MALWARE PREDICTION

Dataset Description

The dataset is downloaded from a finished Kaggle competition, Microsoft Malware Prediction. (Link: https://www.kaggle.com/c/microsoft-malware-prediction)The goal of this competition is to predict a Windows machine's probability of getting infected by various families of malware, based on different properties of that machine. The telemetry data containing these properties and the machine infections was generated by combining heartbeat and threat reports collected by Microsoft's endpoint protection solution, Windows Defender.

Each row in this dataset corresponds to a machine, uniquely identified by a MachineIdentifier. HasDetections is the ground truth and indicates that Malware was detected on the machine. HasDetections is a binary label and also the attribute we want to predict.

The sampling methodology used to create this dataset was designed to meet certain business constraints, both in regards to user privacy as well as the time period during which the machine was running. Malware detection is inherently a time-series problem, but it is made complicated by the introduction of new machines, machines that come online and offline, machines that receive patches, machines that receive new operating systems, etc. Additionally, this dataset is not representative of Microsoft customers' machines in the wild; it has been sampled to include a much larger proportion of malware machines.

The original dataset has more than 8000000 samples. We randomly chose a subset, which has 5000 samples, while adding an additional column "X", which is the index of this row in the original dataset. This column, as well as the "MachineIdentifier", is excluded from potential predictors, which leaves a total of 81 potential predictors in total.

Model Fitting

Logistic Regression

The linear regression is fit with variable selection. Every model is trained with R language on a Cent OS machine with one Intel Xeon CPU E5-2650 v3 @ 2.30GHz core and 64GB of RAM. Training is evaluated by accuracy, false positive/negative, and running time. The model fitting takes 280.145s, which is the majority of the time (the whole time, which include data reading and processing, model fitting and evaluation, takes 280.379s), and the model has the accuracy of 50.56% on the test set and 52.88% on the train set, the false positive rate is 48.77%, and the false negative rate is 50.44% on the test set and 47.96% on the train set.

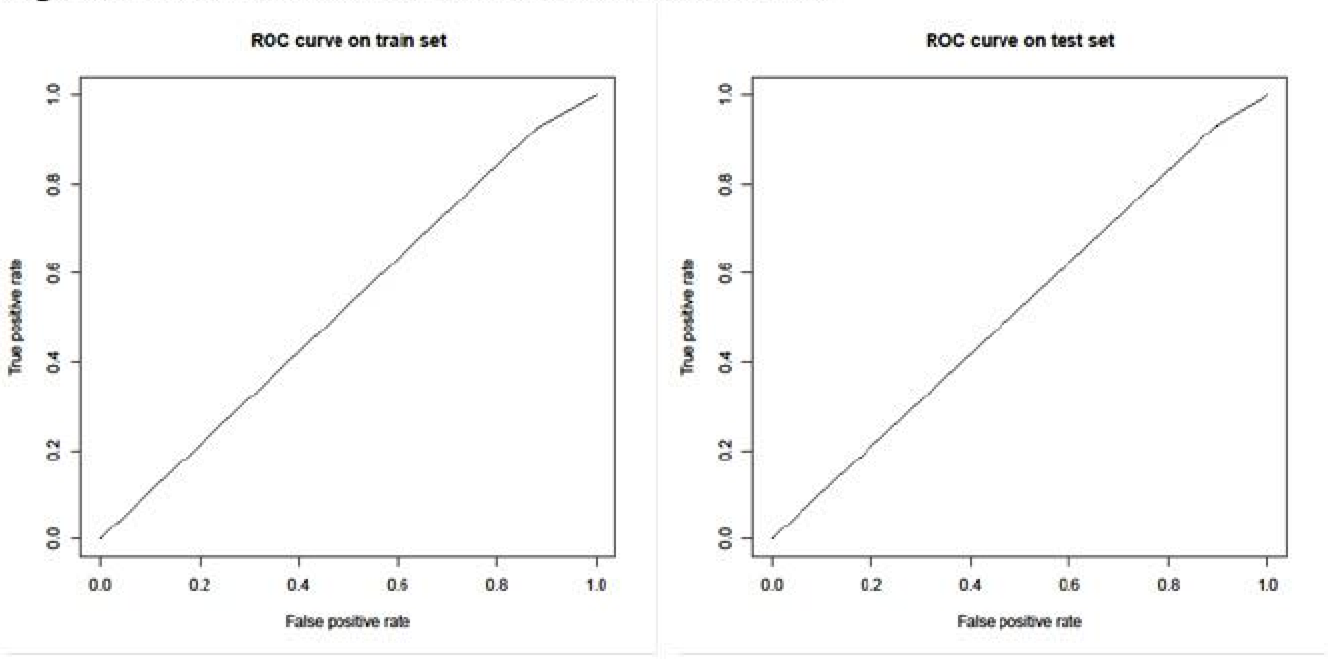


Figure 1: ROC curves for logistic regression

SVM Model

The SVM model is fit with randomly selected 12 predictors, because we have more than 80 predictors, and the number of combinations of chosen 12 predictors is too large and we do not have the time and resources to run a full test.

A total of 100 models are trained with randomly selected 12 predictors at each iteration.Every model is trained with R language on a Cent OS machine with one Intel Xeon CPU E5-2650 v3 @ 2.30GHz core and 64GB of RAM. The training is evaluated by accuracy, false positive/negative rate, and running. The whole process, which include data reading and processing, model training and evaluation, takes 377.403s. Of all the randomly chosen 100 sets of predictors, the model that has the best performance has the an accuracy of 56.40%. The false positive rate is 44.57%, and the false negative rate is 42.63%. The best model takes 2.746s for fitting. Above is the ROC curve on training set and test set for SVM.

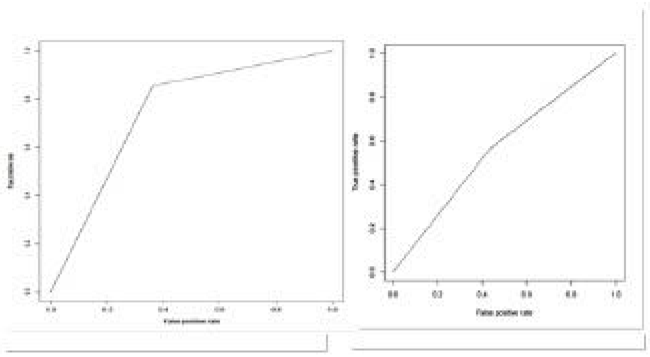


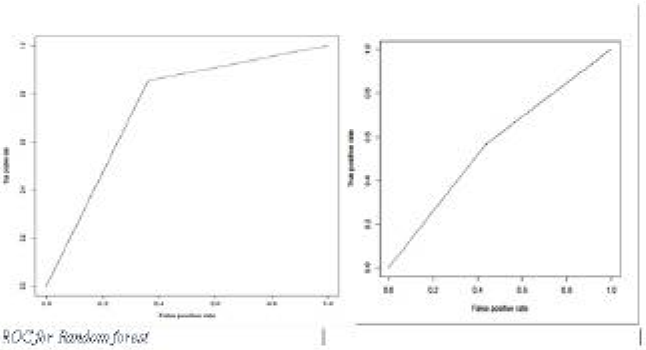
Figure 2: ROC curve for SVM

Random Forest (Tree Structured Classifier)

We now use a tree structured classifier to analyze our data. Once again, we do this by selecting a subset of all the 84 predictors since building the forest with all the available predictors would be very complicated computationally. Like in the SVM Model, we use the same set of 14 predictors here as well: ProductName, IsBeta, HasTpm, RtpStateBitfield, IsSxsPassiveMode, Platform, Processor, IsProtected, PuaMode, Smode, Firewall, Census_DeviceFamily, Census_Processor Class and Census_OSArchitecture.

The random forest method we implement uses a fixed number of variables at each splitting of the tree (each node) but the variables at each node themselves are selected completely randomly from the 14 we're working with. For example, if we specify that we want to try 2 variables at each node, the model randomly selects 2 variables out of the 14 and chooses the best among the two to continue building the tree.

As before, we first split the dataset into training and test data sets (75% and 25% respectively) and use these 14 predictors to predict the HasDetections response variable. If we use the Random Forest function in R to directly build a model on these predictor data, the default is to use 500 trees. The first model we build after removing all the NA values, is this default model. We see that R automatically tries 4 different variables at each split. The MSE for the residuals we obtain in this default model is found to be 0.2492 with only 33% of the variance explained by this model. This MSE is quite high and only a small part of the actual variability is captured by the model.


ROC for Random forest

To see if we can improve our accuracy, we first try to optimize the number of variables tried at each split. In this case, we find that the maximum accuracy is achieved when there are 2 variables tried at each split. For example, if we do 6 variables at each split and apply the resulting model to our test data, we have a 0.2494 MSE for the residuals, a marginal deterioration from the case when we tried 4 variables at a split (the R default). Only 20% of the variability is explained by the model in this case. If we do 2 variables at a split, the MSE for the residuals is 0.2488, and 47% of the variability is explained. As such, we build our model using 2 variables at each split. On applying on the test data, we report a MSE of 0.2488 for the residuals, which is not excellent. The computing time is about 2.53 seconds and the accuracy is 47% (false negative 53% and false positive 49%).

The following is the table showing the MSE of the residuals and the % of variance explained with the number of variables tried at each node.

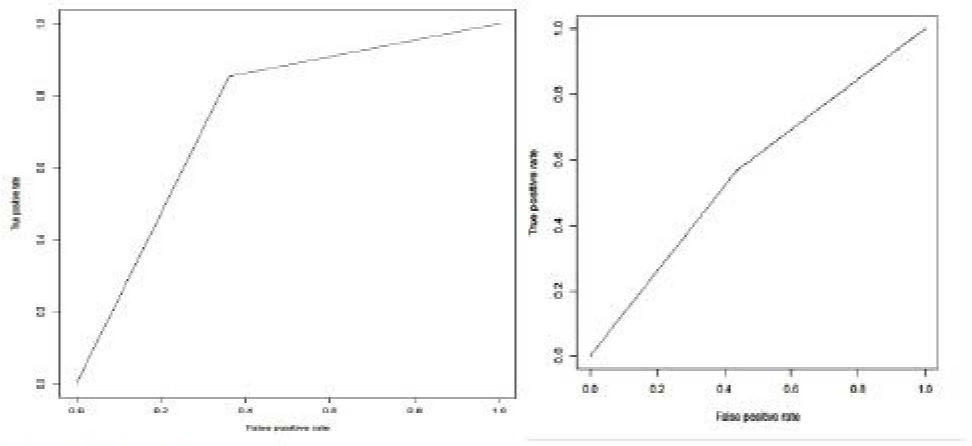
No.of variables at a node	MSE of residuals	% of variance explained
2	0.2488	47
4	0.2492	33
6	0.2495	20
14	0.25	15

It appears that the model becomes worse if we randomly try more variables at each node, and we verify that is indeed the case by writing a program in R.

Boosting

We now use a boosting method to build a new model and compare our results. We use the gbm library in R and, as in the last method, again split our data into training and test data sets. Boosting involves in this case decision trees of pre-decided size as the base learners. We fix the tree size for any one particular run of out algorithm, and also fix the shrinkage and interaction depth. On applying the gbm command with the default Gaussian distribution, shrinkage=0.01 and interaction depth=4, we build our models using the default gbm command. The tree size can be taken to be a variable in this case, but curiously we see that if we apply the resulting models to our test data, the test error does not vary too much on increasing the tree size. It does improve if we consider larger trees, but only marginally.

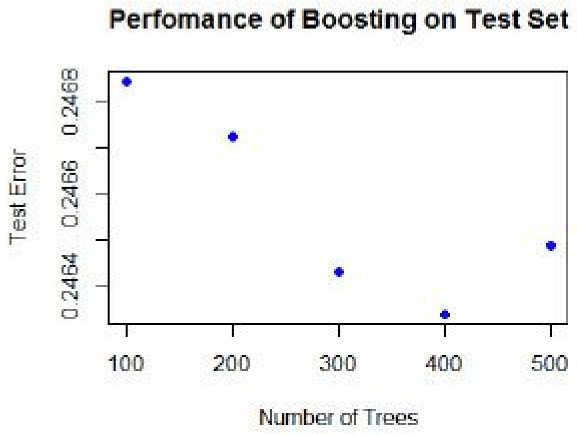
So the optimal case seems to be to consider tree sizes of 500 (any more hardly produces any improvement) assuming the shrinkage is held constant at 0.01 and the interaction depth is kept to 4. In that case, the test error comes to about 24.7%, which is near the error we got in the random



ROC for Boosting

forest method but a little better.

We then try to see if altering the shrinkage changes the accuracy of the model itself. We thus fix the tree size at 500 (larger sizes will typically give more accuracy, but only marginal improvements after 500 is noticed). At a shrinkage of 0.02, the test error at 500 tree size comes to 24.69%, for example. The following table gives the variation of the test errors with shrinkage.



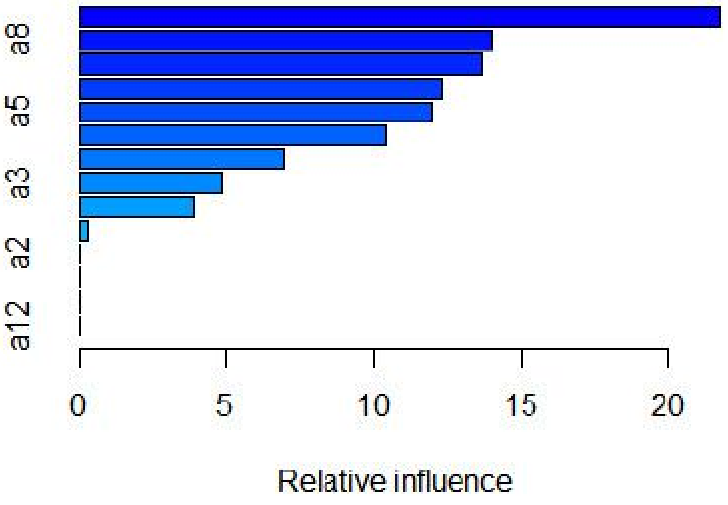
Shrinkage	Test error
0.01	24.71
0.02	24.69
0.03	24.68
0.04	24.67
0.05	24.67
0.47	24.61
0.48	24.62
0.49	24.64

The optimum is attained (assuming interaction depth is held fixed at 4) at shrinkage 0.47 where the test error for 500 trees is 24. 61%. However, at this shrinkage level, the accuracy of our model actually goes down at 500 trees as the following plot shows. The best accuracy is at 400 tree size (we checked only at 100, 200, 300, 400 and 500). The following is for shrinkage=0.47.

Let's now see which of the predictor variables are the most influential when the shrinkage (0.47) and tree size (400) are optimum to give us a test error of 24.61%. We write a code in R and obtain the following relative influences of the 14 predictors. In the following discussion, a1=ProductName, a2=IsBeta, a3=HasTpm, a4=RtpStateBitfield, a5=IsSxsPassiveMode, a6=Platform, a7=Processor, a8=IsProtected, a9=PuaMode, a10=SMode, a11=Firewall, a12=Census_DeviceFamily, a13=Census_ProcessorClass, a14=Census_OSArchitecture.

```
rel.inf
a6 21.7117808
a8 13.9852379
a4 13.6670157
a11 12.2926875
a5 11.9659385
a7 10.3752294
a14  6.9466766
a3  4.8358401
a1  3.9227281
```

```
a13  0.2968655
a2  0.0000000
a9  0.0000000
a10  0.0000000
a12  0.0000000
```



It's clear that the most influential predictors are Platform, IsProtected, RtpStateBitField, Firewall, IsSxsPassiveMode and Processor in that order while IsBeta, PuaMode, SMode and Census_DeviceFamily have little to no influence on the model.

All in all, we conclude that unlike the random forest case where we had a error of 24.88%, introducing boosting and optimizing shrinkage and tree size increase the accuracy. In this case, the optima is at shrinkage 0.47 and tree size around 400, where the test error is 24.61%.

In this case, the computing time is 3.41 seconds, false positive is 41%, false negative is 39% and accuracy is 59%.

Comparison of the models and conclusion

On comparing the four models we study in this report, we've determined that boosting probably gives the best accuracy, but the worst computation time. The SVM method is also good (faster by a little bit: 3.42s for boosting vs 2.74s for SVM) but lacks a little in accuracy (59% for boosting and 56% for SVM). Fastest model to fit is, quite unsurprisingly, linear regression taking just 1.3 seconds and also gives a good accuracy of 52%. Unless we need too much accuracy, regression is probably our best bet, especially if we wanted to run this experiment on all the 84 predictors, where computing time will become an issue.