Permissions-based Detection of Android Malware Using Machine Learning

Matthew Chen, Mark Faynboym, Jasper Tsai

Abstract

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 Android OS (AOS) permissions and developer 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, k-nearest neighbors, random forest, neural network, and ensemble stacking, with similarly high performance across F1, precision, recall, and ROC AUC.

1 Introduction

In 2021, Android had a global market share of 74% in the mobile operating systems market and has 2 billion active devices (Mathur et al., 2021). Further, its open-source nature makes it especially prone to malicious software as 97% of all mobile malware is targeted at Android (Akbar et al., 2022). Although Android curbs the harm of malicious software with regular updates to fix vulnerabilities, it is still of vital importance that malware can be detected on any given system. Mathur et al., 2021 showed that permissions-based malware detection algorithms using the newly created native Android OS (AOS) permissions as well as custom developer permissions could achieve remarkable performance when compared to older research.

The goal of our analysis is to first replicate the results of the original study, as well as explore additional methods that may improve established performance. Namely, we consider implementing a feedforward neural network and a stacking ensemble model. We also prioritize model interpretation to understand key differentiating permissions in detecting malware using a data driven methodology.

1.1 Data Source

In this project, we build predictive models using the NATICUSdroid dataset (Mathur, 2022) which includes the same native and custom permissions as used in the original study. The dataset includes 14700 observations of malware and 14633 observations of benign applications collected between 2010 and 2019. There are 86 permission features represented as binary indicator variables and no missing values. Originally, the benign applications were sourced from the Androzoo database, and the malware applications from the Argus Labs Android Malware Database.

2 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.

2.1 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.

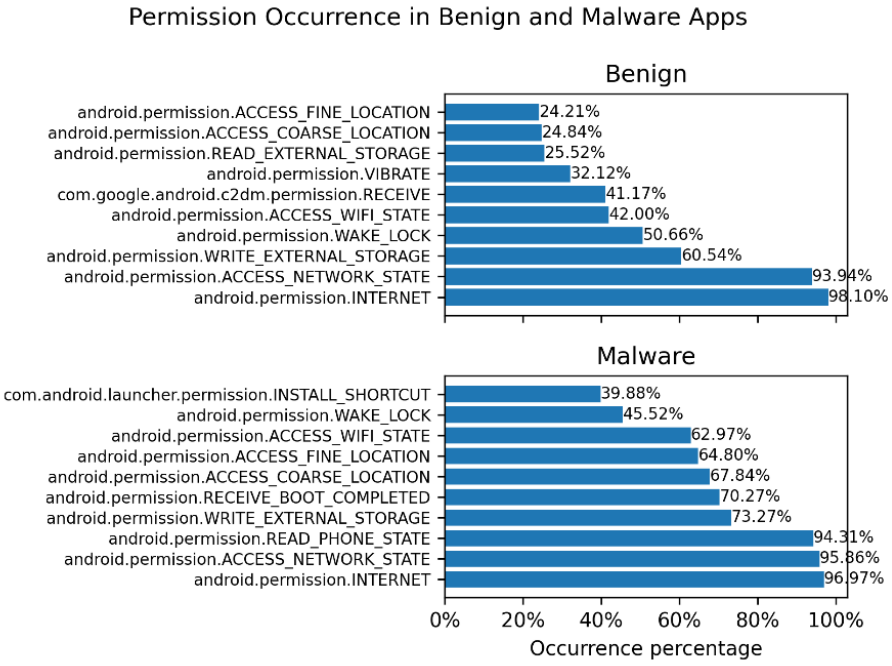

Figure 1. Most frequent permissions in benign and malware applications

2.2 Correlation Matrix

Figure 2 shows a heat map of the correlation matrix of our application permissions. Upon initial inspection we see no obvious issues of multicollinearity with the exception of very few features, specifically, READ, BROADCAST_BADGE, UPDATE_BADGE and UPDATE_SHORTCUT. We will take this into account if our subsequent models perform poorly and have high variance.

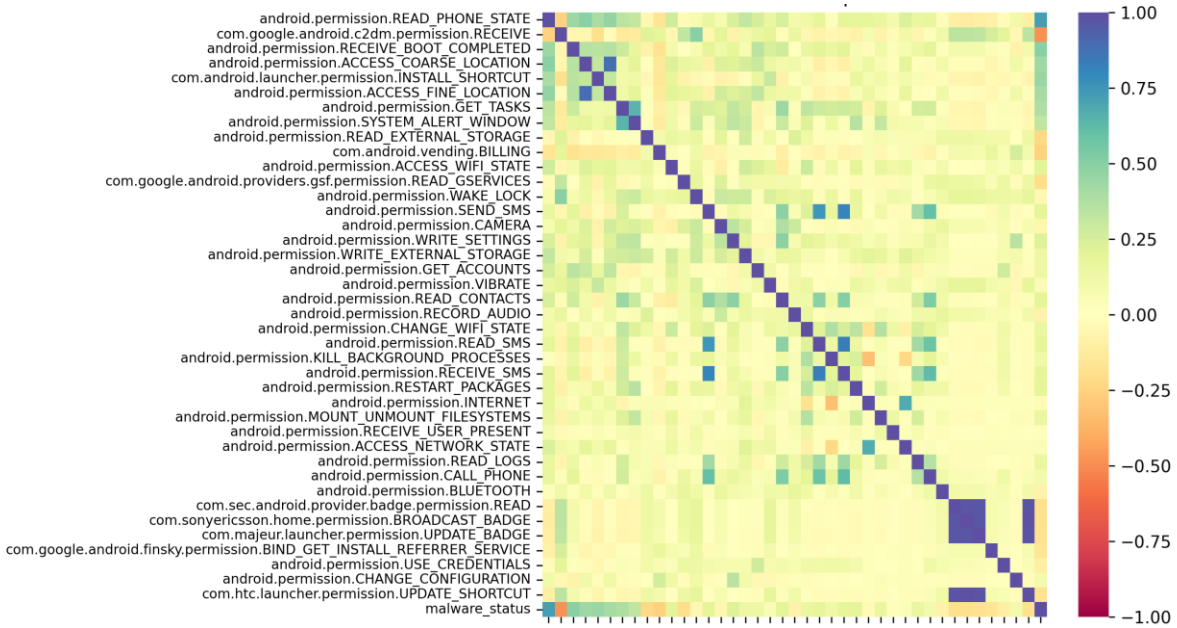

Figure 2. Heatmap of the feature correlation matrix

3 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.

3.1 Proposed Models

3.1.1 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 x we model the log-odds of the outcome probability p using an affine function of x and weights/intercept β (Equation 1).

$$logit(p) = log\left(\frac{p(x;\beta)}{1-p(x;\beta)}\right) = \beta_0 + \beta^T x \qquad Eq.(1)$$

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

$$L(\beta) = \sum_{i=1}^{n} logp_{y_i}(x_i;\beta) = \sum_{i=1}^{n} y_i\, logp(x_i;\beta) + (1-y_i)log\big(1-p(x_i;\beta)\big) \qquad Eq.(2)$$

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

3.1.2 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.

3.1.3 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.

3.1.4 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 (Goodfellow et al., 2016). 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 (σ) 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) W_j, β_j for each layer j (Equation 3). Similar to logistic regression, we train the model on binary cross-entropy (Equation 2).

$$\hat{y} = \sigma\left(ReLU\left(W_2(ReLU(W_1 x + \beta_1) + \beta_2)\right)\right) \qquad Eq.(3)$$

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.

3.1.5 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 3).

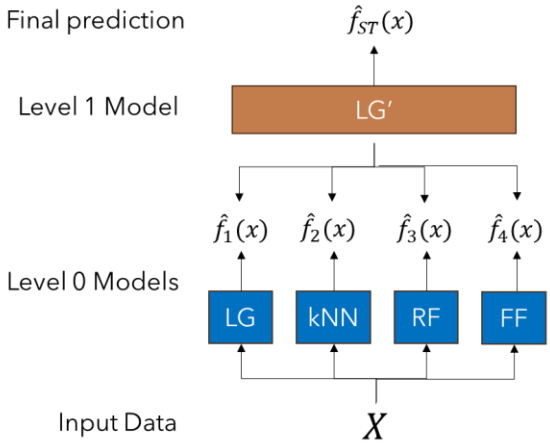

Figure 3. Schematic for ensemble stacking model

Consider a regression example with predictions from M different Level 0 models denoted $\hat{f}_1, \ldots, \hat{f}_M$. Consider the data to be distributed according to some arbitrary distribution $(X, Y) \sim P$. 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.

$$\hat{w} = \underset{w}{argmin} \mathbf{E}_P \left[\left(Y - \sum_{m=1}^{M} w_m \hat{f}_m(x) \right)^2 \right] \qquad Eq.(4)$$

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 (Hastie et al., 2008).

Additionally, to check that the predicted probabilities are well calibrated, we create a probability calibration plot (Figure 4) 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.

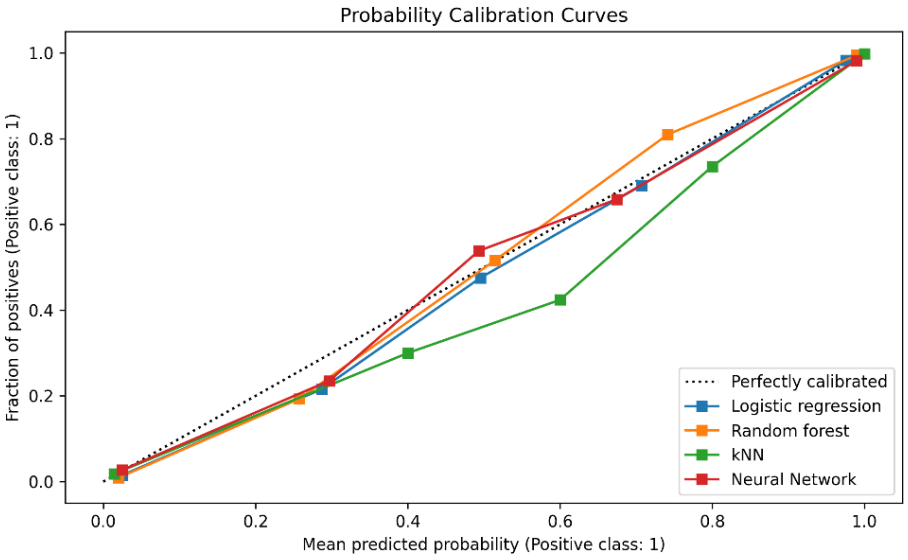

Figure 4. Probability calibration curve for Level 0 models

3.2 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.

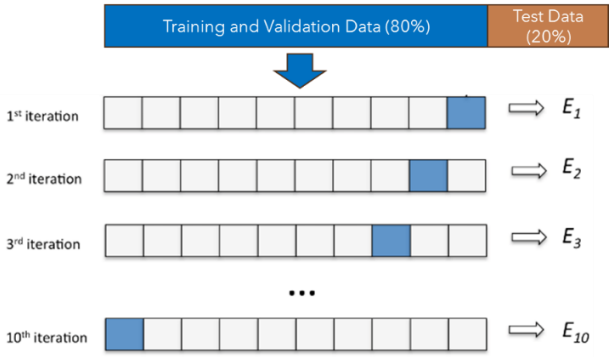

Figure 2. Data split strategy. Image source

3.3 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 5). By doing so, we see that performance generally plateaus after selecting the top 40 features.

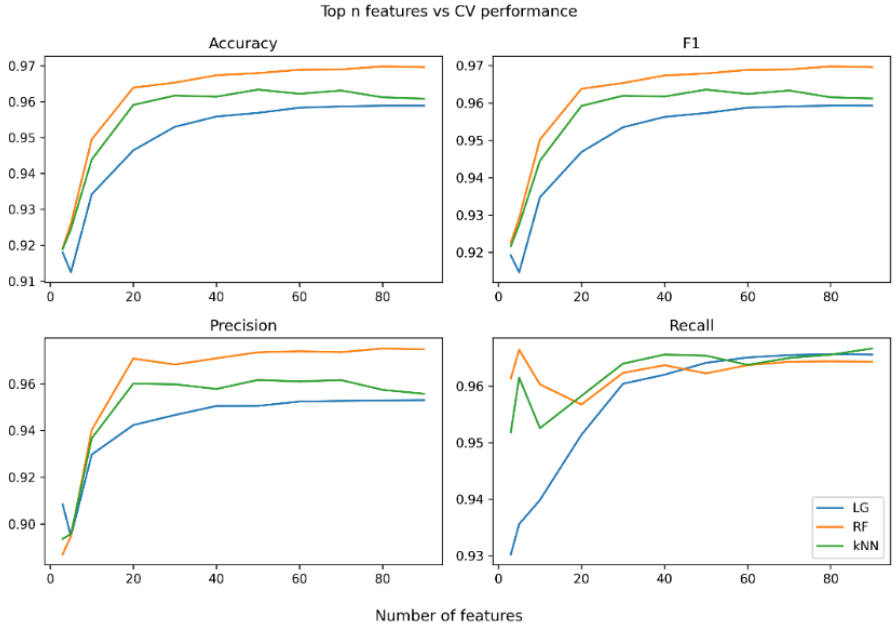

Figure 3. Feature selection vs cross validation performance

3.4 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.

4 Results and Discussion

4.1 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 1. Model performance evaluation

Performance Metrics	Logistic Regression (LG)	Random Forest (RF)	K-nearest neighbors (kNN)	Feed Forward Neural Network (FF)	Ensemble Stacking (ST)
Accuracy	0.960	0.969	0.962	0.970	0.968
F1	0.960	0.969	0.963	0.969	0.968
Precision	0.956	0.973	0.953	0.977	0.971
Recall	0.965	0.965	0.972	0.963	0.966
ROC_AUC	0.988	0.995	0.986	0.993	0.995

4.1.1 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 6, 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.

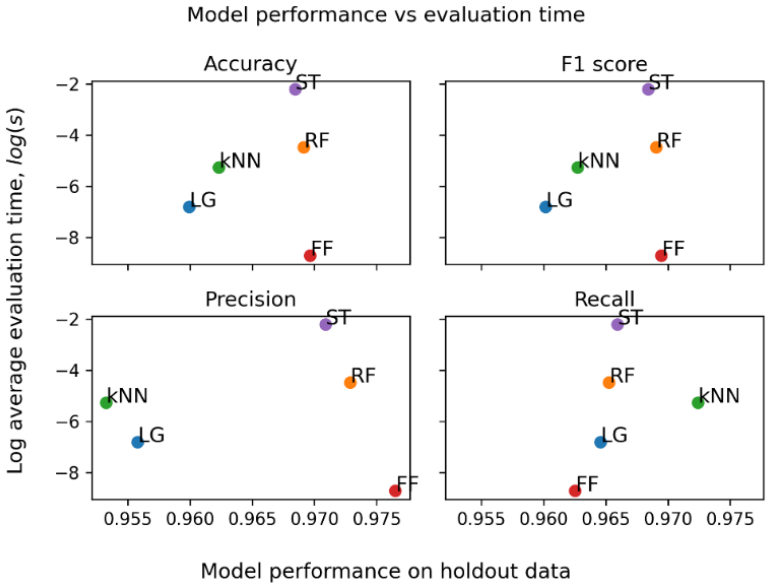

Figure 4. Evaluation time vs. model performance

4.2 Model Interpretation

4.2.1 Logistic Regression

In Figure 7, 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 *RECIEVE* is the most important feature in reducing the probability of predicting malware (i.e. indicative of benign software) and that the 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.

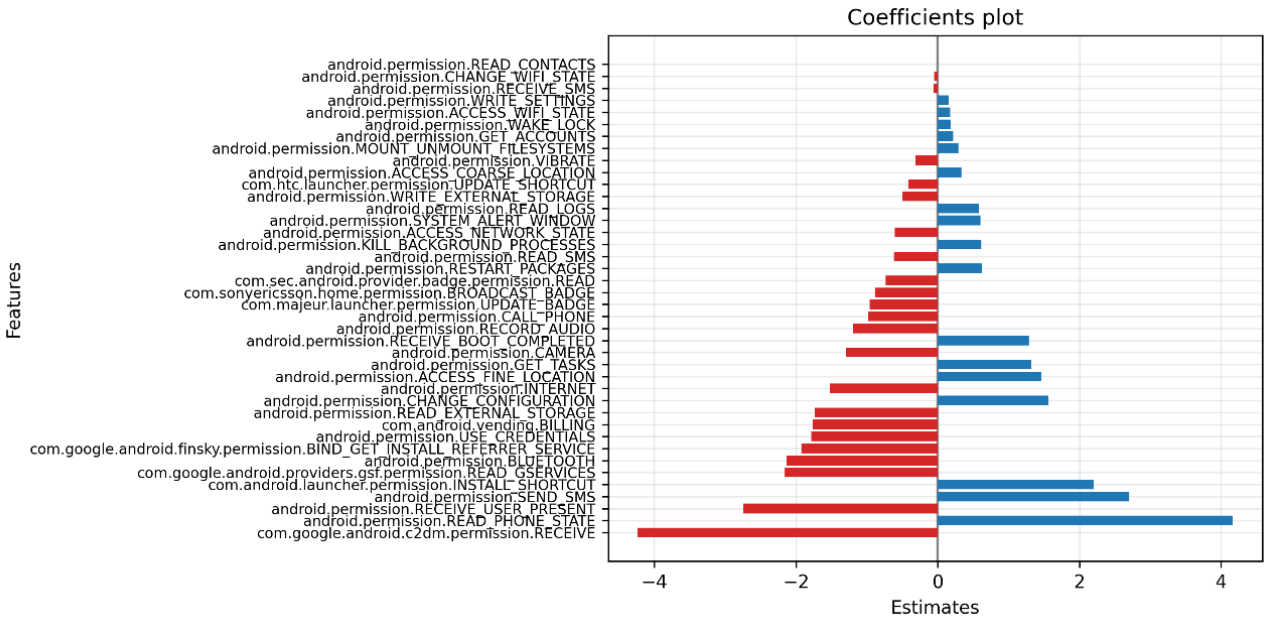

Figure 5. Coefficients of the final logistic regression model

4.3 Random Forest Feature Importance

In Figure 8, we show a plot of the top 20 features in terms of random forest feature importance scores. Not surprisingly, we see that the top 3 most important features, READ_PHONE_STATE, RECEIVE, and RECEIVE_BOOT_COMPLETED, are largely consistent with our observations in the exploratory data analysis section.

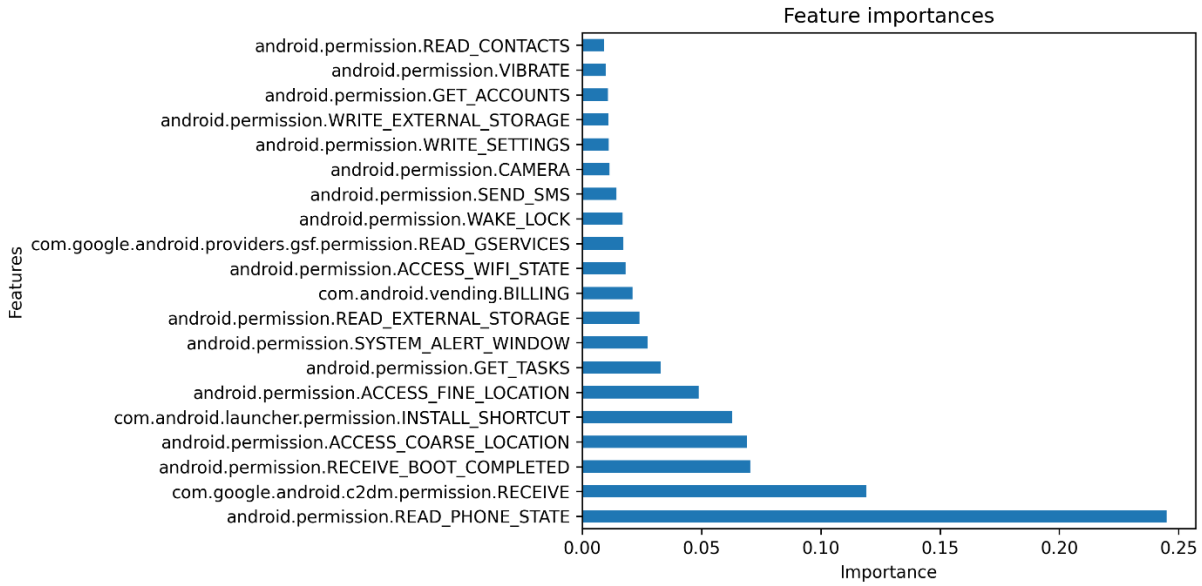

Figure 6. Top 20 random forest feature importance scores

4.4 Decision Tree Interpretation

As aforementioned in Section 3.1.2, the random forest model is not as directly interpretable as a single decision tree. Thus, we fit and interpret a shallow decision to further increase our insight of the data. Below in Figure 9, we show the tree structure of our fitted decision tree, and we see that this tree with only two decision nodes achieves an accuracy of 92.5% on the data which increases the confidence in our previous interpretations. Again, we find that READ_PHONE_STATE and RECEIVE are the most telling features in differentiating malware from benign applications which is consistent with our interpretation of the logistic regression and random forest models.

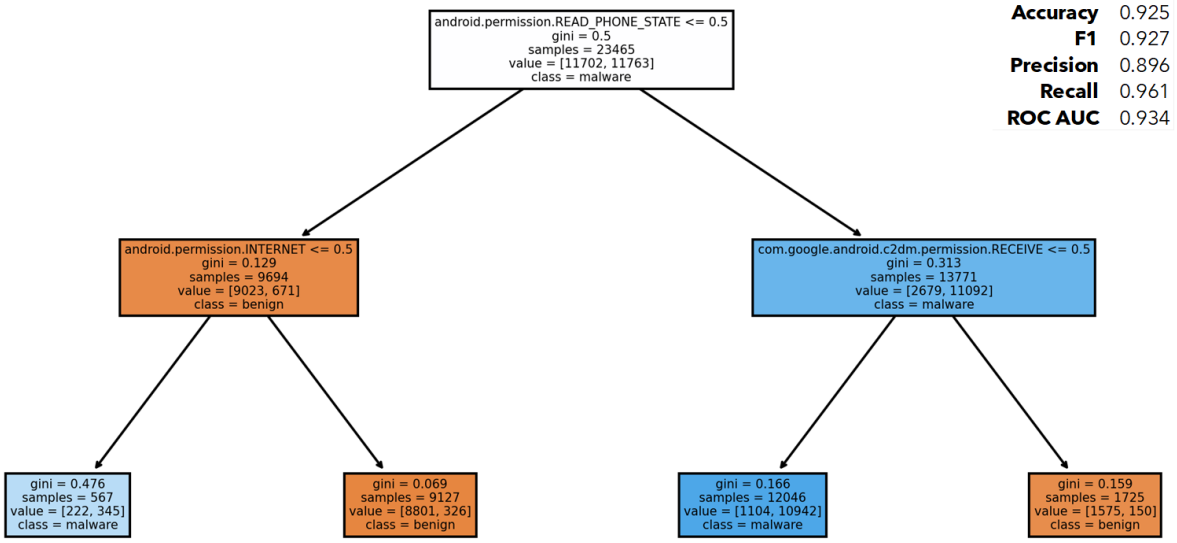

Figure 7. Shallow decision tree

5 Conclusion

Overall, we were able to build a successful malware detection model using Android application permissions building on the work of recent literature. Between the tested models including logistic regression, k-nearest neighbors, random forest, neural network, and ensemble stacking, we achieve accuracies between 96-97% on the holdout test data with similarly high performance across F1 score, precision, recall, and ROC AUC metrics. Additionally, we find that for the logistic regression model, READ_PHONE_STATE is the most important permission feature in distinguishing malware applications while the RECEIVE permission is the most important in distinguishing benign applications. These results are consistent with our interpretation of the Random Forest model, a shallow decision tree, as well as our exploratory data analysis.

These results hold important implications for detecting malware on Android devices in real time, especially when using a logistic regression model that is both fast to train, i.e. update to new malware, and fast to evaluate new applications.

6 References

Akbar, F., Hussain, M., Mumtaz, R., Riaz, Q., Wahab, A. W. A., & Jung, K. H. (2022). Permissions-Based Detection of Android Malware Using Machine Learning. *Symmetry*, *14*(4). https://doi.org/10.3390/sym14040718

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*.

Hastie, T., Tibshirani, R., & Friedman, J. (2008). *The Elements of Statistical Learning Data Mining, Inference, and Prediction*.

Mathur, A. (2022). NATICUSdroid (Android Permissions) Dataset. In *UCI Machine Learning Repository*. https://doi.org/https://doi.org/10.24432/C5FS64

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*. https://doi.org/10.1016/j.jisa.2020.102696