Microsoft Malware Prediction

Abstract

This project intends to predict whether a Windows machine will get infected by malware or not, based on various features of the machine. Understanding all features and handling them separately are some of the big challenges in this classification problem. Most of the features are categorical and about half of them have tens or hundreds of levels. To keep the categorical variables under control, I combined levels and used embeddings. To predict the outcome, I applied knn, decision tree, random forest, gradient boosted trees and neural network on the same preprocessed data. Most of the models can help, though not efficiently, with predicting.

Problem definition and project goals

The goal of this study is to apply different machine learning models to predict if a computer will soon get infected by malware.

The dataset I used is from https://www.kaggle.com/c/malwareiiitb/data. The data and the machine infection status was generated by combining heartbeat and threat reports collected by Microsoft's endpoint protection solution, Windows Defender[1].

The dataset has 10424 observations, each with its unique id. The variable I want to predict is HasDetection, which indicates whether the malware is detected on the machine or not. Other than HasDetection, the dataset has 82 features, such as ProductName, EngineVersion, AppVersion, Processor, Firewall and so on. To view all features and its descriptions, please go to https://www.kaggle.com/c/malwareiiitb/data. I divided all features into three groups: binary variables, categorical variables and numeric variables. Even though binary variables are categorical variables, they will be processed differently. That is why I separated binary variables from categorical variables. Before using the data for this study, I need to handle missing values, normalize numeric variables, and handle categorical variables with many levels by combing levels and using embeddings. Next, I can apply different models on processed data and compare them to find which model works the best.

Related Work

I found a few works related to malware detection that have a complete dataset. One is called EMBER(Endgame Malware BEnchmark for Research). The authors Hyrum S. Anderson and Phil Roth mentioned in their paper that the motivation for creating malware detection using machine learning has not received nearly the same attention in the open research community as other applications. The EMBER dataset is a collection of features from PE files that serve as a benchmark dataset for researchers[2]. The dataset includes features extracted from 1.1M binary

[1] Malware Prediction, https://www.kaggle.com/c/malwareiiitb/data accessed 2 Dec 2019
[2] Endgame Malware BEnchmark for Research, https://github.com/endgameinc/ember accessed 2 Dec 2019

files: 900K training samples (300K malicious, 300K benign, 300K unlabeled) and 200K test samples (100K malicious, 100K benign)[3]. Each object includes the following types in data: the sha256 hash of the original file as a unique identifier; coarse time information (month resolution) that establishes an estimate of when the file was first seen; a label, which may be 0 for benign, 1 for malicious or -1 for unlabeled; and eight groups of raw features that include both parsed values as well as format-agnostic histograms[3]. In 'EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models', the authors experimented on the raw data, trained a gradient boosted decision tree model using LightGBM and released a benchmark LightGBM model.

Another dataset that is related to Malware is from a Microsoft Malware Classification Challenge on Kaggle in 2015[4]. Unlike this project, the goal of the competition is to develop the best mechanism to classify malware attacks into different malware families.

Data Exploration and preprocessing

1. Remove unnecessary variables

When reading in the csv file, I excluded variable MachineIdentifier, OSBuildLab and Census_OSVersion. OSBuildLab is a combined feature of Census_OSAchitecture, OSBuild, Platform and so on. CensusOSVersion is also a combination of other features in the dataset, such as Census_OSBuildNumber, Census_OSBuildReversion and Census_OSBranch. Because they are redundant, I removed them from the dataset. The dataset now has 80 variables and 10424 observations.

2. Handling variables with more than 40% of the missing values and variables with only one level.

There are 7 variable with more than 40% of missing values.

Variable name	Percentage of missing values
DefaultBrowsersIdentifier	0.9556792
PuaMode	0.9997122
Census_ProcessorClass	0.9957790
Census_InternalBatteryType	0.7158480
Census_IsFlightingInternal	0.8337490
Census_ThresholdOptIn	0.6396777
Census_IsWIMBootEnabled	0.6390061

Table1. The percentages of missing values of seven variables

[3] EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models, https://arxiv.org/pdf/1804.04637.pdf accessed 2 Dec 2019

[4] Microsoft Malware Classification Challenge(BIG 2015), https://www.kaggle.com/c/malware-classification accessed 7 Dec 2019

Before deciding whether to remove these 7 variables, I performed chi-square tests between each variable above with HasDetection, the variable we want to predict. I created dummy variables[5] for each of the 7 variables with NAs as 'missing' and observations with any values as 'observed'. If the test result does not show significance between a variable and HasDetection, I remove it. Because none of the chi-square test shows any significance, I removed all seven variables.

There are also three categorical variables in the dataset that only has one level. That is all observations have the same value regarding these three variables. They are AutoSampleOptIn, IsBeta and Census_IsFlightsDisabled. I removed them from the dataset.

3. Clean Binary variables

I divided the rest of the 70 variables into three groups. Binary variables, categorical variables and numeric variables. Binary variables are categorical variables with only two levels. Because I do not need to combine levels or embed them in later steps, I separated them from categorical variables with more than 2 levels.

There are 16 binary variables and 7 of them have missing values. From the following table, we can see that there are not many missing values. I simply replaced NAs with the mode values of the variables.

Variables name	Percentage of missing values
IsProtected	0.0034535687
SMode	0.0602455871
Firewall	0.0110322333
UacLuaenable	0.0009593246
Census_IsVirtualDevice	0.0014389870
Census_IsAlwaysOnAlwaysConnectedCapable	0.0079623945
Wdft_IsGamer	0.0311780507

Table2. The percentages of missing values of incomplete binary variables

4. Clean categorical variables with more than 2 levels

There are 22 categorical variables from the dataset with more than 2 levels. To handle the missing value among categorical variables, I treated them as a separate category by themselves, called 'unknown'.

For categorical variables Census_ActivationChannel, Census_PrimaryDiskTypeName, SkuEdition, Census_OSSkuName, Census_OSEdition, SmartScreen, Census_OSInstallTypeName, Census_OSBranch, Census_ChassisTypeName, I combined some of their levels to reduce the

[5] How to Diagnose the Missing Data Mechanism by Karen Grace-Marthin, https://www.theanalysisfactor.com/missing-data-mechanism/

total number of levels in each variable. The idea is inspired by Stephan Michaels, a participant of the Microsoft malware prediction competition[6]. The following table shows the number of levels of each variables before and after combining and how I combined them.

Variable name	nlevels before change	nlevels after change	Detail (Combine levels on the left side of the arrow to the level on the right side)
Census_ ActivationChannel	6	3	OEM:DM, OEM:NONSLP ->OEM Retail, Retail:TB:Eval -> Retail Volume:GVLK, Volume:MAK -> Volume
Census_ PrimaryDiskTypeName	5	3	UNKNOWN, unknown, Unspecified ->unknown
SkuEdition	8	7	Enterprise, Enterprise LTSB -> Enterprise
Census_OSSkuName	14	6	If level names have string ‘CORE’ in them, combine to a single level ‘CORE’; If a level name has ‘EDUCATION’, combine to a single level ‘EDUCATION; If a level name has ‘ENTERPRISE’, combine to a single level ‘ENTERPRISE’; if a level name has ‘PROFESSIONAL’, combine to a single level ‘PROFESSIONAL’. If a level name has ‘SERVER’, combine them to a single level ‘SERVER’.
Census_OSEdition	15	6	The same as Census_OSSkuName
SmartScreen	9	8	Off, off -> Off
Census_ OSInstallTypeName	9	6	Clean, CleanPCRefrech, IBSClean -> Clean Upgrade, UUPUpgrade -> Upgrade
Census_OSBranch	13	9	Level names contain ‘rs_prerelease’ -> rs_prerelease; Level names contain ‘rs1_release’ ->rs1_release; Level names contain ’rs2_release’ ->rs2_release; level names contain ‘rs3_release’-> rs3_release; level names contain ‘th1’ ->th1
Census_ChassisTypeName	22	21	Unknown, unknown -> Unknown

Table3. Detailed information of combining levels of 8 categorical variables

For the other categorical variables, I cannot combine some of their levels due to the limit of data description or level description. However, variables EngineVersion, AppVersion, AvSigVersion have too many levels that need to be handled. To solve this problem, I combined levels that contain less than 5% of the observations into one class, called ‘Other’. EngineVersion used to have 35 levels, but only 3 after the process. AppVersion used to have 60

[6] Stephan Michaels, https://github.com/imor-de/microsoft_malware_prediction_kaggle_2nd/tree/master/code

levels, but only 3 after. AvSigVersion used to have 1668 levels, but 1 level after. Because AvSigVersion only has 1 level, 'Other', I removed it from the dataset.

The 21 categorical variables we processed above are the ones labeled as factors in the dataset. Some variables are read in as integers, even though they are actually factors. These kinds of variables include 16 variables that have names end with 'identifier' and another 7 variables. Because level names of these variables are just non-negative integers. I replaced all NAs with -1, as a new level. Because most of these variables have more than hundred of levels. Thus, for most of them, I combined levels that contain less than 5% of the observations into one class, called 'Other'. The following table shows the name of the variables that I applied this process, the number of levels before and after the process.

Variable name	nlevels before	nlevels after
AVProductStatesIdentifier	506	3
CountryIdentifier	196	1
CityIdentifier	3717	1
OrganizationIdentifier	23	4
GeoNameIdentifier	172	2
LocaleEnglishNameIdentifier	132	3
IeVerIdentifer	88	5
Census_OEMNameIdentifier	303	7
Census_OEMModelIdentifier	4103	1
Census_ProcessorManufacturerIdentifier	4	3
Census_ProcessorModelIdentifier	988	1
Census_OSInstallLanguageIdentifier	40	5
Census_OSUILocaleIdentifier	46	5
Census_FirmwareManufacturerIdentifier	75	6
Census_FirmwareVersionIdentifier	3612	1
Wdft_RegionIdentifier	16	7
OSBuild	25	5
Census_OSBuildNumber	21	6
Census_OSBuildRevision	173	5

Table4. The number of levels before and after combining certain levels

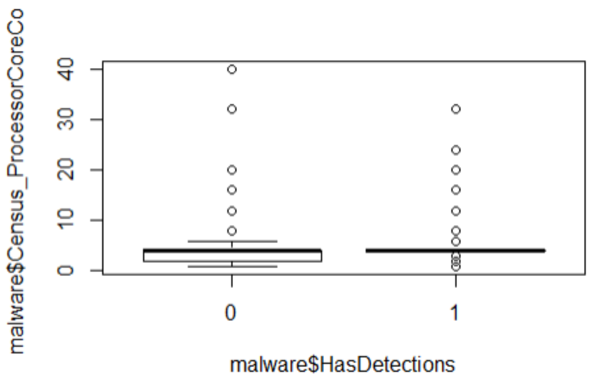
For variables that have number of levels equals to 1 after the process, I removed them from the dataset.

After all the processes above, I ended up with 39 categorical variables.

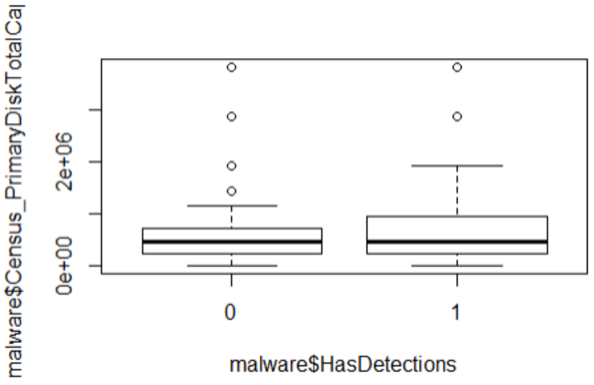
5. Cleaning numeric variables

Other than binary variables and categorical variables, the rest of the variables in the dataset are numeric. There are 8 numeric variables. From the results of t-tests and the following boxplots, we can conclude that variables Census_ProcessorCoreCount, Census_PrimaryDiskTotalCapacity, Census_TotalPhysicalRAM, Census_InternalPrimaryDiagonalDisplaySizeInInces, are associated with the dependent variable, HasDetection.

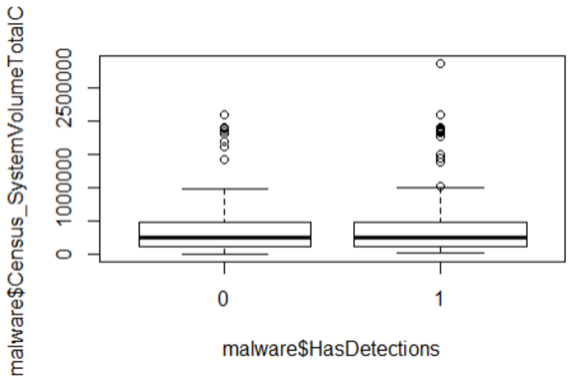
The following boxplots are between HasDetection and each numeric variable.



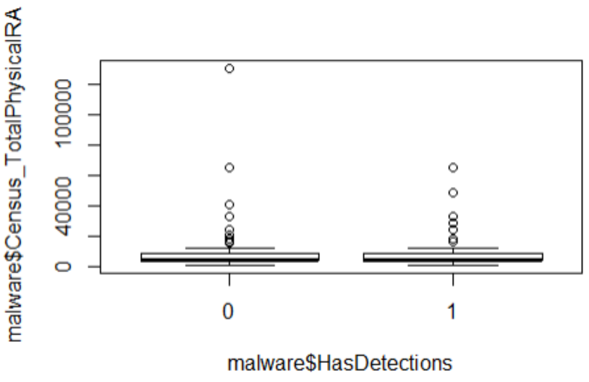
Census_ProcessorCoreCount



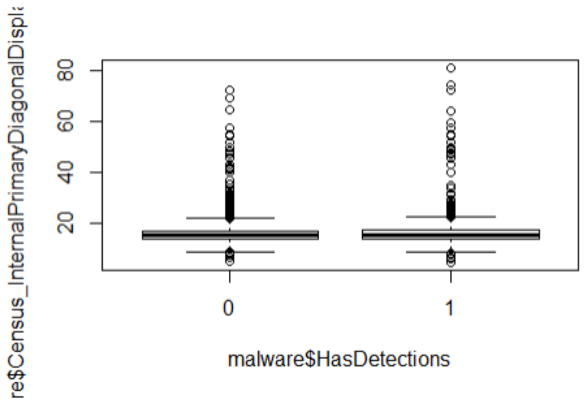
Census_PrimaryDiskTotalCapacity

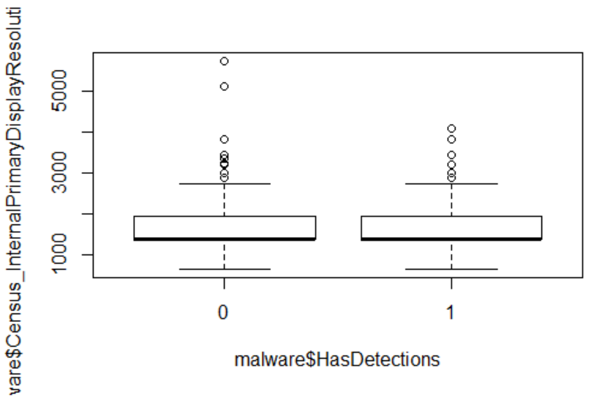


Census_SystemVolumeTotalCapacity

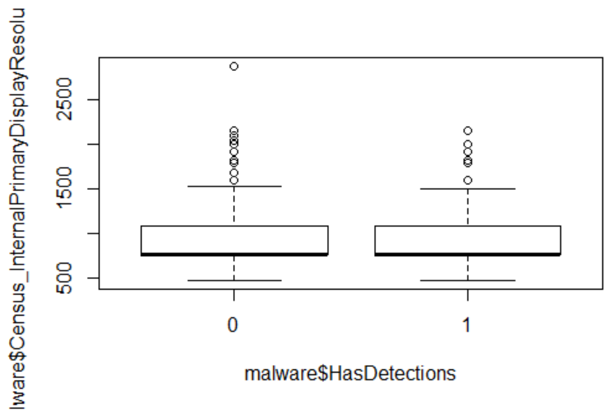


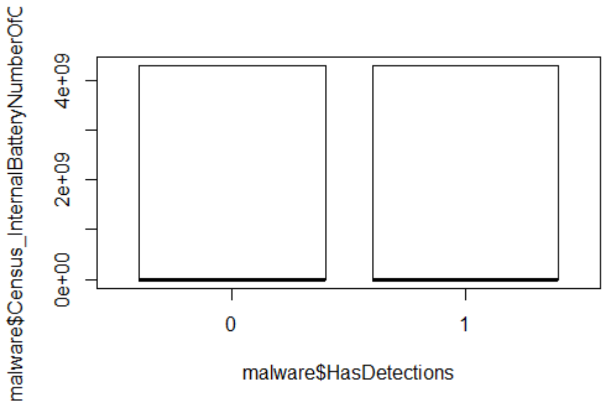
Census_TotalPhysicalRAM





Census_InternalPrimaryDiagonalDisplaySizeInInces Census_InternalPrimaryDisplayResolutionHorizontal





Census_InternalPrimaryDisplayResolutionVertical   Census_InternalBatteryNumberOfCharges

Even though all numeric variables have missing values in them, but most of them have less than 1% of the variables. To impute missing values, I simply used mean of each variable to replace corresponding NAs.

For this project, due to the amount of variables in the dataset, the percentage of missing values, and the limited computing power of my computer, I only chose simple imputation methods to replace missing values.

After processing binary variables, categorical variables and numeric variables, I ended up with a dataset with 63 variables, excluding HasDetection. After splitting the dataset into train and test, I scaled the numeric variables in the training dataset and applied the center and scale resulted from the training dataset on the test dataset.

6. Categorical embedding

Even though I already reduced the number of levels in almost all categorical variables, they still have more than 2 levels. That is to say, I still need to process them before I can fit them to machine learning algorithms. I decided to compute embedding vectors for each categorical variables.

First of all, I divided the training data into two parts, with 90 % of training and 10% of validation data.

I created a input layer for each categorical variable and a input layer for the rest of the variables in the dataset. Since the dataset has 39 categorical variables, there are 39 input layers with shape of c(1) and 1 input layer with a shape of c(24). Because the amount of categorical variables, it is not practical to tune output dimensions of every embedding vectors on my laptop, especially with other hyper-parameters of neural network model to tune. I applied a general rule to keep things simple. The general rule of thumb about the number of embedding dimensions is the 4th root of the number of levels in the variable[7]. Since the number of levels of the categorical levels are between 3 and 21. The embedding dimensions should be between 1.31 and 2.14. Therefore, I set the embedding dimension to 2 for every categorical variables. This way, I could define a embedding layer for each categorical variable. Then I created a R script file, called malware_train.R. There, I concatenated the 39 embedding layers with the input layer for the other variables and passed them through 2 stacked dense layers. In malware_train.R, I defined a flag for all hyper-parameters I wanted to tune. That includes number of nodes in each dense layer, activation function for each layer, dropout rate for each dropout layer, batch size, epochs and learning rate. The following graph shows the neural network architecture.

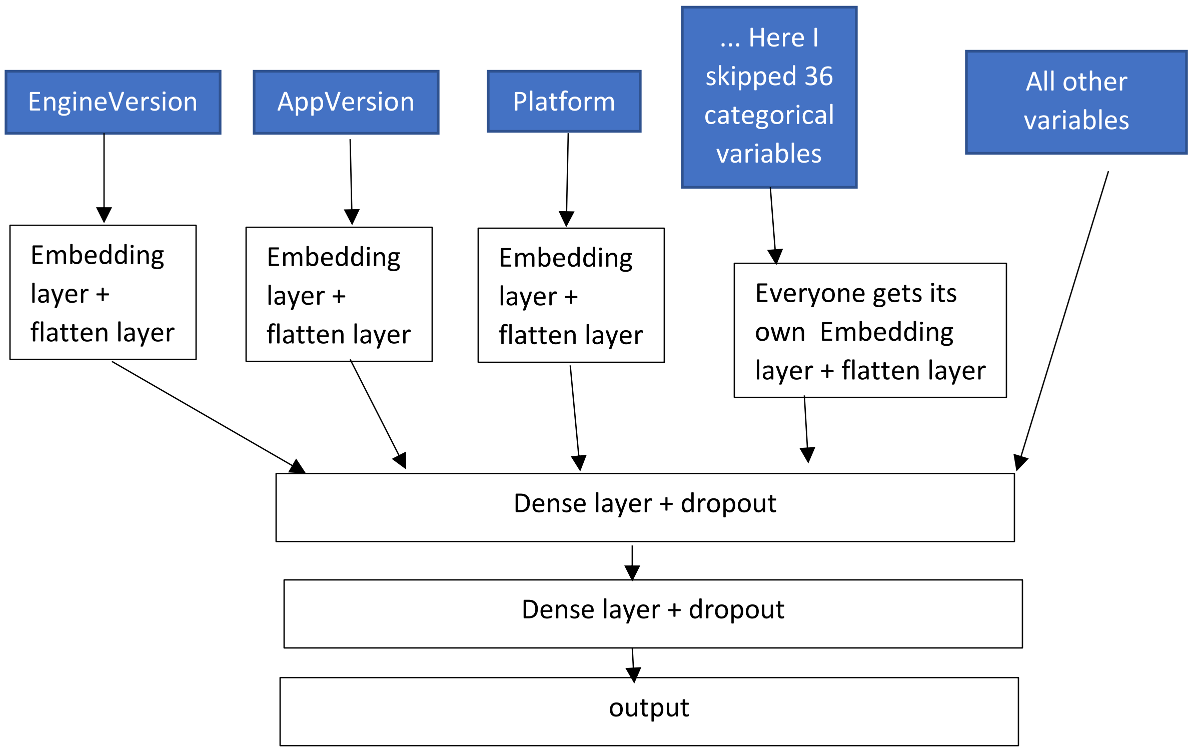
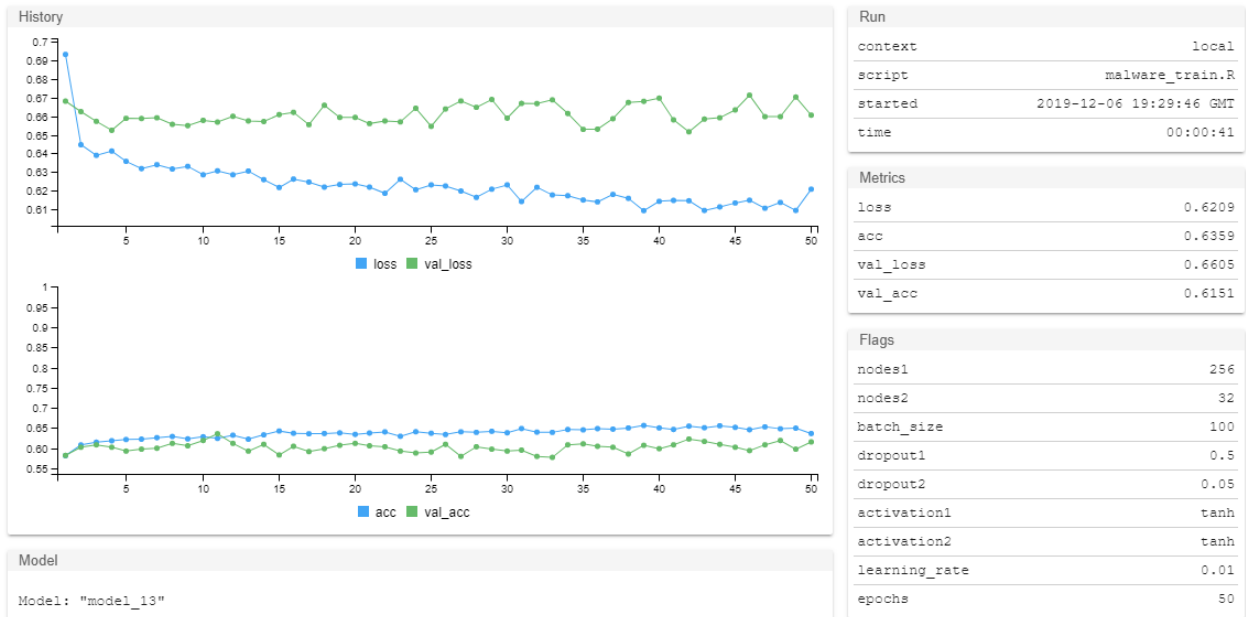

EngineVersion
AppVersion
Platform
... Here I skipped 36 categorical variables
All other variables
Embedding layer + flatten layer
Embedding layer + flatten layer
Embedding layer + flatten layer
Everyone gets its own Embedding layer + flatten layer
Dense layer + dropout
Dense layer + dropout
output

Figure 1. Architecture of the neural network model

The best model has 256 nodes in the first dense layer and uses tanh as activation function. The dropout rate for the first dropout layer is 0.5. The model also has 32 nodes in the second dense layer and uses tanh as activation function. The dropout rate for the second dropout layer is 0.05. Batch size is 100, epochs is 50 and learning rate is 0.01. The following graphs show that the best model overfits.

[7] Introducing TensorFlow Feature Columns, https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html accessed 4 Dec, 2019



Run

```
context                              local
script                      malware_train.R
started            2019-12-06 19:29:46 GMT
time                              00:00:41
```

Metrics

```
loss                                0.6209
acc                                 0.6359
val_loss                            0.6605
val_acc                             0.6151
```

Flags

```
nodes1                                 256
nodes2                                  32
batch_size                             100
dropout1                               0.5
dropout2                              0.05
activation1                           tanh
activation2                           tanh
learning_rate                         0.01
epochs                                  50
```

Model

```
Model: "model_13"
```

Figure2. Summary of the best tuned neural network model

After running the best model on all training data, I could get the categorical embeddings and use them as input to other machine learning models.

Data analysis and experimental results

I fit knn, decision tree, random forest and gradient boosted machine on the same data I got after preprocessing. Because I used neural network to compute categorical embeddings, the data analysis also includes the result from neural network model.

To tune hyper-parameters in these models, I used 10-fold cross-validation and the auto-tune function, train, from caret. Because the variable we want to predict is a factor with 2 levels, I used accuracy and AUC results to compare different models. The following table shows the accuracy and AUC results of each tuned model.

Model	Accuracy	AUC
KNN	0.5715805	0.6203509
Decision tree	0.6059968	0.6753918
Random forest	0.6087482	0.6667201
Gradient boosted machine	0.6116289	0.6787485
Neural network	0.6190019	0.6659516

Table5. Accuracy and AUC of machine learning models

Other than KNN mode, the other four models provide a better result with accuracy around 0.61 and auc around 0.67. The gradient boosted tree model generated the best result with an accuracy of 0.6116289 and AUC of 0.6787485.

When running the random forest model, the model also generated the top 20 most important variables. They are showed in a table below.

SmartScreen.1
SmartScreen.2
AVProductsInstalled.1
AVProductsInstalled.2
Census_IsVirtualDevice
EngineVersion.1
AVProductStatesIdentifier.2
EngineVersion.2
AVProductStatesIdentifier.1
Census_InternalPrimaryDiagonalDisplaySizeInInches
Census_PrimaryDiskTotalCapacity
Census_TotalPhysicalRAM
Census_IsTouchEnabled
AVProductsEnabled.2
Census_OSBuildNumber.1
Census_MDC2FormFactor.1
Census_InternalPrimaryDisplayResolutionHorizontal
AVProductsEnabled.1
RtpStateBitfield.1
Census_GenuineStateName.2

Table 6. The 20 most important variables

Most of these variables are directly related to anti-virus product or malware detection, such as SmartScreen(a cloud-based anti-phishing and anti-malware component included in several Microsoft products[8])or AVProductsInstalled. I am surprised to find that the size of RAM and disk capacity are playing an important role too.

Conclusion

In this project, I fit KNN, decision tree, random forest, gradient boosted tree and neural network models on the processed data. Unfortunately, these models cannot perform efficiently. The AUCs are between 0.6 and 0.7. The results from Kaggle leaderboard are not encouraging either and they are also between 0.6 and 0.7.

[8] Microsoft SmartScreen, https://en.wikipedia.org/wiki/Microsoft_SmartScreen accessed 6 Dec, 2019

Predicting whether a Windows machine will be infected is not a trivial task. In this project, I spent great amount of time preparing data. And I believe there are still many techniques I can try and more work I can do. Future analysis could include different missing value imputation methods, dimensionality reduction and time series analysis.

Reference:

[1] H. Anderson and P. Roth, "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models", in ArXiv e-prints. Apr. 2018.

[2] Endgame Malware BEnchmark for Research, https://github.com/endgameinc/ember accessed 2 Dec 2019

[3] Alvira Swalin , "How to Handle Missing Data", https://towardsdatascience.com/how-to-handle-missing-data-8646b18db0d4, Jan. 2018.

[4] Introducing TensorFlow Feature Columns, https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html accessed 4 Dec 2019.

[5] Karen Grace-Marthin, "How to Diagnose the Missing Data Mechanism", https://www.theanalysisfactor.com/missing-data-mechanism/ accessed 5 Dec 2019.

[6] Stephan Michaels, https://github.com/imor-de/microsoft_malware_prediction_kaggle_2nd/tree/master/code

[7] Microsoft SmartScreen, https://en.wikipedia.org/wiki/Microsoft_SmartScreen accessed 6 Dec 2019.