Permissions-based Detection of Android Malware Using Machine Learning

Matthew Chen, Mark Faynboym, Jasper Tsai

# Abstract

Android had a global market share of 71.8% in the mobile operating systems market in 2023. The rapid growth of Android, its ubiquitous appearance in the smartphone market, and its open-source nature have made it an obvious and accessible target for malware. In recent literature, Mathur et al., (2021) found that malware detection classifiers trained using native AOS permissions and developers’ custom permissions achieve remarkable performance. In this project we replicate the results of Mathur’s findings as well as explore further avenues to increase detection performance. In all, we achieve 96-97% accuracy on holdout test performance using logistic regression, kNN, random forest, neural network, and ensemble stacking, with similarly high performance across F1, precision, recall, and ROC AUC.

# Introduction

# Exploratory Data Analysis

In this section we do a surface exploration of our data to notice any indication of potential hiccups down the pipeline of our analysis and gain a better understanding of certain characteristics of our data like the frequency and signs of multicollinearity. To do this we implement a frequency bar graph and a correlation matrix.

## Frequency Analysis

Figure 1 shows the most frequent permissions for benign and malware applications, respectively. Notably, the permissions READ\_PHONE\_STATE and RECEIVE\_BOOT\_COMPLETED have high frequency in the malware applications but are not present in the top benign permissions. These features are important to note as they may be significant further down the modeling pipeline.

A graph of data on a screen

Description automatically generated with medium confidence

Figure . Most frequent permissions in benign and malware applications

## Correlation Matrix

Figure 2 shows a heat map generated by the correlation matrix of our application permissions. Upon inspection we see no serious issues of multicollinearity with the exception of very few features, READ, BROADCAST\_BADGE, UPDATE\_BADGE and UPDATE\_SHORTCUT which we will also take into account when building our subsequent models.

A screen shot of a graph

Description automatically generated

Figure 2. Heatmap of the feature correlation matrix

# Methodology

In this section we discuss the methodological details of modeling the malware detection problem using machine learning methods, including logistic regression, kNN, random forest, neural network, and ensemble stacking. We briefly introduce the models tested as well as the data validation, feature selection, and hyperparameter tuning strategies employed. Note that in Mathur et al., (2021), the highest performing models were k-Nearest Neighbors and Random Forest.

## Proposed Models

### Logistic Regression

Logistic regression (LG) is the most parsimonious and interpretable model out of all the tested models and serves as a linear baseline for our experiments. In logistic regression, we model the binary classification problem with a linear decision boundary. That is, given a vector of features we model the log-odds of the outcome probability using an affine function of and weights/intercept (Equation 1).

We fit the model using binary cross entropy loss (Equation 2), which corresponds to the maximum likelihood estimation on the predicted probabilities. We have as the target for the i-th observation, representing if the observation is benign (0) or malware (1).

The only hyperparameter of logistic regression that we tune in this project is an L2 penalty on the loss function in Equation 2.

### Random Forest

Random Forest (RF) is a modified bagging method for decision trees. A “forest” of classification decision trees are grown and pruned using some impurity criterion (i.e. Gini Impurity) using bootstrapped data and a randomly chosen pool of candidate features at each data split. The goal of random forest is to build decorrelated decision trees so that by taking a group consensus we effectively reduce the variance of the model while leaving the model bias unaffected. Note that while decision trees are highly interpretable, random forests are not as much since the final prediction is the consensus of many decision trees which changes the model structure. However, by ranking the average contribution of decreasing the impurity criterion for each feature across the forest, we can effectively produce feature importance scores which are useful for feature selection as well as model interpretation.

For random forest, we tune the number of trees in the forest as well as the number of candidate features considered at each split.

### K-Nearest Neighbors

K-Nearest Neighbors (kNN) models are built upon the simple assumption that observations with the same label are close to each other. With this intuition, we rank the distance of a new observation to each training data point with some measure of distance (typically Euclidean distance) and take the consensus of the k-nearest training data points. The number of neighbors to consider is a hyperparameter that needs to be tuned.

### Feedforward Neural Network

Neural Networks (FF) are capable of modeling highly nonlinear functions by representing the data in higher dimensional spaces known as the hidden representations. Neural nets can be expressed as a composite function of nonlinear functions known as activation functions applied to linear transformations of the data. In this study, we use the Rectified Linear Unit (ReLU) function for activation. Note that for the final transformation we use a Sigmoid function ( so that the output is a valid prediction probability between 0 and 1. For example, a two layer neural network can be expressed as the following with weights and bias (intercept) for each layer j (Equation 3). Similar to logistic regression, we train the model on binary cross-entropy (Equation 2).

In this project, we train the neural network with dropout regularization and early stopping, that is, training is terminated when performance plateaus or worsens on the validation data, to prevent overfitting. Hyperparameters that we tune include the number of nodes in the hidden layers, the number of layers, the dropout probability, and the learning rate for the optimization algorithm.

### Ensemble Stacking

Intuitively, ensemble stacking (ST) combines the strengths of several strong predictions and in many cases produces a final model that is stronger than any individual ensemble member, especially if the candidate models are strong in different ways. In practice, we use the predicted probabilities of existing models, known as Level 0 Models, on another algorithm, known as the Level 1 Model, to get the final prediction. In the project, we choose Logistic Regression as the Level 1 Model that combines the Level 0 predictions from logistic regression, random forest, k-nearest neighbors, and neural network (Figure 1).

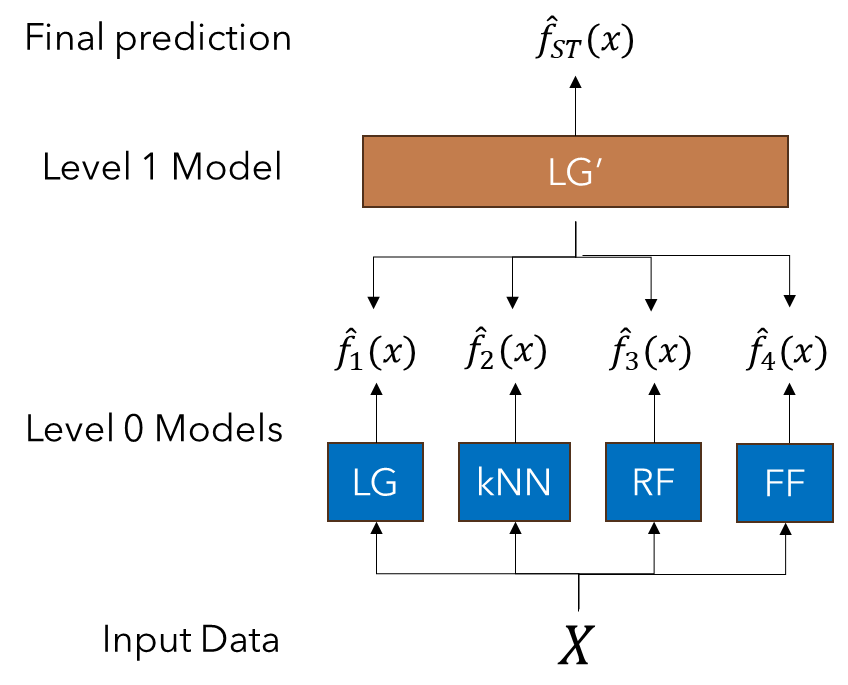


Figure . Schematic for ensemble stacking model

Consider a regression example with predictions from different Level 0 models denoted . Consider the data to be distributed according to some arbitrary distribution . Then, an example of fitting a stacking model would be to find weights such as the following expression of square error loss (Equation 4) is minimized.

At the population level, this solution corresponds to a Level 1 linear regression model. Note that Equation 4 implies that the stacked model is no worse than any individual model at the population level.

Additionally, to check that the predicted probabilities are well calibrated, we create a probability calibration plot (Figure 2) to check that the predicted probabilities are good estimates of the true probability. This also ensures that the scaling of the probabilities is consistent across Level 0 models. In the probability calibration curve, we find that each model is roughly well calibrated, with the exception of k-nearest neighbors.

A graph of a graph showing the number of calibration curves

Description automatically generated

Figure . Probability calibration curve for Level 0 models

## Data Validation Strategy

Prior to modeling, we randomly split the data into two portions, 80% for non-test data used for training and validation, and 20% test data used for estimating generalization error or performance on unseen data. Using the non-test data, we then use 10-fold cross validation for feature selection (see Section 3.3) and hyperparameter tuning (see Section 3.4). The exception is hyperparameter tuning for the neural network, where we simply further split the non-test data into a training portion (60%) and validation portion (20%). This is out of computational concern over 10-fold cross validation and the larger size of the search grid when using grid search hyperparameter tuning.

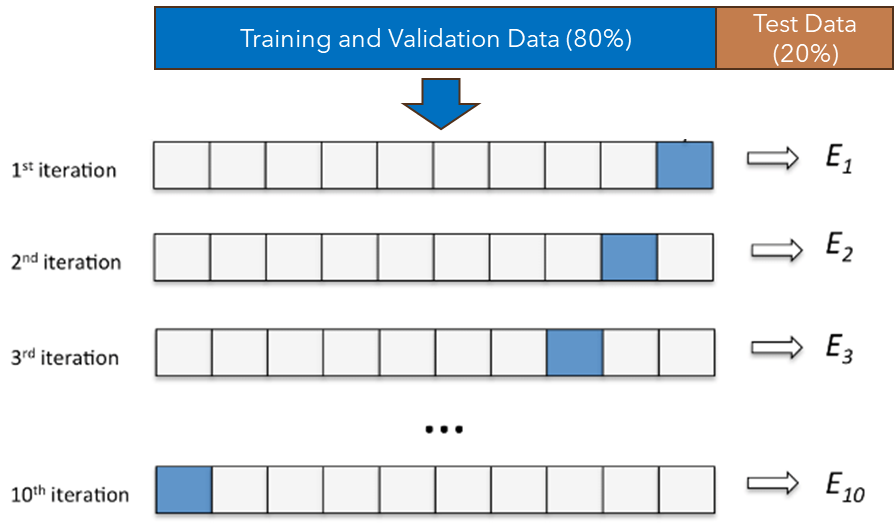


Figure . Data split strategy. Image source: https://www.researchgate.net/figure/Repeated-K-Fold-Cross-Validation-A-repeated-10-fold-CV-was-applied-The-10-fold-CV-works\_fig1\_328798891

## Feature Selection and Dimensionality Reduction

Among the 86 permissions features available in the NATICUSdroid dataset, we aim to reduce the overall dimension by selecting a subset of the top features and discarding the rest. By doing so, we reduce the model variance and promote model parsimony. Additionally, it is unlikely that each of the 40 features in the dataset are informative at representing malware. Thus, we conduct feature selection based on random forest-based feature importance scores (see Section 3.1.2) on the non-test data by ranking their importance and selecting a subset of the top features. To decide on the number of top features to select, we compare model performance (specifically for logistic regression, random forest, and k-nearest neighbors) under 10-fold cross validation on the non-test data (Figure 4). By doing so, we see that performance generally plateaus after selecting the top 40 features.

A graph of different colored lines

Description automatically generated

Figure . Feature selection vs cross validation performance

## Hyperparameter Tuning

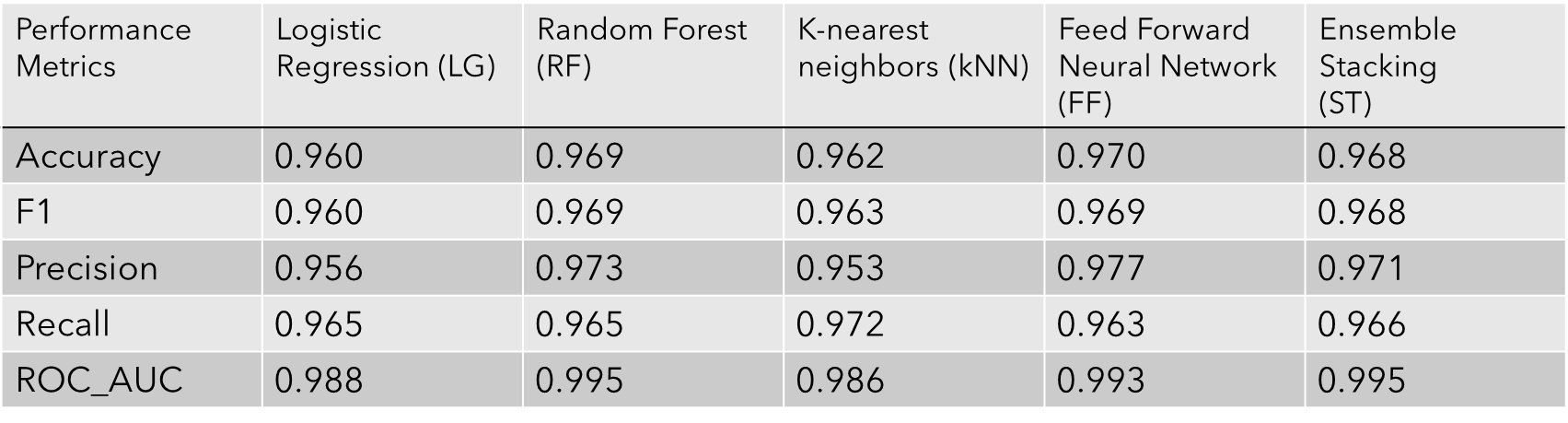
In this project, hyperparameter tuning was done using grid search, that is, to test exhaustively every combination of hyperparameters in a predefined parameter space. As aforementioned, we compare 10-fold cross validation performance on non-test data with the exception of neural network, and select the optimal set of parameters corresponding to the best performance. For logistic regression, we tune the L2 regularization penalty on the loss function. For random forest, we tune the number of trees in the forest and the fraction of features to consider at each data split. For k-nearest neighbors we tune the number of neighbors. Finally, for the neural network, we tune the number of layers, the number of nodes per layer, the dropout regularization probability, and the learning rate for the optimization algorithm.

# Results and Discussion

## Performance Evaluation

Table 1 presents the accuracy, f1 score, precision, recall, and ROC AUC metrics for each tested model. Overall, model performance across all tested models are comparable. Specifically, we find that logistic regression and k-nearest neighbors are about 96% accurate on the test data while random forest, neural network, and ensemble stacking are about 97% accurate on the test data. The other performance metrics are similarly high. Although the neural network has the greatest test performance, the advantage compared to the other tested models are minimal, especially considering the poor interpretability of such models. Logistic regression, on the other hand, performs almost as well and is a highly interpretable model. That is, we know explicitly how a logistic regression model makes its decisions. Further, the high performance of the logistic regression model may suggest that the true decision boundary is in fact linear, and that a linear model may be the most parsimonious and optimal choice.

Table . Model performance evaluation



### Evaluation Speed

It may also be important to consider evaluation speed, especially if the classifier is to be deployed for real-time malware detection. Below in Figure 5, we plot the log average evaluation time (for one sample) over 10 runs against the model performance. The plot shows that the neural network is the optimal choice for evaluation time vs accuracy and f1 score. Surprisingly, the neural network makes evaluations faster than the logistic regression model. This may be due to the fact that the final neural network (a PyTorch object) is more optimized than the final logistic regression, k-nearest neighbors, and random forest models (scikit-learn objects). Theoretically, the number of operations required for a 3-layer neural network is much more than an equivalent logistic regression model.

A chart of performance indicators

Description automatically generated with medium confidence

Figure . Evaluation time vs. model performance

## Model Interpretation

### Logistic Regression

In Figure 6, we plot the coefficients of the final logistic regression model. Note that since all of the features are binary indicator variables and have the same scale, the coefficients are comparable with each other. We interpret these coefficients as a feature’s respective contribution at increasing or decreasing the log-odds, which corresponds to a respective increase or decrease in predicting malware. We find that the permission *com.google.android.c2dm.permission.RECIEVE* is the most important feature in reducing the probability of predicting malware (i.e. indicative of benign software) and that the permission *android.permission.READ\_PHONE\_STATE* is the most important feature in increasing the probability of predicting malware. These results are consistent with what we found in the exploratory data analysis.

A graph with a bar chart and a bar chart with a bar chart

Description automatically generated

Figure . Coefficients of the final logistic regression model

# Conclusion

# References