Detecting Malware in Android Applications

JAKE PITKIN
CS 6350 - Machine Learning
December 14, 2016

Introduction

The Android platform is the most popular mobile platform, holding nearly 85% of the global smartphone marketshare. [1] Most of the available software for Android devices comes from public application markets such as Google Play and the Apple App Store. As a result, Android users are a very popular target for malicious software.

The goal is to classify software as malicious or not as a way to improve the security of the Android platform. This will be accomplished by training a classifier on examples where we observed how often particular system calls were called. The hope is this will expose a patterns of system calls that malicious applications make. Allowing us to prohibit these applications from public markets and protect users.

Approach - Random Forest

The approach I used to classify the maliciousness of an Android application is a random forest. A random forest is an ensemble method based on bagging (short for bootstrap aggregating) and a decision tree using the ID3 algorithm. In the classic ID3 algorithm, all features are considered (using the heuristic of information gain to select a feature at each level) and one tree is trained on all the training examples. For bagging, I randomly selected m examples with replacement and only consider k random features when building the tree. I then built N of these trees.

When it came time to predict a label for a new example, all of the trees in the random forest vote on a label. As I was performing binary classification, I chose N to be odd and took the majority vote of the trees.

Figure 1: Taking the majority vote of an ensemble of decision trees.

Feature Exploration

Each example is defined by 360 features or system calls. But I discovered only 130 of these features appear in the training examples. Of these 130 features, there are a great deal of them that appear in nearly every training example and four features that appear in every training example. The 65 features with the highest frequency appear in at least half of the training examples.

A feature's value is defined by how many times the system call that feature represents is called. As such, the range of values a feature held varied greatly. Some present system calls occurred only a few times where others were called hundreds of thousands of times.

The decision tree classifier is only feasible with numeric feature values by using thresholds or by making the values discrete. I choose to make them discrete by transforming all features into either a 0 or a 1. I calculated the average value a feature held across all the training examples. I considered to be "present" or 1 if it's value is above the average and 0 otherwise. I played with this threshold by introducing a hyper-parameter λ.

$$threshold_j = \lambda * average_j$$

This allowed me to vary how above-or-below the average a feature j should be to be considered "present".

Cross-validation

My random forest contained four different hyper-parameters, all of which were determined by using 6-fold cross validation.

Hyper-parameter	Description	Best Value
λ	Decision boundary for discretization.	0.34
N	Decision tree count.	101
m	Size of training example sample.	10,000
k	Number of randomly selected features.	50

Table 1: The used hyper-parameters for the random forest classifier.

If a system call was called at least 0.34 times the average, that feature was considered present an was given a value of 1 and a value of 0 otherwise.

The random forest consisted of 101 decision trees. Each decision trees was constructed with 50 randomly selected features and was trained on 10,0000 sampled with replacement training examples.

Performance

To determine the performance of my tuned random forest classifier, I considered its accuracy and F1 score on the training and test sets.

Data Set	Accuracy	F1 Score
Training Set	0.881	0.73
Test Set	0.851	0.741

At the time of this report I am currently $9th$ out of $28th$ on the Kaggle competition board with a score of 0.82819.

Improvements

Decision boundaries - An area I would of liked to explore deeper was the discretization of the data. I took a fairly naive approach of using the average value of a feature as the boundary for transforming continuous features values into binary features. Being more analytical by looking at the distribution of labels relative to each feature value could have proven fruitful. Additionally, rather than making all features binary, using multiple decision boundaries to have a handful of possible feature values could of been worthwhile.

SVM on random forest votes - My random forest classifier used a majority vote between all the trees to decide on a final label. I would of liked to

train a SVM classifier using these predictions as features (as we explored in homework 4).

Other Classifiers Attempted

I tried many other classifiers before arriving at using a random forest. Other than decision trees, I tried each classifier with transformed features and unmodified features. Here is how they each ranked according to their perform on the Kaggle submission data.

1. Logistic Regression with binary features
2. Logistic Regression
3. Decision Tree with binary features
4. SVM with binary features
5. Perceptron with binary features
6. Perceptron

With my final classifier, Random Forest with binary features, out performing all of the above classifiers. Logistic Regression with binary features was a very close second and is drastically easier to implement than a Random Forest.

What I Would Like To Try

Neural network

What I Learned

References

[1] Dimjašević, M., Atzeni, S., Ugrina, I. and Rakamaric, Z., 2016, March. Evaluation of Android Malware Detection Based on System Calls. In Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics (pp. 1-8). ACM.