**Project: Microsoft Malware Prediction, a Kaggle Competition**

Name: Jason Weidner

Class and Semester: CSPB3022, Summer

**Intro**

Kaggle has partnered with Georgia Tech and Microsoft to release a large volume of data that includes information about machines infected with malware and their characteristics. The intention is to provide the public an opportunity to model which characteristics of the computers are most useful in determining a machine’s risk of infection.

This report summarises my analysis of the data, including some basic data exploration as well as some feature engineering to determine which features of a computer can be used to estimate the machine’s likelihood of infection.

**Data Size and Quality**

Kaggle has provided two main data files – test and train -- each of which are very large (3.8 GB and 4.4GB) even in a comma-separated value format. As a result, it is difficult to process the data in their entirety in memory on a standard laptop.

The dataset includes 84 columns of data. Appendix A shows the column names and datatypes.

Quick exploration of the data (see reference 1) shows that several features contain little data. The dataset also contains information for that are touchscreen-enabled.

I had initial challenges creating simple models due to the presence of NaN, infinity and very large values.

Also, the training data includes several categorical features, where conceptually unique values of 1, 2, 3, etc. are enough to describe the feature.

**Data Cleaning**

*Initial cleaning*

Columns containing <70% of data were dropped. These include:

* PuaMode
* Cesus\_ProcessorClass
* DefaultBrowserIdentifier
* Census\_IsFlightingInteral
* Census\_InternalBetteryType
* Census\_ThresholdOptIn
* Census\_IsWINBoostEnabled
* SmartScreen
* OrganizationIdetnfier

To simplify the analysis only non-touch screen devices were considered so Census\_IsTouchEnabled was dropped too.

Features were compared for correlation so that highly correlated (>60%) features could be eliminated from consideration. In this way the variation in the dataset captured by only a single variable would be used rather than the full set of correlated features. Figure 1 (see reference 7) shows the correlations between features (that have not already been dropped due to sparse data).

A picture containing clock

Description automatically generated

Because of the large number of features in the image, here is a list of the pairs of features (and their correlation coefficient) that have correlations within (0.6, 1.0):

|  |  |  |
| --- | --- | --- |
|  | Corr. coeff. | Feature Pairs |
| 1 | 0.988 | Census\_OSUILocaleIdentifier |
| Census\_OSInstallLanguageIdentifier |
| 2 | 0.938 | Census\_OSBuildNumber |
| OsBuild |
| 3 | 0.902 | Census\_InternalPrimaryDisplayResolutionVertical |
| Census\_InternalPrimaryDisplayResolutionHorizontal |
| 4 | 0.633 | AVProductStatesIdentifier |
| AVProductsInstalled |
| 5 | 0.599 | GeoNameIdentifier |
| CountryIdentifier |
| 6 | 0.599 | Census\_TotalPhysicalRAM |
| Census\_ProcessorCoreCount |

If two attributes were correlated, one of them was dropped from use in training data.

Subsequent to this cleaning process model builds were attempted – but none would work due to memory challenges.

*Second cleaning approach*

As a second approach I followed the process outlined by ref 10 et al before adding my own data processing. First, the Pandas method *factorize* was used to re-interpret all categorical data (ref 8) to simplify the values in those columns. After factorization, in-memory usage dropped by over 1.6 GB and the size of the training file decreased from 4.38 GB to 2.67 GB. Run-times for factorization were significant so raw data files were processed individually with results saved to new files.

After factorization, training data was examined again for fraction of missing values. Garbage collection (ref 9) was used extensively to manage laptop memory while processing the raw data files.

On this pass of data processing, fewer columns needed to be dropped due to missing values. Unsurprisingly, the correlations between data features were unchanged, so one feature of each pair of highly correlated features were dropped from the training data too.

Critically, the method *dropna* was used to further simplify the training data. Without this step model builds kept failing. Even with *dropna* being used there were ~3M rows of data in the training dataset alone.

Samples of 20k rows were randomly selected from the processed training data and used

to facilitate quick model run times.

**Summary of columns dropped**

|  |  |  |
| --- | --- | --- |
|  | Feature | Reason dropped |
| 1 | PuaMode | >30% data missing |
| 2 | Census\_ProcessorClass | >30% data missing |
| 3 | DefaultBrowserIdentifier | >30% data missing |
| 4 | Census\_IsFlightingInteral | >30% data missing |
| 5 | Census\_InternalBatteryType | >30% data missing |
| 6 | Census\_ThresholdOptIn | >30% data missing |
| 7 | Census\_IsWINBoostEnabled | >30% data missing |
| 8 | SmartScreen | >30% data missing |
| 9 | OrganizationIdentifier | >30% data missing |
| 10 | Census\_IsTouchEnabled | Touch screen |
| 11 | AVProductStatesIdentifier | Correlated feature |
| 12 | ProcessorCoreCount | Correlated feature |
| 13 | Census\_OSInstallLanguageIdentifier | Correlated feature |
| 14 | OsBuild | Correlated feature |
| 15 | Census\_InternalPrimaryDisplayResolutionHorizontal | Correlated feature |
| 16 | CountryIdentifier | Correlated feature |

Appendix A identifies which columns of the dataset were dropped.

Also ‘Machine Identifier’ was used as an index, not as a training feature. Lastly, ‘Has Infection’ was used as the label and not as a training feature.

**General Approach**

Initial attempts were made to use a step-forward method to identify which features were most significant in generating a linear regression of the data – but model building functions repeatedly returned errors without results. Only when the data was fully processed (as described above) would model builds generate results.

On a second attempt, after significant challenges due to dataset size, models were built using k-means clustering and k-nearest neighbors.

*K-Means*

K-means is a clustering algorithm that allows a user to define, upfront, how many clusters to in which the training data is divided. Cluster centers are then determined iteratively and reported back as an output of the training. New data is then compared to the cluster centers and are labelled based on the shortest distance to a cluster center. Importantly, if there are n-features in the dataset, then the cluster centers are in n-dimensional space (with similarly n-dimensional distances from one another).

Since the selection of the number of clusters is arbitrary in this case (maybe there are, generally, 8 ‘clusters’ of computer users?) I tested k-means over a range of cluster number from 1 to 10. Having a cluster of 1 would mean all training data were in the same cluster, and any test data would be a random guess as to whether it would be infected or not.

Below are the results:

|  |  |
| --- | --- |
| No. of clusters | Avg. correct |
| 1 | 0.5071 |
| 2 | 0.51165 |
| 3 | 0.44 |
| 4 | 0.166 |
| 5 | 0.2421 |
| 6 | 0.19635 |
| 7 | 0.1447 |
| 8 | 0.18335 |
| 9 | 0.15475 |

Even at best (i.e. 2 clusters) the chance of predicting infection is only slightly better than randomly guessing.

*K-Nearest Neighbors*

Because of the disappointing approach using k-means, I also built models using k-nearest neighbors. KNN, for short, is a clustering algorithm that classifies a test data point based on the classification of its k-nearest neighbors. Similar to k-means, the user must define upfront how many neighbrs to consider when classifying test data. In turn I tested several models that consider 1 to 10 neighbors when classifying a new data point.

Here are the results:

|  |  |
| --- | --- |
| No. of neighbors | Avg. correct |
| 1 | 0.5063642644970974 |
| 2 | 0.5190501160768115 |
| 3 | 0.5063642644970974 |
| 4 | 0.5136770570507103 |
| 5 | 0.5113163628533621 |
| 6 | 0.5214597956542668 |
| 7 | 0.5132332133211863 |
| 8 | 0.5163386572478043 |
| 9 | 0.5169175137584544 |

Similar to k-means, the results are only slightly above randomly guessing if a computer is infected or not.

**Possible Improvements**

There are several improvements possible with my approach to analysing the data. First, k-means becomes less effective at higher dimensional data (the so called ‘curse of dimensionality’). To address this more carefully, effort could be made to sensibly reduce the dimensionality of the training data. Either sensitivity analysis could be executed – where individual features are removed systematically – or subject matter expert input could help aggregate or simplify the training dataset itself. For example, it would be preferable if three features of the existing training data could be aggregated into one feature (while still honouring the underlying data). It is likely that such aggregation would need expert evaluation to ensure it honors the data.

Second, there are other methods for classifying data that very well may work better than clustering. Logistic regression, in particular, comes to mind.

Third, I only generated models using samples of the training data because I was struggling with memory-related challenges. After having used *dropna* it is conceivable that the entire dataset could be used for training and result in better clustering.

**Conclusions**

While the models I generated were quite poor, there is value in having processed the training and test data. At this point it would be relatively easy to examine improvements to existing modelling approaches (i.e. k-means or k-nearest neighbors) or to evaluate other methods (logistic regression).

Furthermore, data processing and data scarcity were critical challenges that had to be addressed before modelling commenced. Indeed, the vast majority of the effort to generate models was preparing the input data so that model-building functions would run.

Additionally, clustering approaches may not be the best technique to use with the training data as-is. It is possible, though, that features may be aggregated with subject matter expert input. Such improvements of the training data would provide options for continuing with cluster-based methods for malware infection prediction.

**References**

1. Exploratory analysis of the data:
   * <https://www.kaggle.com/artgor/is-this-malware-eda-fe-and-lgb-updated>
2. Exploratory analysis of the data
   * <https://www.kaggle.com/youhanlee/my-eda-i-want-to-see-all>
3. Method for quickly calculating AUC ROC
   * <https://www.kaggle.com/c/microsoft-malware-prediction/discussion/76013>
4. Light gradient boost analysis of the full MSFT dataset
   * <https://www.kaggle.com/sjb1988/lgb-python-basic-features-only>
5. Explanation of boosting vs bagging approaches to modeling, as well as justification for light gradient boost rather than XGBoost
   * <https://www.kaggle.com/questions-and-answers/103834>
6. Comparison of LGB and XGB
   * <https://www.codeastar.com/lgb-winning-gradient-boosting-model/>
7. Correlation between features:
   * <https://www.kaggle.com/hrmello/eda-most-correlated-variables-to-the-target>
8. Garbage collection:
   * <https://stackify.com/python-garbage-collection/>
9. Pandas factorize method:
   * <https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.factorize.html>
10. Input data processing with guidance on garbage collection and factorization:
    * <https://www.kaggle.com/tunguz/malware-feature-engineering-full-train-and-test>

**Appendix A**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Column Name | Data Type | Dropped |
| 1 | 'MachineIdentifier' | 'category', | Index |
| 2 | 'ProductName' | 'category', |  |
| 3 | 'EngineVersion' | 'category', |  |
| 4 | 'AppVersion' | 'category', |  |
| 5 | 'AvSigVersion' | 'category', |  |
| 6 | 'IsBeta' | 'int8', |  |
| 7 | 'RtpStateBitfield' | 'float16', |  |
| 8 | 'IsSxsPassiveMode' | 'int8', |  |
| 9 | 'DefaultBrowsersIdentifier' | 'float16', | Dropped |
| 10 | 'AVProductStatesIdentifier' | 'float32', | Dropped |
| 11 | 'AVProductsInstalled' | 'float16', |  |
| 12 | 'AVProductsEnabled' | 'float16', |  |
| 13 | 'HasTpm' | 'int8', |  |
| 14 | 'CountryIdentifier' | 'int16', | Dropped |
| 15 | 'CityIdentifier' | 'float32', |  |
| 16 | 'OrganizationIdentifier' | 'float16', | Dropped |
| 17 | 'GeoNameIdentifier' | 'float16', |  |
| 18 | 'LocaleEnglishNameIdentifier' | 'int8', |  |
| 19 | 'Platform' | 'category', |  |
| 20 | 'Processor' | 'category', |  |
| 21 | 'OsVer' | 'category', |  |
| 22 | 'OsBuild' | 'int16', | Dropped |
| 23 | 'OsSuite' | 'int16', |  |
| 24 | 'OsPlatformSubRelease' | 'category', |  |
| 25 | 'OsBuildLab' | 'category', |  |
| 26 | 'SkuEdition' | 'category', |  |
| 27 | 'IsProtected' | 'float16', |  |
| 28 | 'AutoSampleOptIn' | 'int8', |  |
| 29 | 'PuaMode' | 'category', | Dropped |
| 30 | 'SMode' | 'float16', |  |
| 31 | 'IeVerIdentifier' | 'float16', |  |
| 32 | 'SmartScreen' | 'category', | Dropped |
| 33 | 'Firewall' | 'float16', |  |
| 34 | 'UacLuaenable' | 'float32', |  |
| 35 | 'Census\_MDC2FormFactor' | 'category', |  |
| 36 | 'Census\_DeviceFamily' | 'category', |  |
| 37 | 'Census\_OEMNameIdentifier' | 'float16', |  |
| 38 | 'Census\_OEMModelIdentifier' | 'float32', |  |
| 39 | 'Census\_ProcessorCoreCount' | 'float16', | Dropped |
| 40 | 'Census\_ProcessorManufacturerIdentifier' | 'float16', |  |
| 41 | 'Census\_ProcessorModelIdentifier' | 'float16', |  |
| 42 | 'Census\_ProcessorClass' | 'category', | Dropped |
| 43 | 'Census\_PrimaryDiskTotalCapacity' | 'float32', |  |
| 44 | 'Census\_PrimaryDiskTypeName' | 'category', |  |
| 45 | 'Census\_SystemVolumeTotalCapacity' | 'float32', |  |
| 46 | 'Census\_HasOpticalDiskDrive' | 'int8', |  |
| 47 | 'Census\_TotalPhysicalRAM' | 'float32', |  |
| 48 | 'Census\_ChassisTypeName' | 'category', |  |
| 49 | 'Census\_InternalPrimaryDiagonalDisplaySizeInInches' | 'float16', |  |
| 50 | 'Census\_InternalPrimaryDisplayResolutionHorizontal' | 'float16', | Dropped |
| 51 | 'Census\_InternalPrimaryDisplayResolutionVertical' | 'float16', |  |
| 52 | 'Census\_PowerPlatformRoleName' | 'category', |  |
| 53 | 'Census\_InternalBatteryType' | 'category', | Dropped |
| 54 | 'Census\_InternalBatteryNumberOfCharges' | 'float32', |  |
| 55 | 'Census\_OSVersion' | 'category', |  |
| 56 | 'Census\_OSArchitecture' | 'category', |  |
| 57 | 'Census\_OSBranch' | 'category', |  |
| 58 | 'Census\_OSBuildNumber' | 'int16', |  |
| 59 | 'Census\_OSBuildRevision' | 'int32', |  |
| 60 | 'Census\_OSEdition' | 'category', |  |
| 61 | 'Census\_OSSkuName' | 'category', |  |
| 62 | 'Census\_OSInstallTypeName' | 'category', |  |
| 63 | 'Census\_OSInstallLanguageIdentifier' | 'float16', | Droppped |
| 64 | 'Census\_OSUILocaleIdentifier' | 'int16', |  |
| 65 | 'Census\_OSWUAutoUpdateOptionsName' | 'category', |  |
| 66 | 'Census\_IsPortableOperatingSystem' | 'int8', |  |
| 67 | 'Census\_GenuineStateName' | 'category', |  |
| 68 | 'Census\_ActivationChannel' | 'category', |  |
| 69 | 'Census\_IsFlightingInternal' | 'float16', | Dropped |
| 70 | 'Census\_IsFlightsDisabled' | 'float16', |  |
| 71 | 'Census\_FlightRing' | 'category', |  |
| 72 | 'Census\_ThresholdOptIn' | 'float16', | Dropped |
| 73 | 'Census\_FirmwareManufacturerIdentifier' | 'float16', |  |
| 74 | 'Census\_FirmwareVersionIdentifier' | 'float32', |  |
| 75 | 'Census\_IsSecureBootEnabled' | 'int8', |  |
| 76 | 'Census\_IsWIMBootEnabled' | 'float16', | Dropped |
| 77 | 'Census\_IsVirtualDevice' | 'float16', |  |
| 78 | 'Census\_IsTouchEnabled' | 'int8', | Dropped |
| 79 | 'Census\_IsPenCapable' | 'int8', |  |
| 80 | 'Census\_IsAlwaysOnAlwaysConnectedCapable' | 'float16', |  |
| 81 | 'Wdft\_IsGamer' | 'float16', |  |
| 82 | 'Wdft\_RegionIdentifier' | 'float16', |  |
| 83 | 'HasDetections' | 'int8' |  |
| 84 | 'HasDetections' | 'int8' |  |