem than the other. Doing so will likely sacrifice some of the overall accuracy of the model while striving to classify as many malware programs as possible correctly.

[ ]:
```
weight = [{0:0.5,1:0.5},{0:0.6,1:0.4},{0:0.7,1:0.3},{0:0.75,1:0.25},{0:0.8,1:0.
↪2},
          {0:0.85,1:0.15},{0:0.9,1:0.1},{0:0.95,1:0.05}]
```

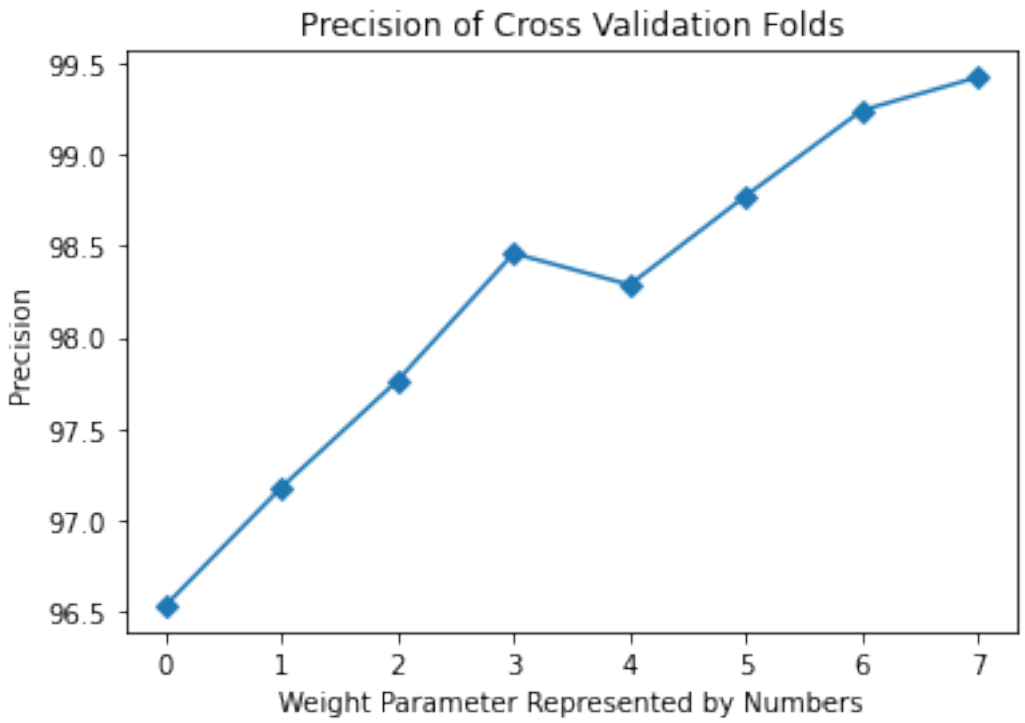
[ ]:
```
wmlp_results = []

weightMLP = keras.models.Sequential([
    keras.Input(shape=55),
    keras.layers.Dense(50, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(200, activation='relu'),
    keras.layers.Dense(1, activation = 'sigmoid')
])


for w in weight:
```

```
 weightMLP.compile(loss = 'binary_crossentropy', optimizer=keras.optimizers.
↪Adam(learning_rate = 0.0005,epsilon = 0.001),metrics=['accuracy'])
 estimator_w = KerasClassifier(build_fn = weightMLP, epochs = 8, class_weight␣
↪= w)

 cvw = cross_val_score(estimator_w, X_train, y_train, cv=10, n_jobs=2, scoring␣
↪= 'precision')
 wmlp_results.append({'Weight': w, 'Precision': cvw.mean()})
```

[ ]:
```
wmlp_results_df = pd.DataFrame(wmlp_results)
wmlp_results
```

[ ]:
```
[{'Precision': 0.9653610804672631, 'Weight': {0: 0.5, 1: 0.5}},
 {'Precision': 0.9718204475312255, 'Weight': {0: 0.6, 1: 0.4}},
 {'Precision': 0.9776464594496534, 'Weight': {0: 0.7, 1: 0.3}},
 {'Precision': 0.9845854892068013, 'Weight': {0: 0.75, 1: 0.25}},
 {'Precision': 0.9828716776959432, 'Weight': {0: 0.8, 1: 0.2}},
 {'Precision': 0.9877217984149975, 'Weight': {0: 0.85, 1: 0.15}},
 {'Precision': 0.992389140670114, 'Weight': {0: 0.9, 1: 0.1}},
 {'Precision': 0.9942047871099934, 'Weight': {0: 0.95, 1: 0.05}}]
```

[ ]:
```
tick = range(len(wmlp_results_df['Weight']))
plt.plot(wmlp_results_df['Precision']*100, marker = 'D')
plt.xlabel('Weight Parameter Represented by Numbers')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.xticks(ticks=tick)
plt.show()
```



The precision is best optimized for 0.95 malware class, 0.05 non-malware weighting.

[ ]:
```
finalwMLP = keras.models.Sequential([
    keras.Input(shape=55),
    keras.layers.Dense(50, activation='relu'),
    keras.layers.Dense(100, activation='relu'),
    keras.layers.Dense(200, activation='relu'),
    keras.layers.Dense(1, activation = 'sigmoid')
])

finalwMLP.compile(loss = 'binary_crossentropy', optimizer=keras.optimizers.
 ↪Adam(learning_rate = 0.0005,epsilon = 0.001),metrics=['accuracy'])

finalwMLP.fit(X_train, y_train, epochs = 8, class_weight = {0: 0.95, 1: 0.05})

pred_finalwMLP = np.round(finalwMLP.predict(X_test))
```

```
Epoch 1/8
642/642 [==============================] - 5s 5ms/step - loss: 0.0767 -
accuracy: 0.6149
Epoch 2/8
642/642 [==============================] - 3s 5ms/step - loss: 0.0251 -
accuracy: 0.8855
Epoch 3/8
```

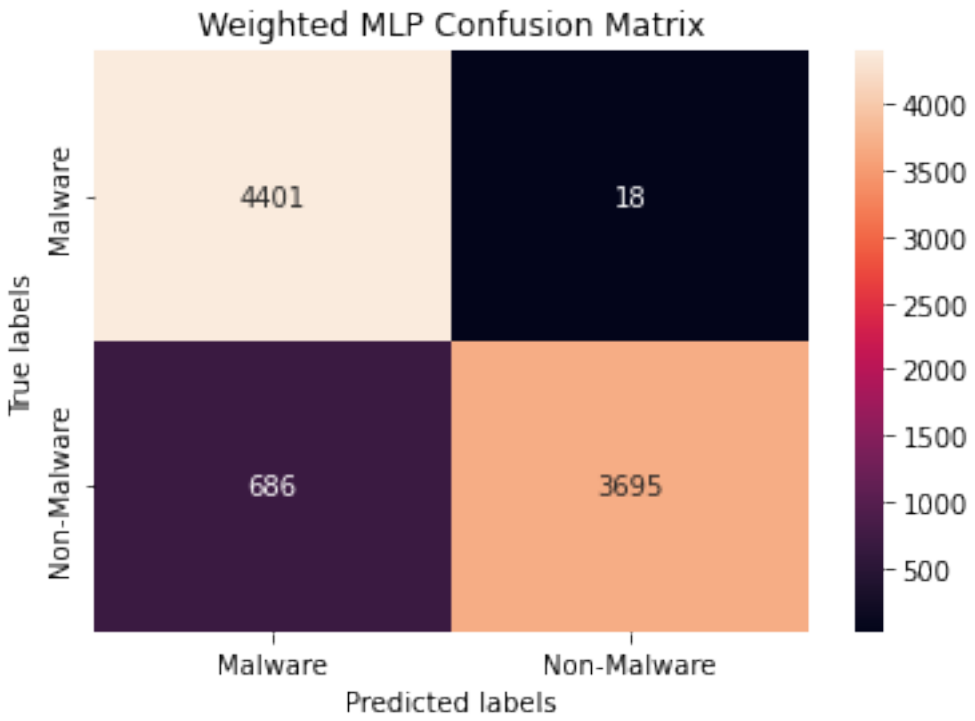
```
642/642 [==============================] - 3s 5ms/step - loss: 0.0220 -
accuracy: 0.9042
Epoch 4/8
642/642 [==============================] - 3s 5ms/step - loss: 0.0204 -
accuracy: 0.9108
Epoch 5/8
642/642 [==============================] - 4s 7ms/step - loss: 0.0193 -
accuracy: 0.9131
Epoch 6/8
642/642 [==============================] - 4s 6ms/step - loss: 0.0183 -
accuracy: 0.9171
Epoch 7/8
642/642 [==============================] - 4s 6ms/step - loss: 0.0174 -
accuracy: 0.9196
Epoch 8/8
642/642 [==============================] - 4s 6ms/step - loss: 0.0165 -
accuracy: 0.9247
```

[ ]:
```
w_mlp_acc = accuracy_score(y_test, pred_finalwMLP)
w_mlp_prec = metrics.precision_score(y_test, pred_finalwMLP)
w_mlp_rec = metrics.recall_score(y_test, pred_finalwMLP)
w_mlp_f1 = metrics.f1_score(y_test, pred_finalwMLP)

print(accuracy_score(y_test, pred_finalwMLP))
print(metrics.precision_score(y_test, pred_finalwMLP))
```

```
0.92
0.995152168058174
```

Weighted MLP Confusion Matrix

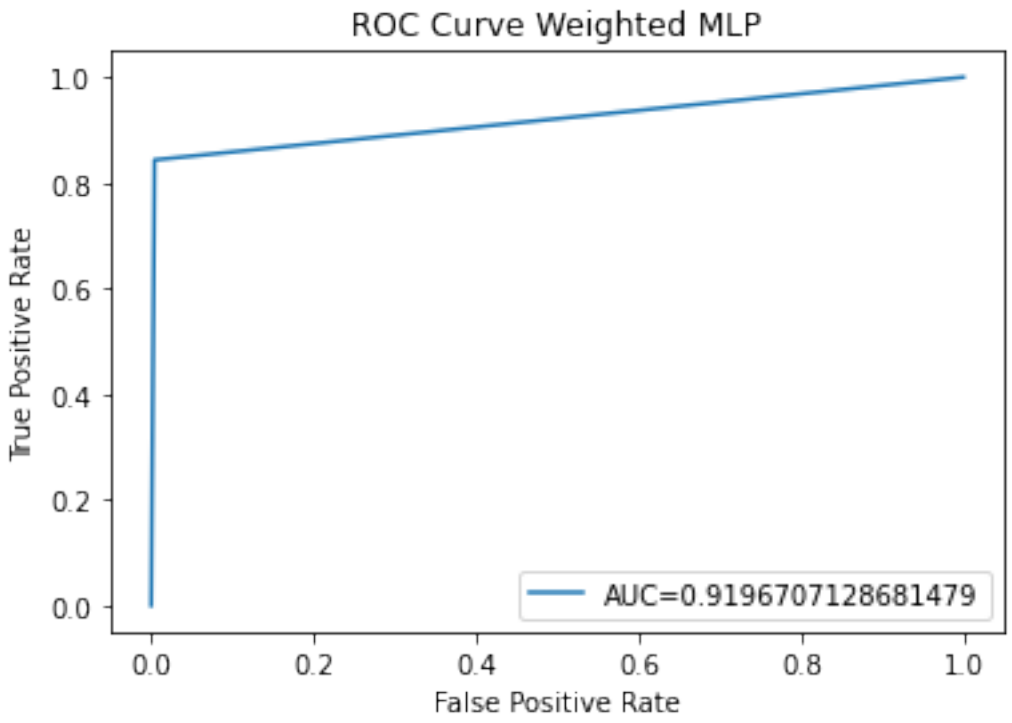
[ ]:
```
mlpw_cm = confusion_matrix(y_test, pred_finalwMLP, labels = [0,1])


ax = plt.subplot()
sns.heatmap(mlpw_cm, annot=True, fmt='g', ax=ax)
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Weighted MLP Confusion Matrix')
ax.xaxis.set_ticklabels(['Malware', 'Non-Malware'])
ax.yaxis.set_ticklabels(['Malware', 'Non-Malware'])
plt.show()
```



While the model does only have 6 miss classifications on the malware class, it does come at the cost of flagging many legitimate applications as malware. In a real business setting, the financial cost of flagging that many legitimate applications as malware would have to taken into account. This could potentially flag the application for the smartphone user to decide whether yes, they would like to continue the download, or no, they think the application might be risky as a computer defense application would.

Weighted MLP ROC Curve

[ ]:
```
mlpwfalsepr, mlpwtruepr, _ = metrics.roc_curve(y_test,  pred_finalwMLP)
mlpwauc = metrics.roc_auc_score(y_test, pred_finalwMLP)

plt.plot(mlpwfalsepr,mlpwtruepr,label="AUC="+str(mlpwauc))
plt.title('ROC Curve Weighted MLP')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```



0.3 SVM

[ ]:
```
from sklearn.svm import SVC
```

SVM Kernel Tuning with Cost

[ ]:
```
cost = [0.01,0.05,0.1,0.5,1,3,5,10,15,20]
svm_results = []
for c in cost:
  # iterating over each kernel with different costs
  svmline = SVC(C = c, kernel = 'linear',random_state=3).fit(X_train, y_train)
  svmpoly = SVC(C = c, kernel = 'poly',random_state=3).fit(X_train, y_train)
  svmrbf = SVC(C = c, kernel = 'rbf',random_state=3).fit(X_train, y_train)
  svmsig = SVC(C = c, kernel = 'sigmoid',random_state=3).fit(X_train, y_train)

  # iterating over cross validation scoring
  svmlcv = cross_val_score(svmline, X_train, y_train, cv = 10, n_jobs = 2,␣
 ↪scoring = 'precision')
  svmpcv = cross_val_score(svmpoly, X_train, y_train, cv = 10, n_jobs = 2,␣
 ↪scoring = 'precision')
  svmrcv = cross_val_score(svmrbf, X_train, y_train, cv = 10, n_jobs = 2,␣
 ↪scoring = 'precision')
```

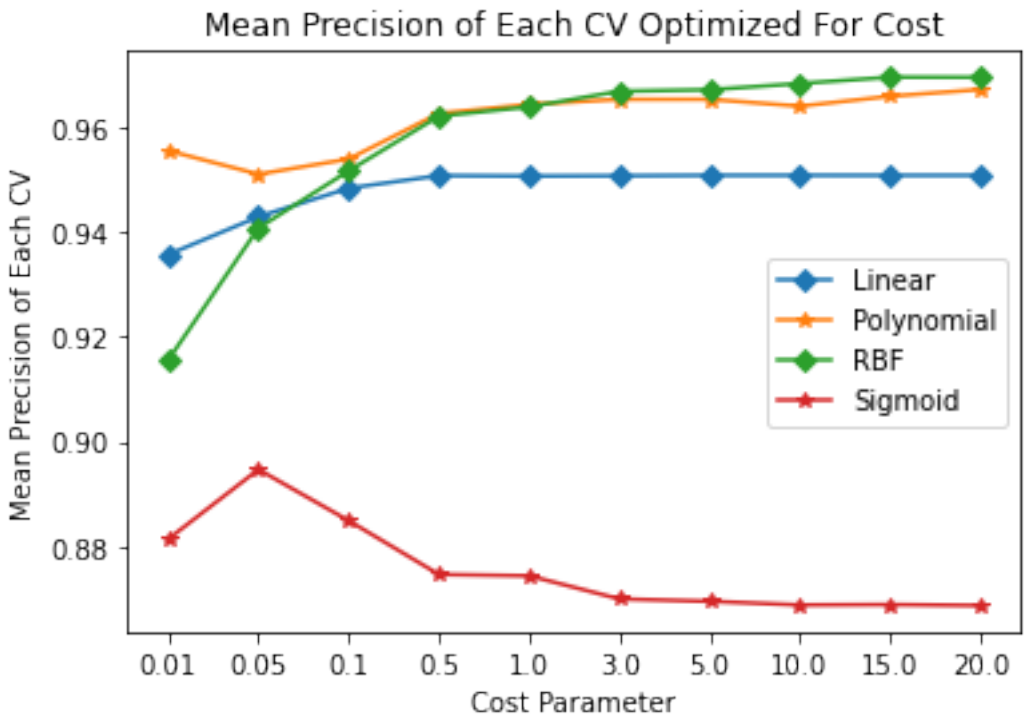
```
 svmscv = cross_val_score(svmsig, X_train, y_train, cv = 10, n_jobs = 2,␣
↪scoring = 'precision')
 svm_results.append({'Cost': c, 'Linear Precision': svmlcv.mean(),
                     'Polynomial Precision': svmpcv.mean(), 'RBF Precision':␣
↪svmrcv.mean(),
                     'Sigmoid Precision': svmscv.mean()})
```

[ ]:
```
svm_results_df = pd.DataFrame(svm_results)
svm_results_df
```

[ ]:

	Cost	Linear Precision	Polynomial Precision	RBF Precision	Sigmoid Precision
0	0.01	0.935719	0.955553	0.915598	0.881399
1	0.05	0.942949	0.951068	0.940790	0.894696
2	0.10	0.948400	0.953966	0.951712	0.884924
3	0.50	0.950803	0.962679	0.962147	0.874653
4	1.00	0.950713	0.964391	0.963925	0.874391
5	3.00	0.950768	0.965357	0.966906	0.869976
6	5.00	0.950849	0.965370	0.967190	0.869523
7	10.00	0.950854	0.964048	0.968378	0.868829
8	15.00	0.950844	0.965954	0.969591	0.868903
9	20.00	0.950846	0.967238	0.969568	0.868728

[ ]:
```
tick = range(len(svm_results_df['Cost']))
plt.plot(svm_results_df['Linear Precision'], marker = 'D', label = 'Linear')
plt.plot(svm_results_df['Polynomial Precision'], marker = '*', label =␣
 ↪'Polynomial')
plt.plot(svm_results_df['RBF Precision'], marker = 'D', label = 'RBF')
plt.plot(svm_results_df['Sigmoid Precision'], marker = '*', label = 'Sigmoid')
plt.xlabel('Cost Parameter')
plt.ylabel('Mean Precision of Each CV')
plt.title('Mean Precision of Each CV Optimized For Cost')
plt.legend(loc = 'top left')
plt.xticks(ticks=tick, labels=svm_results_df['Cost'])
plt.show()
```



SVM Cost Tuning For RBF In order to save time training the models for tuning to get a more precise cost, a new training loop has been creating below for the Kernel of choice in this instance, RBF, as it offered the best precision above. The RBF kernel uses distance to compute similarity.

The cost parameter of the SVM regularizes the data within the algorithm. The higher the cost, the greater the regularization.

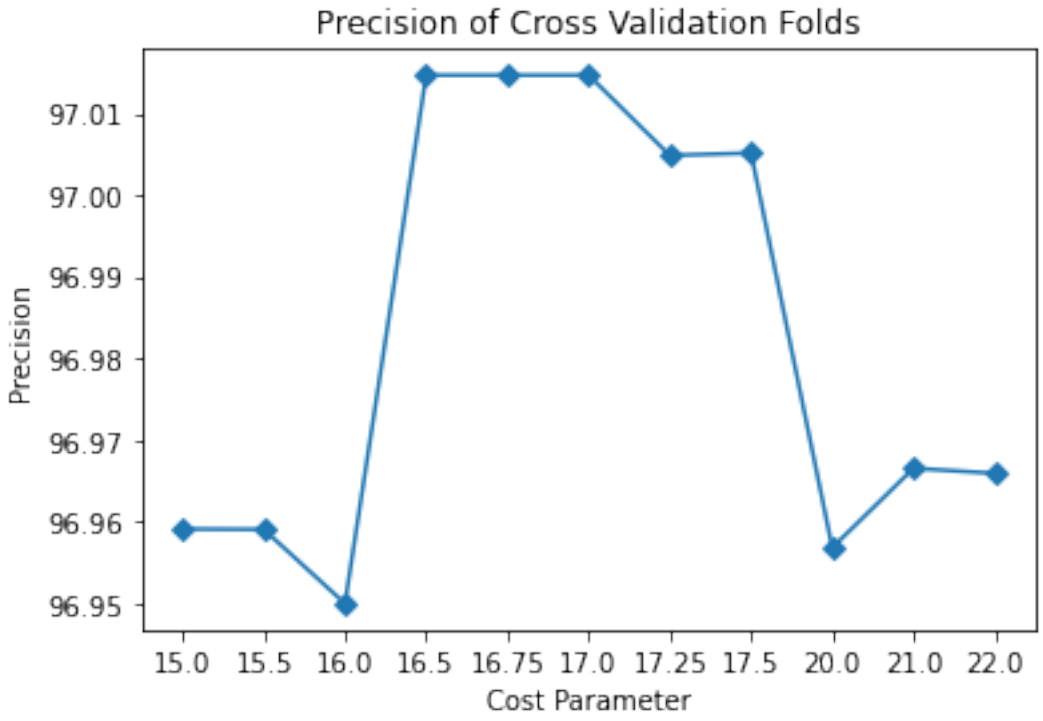
[ ]:
```
costs = [15,15.5,16,16.5,16.75,17,17.25,17.5,20,21,22]

Crbf_results = []

for c in costs:
  costtune = SVC(C= c, kernel = 'rbf',random_state=3).fit(X_train,y_train)
  costcv = cross_val_score(costtune, X_train, y_train, n_jobs=2, cv=10,␣
 ↪scoring='precision')
  Crbf_results.append({'Cost': c,'Precision': costcv.mean()})
```

[ ]:
```
Crbf_results_df = pd.DataFrame(Crbf_results)
Crbf_results_df
```

[ ]:
```
     Cost  Precision
0   15.00   0.969591
1   15.50   0.969590
```

```
2   16.00   0.969500
3   16.50   0.970147
4   16.75   0.970147
5   17.00   0.970147
6   17.25   0.970048
7   17.50   0.970051
8   20.00   0.969568
9   21.00   0.969665
10  22.00   0.969659
```

[ ]:
```
tick = range(len(Crbf_results_df['Cost']))
plt.plot(Crbf_results_df['Precision']*100, marker = 'D')
plt.xlabel('Cost Parameter')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.xticks(ticks=tick, labels=Crbf_results_df['Cost'])
plt.show()
```



Due to the time constraints when training an SVM, further fine tuning of the cost parameter was done on the kernel of choice RBF. The fine tune revealed that a cost parameter of 17 offered the best precision over the cross validation.

0.3.1 RBF Gamma Tuning

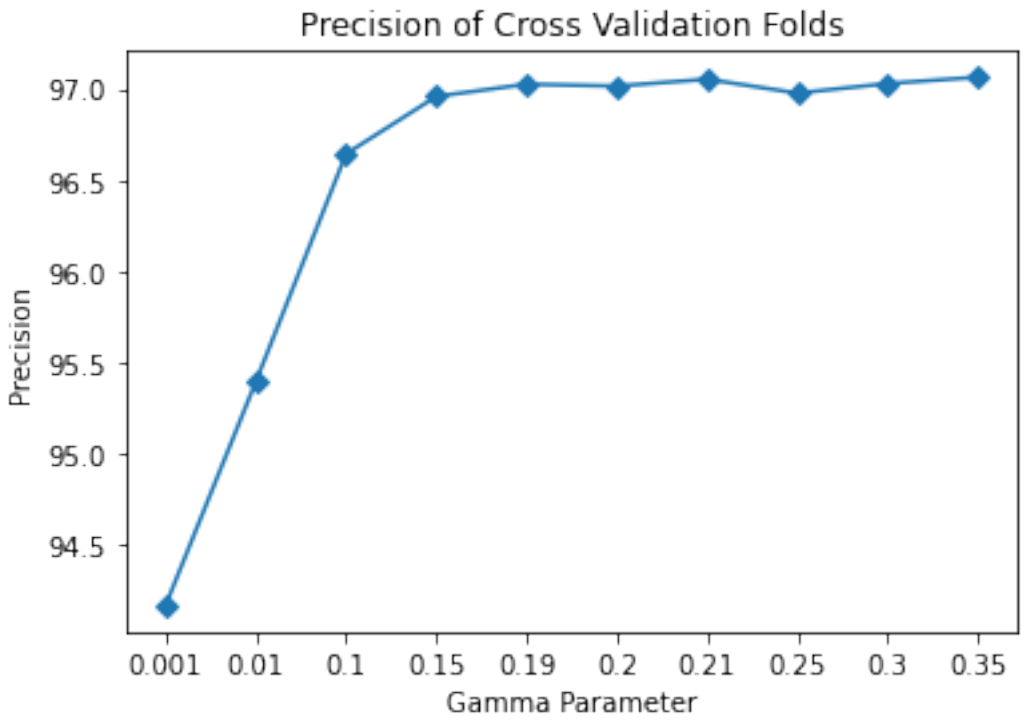
[ ]:
```
gammas = [0.001,0.01,0.1,0.15,0.19,0.2,0.21,0.25,0.3,0.35]

gamsvm_results = []

for g in gammas:
  gammatune = SVC(C= 17, gamma=g, kernel = 'rbf',random_state=3).
 ↪fit(X_train,y_train)
  gammacv = cross_val_score(gammatune, X_train, y_train, n_jobs=2, cv=10,␣
 ↪scoring='precision')
  gamsvm_results.append({'Gamma': g,'Precision': gammacv.mean()})
```

[ ]:
```
gamsvm_results_df = pd.DataFrame(gamsvm_results)
gamsvm_results_df
```

[ ]:

	Gamma	Precision
0	0.001	0.941667
1	0.010	0.953930
2	0.100	0.966423
3	0.150	0.969574
4	0.190	0.970246
5	0.200	0.970141
6	0.210	0.970519
7	0.250	0.969757
8	0.300	0.970280
9	0.350	0.970630

[ ]:
```
tick = range(len(gamsvm_results_df['Gamma']))
plt.plot(gamsvm_results_df['Precision']*100, marker = 'D')
plt.xlabel('Gamma Parameter')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.xticks(ticks=tick, labels=gamsvm_results_df['Gamma'])
plt.show()
```



The gamma of the RBF kerel controls the spread of the decision boundaries. The lower the number, the wider the boundary. 0.35 proved to have the highest precision across the cross validation folds.

0.3.2 Final SVM Model

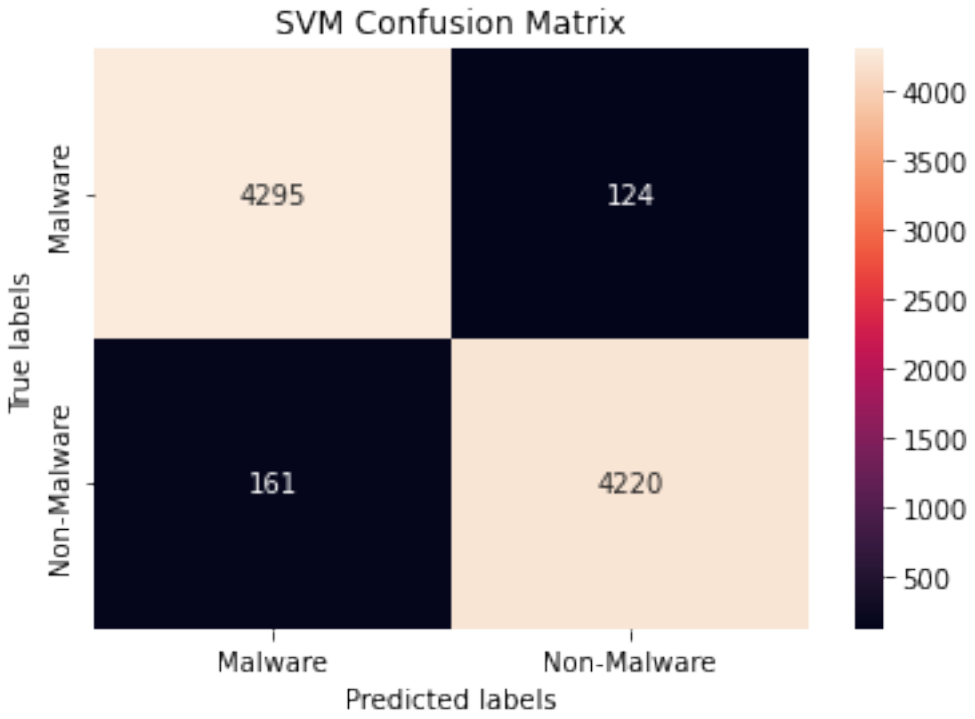
[ ]:
```
final_svm = SVC(C = 17, gamma=0.35, kernel = 'rbf',random_state=3).fit(X_train,␣
↪y_train)
svm_pred = final_svm.predict(X_test)
```

[ ]:
```
svm_acc = accuracy_score(y_test, svm_pred)
svm_prec = metrics.precision_score(y_test, svm_pred)
svm_rec = metrics.recall_score(y_test, svm_pred)
svm_f1 = metrics.f1_score(y_test, svm_pred)

print(accuracy_score(y_test, svm_pred))
print(metrics.precision_score(y_test, svm_pred))
```

```
0.9676136363636364
0.9714548802946593
```

Confusion Matrix

```
[ ]: svm_cm = confusion_matrix(y_test, svm_pred, labels = [0,1])


ax = plt.subplot()
sns.heatmap(svm_cm, annot=True, fmt='g', ax=ax)
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('SVM Confusion Matrix')
ax.xaxis.set_ticklabels(['Malware', 'Non-Malware'])
ax.yaxis.set_ticklabels(['Malware', 'Non-Malware'])
plt.show()
```

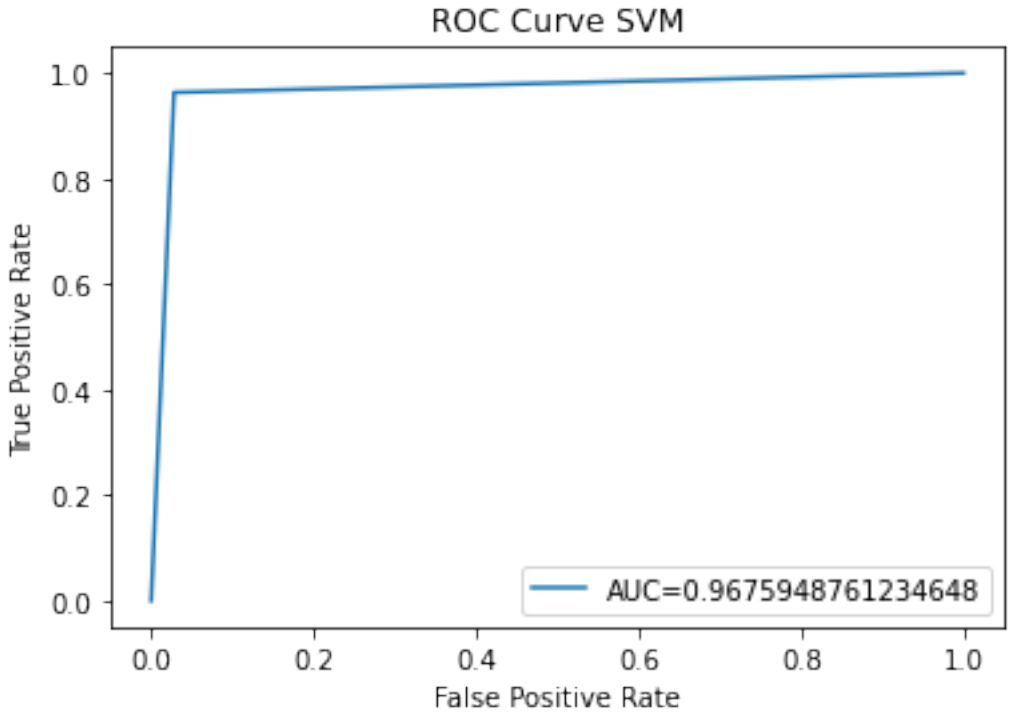


Compared to the weighted model, this confusion matrix has more overall accuracy as the recall is improved.

ROC Curve

```
[ ]: svmfalsepr, svmtruepr, _ = metrics.roc_curve(y_test,  svm_pred)
svmauc = metrics.roc_auc_score(y_test, svm_pred)

plt.plot(svmfalsepr,svmtruepr,label="AUC="+str(svmauc))
plt.title('ROC Curve SVM')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
```

```
plt.legend(loc=4)
plt.show()
```



0.3.3 Weighted SVM

Again, using weights can tune the model to be even more focused on precision than tuning other hyperparameters.

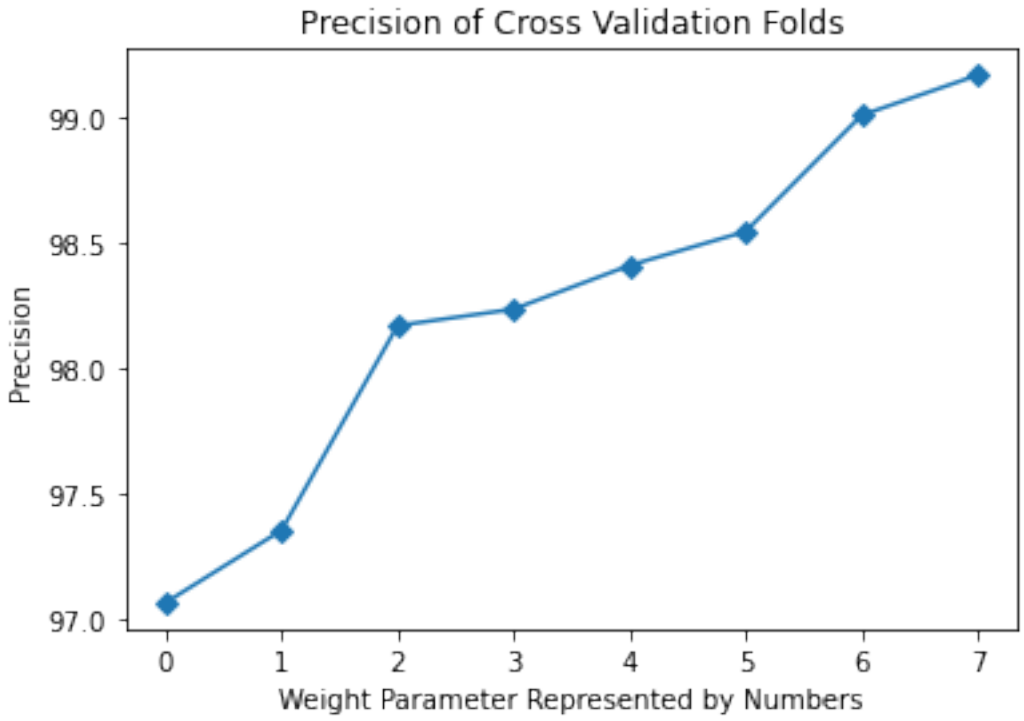
[ ]:
```
weight_svm_results = []
for w in weight:
  weight_svm = SVC(class_weight = w, C = 17, gamma=0.35, kernel =␣
 ↪'rbf',random_state=3).fit(X_train, y_train)
  svmwcv = cross_val_score(weight_svm, X_train, y_train, cv=10, n_jobs=2,␣
 ↪scoring='precision')
  weight_svm_results.append({'weight': w, 'Precision': svmwcv.mean()})
```

[ ]:
```
weight_svm_results_df = pd.DataFrame(weight_svm_results)
weight_svm_results_df
```

[ ]:
```
                weight  Precision
0    {0: 0.5, 1: 0.5}   0.970671
1    {0: 0.6, 1: 0.4}   0.973556
```

```
2    {0: 0.7, 1: 0.3}   0.981705
3   {0: 0.75, 1: 0.25}   0.982362
4    {0: 0.8, 1: 0.2}   0.984104
5   {0: 0.85, 1: 0.15}   0.985455
6    {0: 0.9, 1: 0.1}   0.990084
7   {0: 0.95, 1: 0.05}   0.991683
```

[ ]:
```
tick = range(len(weight_svm_results_df['weight']))
plt.plot(weight_svm_results_df['Precision']*100, marker = 'D')
plt.xlabel('Weight Parameter Represented by Numbers')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.xticks(ticks=tick)
plt.show()
```



The 0.95, 0.5 weight again offered the highest precision over the cross validation.

[ ]:
```
weight_svm = SVC(class_weight = {0: 0.95, 1: 0.05}, C = 17, gamma=0.35,
                 kernel = 'rbf',random_state=3).fit(X_train, y_train)
```

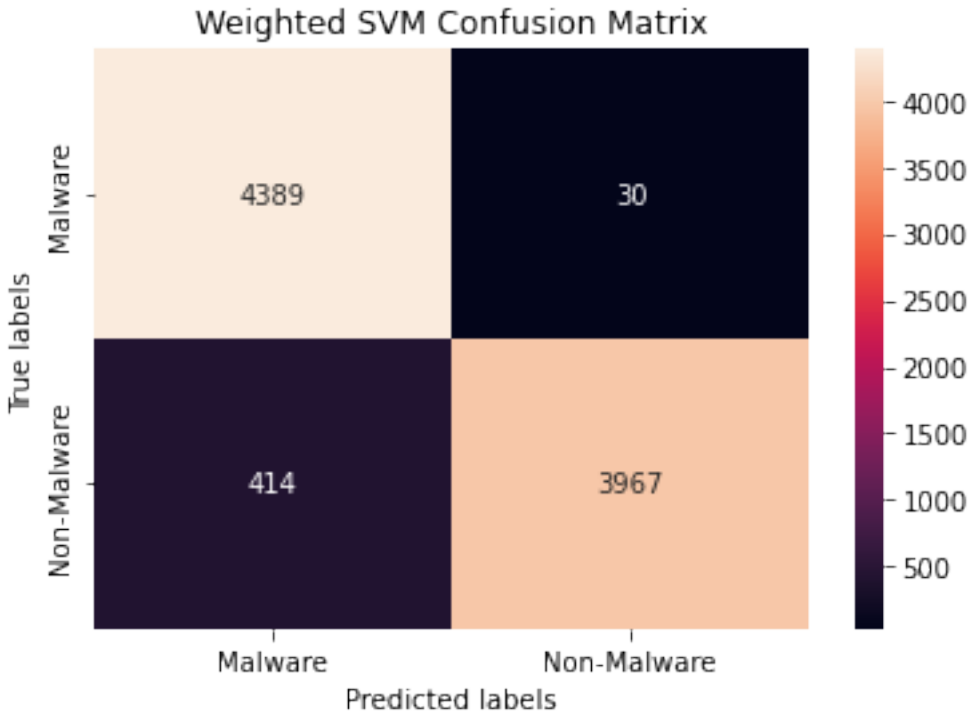
[ ]:
```
w_svm_pred = weight_svm.predict(X_test)


w_svm_acc = accuracy_score(y_test, w_svm_pred)
```

```
w_svm_prec = metrics.precision_score(y_test, w_svm_pred)
w_svm_rec = metrics.recall_score(y_test, w_svm_pred)
w_svm_f1 = metrics.f1_score(y_test, w_svm_pred)



print(accuracy_score(y_test, w_svm_pred))
print(metrics.precision_score(y_test, w_svm_pred))
```

```
0.9495454545454546
0.9924943707780836
```

Weight SVM Confusion Matrix

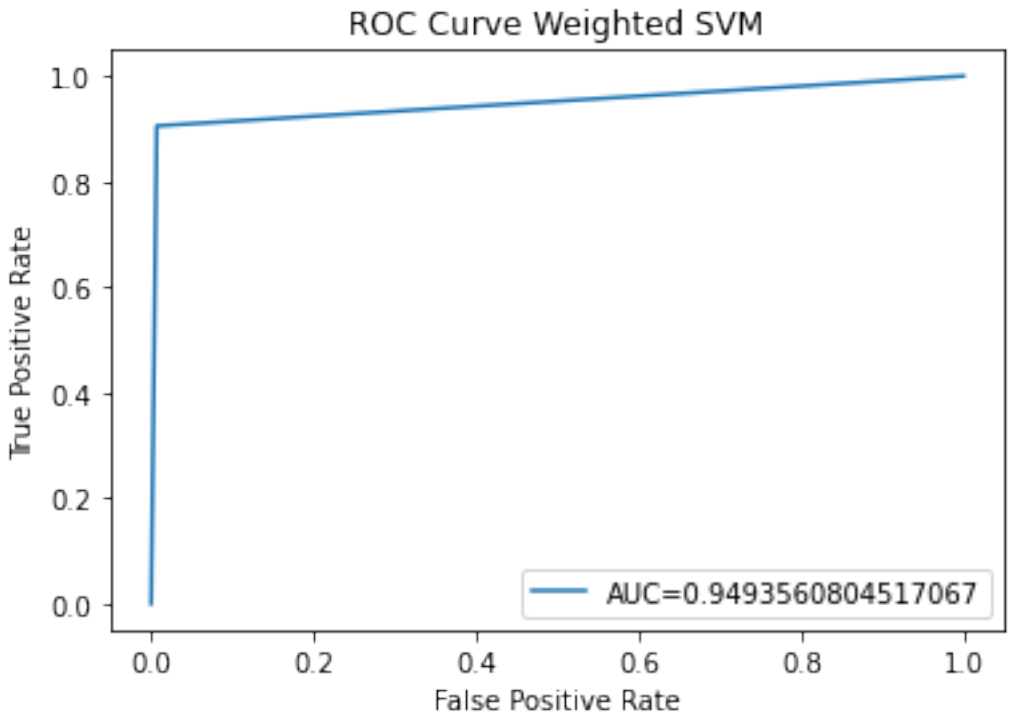
[ ]:
```
wsvm_cm = confusion_matrix(y_test, w_svm_pred, labels = [0,1])


ax = plt.subplot()
sns.heatmap(wsvm_cm, annot=True, fmt='g', ax=ax)
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Weighted SVM Confusion Matrix')
ax.xaxis.set_ticklabels(['Malware', 'Non-Malware'])
ax.yaxis.set_ticklabels(['Malware', 'Non-Malware'])
plt.show()
```



The weighted SVM created a less precise confusion matrix over the test set than the MLP, but it is much more accurate overall with 94% accuracy vs. the weighted MLP's 89% accuracy.

Weighted SVM ROC Curve

[ ]:
```
wsvmfalsepr, wsvmtruepr, _ = metrics.roc_curve(y_test,  w_svm_pred)
wsvmauc = metrics.roc_auc_score(y_test, w_svm_pred)

plt.plot(wsvmfalsepr,wsvmtruepr,label="AUC="+str(wsvmauc))
plt.title('ROC Curve Weighted SVM')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```



0.4 Decision Tree

[ ]:
```
from sklearn.tree import DecisionTreeClassifier
```

Below both max depth and ccp_alpha will be tuned to compare the results over the cross validation

and the test set. Both parameter tune the complexity of the decision tree in slightly different ways. From previous experience, ccp_alpha will offer a better tune because it is a little more specific than simply specifying the max depth of the tree.

0.4.1 Tuning for Max Depth

[ ]:
```
DTC_results = []

for d in range(2, 52):
  dtcg = DecisionTreeClassifier(criterion='gini', max_depth = d,random_state=3).
↪fit(X_train, y_train)
  dtce = DecisionTreeClassifier(criterion='entropy', max_depth =␣
↪d,random_state=3).fit(X_train, y_train)
  cvg = cross_val_score(dtcg, X_train, y_train, cv = 10, n_jobs=2,␣
↪scoring='precision')
  cve = cross_val_score(dtce, X_train, y_train, cv = 10, n_jobs=2,␣
↪scoring='precision')
  DTC_results.append({'Depth': d, 'Gini Precision': cvg.mean(), 'Entropy␣
↪Precision': cve.mean()})
```

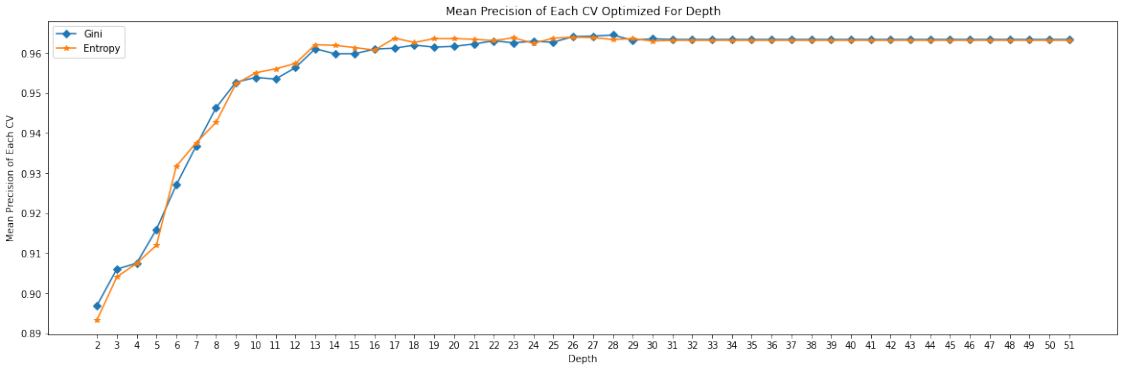
[ ]:
```
DTC_results_df = pd.DataFrame(DTC_results)
DTC_results_df
```

[ ]:

	Depth	Gini Precision	Entropy Precision
0	2	0.896935	0.893312
1	3	0.906028	0.904054
2	4	0.907479	0.907469
3	5	0.915952	0.911965
4	6	0.927056	0.931714
5	7	0.936757	0.937558
6	8	0.946284	0.942628
7	9	0.952660	0.952234
8	10	0.953878	0.955032
9	11	0.953454	0.956012
10	12	0.956308	0.957347
11	13	0.961031	0.962052
12	14	0.959779	0.961878
13	15	0.959803	0.961327
14	16	0.960944	0.960685
15	17	0.961173	0.963675
16	18	0.961941	0.962610
17	19	0.961441	0.963553
18	20	0.961634	0.963563
19	21	0.962222	0.963399
20	22	0.963007	0.963069
21	23	0.962523	0.963768

22	24	0.962903	0.962392
23	25	0.962666	0.963671
24	26	0.964073	0.963923
25	27	0.964180	0.963817
26	28	0.964460	0.963340
27	29	0.963145	0.963547
28	30	0.963528	0.962959
29	31	0.963347	0.963137
30	32	0.963347	0.963137
31	33	0.963347	0.963137
32	34	0.963347	0.963137
33	35	0.963347	0.963137
34	36	0.963347	0.963137
35	37	0.963347	0.963137
36	38	0.963347	0.963137
37	39	0.963347	0.963137
38	40	0.963347	0.963137
39	41	0.963347	0.963137
40	42	0.963347	0.963137
41	43	0.963347	0.963137
42	44	0.963347	0.963137
43	45	0.963347	0.963137
44	46	0.963347	0.963137
45	47	0.963347	0.963137
46	48	0.963347	0.963137
47	49	0.963347	0.963137
48	50	0.963347	0.963137
49	51	0.963347	0.963137

[ ]:
```
tick = range(len(DTC_results_df['Depth']))
plt.figure(figsize=(20,6))
plt.plot(DTC_results_df['Gini Precision'], marker = 'D', label = 'Gini')
plt.plot(DTC_results_df['Entropy Precision'], marker = '*', label = 'Entropy')
plt.xlabel('Depth')
plt.ylabel('Mean Precision of Each CV')
plt.title('Mean Precision of Each CV Optimized For Depth')
plt.legend(loc = 'top left')
plt.xticks(ticks=tick, labels=DTC_results_df['Depth'])
plt.show()
```



The most optimized criterion and depth for the Decision Tree Precision is gini and max_depth = 27. Gini will be the criterion that decides the splits for the model.

0.4.2 Tuning for CCP Alpha

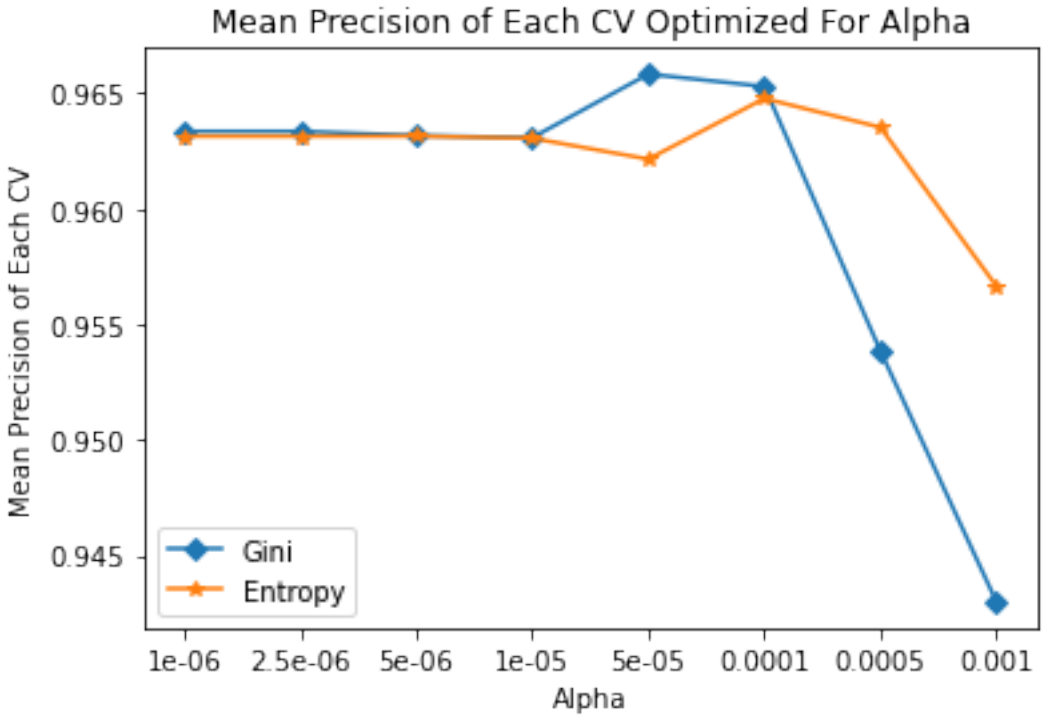
[ ]:
```
alpha = [0.000001,0.0000025,0.000005,0.00001,0.00005,0.0001,0.0005,0.001]
DTCa_results = []

for a in alpha:
  dtcga = DecisionTreeClassifier(criterion='gini', ccp_alpha =␣
↪a,random_state=3).fit(X_train, y_train)
  dtcea = DecisionTreeClassifier(criterion='entropy', ccp_alpha =␣
↪a,random_state=3).fit(X_train, y_train)
  cvga = cross_val_score(dtcga, X_train, y_train, cv = 10, n_jobs=2,␣
↪scoring='precision')
  cvea = cross_val_score(dtcea, X_train, y_train, cv = 10, n_jobs=2,␣
↪scoring='precision')
  DTCa_results.append({'Alpha': a, 'Gini Precision': cvga.mean(), 'Entropy␣
↪Precision': cvea.mean()})
```

[ ]:
```
DTCa_results_df = pd.DataFrame(DTCa_results)
DTCa_results_df
```

[ ]:

	Alpha	Gini Precision	Entropy Precision
0	0.000001	0.963347	0.963137
1	0.000003	0.963347	0.963137
2	0.000005	0.963163	0.963141
3	0.000010	0.963073	0.963048
4	0.000050	0.965826	0.962145
5	0.000100	0.965288	0.964768
6	0.000500	0.953852	0.963523
7	0.001000	0.942996	0.956652

```
[ ]: tick = range(len(DTCa_results_df['Alpha']))
plt.plot(DTCa_results_df['Gini Precision'], marker = 'D', label = 'Gini')
plt.plot(DTCa_results_df['Entropy Precision'], marker = '*', label = 'Entropy')
plt.xlabel('Alpha')
plt.ylabel('Mean Precision of Each CV')
plt.title('Mean Precision of Each CV Optimized For Alpha')
plt.legend(loc = 'lower left')
plt.xticks(ticks=tick, labels=DTCa_results_df['Alpha'])
plt.show()
```



Similarly, the ccp_alpha complexity tune performed best using gini criterion for deciding splits. The most optimal alpha is 0.00005.

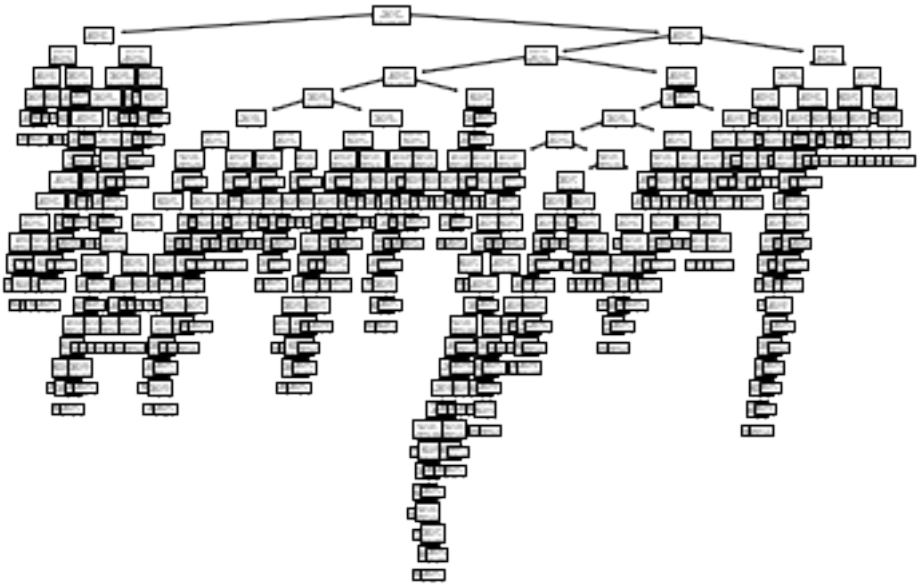
0.4.3 CCP Alpha Decision Tree Model

```
[ ]: from sklearn.tree import plot_tree
final_dt = DecisionTreeClassifier(criterion = 'gini', ccp_alpha=0.00005,␣
 ↪random_state = 3)
final_dt.fit(X_train, y_train)
pred_final_dt = final_dt.predict(X_test)
print(accuracy_score(y_test, pred_final_dt))
print(metrics.precision_score(y_test, pred_final_dt))
```

```
0.965
0.9695642148950888
```

```
[ ]: dt_acc = accuracy_score(y_test, pred_final_dt)
dt_prec = metrics.precision_score(y_test, pred_final_dt)
dt_rec = metrics.recall_score(y_test, pred_final_dt)
dt_f1 = metrics.f1_score(y_test, pred_final_dt)
```

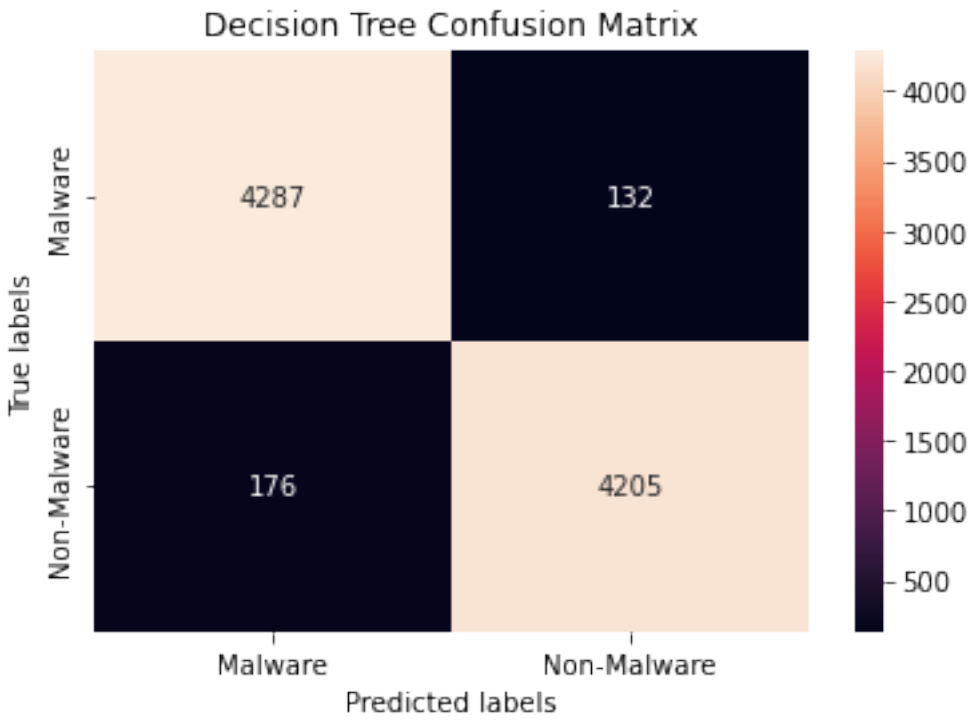
```
[ ]: plot_tree(final_dt)
plt.show()
```



A general look at the overall complexity of both final trees shows that in addition to performing better on the test set, this decision tree also has a simplier split complexity.

Confusion Matrix CCP Alpha Model

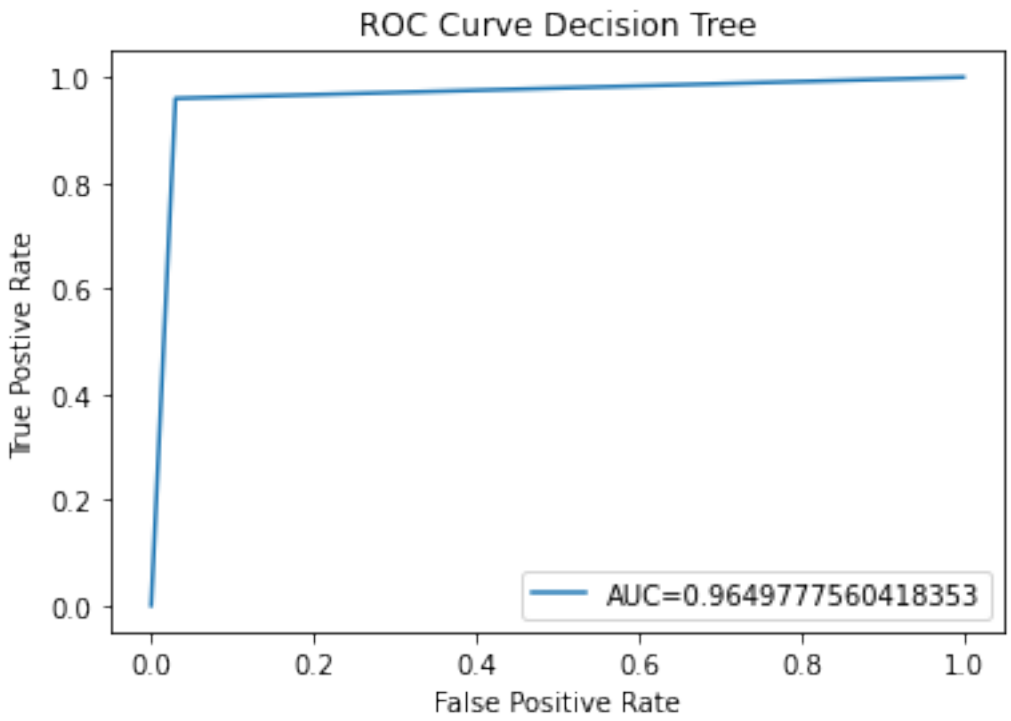
```
[ ]: final_dt_cm = confusion_matrix(y_test, pred_final_dt, labels = [0,1])


ax = plt.subplot()
sns.heatmap(final_dt_cm, annot=True, fmt='g', ax=ax)
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Decision Tree Confusion Matrix')
ax.xaxis.set_ticklabels(['Malware', 'Non-Malware'])
ax.yaxis.set_ticklabels(['Malware', 'Non-Malware'])
plt.show()
```



The balanced model with a tune towards precision still offered more balanced results on the classes missing around 150 of both class. The precision tune still allows to many false negatives for the business to move forward with.

ROC Curve CCP Alpha Model

[ ]:
```
dtafalsepr, dtatruepr, _ = metrics.roc_curve(y_test,  pred_final_dt)
dtaauc = metrics.roc_auc_score(y_test, pred_final_dt)

plt.plot(dtafalsepr,dtatruepr,label="AUC="+str(dtaauc))
plt.title('ROC Curve Decision Tree')
plt.ylabel('True Postive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```

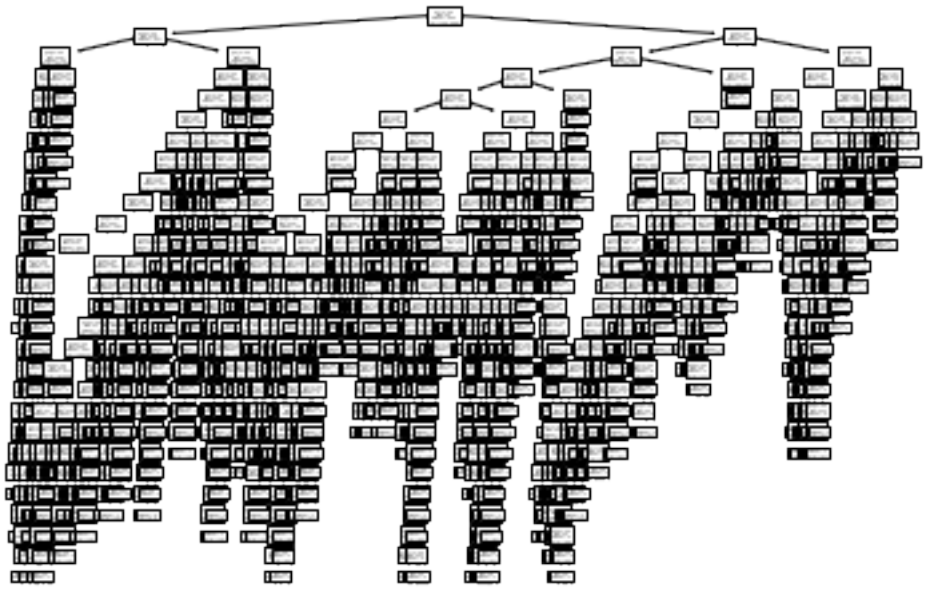


0.4.4 Max Depth Decision Tree

[ ]:
```
fin_dt = DecisionTreeClassifier(criterion='gini', max_depth = 27, random_state␣
↪= 3)
fin_dt.fit(X_train, y_train)
pred_fin_dt = fin_dt.predict(X_test)
print(accuracy_score(y_test, pred_fin_dt))
print(metrics.precision_score(y_test, pred_fin_dt))
```

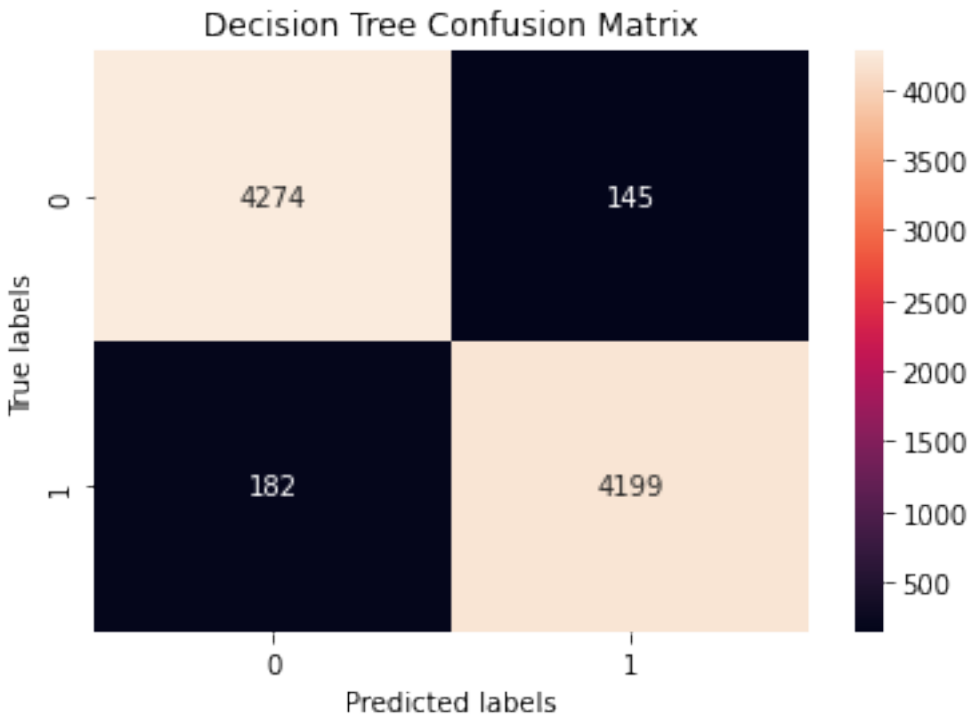
```
0.9628409090909091
0.9666206261510129
```

[ ]:
```
plot_tree(fin_dt)
plt.show()
```



Max Depth Confusion Matrix

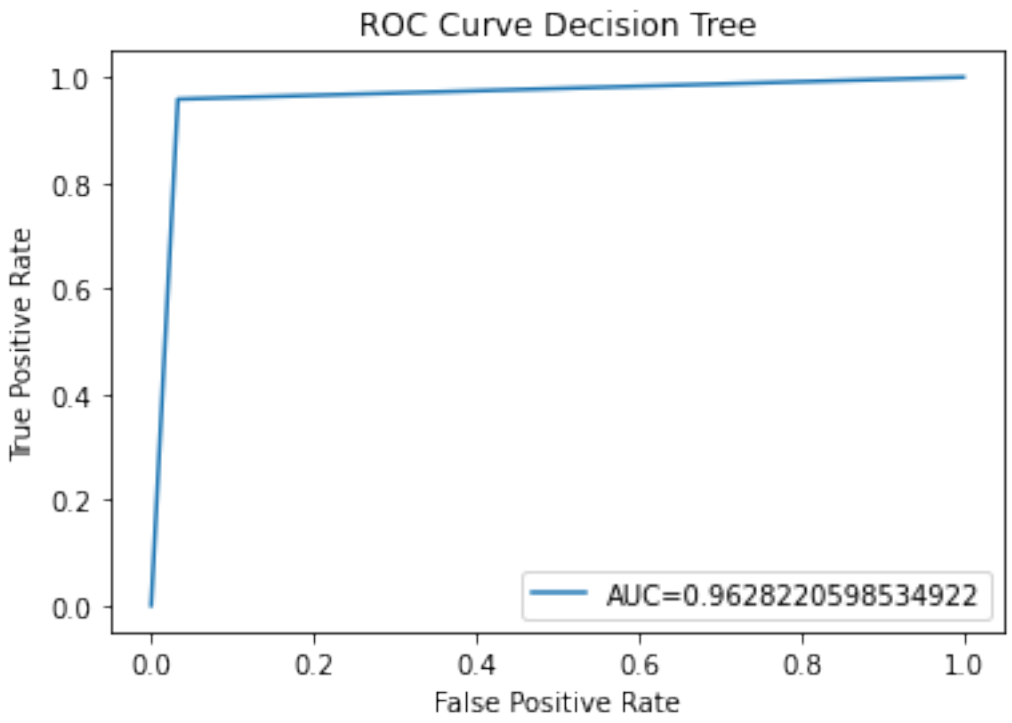
[ ]:
```
fin_dt_cm = confusion_matrix(y_test, pred_fin_dt, labels = [0,1])


ax = plt.subplot()
sns.heatmap(fin_dt_cm, annot=True, fmt='g', ax=ax)
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Decision Tree Confusion Matrix')
ax.xaxis.set_ticklabels(['0', '1'])
ax.yaxis.set_ticklabels(['0', '1'])
plt.show()
```



Using max_depth offered similarly balanced and worse overall results than the ccp_alpha tune.

Max Depth ROC Curve

[ ]:
```
dtfalsepr, dttruepr, _ = metrics.roc_curve(y_test,  pred_fin_dt)
dtauc = metrics.roc_auc_score(y_test, pred_fin_dt)

plt.plot(dtfalsepr,dttruepr,label="AUC="+str(dtauc))
plt.title('ROC Curve Decision Tree')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```



0.4.5 Weighted Decision Tree

[ ]:
```
weight_dt_results = []

for w in weight:
  weight_dt = DecisionTreeClassifier(class_weight = w, criterion = 'gini',␣
 ↪ccp_alpha=0.00005,random_state=3).fit(X_train,y_train)
  dtwcv = cross_val_score(weight_dt, X_train, y_train, cv = 10, n_jobs=2,␣
 ↪scoring='precision')
  weight_dt_results.append({'Weight': w, 'Precision': dtwcv.mean()})
```

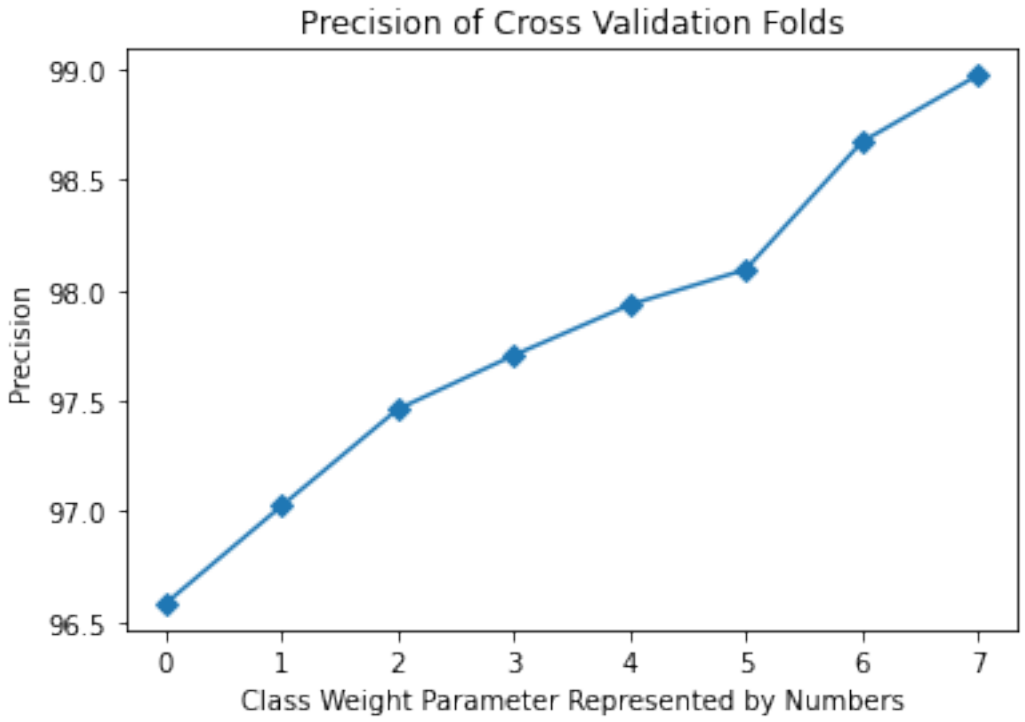
[ ]:
```
weight_dt_results_df = pd.DataFrame(weight_dt_results)
weight_dt_results_df
```

[ ]:

	Weight	Precision
0	{0: 0.5, 1: 0.5}	0.965826
1	{0: 0.6, 1: 0.4}	0.970242
2	{0: 0.7, 1: 0.3}	0.974611
3	{0: 0.75, 1: 0.25}	0.977052
4	{0: 0.8, 1: 0.2}	0.979339
5	{0: 0.85, 1: 0.15}	0.980934
6	{0: 0.9, 1: 0.1}	0.986701

```
7  {0: 0.95, 1: 0.05}   0.989725
```

[ ]:
```
tick = range(len(weight_dt_results_df['Weight']))
plt.plot(weight_dt_results_df['Precision']*100, marker = 'D')
plt.xlabel('Class Weight Parameter Represented by Numbers')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.xticks(ticks=tick)
plt.show()
```



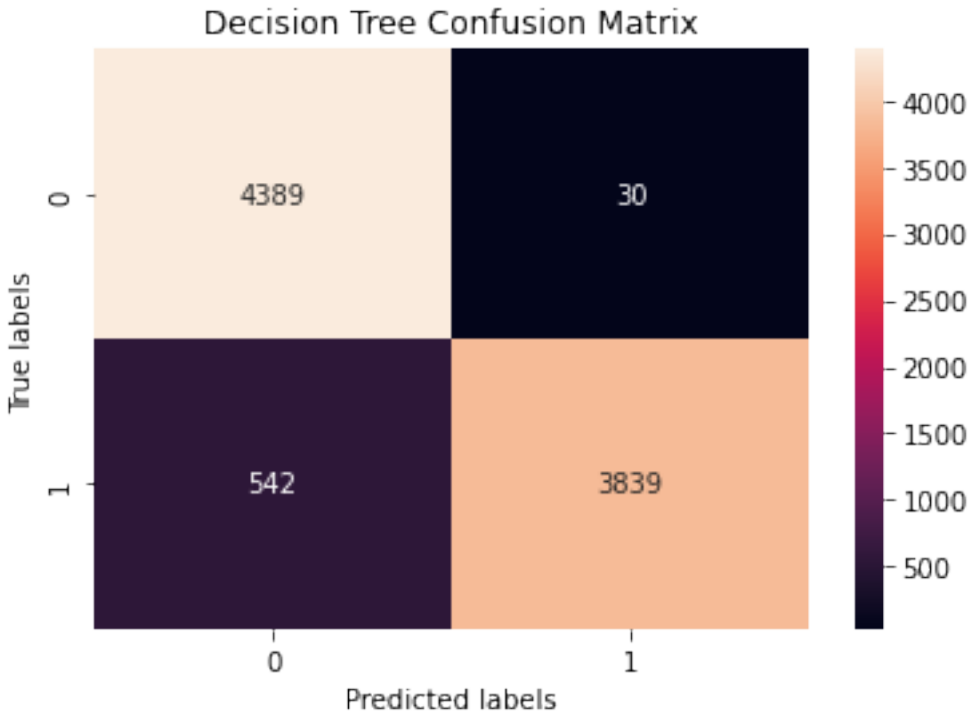
[ ]:
```
weight_dt = DecisionTreeClassifier(class_weight = {0: 0.95, 1: 0.05},
                                   criterion = 'gini', ccp_alpha=0.00005,
                                   random_state=3).fit(X_train,y_train)

weight_dt_pred = weight_dt.predict(X_test)

print(accuracy_score(y_test, weight_dt_pred))
print(metrics.precision_score(y_test, weight_dt_pred))
```

```
0.935
0.9922460584130266
```

```
[ ]: w_dt_acc = accuracy_score(y_test, weight_dt_pred)
     w_dt_prec = metrics.precision_score(y_test, weight_dt_pred)
     w_dt_rec = metrics.recall_score(y_test, weight_dt_pred)
     w_dt_f1 = metrics.f1_score(y_test, weight_dt_pred)
```

Weighted Decision Tree Confusion Matrix

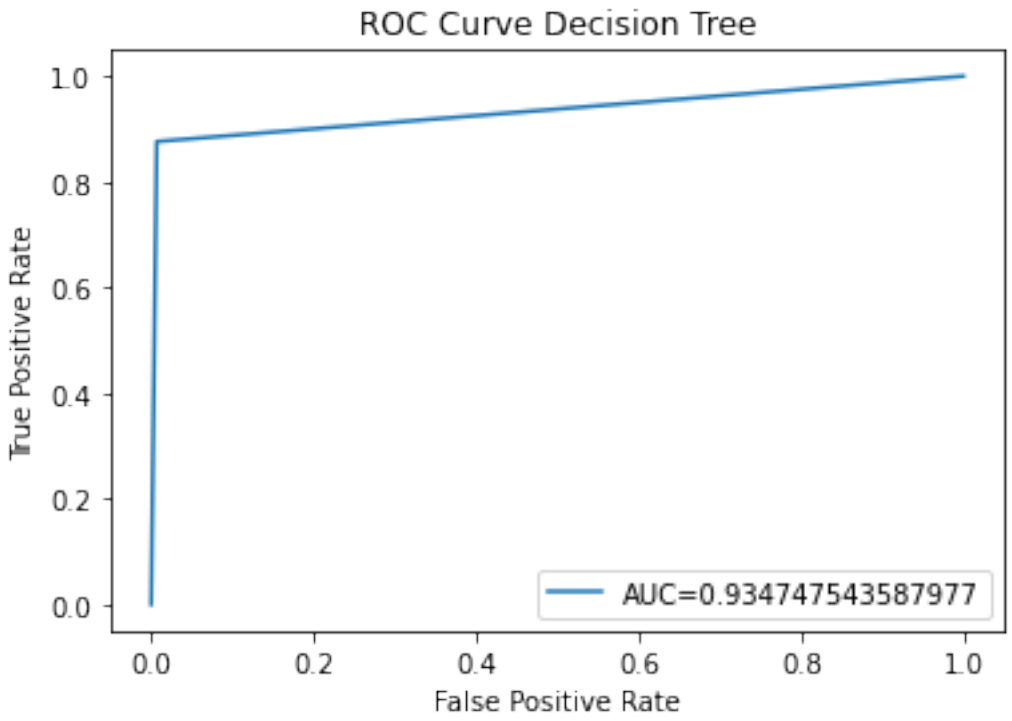
```
[ ]: weight_dt_cm = confusion_matrix(y_test, weight_dt_pred, labels = [0,1])


     ax = plt.subplot()
     sns.heatmap(weight_dt_cm, annot=True, fmt='g', ax=ax)
     ax.set_xlabel('Predicted labels')
     ax.set_ylabel('True labels')
     ax.set_title('Decision Tree Confusion Matrix')
     ax.xaxis.set_ticklabels(['0', '1'])
     ax.yaxis.set_ticklabels(['0', '1'])
     plt.show()
```



Weighting the tree did offer an improved precision at the cost of the recall and overall accuracy of the model. The main issue is other models when weighted have found a better precision and better accuracy.

Weighted Decision Tree Roc Curve

```
[ ]: dtwfalsepr, dtwtruepr, _ = metrics.roc_curve(y_test,  weight_dt_pred)
dtwauc = metrics.roc_auc_score(y_test, weight_dt_pred)

plt.plot(dtwfalsepr,dtwtruepr,label="AUC="+str(dtwauc))
plt.title('ROC Curve Decision Tree')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```



0.5 Random Forest Classifier

```
[ ]: from sklearn.ensemble import RandomForestClassifier
```

0.5.1 Random Forest Max Depth Tuning

```
[ ]: RF_results = []

for d in range(2, 52):
```

```
 RFg = RandomForestClassifier(criterion='gini', max_depth = d).fit(X_train,␣
↪y_train)
 RFe = RandomForestClassifier(criterion='entropy', max_depth = d).fit(X_train,␣
↪y_train)
 rfcvg = cross_val_score(RFg, X_train, y_train, cv = 10, n_jobs=2,␣
↪scoring='precision')
 rfcve = cross_val_score(RFe, X_train, y_train, cv = 10, n_jobs=2,␣
↪scoring='precision')
 RF_results.append({'Depth': d, 'Gini Precision': rfcvg.mean(), 'Entropy␣
↪Precision': rfcve.mean()})
```

[ ]:
```
RF_results_df = pd.DataFrame(RF_results)
RF_results_df
```

[ ]:

	Depth	Gini Precision	Entropy Precision
0	2	0.917183	0.920703
1	3	0.928546	0.932540
2	4	0.937779	0.938326
3	5	0.945376	0.944780
4	6	0.945807	0.944022
5	7	0.948557	0.948084
6	8	0.949255	0.949567
7	9	0.953524	0.952274
8	10	0.957436	0.955845
9	11	0.958565	0.959451
10	12	0.960718	0.961113
11	13	0.962927	0.963219
12	14	0.963387	0.963120
13	15	0.964643	0.964239
14	16	0.965370	0.964905
15	17	0.965658	0.965114
16	18	0.965560	0.965734
17	19	0.966222	0.966105
18	20	0.966681	0.965589
19	21	0.966215	0.966292
20	22	0.966180	0.965362
21	23	0.966450	0.965985
22	24	0.967089	0.966452
23	25	0.968752	0.967185
24	26	0.969420	0.968762
25	27	0.969045	0.968664
26	28	0.969404	0.968366
27	29	0.968905	0.969121
28	30	0.969010	0.969379
29	31	0.970036	0.969393
30	32	0.969114	0.970135
31	33	0.969945	0.969662

32	34	0.969482	0.969473
33	35	0.969106	0.969472
34	36	0.970236	0.968911
35	37	0.969353	0.969489
36	38	0.970493	0.969658
37	39	0.969180	0.969657
38	40	0.969481	0.969186
39	41	0.969674	0.969015
40	42	0.969565	0.970129
41	43	0.969556	0.969208
42	44	0.969395	0.969655
43	45	0.970339	0.969834
44	46	0.970332	0.969089
45	47	0.970594	0.970143
46	48	0.970398	0.968908
47	49	0.970120	0.968717
48	50	0.969966	0.969948
49	51	0.969588	0.968635

0.5.2 Random Forest Alpha Tuning

[ ]:
```
alpha = [0.000001,0.0000025,0.000005,0.00001,0.00005,0.0001,0.0005,0.001]
rfa_results = []

for a in alpha:
  rfga = RandomForestClassifier(criterion='gini', ccp_alpha = a).fit(X_train,␣
 ↪y_train)
  rfea = RandomForestClassifier(criterion='entropy', ccp_alpha = a).
 ↪fit(X_train, y_train)
  rfcvga = cross_val_score(rfga, X_train, y_train, cv = 10, n_jobs=2,␣
 ↪scoring='precision')
  rfcvea = cross_val_score(rfea, X_train, y_train, cv = 10, n_jobs=2,␣
 ↪scoring='precision')
  rfa_results.append({'Alpha': a, 'Gini Precision': rfcvga.mean(), 'Entropy␣
 ↪Precision': rfcvea.mean()})
```

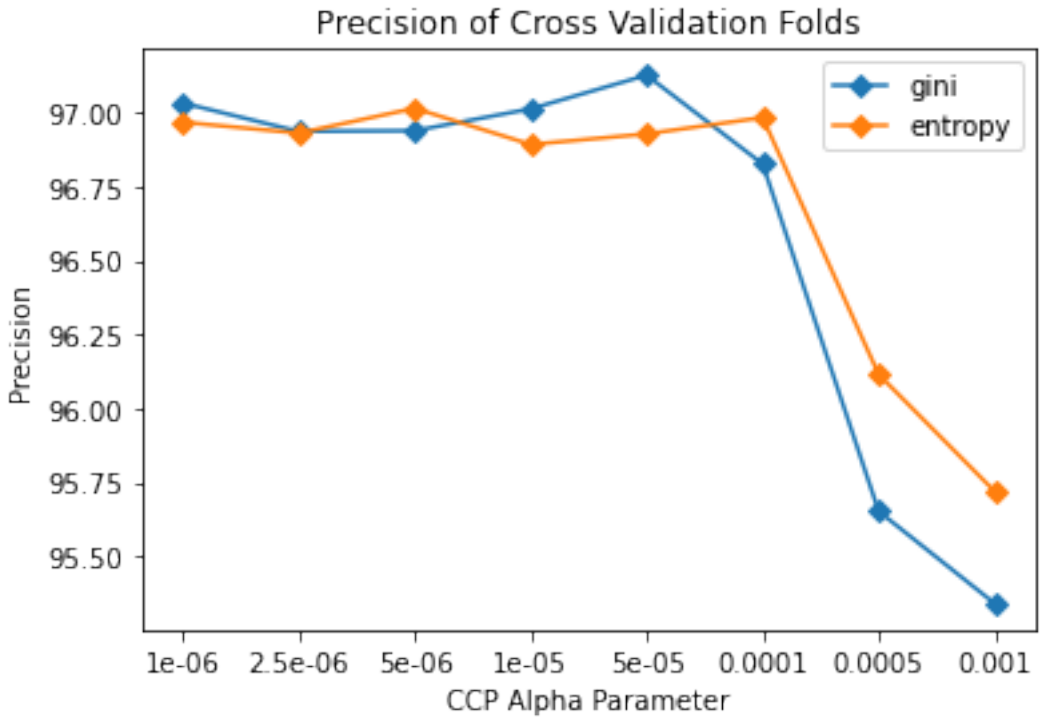
[ ]:
```
rfa_results_df = pd.DataFrame(rfa_results)
rfa_results_df
```

[ ]:

	Alpha	Gini Precision	Entropy Precision
0	0.000001	0.970321	0.969671
1	0.000003	0.969360	0.969302
2	0.000005	0.969382	0.970141
3	0.000010	0.970122	0.968917
4	0.000050	0.971260	0.969275

```
5   0.000100          0.968258          0.969828
6   0.000500          0.956519          0.961144
7   0.001000          0.953402          0.957191
```

```
[ ]: tick = range(len(rfa_results_df['Alpha']))
plt.plot(rfa_results_df['Gini Precision']*100, marker = 'D', label = 'gini')
plt.plot(rfa_results_df['Entropy Precision']*100, marker = 'D', label =␣
 ↪'entropy')
plt.xlabel('CCP Alpha Parameter')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.legend()
plt.xticks(ticks=tick, labels=rfa_results_df['Alpha'])
plt.show()
```



The CCP_alpha tune offered better results, it will be used going forward in building the final RF and weighted RF model. Gini criterion was the best splitting process like the Decision Tree and an alpha of 0.00005 again was the best complexity control parameter.

0.6 Random Forest Final Model

[ ]:
```
final_rf = RandomForestClassifier(criterion = 'gini',ccp_alpha = 0.00005).
↪fit(X_train,y_train)
```

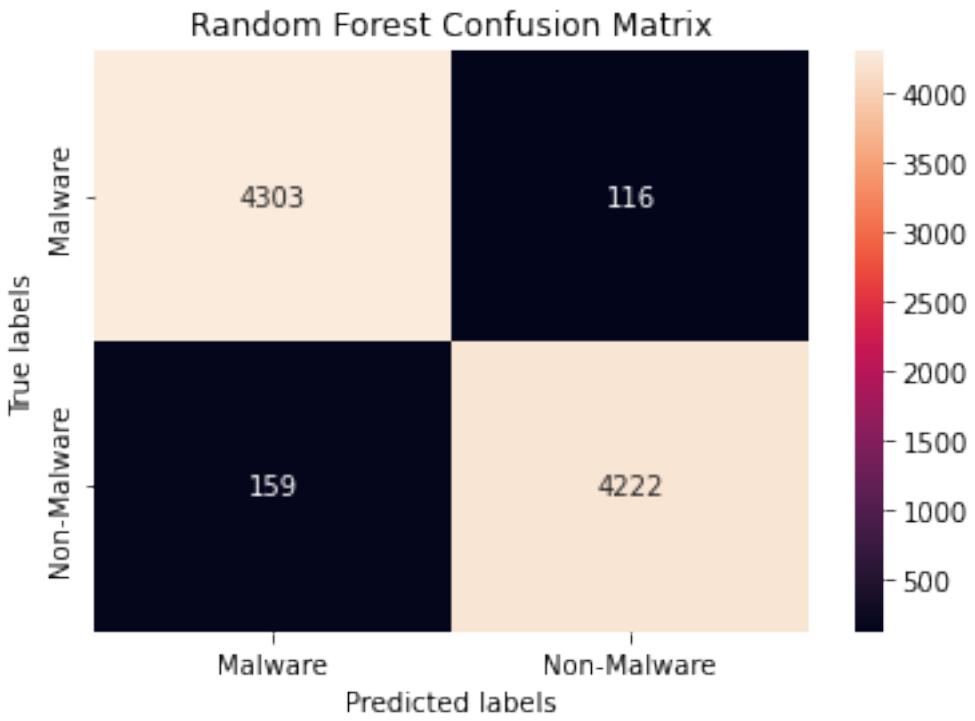
[ ]:
```
pred_rf = final_rf.predict(X_test)
print(accuracy_score(y_test, pred_rf))
print(metrics.precision_score(y_test, pred_rf))
```

```
0.96875
0.9732595666205625
```

[ ]:
```
rf_acc = accuracy_score(y_test, pred_rf)
rf_prec = metrics.precision_score(y_test, pred_rf)
rf_rec = metrics.recall_score(y_test, pred_rf)
rf_f1 = metrics.f1_score(y_test, pred_rf)
```

Random Forest Confusion Matrix

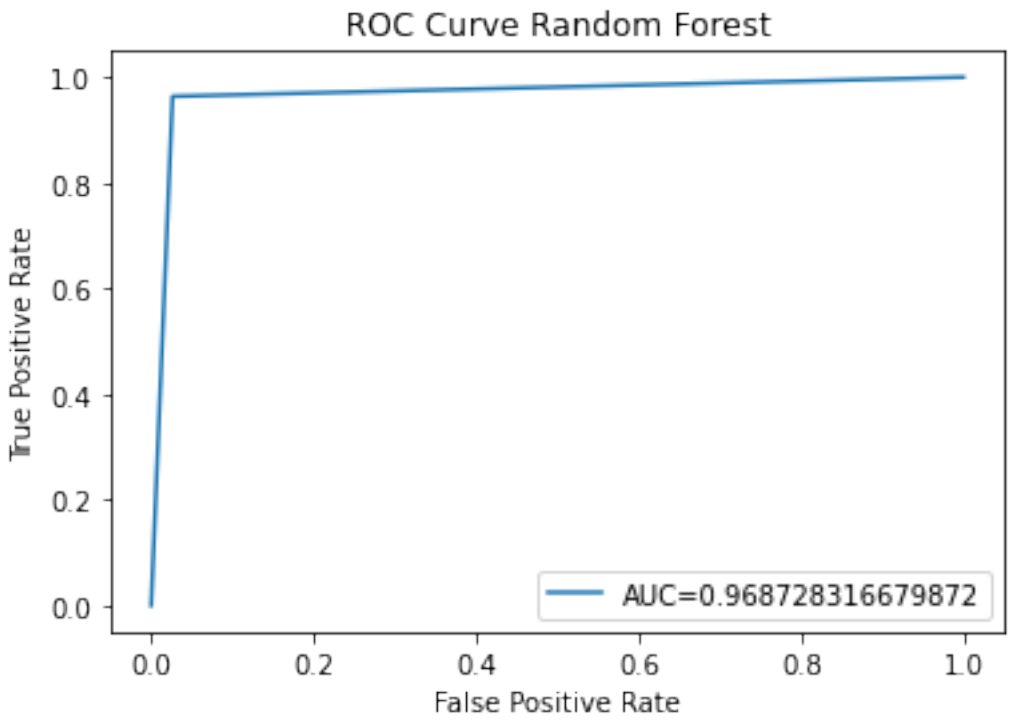
[ ]:
```
fin_rf_cm = confusion_matrix(y_test, pred_rf, labels = [0,1])


ax = plt.subplot()
sns.heatmap(fin_rf_cm, annot=True, fmt='g', ax=ax)
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Random Forest Confusion Matrix')
ax.xaxis.set_ticklabels(['Malware', 'Non-Malware'])
ax.yaxis.set_ticklabels(['Malware', 'Non-Malware'])
plt.show()
```



Of the non-weighted final models, the Random Forest offered the best precision, with false negatives nearly dipping to 100. Still it seems that weighting the classes could offer a better solution to our problem.

Random Forest ROC Curve

[ ]:
```
rffalsepr, rftruepr, _ = metrics.roc_curve(y_test,  pred_rf)
rfauc = metrics.roc_auc_score(y_test, pred_rf)

plt.plot(rffalsepr,rftruepr,label="AUC="+str(rfauc))
plt.title('ROC Curve Random Forest')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```



0.6.1 Weighted Random Forest

[ ]:
```
weight_rf_results = []

for w in weight:
  weight_rf = RandomForestClassifier(class_weight = w, criterion =␣
 ↪'entropy',max_depth = 37).fit(X_train,y_train)
  rfwcv= cross_val_score(weight_rf, X_train, y_train, cv = 10, n_jobs=2,␣
 ↪scoring='precision')
  weight_rf_results.append({'Weight': w, 'Precision': rfwcv.mean()})
```

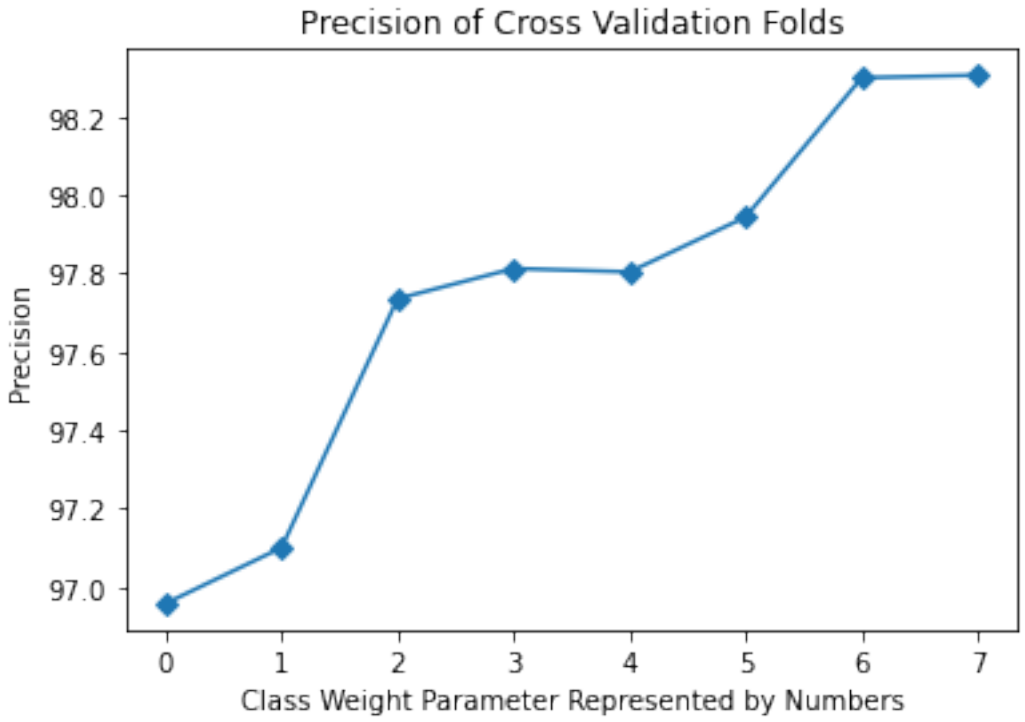
[ ]:
```
weight_rf_results_df = pd.DataFrame(weight_rf_results)
weight_rf_results_df
```

[ ]:

	Weight	Precision
0	{0: 0.5, 1: 0.5}	0.969583
1	{0: 0.6, 1: 0.4}	0.971009
2	{0: 0.7, 1: 0.3}	0.977360
3	{0: 0.75, 1: 0.25}	0.978131
4	{0: 0.8, 1: 0.2}	0.978044
5	{0: 0.85, 1: 0.15}	0.979445
6	{0: 0.9, 1: 0.1}	0.982998

```
7  {0: 0.95, 1: 0.05}   0.983064
```

```
[ ]: tick = range(len(weight_rf_results_df['Weight']))
plt.plot(weight_rf_results_df['Precision']*100, marker = 'D')
plt.xlabel('Class Weight Parameter Represented by Numbers')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.xticks(ticks=tick)
plt.show()
```



Again, the 95:05 class weights offered the best precision.

```
[ ]: weight_rf = RandomForestClassifier(class_weight = {0:0.9,1:0.1}, criterion =␣
 ↪'gini',ccp_alpha = 0.00005, random_state=3).fit(X_train,y_train)
pred_w_rf = weight_rf.predict(X_test)
print(accuracy_score(y_test, pred_w_rf))
print(metrics.precision_score(y_test, pred_w_rf))
```

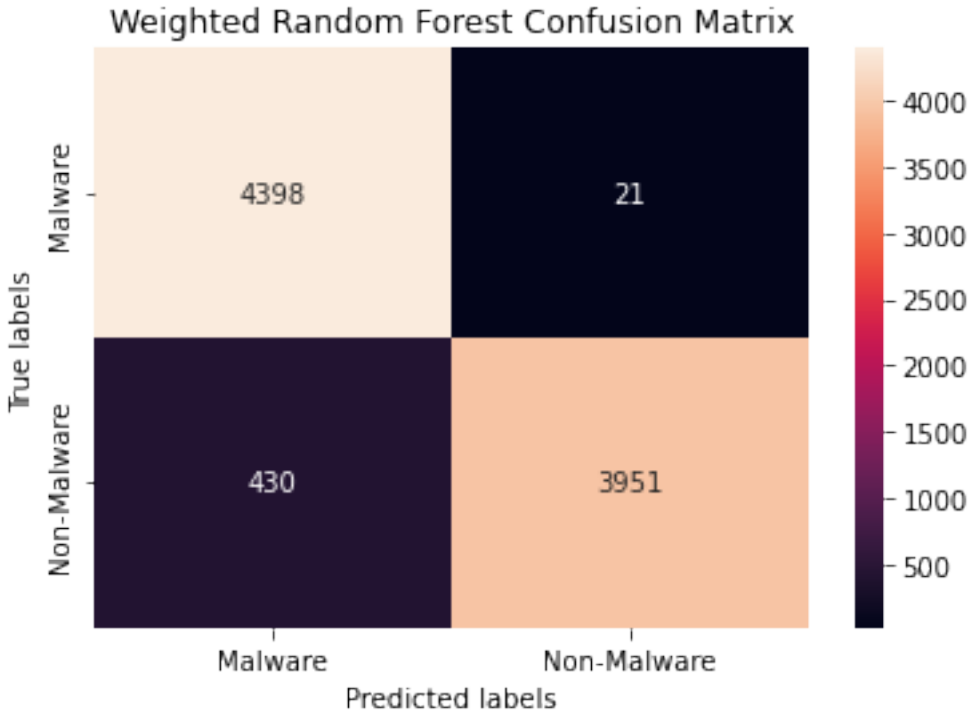
```
0.94875
0.9947129909365559
```

```
[ ]: w_rf_acc = accuracy_score(y_test, pred_w_rf)
w_rf_prec = metrics.precision_score(y_test, pred_w_rf)
w_rf_rec = metrics.recall_score(y_test, pred_w_rf)
```

```
w_rf_f1 = metrics.f1_score(y_test, pred_w_rf)
```

Weighted Random Forest Confusion Matrix

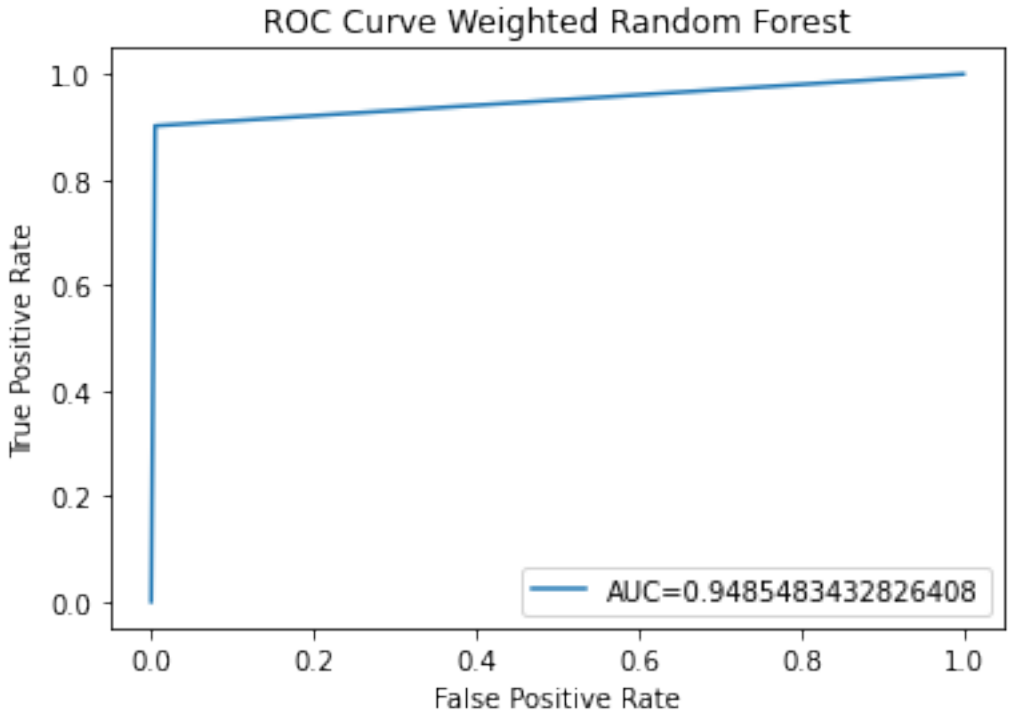
```
[ ]: w_rf_cm = confusion_matrix(y_test, pred_w_rf, labels = [0,1])


ax = plt.subplot()
sns.heatmap(w_rf_cm, annot=True, fmt='g', ax=ax)
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Weighted Random Forest Confusion Matrix')
ax.xaxis.set_ticklabels(['Malware', 'Non-Malware'])
ax.yaxis.set_ticklabels(['Malware', 'Non-Malware'])
plt.show()
```



The false negatives were reduced to 21 applications while the false positives only increased to 430, compared to other weighted models that were over 600 false positives. This model can potentially flag an extremely high percentage of malware applications, about 99.4% while maintaining a high overall accuracy of 94.5% to not spam consumers with flagged malware applications for their attention

Weighted Random Forest Roc Curve

```
[ ]: rfwfalsepr, rfwtruepr, _ = metrics.roc_curve(y_test,  pred_w_rf)
rfwauc = metrics.roc_auc_score(y_test, pred_w_rf)

plt.plot(rfwfalsepr,rfwtruepr,label="AUC="+str(rfwauc))
plt.title('ROC Curve Weighted Random Forest')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```



0.7 Extra Tree Classifier

```
[ ]: from sklearn.ensemble import ExtraTreesClassifier
```

0.7.1 Extra Tree Depth Tuning

```
[ ]: ET_results = []

for d in range(2, 52):
  etg = ExtraTreesClassifier(criterion='gini', max_depth = d, random_state=3).
 ↪fit(X_train, y_train)
```

```
  ete = ExtraTreesClassifier(criterion='entropy', max_depth = d,random_state=3).
↪fit(X_train, y_train)
  etcvg = cross_val_score(etg, X_train, y_train, cv = 10, n_jobs=2,␣
↪scoring='precision')
  etcve = cross_val_score(ete, X_train, y_train, cv = 10, n_jobs=2,␣
↪scoring='precision')
  ET_results.append({'Depth': d, 'Gini Precision': etcvg.mean(), 'Entropy␣
↪Precision': etcve.mean()})
```

[ ]:
```
et_results_df = pd.DataFrame(ET_results)
et_results_df
```

0.7.2 Extra Tree Alpha Tuning

[ ]:
```
alpha = [0.000001,0.0000025,0.000005,0.00001,0.00005,0.0001,0.0005,0.001]
eta_results = []

for a in alpha:
  etga = ExtraTreesClassifier(criterion='gini', ccp_alpha = a,random_state=3).
↪fit(X_train, y_train)
  etea = ExtraTreesClassifier(criterion='entropy', ccp_alpha =␣
↪a,random_state=3).fit(X_train, y_train)
  etcvga = cross_val_score(etga, X_train, y_train, cv = 10, n_jobs=2,␣
↪scoring='precision')
  etcvea = cross_val_score(etea, X_train, y_train, cv = 10, n_jobs=2,␣
↪scoring='precision')
  eta_results.append({'Alpha': a, 'Gini Precision': etcvga.mean(), 'Entropy␣
↪Precision': etcvea.mean()})
```

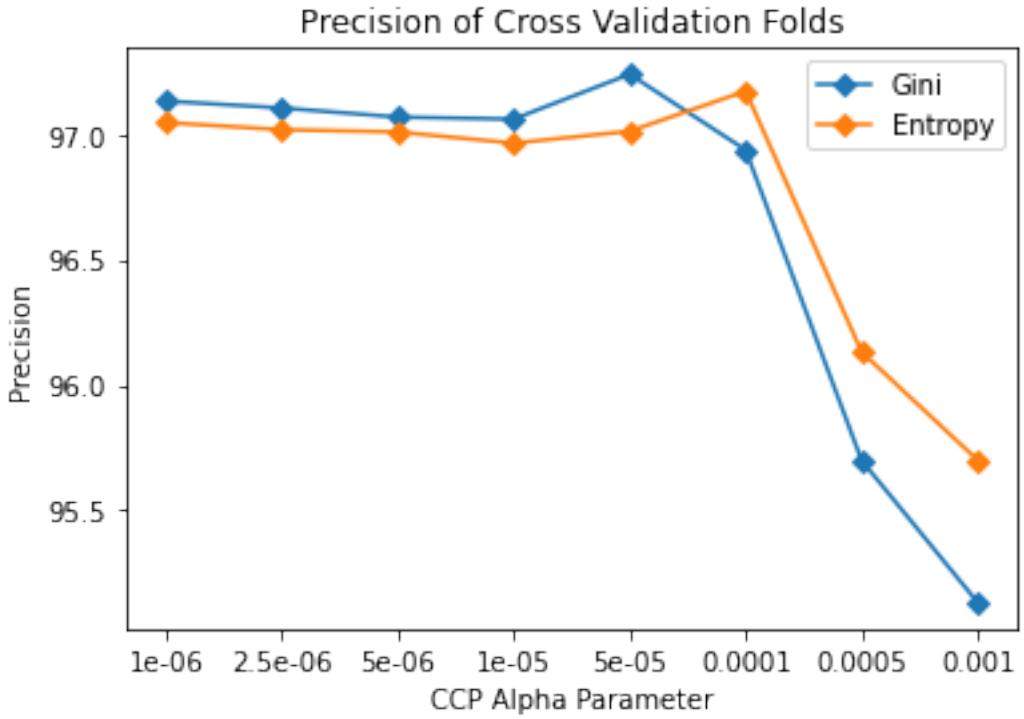
[ ]:
```
eta_results_df = pd.DataFrame(eta_results)
eta_results_df
```

[ ]:

	Alpha	Gini Precision	Entropy Precision
0	0.000001	0.971325	0.970467
1	0.000003	0.971049	0.970184
2	0.000005	0.970685	0.970092
3	0.000010	0.970604	0.969646
4	0.000050	0.972396	0.970121
5	0.000100	0.969374	0.971721
6	0.000500	0.956990	0.961317
7	0.001000	0.951330	0.957006

[ ]:
```
tick = range(len(eta_results_df['Alpha']))
plt.plot(eta_results_df['Gini Precision']*100, marker = 'D', label = 'Gini')
```

```
plt.plot(eta_results_df['Entropy Precision']*100, marker = 'D', label =␣
↪'Entropy')
plt.xlabel('CCP Alpha Parameter')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.legend()
plt.xticks(ticks=tick, labels=eta_results_df['Alpha'])
plt.show()
```



Similar to the RF, ET offered better results by tuning the complexity on CCP alpha rather than max_depth. Again Gini splits suited the data better for the criterion, and the most precise CCP Alpha tune is again 0.00005 for Extra Trees, same as DT and RF.

0.7.3 Extra Tree Final Model

[ ]:
```
final_et = ExtraTreesClassifier(criterion='gini',ccp_alpha = 0.
↪00005,random_state=3).fit(X_train,y_train)
```

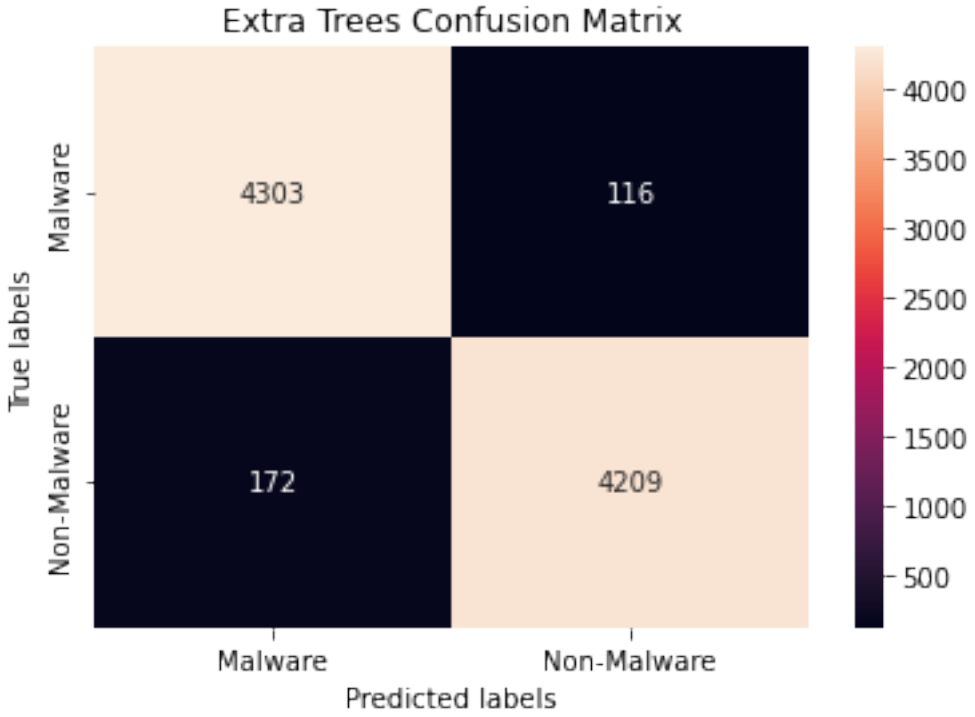
[ ]:
```
pred_et = final_et.predict(X_test)

print(accuracy_score(y_test, pred_et))
print(metrics.precision_score(y_test, pred_et))
```

```
0.9672727272727273
0.9731791907514451
```

```
[ ]: et_acc = accuracy_score(y_test, pred_et)
et_prec = metrics.precision_score(y_test, pred_et)
et_rec = metrics.recall_score(y_test, pred_et)
et_f1 = metrics.f1_score(y_test, pred_et)
```

Extra Trees Confusion Matrix

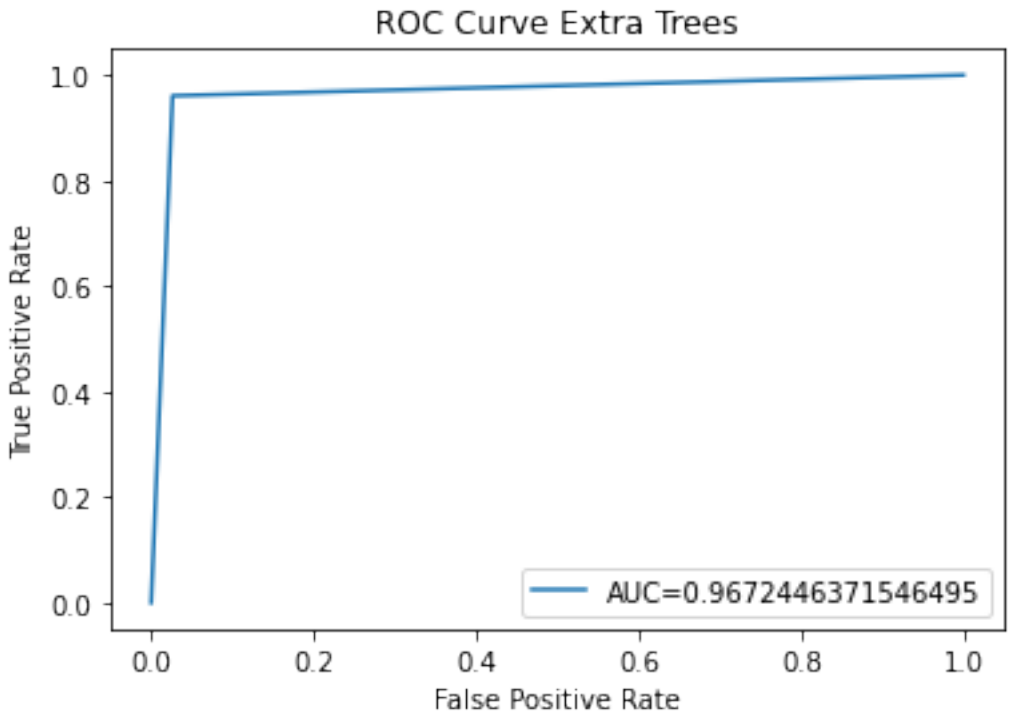
```
[ ]: fin_et_cm = confusion_matrix(y_test, pred_et, labels = [0,1])


ax = plt.subplot()
sns.heatmap(fin_et_cm, annot=True, fmt='g', ax=ax)
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Extra Trees Confusion Matrix')
ax.xaxis.set_ticklabels(['Malware', 'Non-Malware'])
ax.yaxis.set_ticklabels(['Malware', 'Non-Malware'])
plt.show()
```



The Extra Trees Final model offered similar results to the random forest. A strong model for both precision that also minimizes false positive flagging with its overall accuracy.

Extra Trees ROC Curve

[ ]:
```
etfalsepr, ettruepr, _ = metrics.roc_curve(y_test,  pred_et)
etauc = metrics.roc_auc_score(y_test, pred_et)

plt.plot(etfalsepr,ettruepr,label="AUC="+str(etauc))
plt.title('ROC Curve Extra Trees')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```

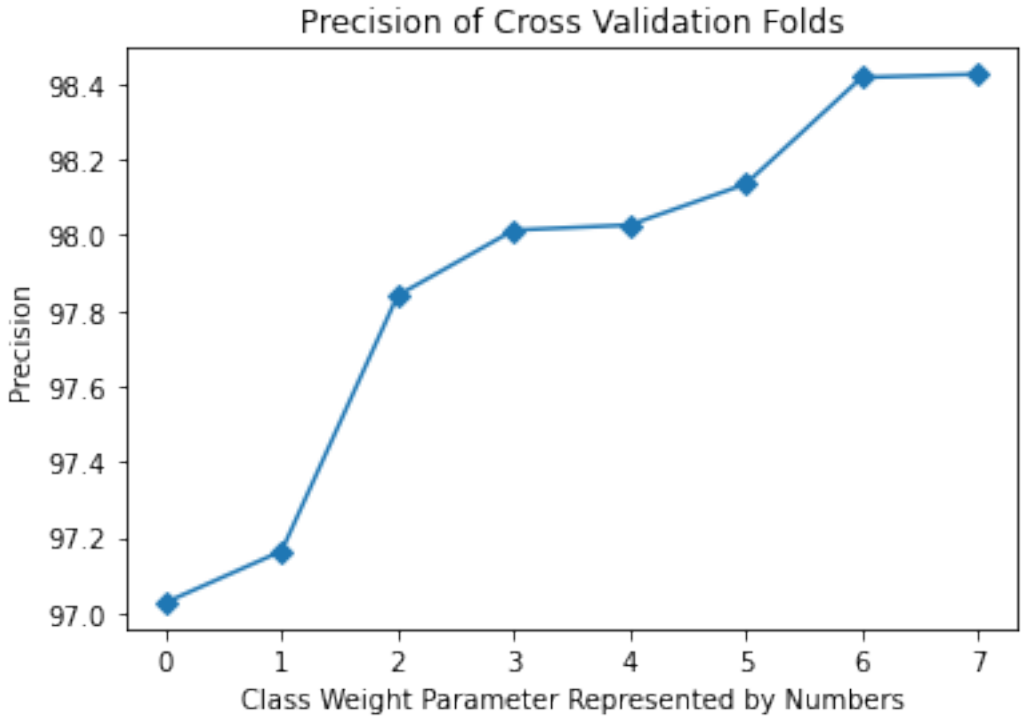


0.7.4 Weighted Extra Trees

[ ]:
```
weight_et_results = []

for w in weight:
  weight_et = ExtraTreesClassifier(class_weight = w, criterion =␣
 ↪'entropy',max_depth = 37,random_state=3).fit(X_train,y_train)
  etwcv= cross_val_score(weight_et, X_train, y_train, cv = 10, n_jobs=2,␣
 ↪scoring='precision')
  weight_et_results.append({'Weight': w, 'Precision': etwcv.mean()})
```

[ ]:
```
weight_et_results_df = pd.DataFrame(weight_et_results)
```

[ ]:
```
tick = range(len(weight_et_results_df['Weight']))
plt.plot(weight_et_results_df['Precision']*100, marker = 'D')
plt.xlabel('Class Weight Parameter Represented by Numbers')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.xticks(ticks=tick)
plt.show()
```



The chosen weight is again 95:05 as the most optimized precision.

[ ]:
```
weight_et = ExtraTreesClassifier(criterion='gini',ccp_alpha = 0.
↪00005,class_weight = {0:.95,1:.05},random_state=3).fit(X_train,y_train)
```

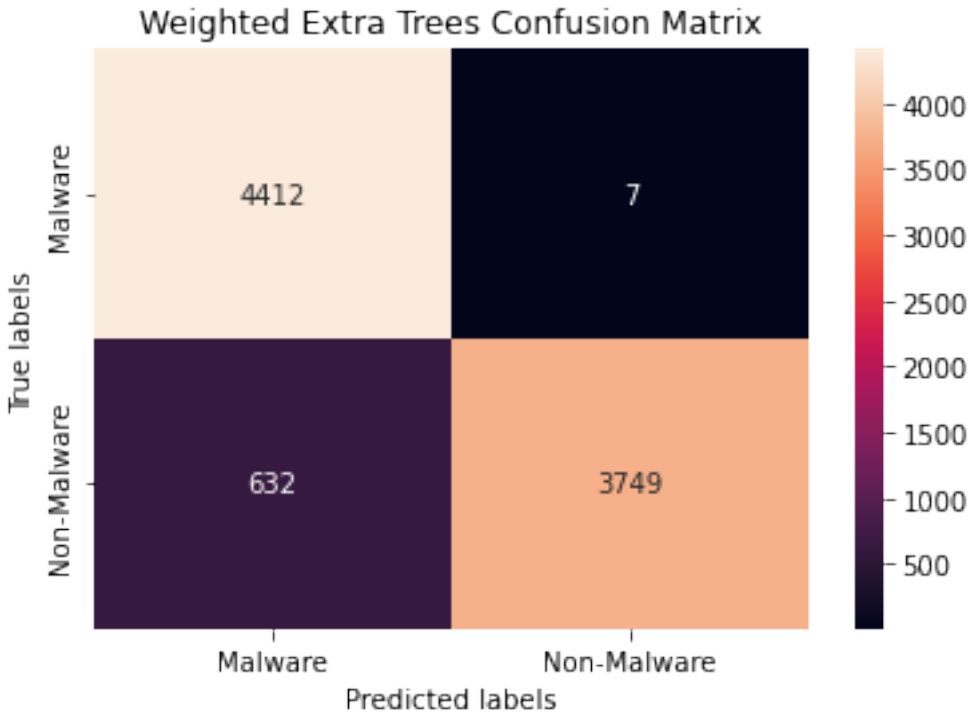
[ ]:
```
pred_weight_et = weight_et.predict(X_test)

print(accuracy_score(y_test, pred_weight_et))
print(metrics.precision_score(y_test, pred_weight_et))
```

```
0.9273863636363636
0.998136315228967
```

```
[ ]: w_et_acc = accuracy_score(y_test, pred_weight_et)
     w_et_prec = metrics.precision_score(y_test, pred_weight_et)
     w_et_rec = metrics.recall_score(y_test, pred_weight_et)
     w_et_f1 = metrics.f1_score(y_test, pred_weight_et)
```

Weighted Extra Trees Confusion Matrix

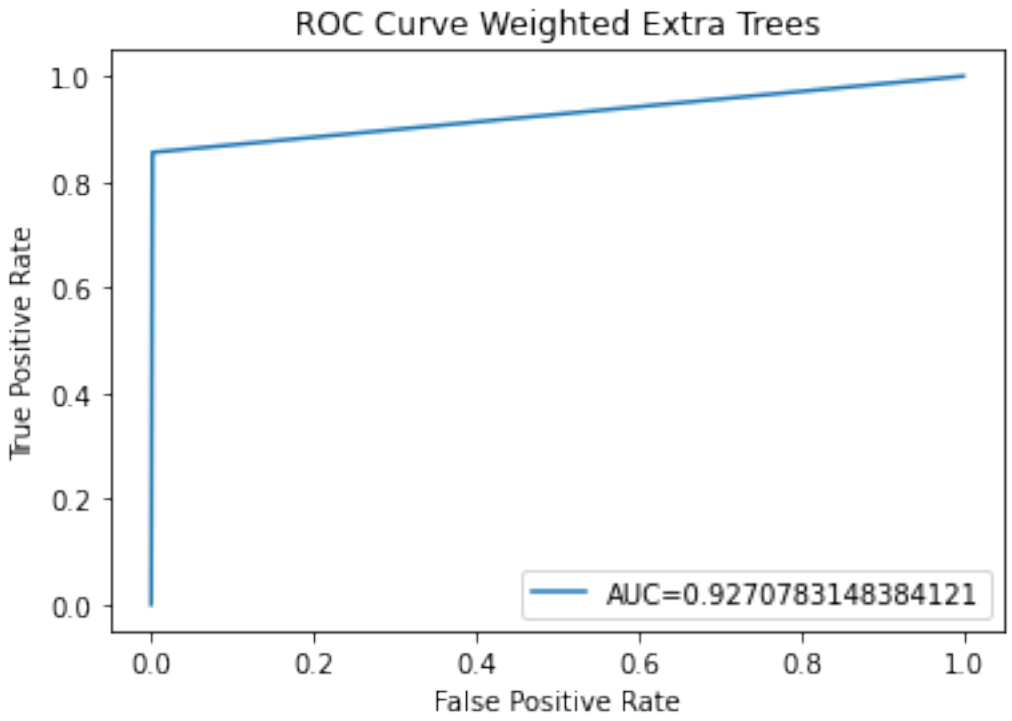
```
[ ]: weight_et_cm = confusion_matrix(y_test, pred_weight_et, labels = [0,1])


     ax = plt.subplot()
     sns.heatmap(weight_et_cm, annot=True, fmt='g', ax=ax)
     ax.set_xlabel('Predicted labels')
     ax.set_ylabel('True labels')
     ax.set_title('Weighted Extra Trees Confusion Matrix')
     ax.xaxis.set_ticklabels(['Malware', 'Non-Malware'])
     ax.yaxis.set_ticklabels(['Malware', 'Non-Malware'])
     plt.show()
```



The Extra Trees offers both a strong precision with only 7 false negative classifications and a decent overall accuracy at 92%. Compared to the RF, the accuracy decrease cause about 200 more false positives while only removing 9 false negatives. Ultimately this trade off may not be worthwhile when flagging applications to notify consumers of when compared to the Random Forest.

Weighted Extra Trees ROC Curve

[ ]:
```
etwfalsepr, etwtruepr, _ = metrics.roc_curve(y_test,  pred_weight_et)
etwauc = metrics.roc_auc_score(y_test, pred_weight_et)

plt.plot(etwfalsepr,etwtruepr,label="AUC="+str(etwauc))
plt.title('ROC Curve Weighted Extra Trees')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```



1 Logistic Regression

[ ]:
```
from sklearn.linear_model import LogisticRegression
```

1.0.1 Tuning For Penalty

[ ]:
```
lr_results = []

lrl1 = LogisticRegression(penalty = 'l1', solver = 'saga',random_state=3).
 ↪fit(X_train, y_train)
lrl2 = LogisticRegression(penalty = 'l2',random_state=3).fit(X_train, y_train)
l1cv = cross_val_score(lrl1, X_train, y_train, cv = 10, n_jobs = 2, scoring =␣
 ↪'precision')
l2cv = cross_val_score(lrl2, X_train, y_train, cv = 10, n_jobs = 2, scoring =␣
 ↪'precision')
lr_results.append({'LR L1 Precision': l1cv.mean(), 'LR L2 Precision': l2cv.
 ↪mean()})
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:354:
ConvergenceWarning: The max_iter was reached which means the coef_ did not
converge
  ConvergenceWarning,
```

[ ]:
```
pd.DataFrame(lr_results)
```

[ ]:

	LR L1 Precision	LR L2 Precision
0	0.952893	0.951387

L1 penalty with saga solver offered stronger precision.

1.0.2 Final Logistic Regression Model

[ ]:
```
lr = LogisticRegression(penalty = 'l1',solver = 'saga', random_state=3).
 ↪fit(X_train, y_train)
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/linear_model/_sag.py:354:
ConvergenceWarning: The max_iter was reached which means the coef_ did not
converge
  ConvergenceWarning,
```

[ ]:
```
lr_pred = lr.predict(X_test)
print(accuracy_score(y_test, lr_pred))
print(metrics.precision_score(y_test, lr_pred))
```

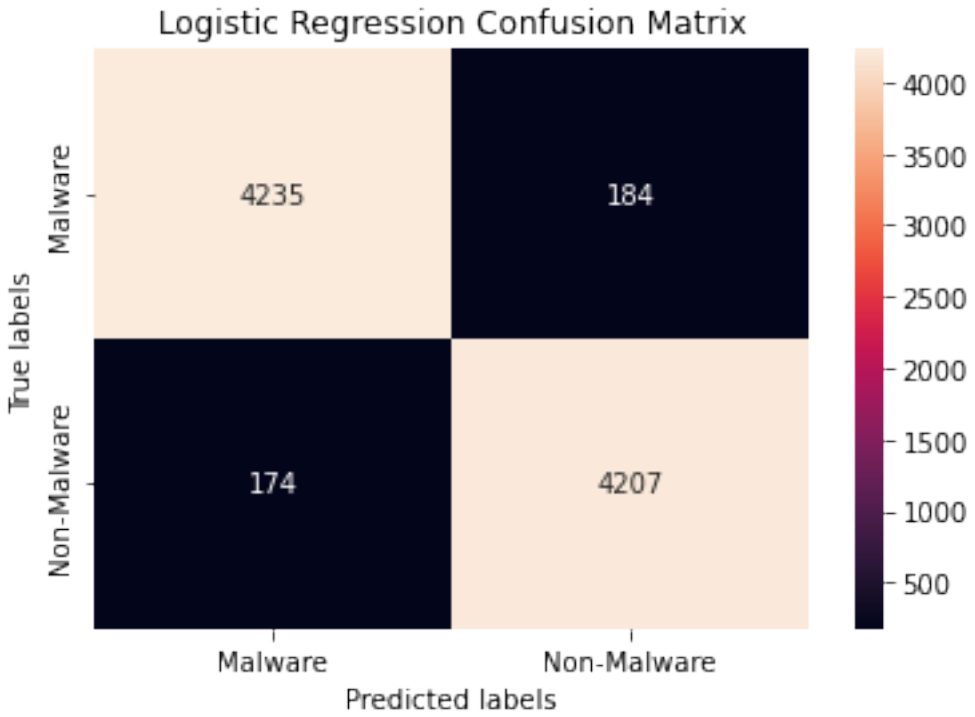
```
0.9593181818181818
0.95809610567069
```

[ ]:
```
lr_acc = accuracy_score(y_test, lr_pred)
lr_prec = metrics.precision_score(y_test, lr_pred)
lr_rec = metrics.recall_score(y_test, lr_pred)
```

```
lr_f1 = metrics.f1_score(y_test, lr_pred)
```

LR Confusion Matrix

```
[ ]: lrcm = confusion_matrix(y_test, lr.predict(X_test), labels = [0,1])


ax = plt.subplot()
sns.heatmap(lrcm, annot=True, fmt='g', ax=ax)
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Logistic Regression Confusion Matrix')
ax.xaxis.set_ticklabels(['Malware', 'Non-Malware'])
ax.yaxis.set_ticklabels(['Malware', 'Non-Malware'])
plt.show()
```

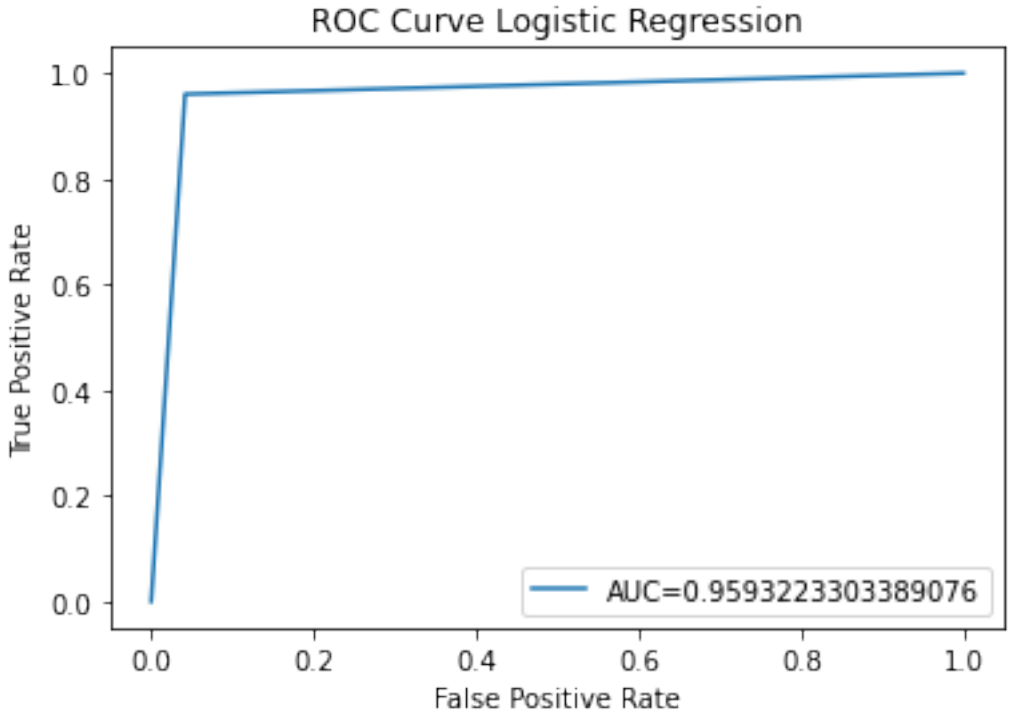


This model offered balanced results that did not meet the standards of any other model.

LR ROC Curve

```
[ ]: lrfalsepr, lrtruepr, _ = metrics.roc_curve(y_test,  lr_pred)
lrauc = metrics.roc_auc_score(y_test, lr_pred)

plt.plot(lrfalsepr,lrtruepr,label="AUC="+str(lrauc))
```

```
plt.title('ROC Curve Logistic Regression')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```



1.0.3 Weighted Logistic Regression Model

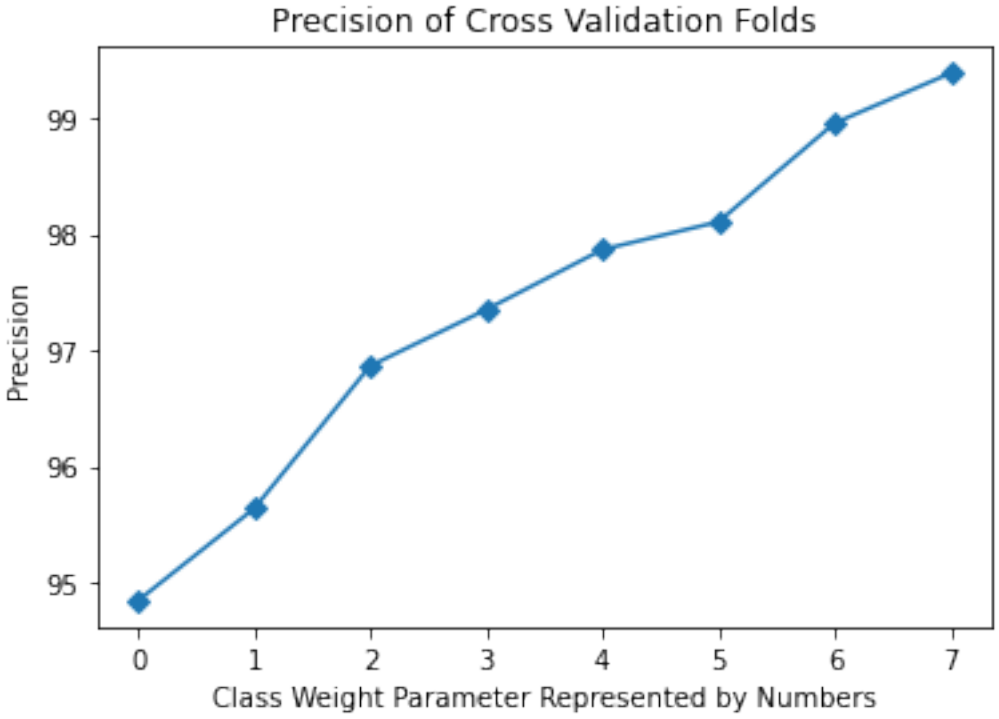
[ ]:
```
lrw_results = []

for w in weight:
  wlr = LogisticRegression(class_weight = w, random_state=3).fit(X_train,␣
 ↪y_train)
  wlrcv = cross_val_score(wlr, X_train, y_train, cv = 10, n_jobs = 2, scoring =␣
 ↪'precision')
  lrw_results.append({'weight': w,'Precision': wlrcv.mean()})
```

[ ]:
```
lrw_results_df = pd.DataFrame(lrw_results)
lrw_results_df
```

[ ]:
```
                weight  Precision
0    {0: 0.5, 1: 0.5}   0.948419
```

```
1    {0: 0.6, 1: 0.4}   0.956354
2    {0: 0.7, 1: 0.3}   0.968652
3   {0: 0.75, 1: 0.25}   0.973507
4    {0: 0.8, 1: 0.2}   0.978677
5   {0: 0.85, 1: 0.15}   0.981070
6    {0: 0.9, 1: 0.1}   0.989579
7   {0: 0.95, 1: 0.05}   0.993890
```

[ ]:
```
tick = range(len(lrw_results_df['weight']))
plt.plot(lrw_results_df['Precision']*100, marker = 'D')
plt.xlabel('Class Weight Parameter Represented by Numbers')
plt.ylabel('Precision')
plt.title('Precision of Cross Validation Folds')
plt.xticks(ticks=tick)
plt.show()
```



[ ]:
```
weight_lr = LogisticRegression(class_weight = {0:.95,1:.05}, random_state=3).
↪fit(X_train, y_train)
```

[ ]:
```
wlr_pred = weight_lr.predict(X_test)
print(accuracy_score(y_test, wlr_pred))
print(metrics.precision_score(y_test, wlr_pred))
```

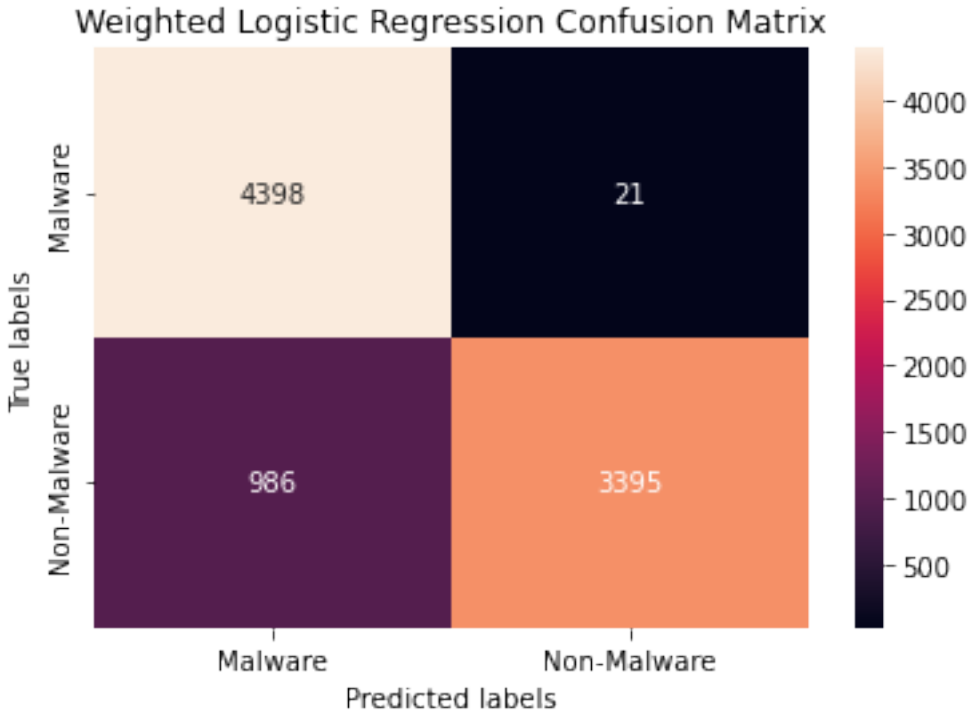
```
0.8855681818181819
```

```
0.9938524590163934
```

```
[ ]: w_lr_acc = accuracy_score(y_test, wlr_pred)
w_lr_prec = metrics.precision_score(y_test, wlr_pred)
w_lr_rec = metrics.recall_score(y_test, wlr_pred)
w_lr_f1 = metrics.f1_score(y_test, wlr_pred)
```

Weighted LR Confusion Matrix

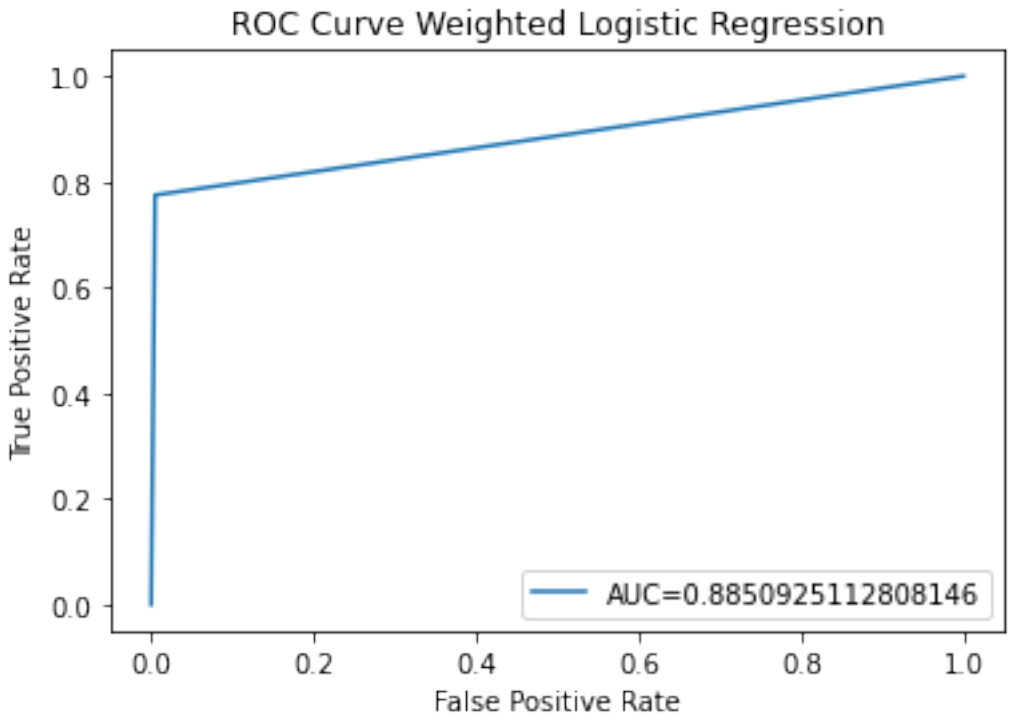
```
[ ]: lrwcm = confusion_matrix(y_test, wlr_pred, labels = [0,1])


ax = plt.subplot()
sns.heatmap(lrwcm, annot=True, fmt='g', ax=ax)
ax.set_xlabel('Predicted labels')
ax.set_ylabel('True labels')
ax.set_title('Weighted Logistic Regression Confusion Matrix')
ax.xaxis.set_ticklabels(['Malware', 'Non-Malware'])
ax.yaxis.set_ticklabels(['Malware', 'Non-Malware'])
plt.show()
```



The Weighted Logistic Regression made too many errors on the non-malware class to be further considered.

Weighted LR ROC Curve

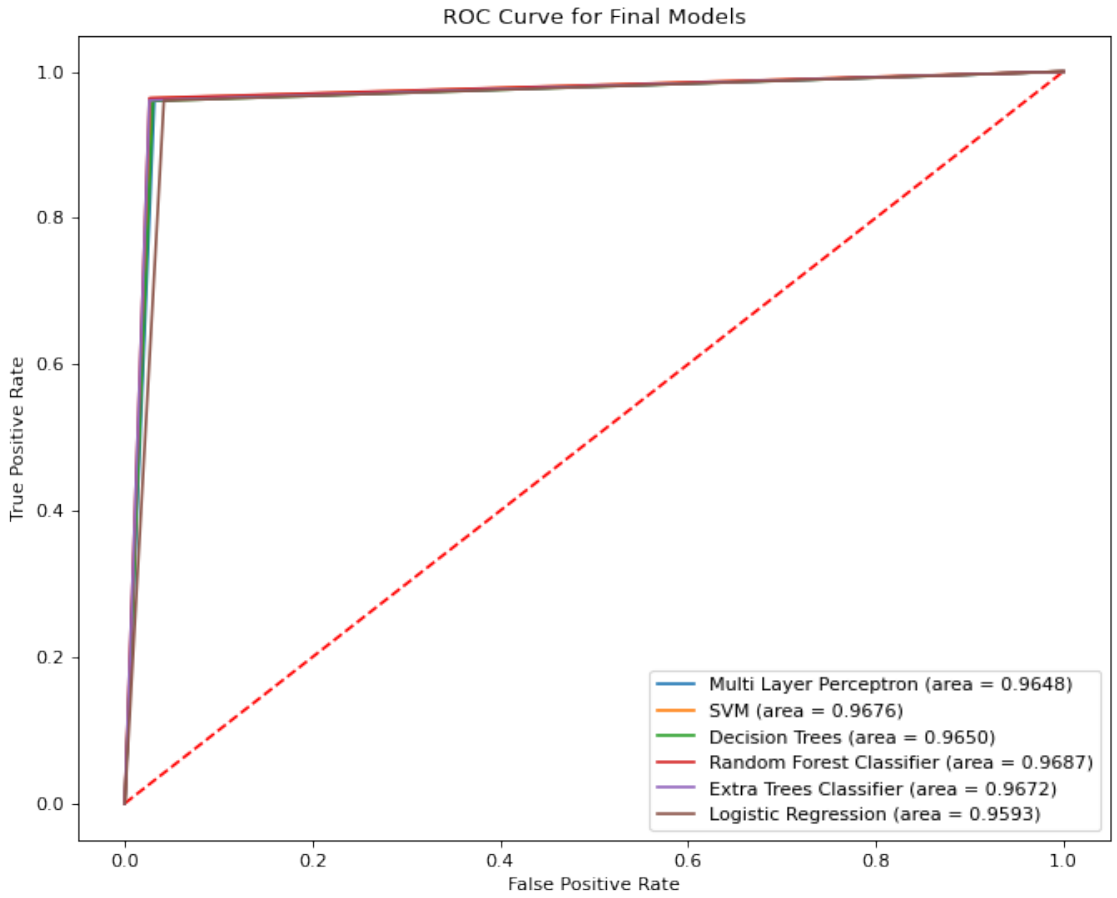
```
[ ]: wlrfalsepr, wlrtruepr, _ = metrics.roc_curve(y_test,  wlr_pred)
wlrauc = metrics.roc_auc_score(y_test, wlr_pred)

plt.plot(wlrfalsepr,wlrtruepr,label="AUC="+str(wlrauc))
plt.title('ROC Curve Weighted Logistic Regression')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.legend(loc=4)
plt.show()
```



1.1 ROC Curve Final Models

```
[ ]: plt.figure(figsize=(10,8),dpi=80)
plt.plot([0,1],[0,1],'r--')

plt.plot(mlpfalsepr, mlptruepr, label = 'Multi Layer Perceptron (area = %0.4f)'␣
 ↪%mlpauc);
plt.plot(svmfalsepr, svmtruepr, label = 'SVM (area = %0.4f)' %svmauc);
plt.plot(dtafalsepr, dtatruepr, label = 'Decision Trees (area = %0.4f)'␣
 ↪%dtaauc);
```

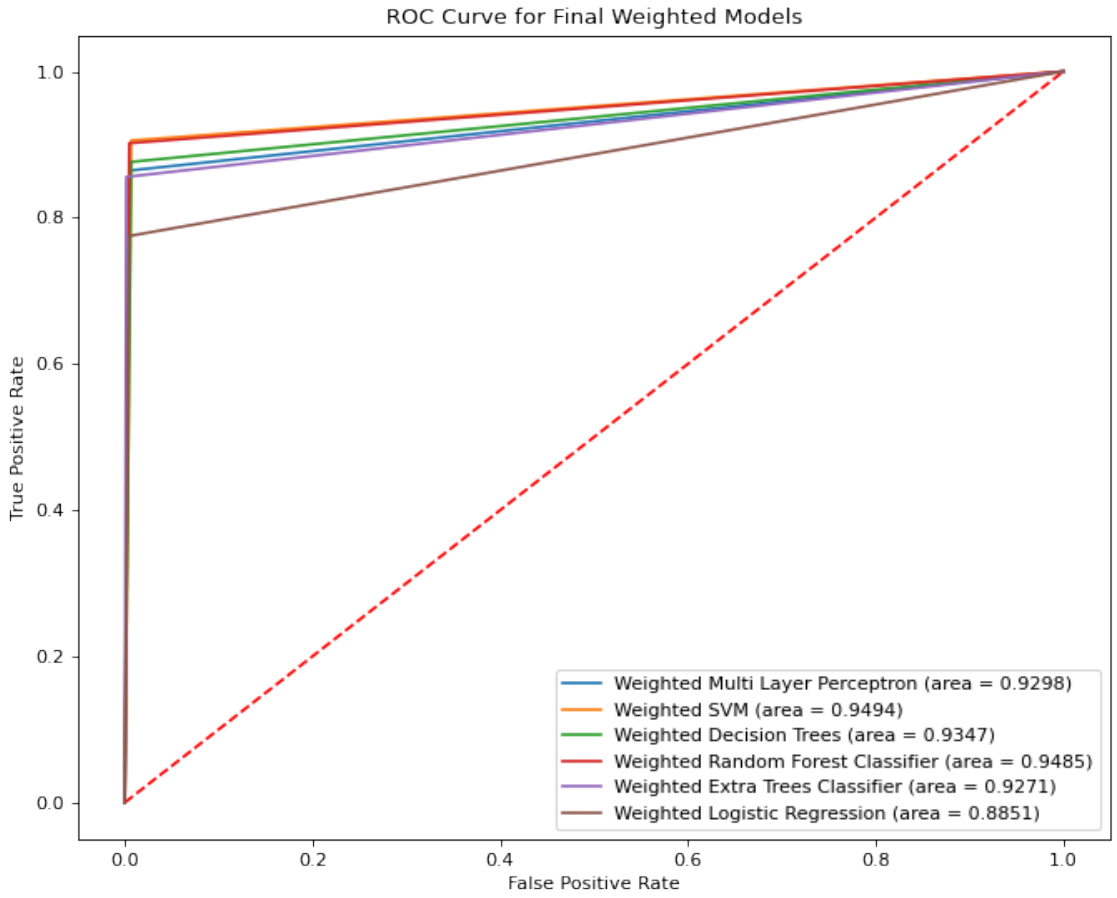
```
plt.plot(rffalsepr, rftruepr, label = 'Random Forest Classifier (area = %0.4f)'␣
 ↪%rfauc);
plt.plot(etfalsepr, ettruepr, label = 'Extra Trees Classifier (area = %0.4f)'␣
 ↪%etauc);
plt.plot(lrfalsepr,lrtruepr, label = 'Logistic Regression (area = %0.4f)'␣
 ↪%lrauc);
plt.xlabel('False Positive Rate');
plt.ylabel('True Positive Rate');
plt.title('ROC Curve for Final Models');
plt.legend();
```



The final model tuned for precision offered strong, balanced solutions when compared to the weighted models below. The strongest at making the most precise and overall accurate predictions was the Random Forest Classifier. Although, these models are all very close to each other as far as AUC. The Random Forest is quick to train and easy to deploy while offering the best solution out of this group of models, due to this Random Forest would need to be chosen out of these final models.

1.1.1 ROC Curve Weighted Models

[ ]:
```
plt.figure(figsize=(10,8),dpi=80)
plt.plot([0,1],[0,1],'r--')

plt.plot(mlpwfalsepr, mlpwtruepr, label = 'Weighted Multi Layer Perceptron␣
↪(area = %0.4f)' %mlpwauc);
plt.plot(wsvmfalsepr, wsvmtruepr, label = 'Weighted SVM (area = %0.4f)'␣
↪%wsvmauc);
plt.plot(dtwfalsepr, dtwtruepr, label = 'Weighted Decision Trees (area = %0.
↪4f)' %dtwauc);
plt.plot(rfwfalsepr, rfwtruepr, label = 'Weighted Random Forest Classifier␣
↪(area = %0.4f)' %rfwauc);
plt.plot(etwfalsepr, etwtruepr, label = 'Weighted Extra Trees Classifier (area␣
↪= %0.4f)' %etwauc);
plt.plot(wlrfalsepr,wlrtruepr, label = 'Weighted Logistic Regression (area = %0.
↪4f)' %wlrauc);
plt.xlabel('False Positive Rate');
plt.ylabel('True Positive Rate');
plt.title('ROC Curve for Final Weighted Models');
plt.legend();
```



Looking at the weighted models that offer better precision than the models graphed above, the random forest and SVM clearly offer the best auc. The SVM did not reach the same precision as the Random Forest. Additionally when considering deploying the model for the business, the SVM is far more costly and time consuming to train when compared to the random forest. Both also do not offer great interpretability. That is why when choosing from the weighted models based off of auc, the Random Forest offered the best results.

1.2 Performance Metrics For Both Final and Weighted Models

```
[ ]: final_metrics = pd.DataFrame({'Metrics':␣
     ↪['Accuracy','Precision','Recall','F1','AUC'],
                  'MLP': [mlp_acc,mlp_prec,mlp_rec,mlp_f1,mlpauc],
                  'Weighted MLP': [w_mlp_acc,w_mlp_prec,w_mlp_rec,w_mlp_f1,mlpwauc],
                  'SVM': [svm_acc,svm_prec,svm_rec,svm_f1,svmauc],
                  'Weighted SVM': [w_svm_acc,w_svm_prec,w_svm_rec,w_svm_f1,wsvmauc],
                  'Decision Tree': [dt_acc,dt_prec,dt_rec,dt_f1,dtaauc],
                  'Weighted Decision Tree':␣
     ↪[w_dt_acc,w_dt_prec,w_dt_rec,w_dt_f1,dtwauc],
```

```
                'Random Forest': [rf_acc,rf_prec,rf_rec,rf_f1,rfauc],
                'Weighted Random Forest':␣
 ↪[w_rf_acc,w_rf_prec,w_rf_rec,w_rf_f1,rfwauc],
                'Extra Trees': [et_acc,et_prec,et_rec,et_f1,etauc],
                'Weighted Extra Trees':␣
 ↪[w_et_acc,w_et_prec,w_et_rec,w_et_f1,etwauc],
                'Logistic Regression': [lr_acc,lr_prec,lr_rec,lr_f1,lrauc],
                'Weighted Logistic Regression':␣
 ↪[w_lr_acc,w_lr_prec,w_lr_rec,w_lr_f1,wlrauc]})
final_metrics
```

[ ]:

	Metrics	MLP	Weighted MLP	SVM	Weighted SVM	Decision Tree	\
0	Accuracy	0.964773	0.920000	0.967614	0.949545	0.965000	
1	Precision	0.968039	0.995152	0.971455	0.992494	0.969564	
2	Recall	0.960968	0.843415	0.963250	0.905501	0.959827	
3	F1	0.964490	0.913022	0.967335	0.947004	0.964671	
4	AUC	0.964756	0.919671	0.967595	0.949356	0.964978	

	Weighted Decision Tree	Random Forest	Weighted Random Forest	Extra Trees	\
0	0.935000	0.968750	0.948750	0.967273	
1	0.992246	0.973260	0.994713	0.973179	
2	0.876284	0.963707	0.901849	0.960740	
3	0.930667	0.968460	0.946007	0.966919	
4	0.934748	0.968728	0.948548	0.967245	

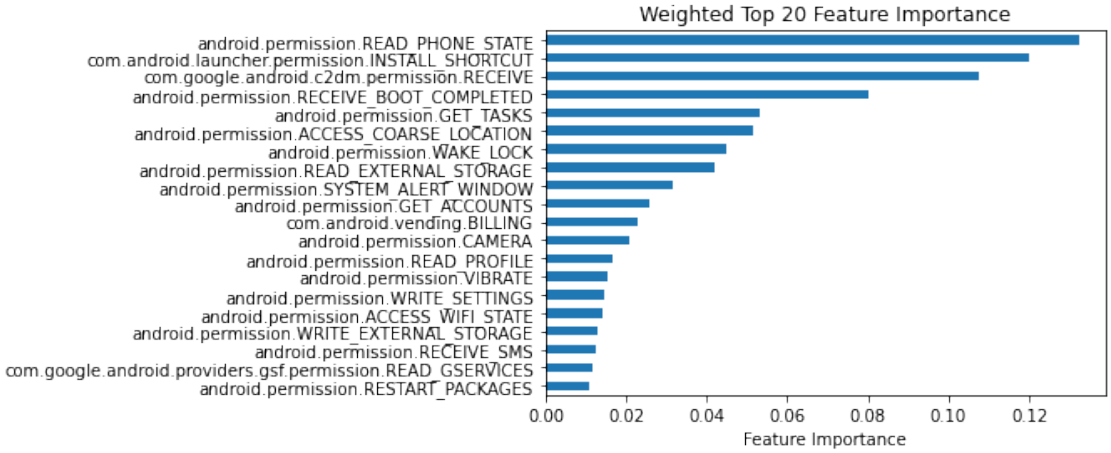
	Weighted Extra Trees	Logistic Regression	Weighted Logistic Regression
0	0.927386	0.959318	0.885568
1	0.998136	0.958096	0.993852
2	0.855741	0.960283	0.774937
3	0.921470	0.959188	0.870848
4	0.927078	0.959322	0.885093

Looking at our model metrics. The weighted MLP, Weighted Random Forest, Weighted SVM, and Weighted Etra Trees have offered the strongest precision out of the grouping. The business plan has dictated that a model must meet a minimum threshold of 93% overall accuracy in order to avoid spamming consumers with flagged applications. Due to this, the weighted SVM and weighted RF offered the best precision above the 93% threshold. Weighted RF had better precision slightly at 99.47% vs. 99.25% while the SVM offered slightly better accuracy and recall at 94.95% & 90.55% vs. 94.88% & 90.18%. Considering that the weighted RF offers the best precision in addition to faster training and easier deployment while reaching the minimum threshold as dictated by the business plan, the weighted RF is then the best choice given our choices here.

1.3 Feature Importance for Weighted Random Forest (selected model)

```
[ ]: feat_importances = pd.Series(weight_rf.feature_importances_, index=X_train.
     ↪columns)
     feat_importances.nlargest(20).plot(kind='barh').invert_yaxis()
     plt.xlabel("Feature Importance")
     plt.title("Weighted Top 20 Feature Importance")
```

```
[ ]: Text(0.5, 1.0, 'Weighted Top 20 Feature Importance')
```



Considering the final model's most important features will be important when working with the data further and pushing the model into production. RF does perform feature selection, and there are clearly a handful of features deemed more important to model creation.

The top two permission read_phone_state and install_shortcut also were most represented in only malware applications. Perhaps future research can be done to lock these permissions entirely aside from certified applications that are safe in order to try and avoid more malicious applications entirely.