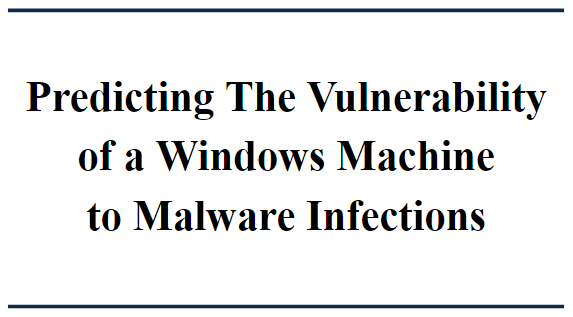


**Predicting the Vulnerability of**

**Windows Machine to Malware Infections**

**using different Classification Models**



**IS424 Data Mining and Business Analytics**

**G1 Team 2**

Tan Kin Meng

Alvin Lim

Eugene Kim

Zican Shi

Aakshi Narang

# Abstract

Cyberattacks such as malware are an inevitable source of worry for many users of smart devices. While antivirus software does aim to defend machines against malicious infections, there is scope for us to identify machines that are vulnerable early, depending on their various properties. Data mining and machine learning enables us to develop classification models that aid in such predictions. We obtained a train dataset of around 8.9 million rows of data with each observation corresponding to a unique Windows machine identifier and other 83 different features, including one target variable, HasDetection 0 or 1, indicating whether malware is present or not. Most features in the dataset are categorical variables. During data-preprocessing, we eliminated features with a large percentage of missing values, removed categorical features with a large percentage of values belonging to a single class and removed irrelevant features according to our domain knowledge and research on what affects the vulnerability of a device to malware infection.

Besides filtering out irrelevant features for the purpose of target variable prediction, we performed dimensionality reduction such as Principal Component Analysis (PCA), feature encoding (Label Encoding and One-Hot Encoding) for categorical values, standardized and did a log-transformation on continuous features “Census\_SystemVolumeTotalCapacity” and ''Census\_InternalBatteryNumberOfCharges”. In the end, we decided not to perform One-Hot Encoding and PCA as they reduced the overall model performance when tested on a sample dataset of 100,000.

We then built a variety of classification models - XGBoost, CatBoost, LightGBM, Support Vector Machine, Multi-Layer Perceptron, Random Forest, Logistic Regression and K-NN. Evaluating those models by performance metrics with mainly accuracy (because the distribution of target variables is balanced) and AUC-ROC curves, which provides a clear visualization of the performance of the models created. From this research, we conclude LightGBM is the best performing classification model which successfully classifies whether a Windows machine will have malware infections detected based on 38 different properties of the machine with 63.48% accuracy. However, there are plenty of future work that could be looked at and we will focus in our research on discussing (1) how to possibly use feature engineering and modeling to achieve improved performance, (2) other ways of feature encoding and managing sparse matrix and (3) implement custom stacking ensembles using the models we have already created.

# Table of Contents

1. Introduction / Problem Statement ---------------------------------------------- 4
2. Motivation ------------------------------------------------------------------------- 4
3. Literature Review ----------------------------------------------------------------- 5
4. Dataset ------------------------------------------------------------------------------ 6

4.1 Source of the Dataset --------------------------------------------------------- 6

4.2 Overview of the Dataset ------------------------------------------------------ 6

1. Methodology ------------------------------------------------------------------------ 8

5.1 Exploratory Data Analysis --------------------------------------------------- 8

5.2 Data Pre-processing----------------------------------------------------------- 11

5.3 Model Creation ---------------------------------------------------------------- 13

1. Results and Discussion ----------------------------------------------------------- 23
2. Conclusion and Future Work ---------------------------------------------------- 25
3. References ------------------------------------------------------------------------- 26

# Introduction / Problem statement

With the high adoption of personal computing devices, cyberthreat is becoming more prevalent. According to Microsoft, the malware industry is becoming increasingly more well-organized and better funded. There are many syndicates who invest heavily in technologies and capabilities built to circumvent conventional security protection. Hence, enterprises need to take actions against emerging cyber threats (Microsoft, 2018). Although developing counter mechanisms against the various malware is important, the best case scenario to fight against the malware industry is thought to be detecting and locating the causes and contributing factors of computer malware thus predicting and preventing malware from users of Microsoft.

With this in mind, our aim is to employ classification algorithms such as **eXtreme Gradient Boosting, Category Boosting, Light Gradient Boosting Machine (LGBM), Logistic Regression, Support Vector Machines (SVM), Multi-Layered Perceptrons** to detect the presence of malware in client’s personal computing device (0 signifying no presence of malware and 1 signifying presence of malware) and evaluate the methods with **Area Under the Receiver Operating Characteristics Curve (AUC ROC)** as well as **accuracy** (with the balance dataset on hand).

# Motivation

While plenty of antivirus programmes aim to detect malware on machines by scanning through all applications, data mining techniques can be used to go a step further and predict the vulnerability of a machine to being infected by malware. By identifying machines with higher probability of malware infection based on its various properties, suitable steps can be taken to improve the machine’s resilience and reduce its probability of being infected, thus keeping systems as well as users secure. This will lead to real world benefits such as **reduction in financial losses, increased protection of intellectual property and brand property** (Vijayanand & Arunlal, 2019) . With more than **one billion enterprise** and consumer customers (Microsoft, 2018)., anticipating a vulnerable machine can help Microsoft make a tremendous impact on their clients.

# Literature Review

**Background information and Current Research on Malware**

It has been found that 350,000 new pieces of malware are detected every day with over 7 billion malware attacks being reported in 2019 (Jovanović, 2019). With nearly 980 million malware programs out in the world, the need for more awareness and understanding of the different kinds of malware has become more significant (Jovanović, 2019).

This has, in turn, brought a rapid increase in the number of studies related to malware detection and analysis (O¨ mer et al., 2020). Researchers are trying to understand the nature of malwares as well as using data mining techniques to develop new techniques for detecting malware based on behavior, heuristics, and model checking (O¨ mer et al., 2020). In addition, models to predict the percentage of hosts in a given population (we use country in our experiments) that will be infected by a particular piece of malware, given some historical data about the malware and the hosts (Kang et al., 2016).

We note that most current research directly identifies the characteristics of the malware itself as well as the classification of the types of malware. Our research differs from other popular research as we are predicting based on the characteristics of the machine itself.

To summarize the findings directly related to our dataset, we looked through top Kaggle submission as well as medium articles written by participants of the competition.

**Machine learning models to be applied in our dataset**

One of the highly recommended machine learning models is LightGBM due to the nature of the dataset . The dataset contains multiple features with high cardinality (high number of unique values) which *Light Gradient Boosting Model (LGBM)* can handle and it can also handle categorical features with minimal memory utilization (Yee, 2019). *eXtreme Gradient Boosting (XGboost)* is also another frequently used machine learning model (Kouroupetroglou, 2019) due to its fast processing speed and parallel processing capabilities which is required for the huge volume of dataset. Others also tried stacking ensembles by combining two or more base classifiers to obtain better results. (Jugaloza, 2020). Hence, in summary, many of the participants used Boosting classifiers and we can try using them in our project.

**Encoding methods and data-preprocessing steps**

Several methods of encoding are also extensively discussed by multiple participants given the high number of categorical variables in the dataset, mainly nominal. In an article written by Yee (2019), it was stated that One-Hot Encoding could result in a sparse matrix and high dimensionality, on top of the already high-dimensional dataset of (83 columns). As such, the author stated that the options available are frequency, ordinal and mean target encoding, but each of them has their pros and cons. For example, ordinal encoding might not work for the nominal categorical features in the dataset which have no ordinal relationship. It is also interesting to note that one of the Kaggle participants, Vladislav Bogorod, managed to overcome the sparse matrix problems presented in the dataset from performing one-hot encoding and his work can be considered for future work (Bogorod, 2019).

**Other relevant research**

The research paper titled “RiskTeller: Predicting the Risk of Cyber Incidents” (Bilge et al., 2017) in particular, proposed a system called ‘RiskTeller’ that leverages both supervised and semi-supervised learning methodologies to predict which machines are at risk of being infected by malware (Bilge et al., 2017).

In this study, by extracting 89 features (such as app count, volume of downloaded apps, volume of events and their temporal patterns, past threat history, etc.) per-machine from the data generated by enterprise customers of a large antivirus company that opted in to share their data, users’ machine’s patterns of usage and security awareness of the users were captured. Then, machine learning techniques such as Random Forest Classifiers were used to build up the RiskTeller, for it to predict which machines will get infected with high precision (Bilge et al., 2017).

# Dataset

## **4.1 Source of the Dataset**

The dataset for this project was obtained from Kaggle from the link provided below:

<https://www.kaggle.com/c/microsoft-malware-prediction/data> (Microsoft, 2018)

## **4.2 Overview of the Dataset**

The training dataset used for this project contains **8,921,483 rows of data**, each row corresponding to a Windows machine identified with the feature ‘MachineIdentifier’. There are **83 columns** in the training dataset with one target variable, each column describes different properties of the machine such as ‘ProductName’, ‘OsVer’, ‘Census\_PrimaryDiskTotalCapacity’, etc. The **target variable called ‘HasDetections’** was used as a label of the dataset whereby it contains a binary value that indicates whether malware is or isn’t detected in the machine (0 and 1).

The test dataset (separate from the training dataset) contains 7,853,253 rows of data *without the label column* (i.e. ‘HasDetections’ column). This is because the score for the Kaggle competition is determined upon submission. Hence, we cannot use the test dataset to evaluate our model performance. **Instead, we will only be using 1 million observations from the training dataset split into train/test datasets, where 30% will be train set and 70% will be test set.**

The following can be viewed in the metadata table below.

**Train Dataset:**

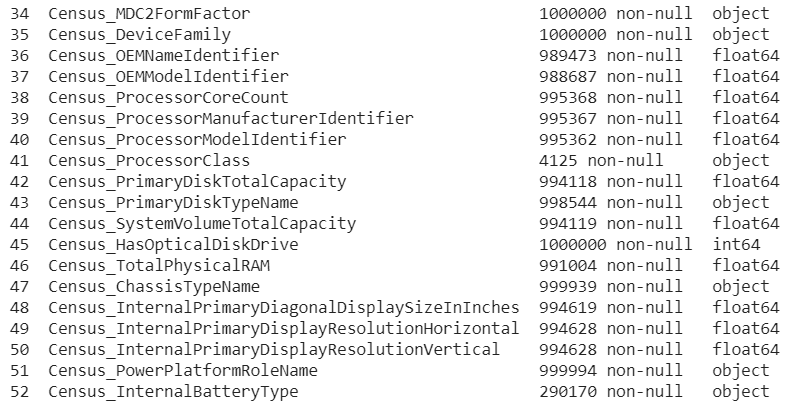
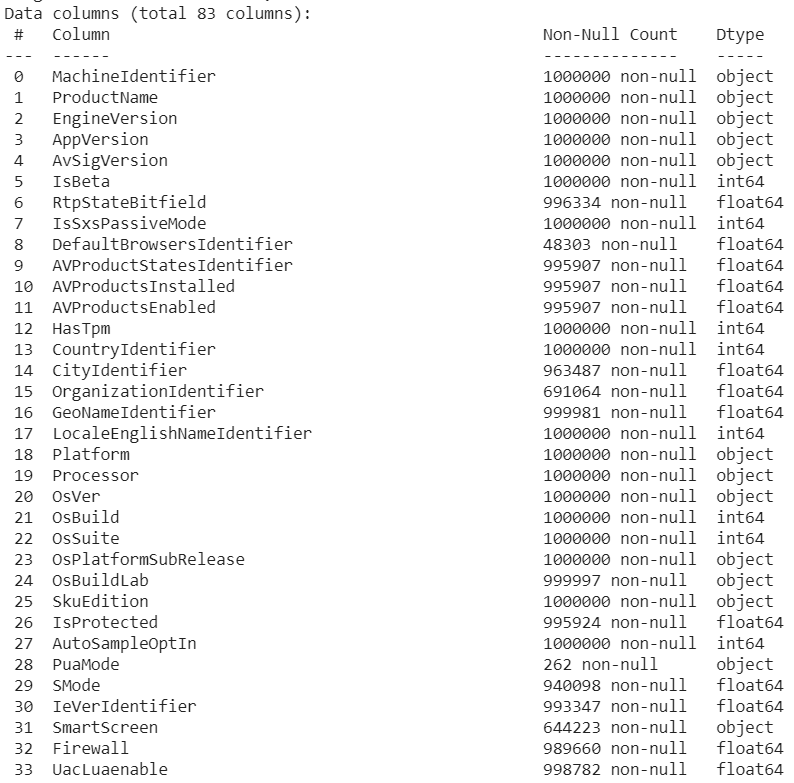
****

Figure 1. Metadata for Train Dataset (0-51)

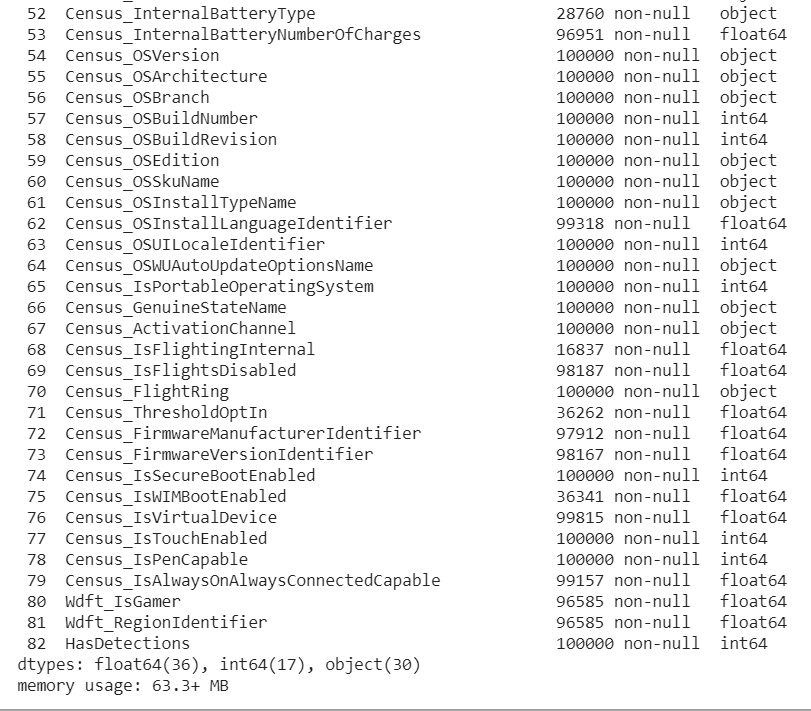
****

Figure 2. Metadata for Train Dataset (52-82)

# Methodology

## **5.1 Exploratory Data Analysis**

### **5.1.1 Missing Values in the Dataset**

After computing the number of missing values and calculating the percentage of missing values per column the attributes were ranked in order of highest to lowest percentage of missing values. It was found that ‘PuaMode’, ‘Census\_ProcessorClass’ and ‘DefaultBrowserIdentifier’ were some of the attributes with the highest percentages of missing values.

For the dataset on hand, as it was not suitable to fill in the missing values due to the large proportion of it (whether it be by predicting the missing values, imputing average values or most frequent values), it was concluded that the attributes with high percentage of missing values will be dropped from our dataset. This achieves two main objectives (1) reduces dimensions (2) if we choose to impute values, the features with high number of missing values could end up as a less meaningful splitting criterion.

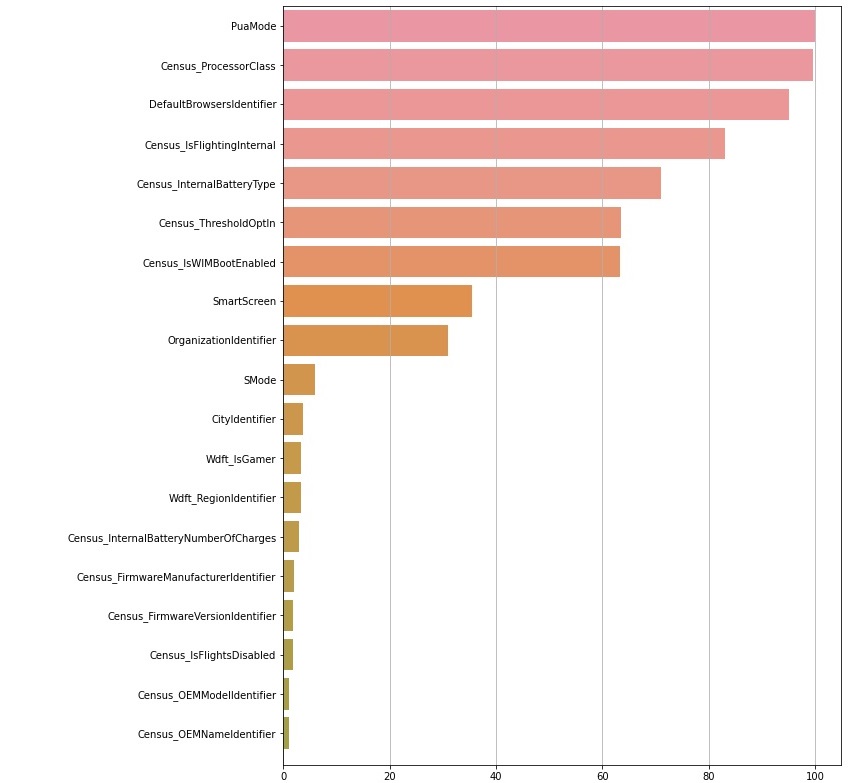


Figure 3. Horizontal bar chart showing the percentage of missing values for attributes that have high percentage of missing values out of the 82 attributes in total (without the target variable) Note: features with little missing values are omitted from this bar chart. Please refer to our code for the full chart.

### **5.1.2 Number of Unique Values & Relative Frequencies**

In order to better understand each of the attributes, the number of unique values and the largest relative frequency of the unique values for that particular attribute was found by implementing the following code:

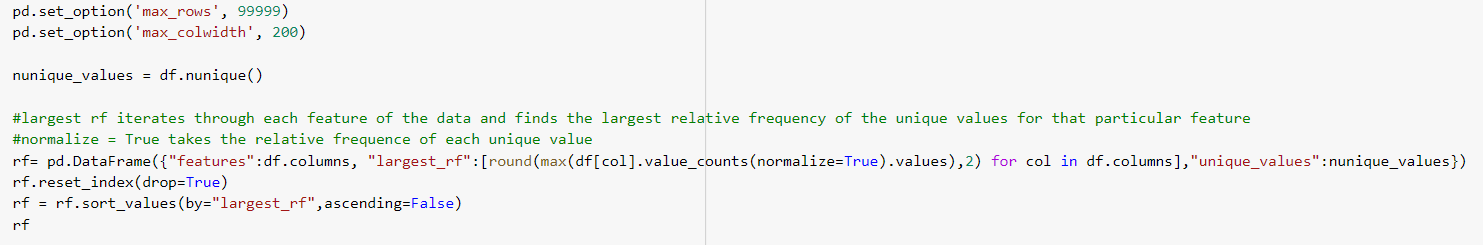


Figure 4. Snippet of the code to find the number of unique values and largest relative frequency per feature

After running the code, the following table was obtained:



Figure 5. Snippet of the result obtained from running the code chunk in Figure 4. Note: not all features are included here

From the table, it was found that the largest relative frequency (represented by the variable name largest\_rf) of each unique value for some of the features such as AutoSampleOptIn is close to 1.00 (100%). This means that all the values in the particular feature belonged to one particular class/category. This would likely mean that this category would not be useful as a decision criterion as it does not provide information for splitting/classifying. Given that our models are mostly classification based, we decided to drop these features as they are unlikely to help in the prediction of the target variable.

### **5.1.3 Class Distribution of ‘HasDetections’**

As mentioned in section *4.2 Overview of the Dataset*, the dataset had one target variable ‘HasDetections’. This target variable contained binary class values of 0 and 1, whereby value 0 signified that there were no malwares detected in the machine and value 1 signified that malware was detected in the machine.

A bar chart was generated to better understand the distribution of instances per class.

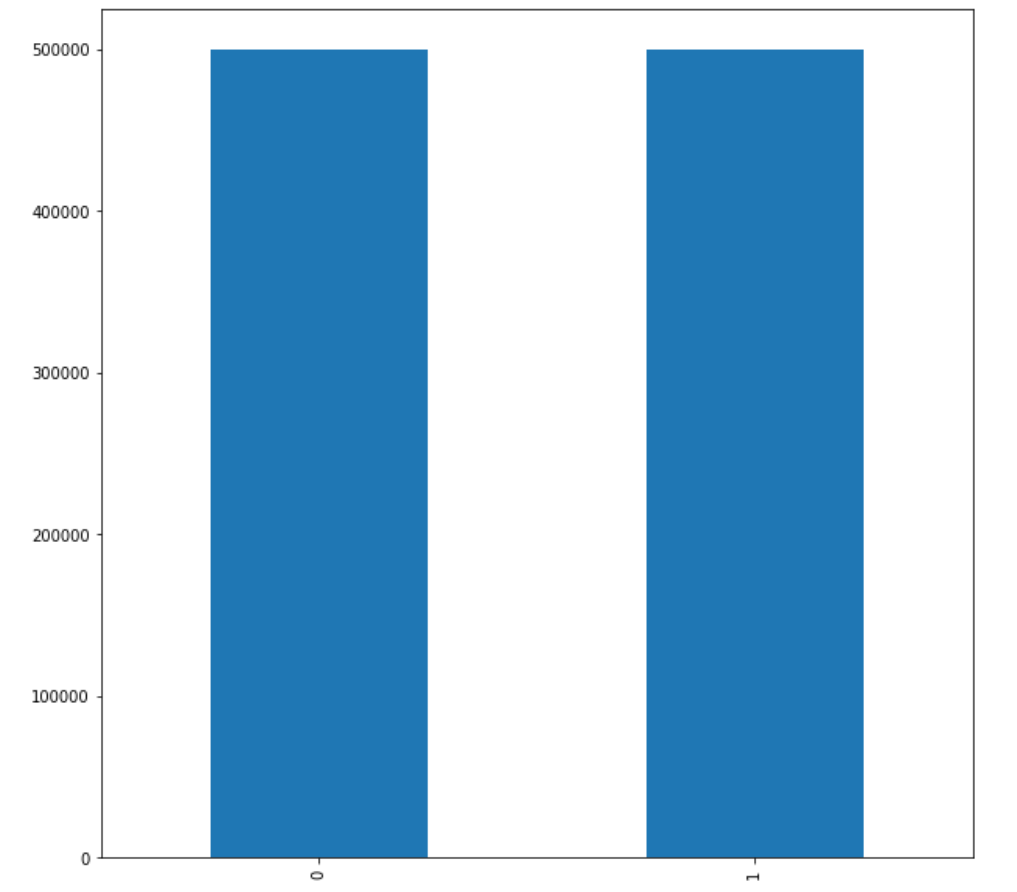


Figure 6. Bar chart showing the distribution of instances for class 0 and 1

With 500187 number of instances in class 0 and 499183 number of instances with class 1, the bar chart showed that the dataset had a fairly equal distribution of each class. From this, we may infer that accuracy can be used to evaluate our classification models. In addition, this characteristic of the dataset also allowed us to use the AUC-ROC curve to evaluate the different models in section *6. Results and Discussion* of the report.

## **5.2 Data Pre-processing**

Before diving into building our models, we decided to do some data pre-processing to prepare our dataset using the knowledge we had gained from the Exploratory Data Analysis (EDA) of the dataset.

### **5.2.1 Dimensionality Reduction**

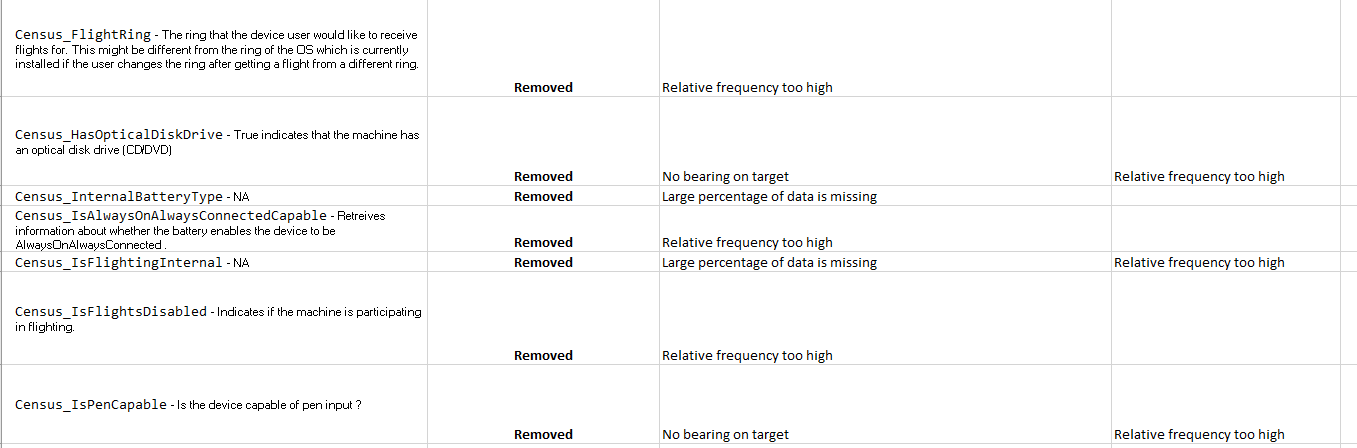


Figure 7. Snippet of the table showing the column names we dropped with reasoning

From the first look at the dataset, we realized that we need to remove some attributes as our dataset had high dimensionality (with 83 columns excluding the target variable). Hence, besides what we had done in the earlier section where we removed features with a high number of missing values, we also removed features which have no relevance in predicting the target variable (‘HasDetection’). A total of 44 columns were removed in total, leaving us with 38 columns.

Some of the columns that were removed due to irrelevance are ‘Census\_IsPenCapable’ which indicated whether a device can be used with an external stylus and ‘Census\_InternalPrimaryDisplayResolutionVertical’ which indicated the machine’s internal resolutions with values corresponding to the number of pixels. These features are unlikely to affect the target variable (‘HasDetection’), hence, they are removed.

Furthermore, we also tried Principal Component Analysis (PCA) on a sample size of 100,000 with varying n such as n=2, n=5, n=10, n=15, n=20. However, based on the outcomes tested on XGBoost, the model performance declined as compared to not performing PCA, regardless of the n chosen. Moreover, another disadvantage of using PCA is that each feature could no longer clearly explain the model outcomes, which made it challenging for us to evaluate the outcomes of each model. Having tried PCA to reduce dimensions, we decided not to go ahead with it.

### **5.2.2 Feature Encoding**

After dimensionality reduction, we identified categorical variables having text values within the remaining features.

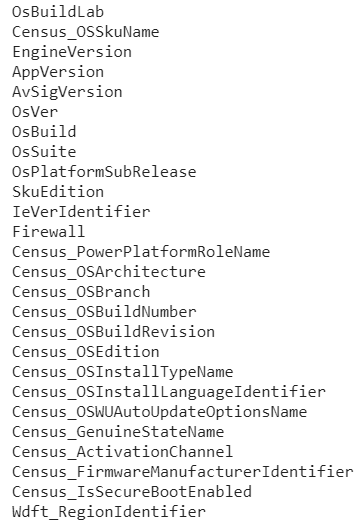


Figure 8. List of categorical variables with text values

As classification models cannot be trained with text inputs, the values of the attributes were encoded into numerical labels using LabelEncoder, a function available on *sklearn.preprocessing* in Python. Subsequently *sklearn.preprocessing.OneHotEncoder()* was used to ensure that the model does not perceive the variables to be in any ordinal sequence, which could affect the model outcome.

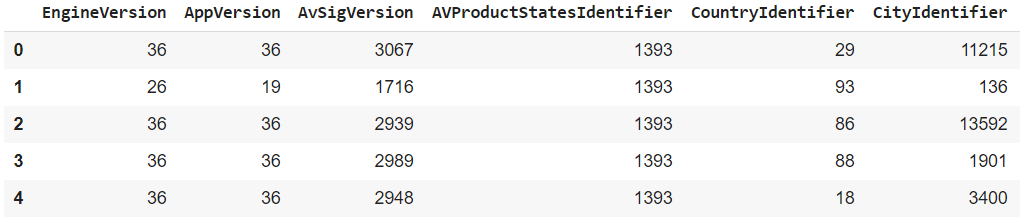


Figure 9. Snippet of the output of encoding and applying OneHotEncoder

As noted in our literature review, One-Hot Encoding would take too long if we had applied to all nominal variables, in addition, to the problem of a sparse matrix which has to be solved. Nevertheless, we still applied it to features with less than 100 unique values and tested the encoded data on XGboost with a sample size of 100,000.

However, the performance of XGboost declined after One-Hot Encoding was performed. Hence, we decided to still retain LabelEncoder as the method of encoding instead, even though we are aware that the ordinal relationship is not eliminated and could affect the model performance. Bearing this in mind, it would be good in future research to see how to make one-hot encoding work in the context of large dataset like ours and how to manage the sparse matrix resulting from One-Hot Encoding (Bogorod, 2019). We will discuss this in further details under future research.

### **5.2.3 Feature Scaling**

As for continuous variables consisting of 'Census\_SystemVolumeTotalCapacity and Census\_InternalBatteryNumberOfCharges', they were standardized using *sklearn.preprocessing.StandardScalar()* on python.

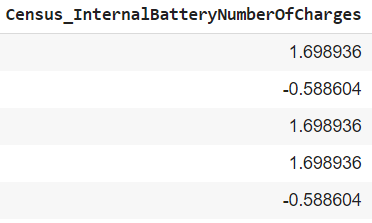
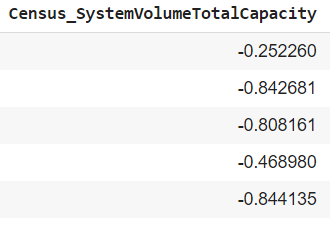


Figure 10. Outcome of applying *StandardScalar()* on two continuous variables

This was to ensure that these variables would not dominate other variables, affecting building a reliable model. However, do note that later on we discovered that a log transformation would change the skewness for these two features which gave us a better understanding of our model. Standardization was commented out from our code as a result because log transformation could not be done on negative values.

## **5.3. Model Creation**

To create the best possible model to correctly predict whether a machine is infected by malware or not, we decided to explore XGBoost (and XGBoost with Hyperparameter Tuning), CatBoost (and CatBoost with Hyperparameter Tuning), LightGBM, Multi-layer Perceptron, Random Forest, Support Vector Machine (SVM), K-NN and last but not least, Logistic Regression.

From our research, we first identified the main differences between XGBoost, CatBoost and LightGBM.

|  |  |  |
| --- | --- | --- |
| XGBoost | CatBoost | LightGBM |
| Continuously splits its nodes until it satisfies the stated max\_depth and then starts pruning. | Splits according to the penalty function. This can, however, be changed with parameter | Uses leaf-wise (best-first) tree-growth, which can lead to an imbalance tree. There are chances of overfitting the model if the sample size is small. Max\_depth needs to be controlled |

Figure 11. Table showing the main difference between XGBoost, CatBoost and LightGBM (Nahon,2020)

### **5.3.1 XGBoost & XGBoost with Hyperparameter Tuning**

XGBoost is different from other boosting models in that it is one of the Standard Gradient Boosting Models (GBM), which does not use regularization to help reduce the chances of overfitting the models (Jain, 2020). Also, XGBoost allows for faster processing utilizing what is known as parallel processing. This was essential for us as we had a large dataset, even after the dimensionality reduction (Jain, 2020).

Other unique features of XGBoost included how it would expand until the specified max\_depth is met and only then start pruning, and its built-in cross-validation feature whereby it is possible to get the exact number of boosting iterations in a single run without the need for Grid Search CV (Worcester, 2019).

Even though Grid Search CV is not required, we still explored using the Randomized Search optimization method and split the dataset by stratified 3-folds to get the most optimal hyperparameters. This method searches for the specified subset of hyperparameters in a randomized manner, lowering the processing time as compared to Grid Search CV which conducts exhaustive search. More often than not, it would still give us good results, according to research (Worcester, 2019) as compared to Grid Search CV, despite taking less time.

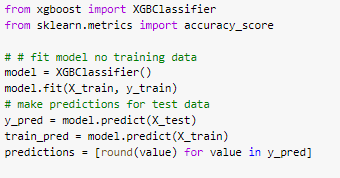


Figure 12: This figure contains the code for the baseline model



Figure 13. Finding the best parameters for XGBoost

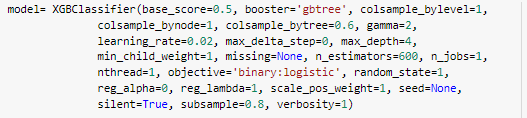


Figure 14: showing best hyperparameters for XGBoost

|  |  |  |
| --- | --- | --- |
|  | XGBoost | XGBoost + Hyperparameter Tuning |
| Accuracy | 61.81% | 62.52% |
| Precision | 60.46% | 61.18% |
| Recall | 70.5% | 70.65% |
| F1-Score | 65.1% | 65.57% |

Figure 15. Accuracy, Precision, Recall and F1-Score of XGBoost and XGBoost with Hyperparameter Tuning

Implementing XGBoost with the tuned hyperparameters resulted in a 0.71% increase in model accuracy and 0.72% increase in model precision.

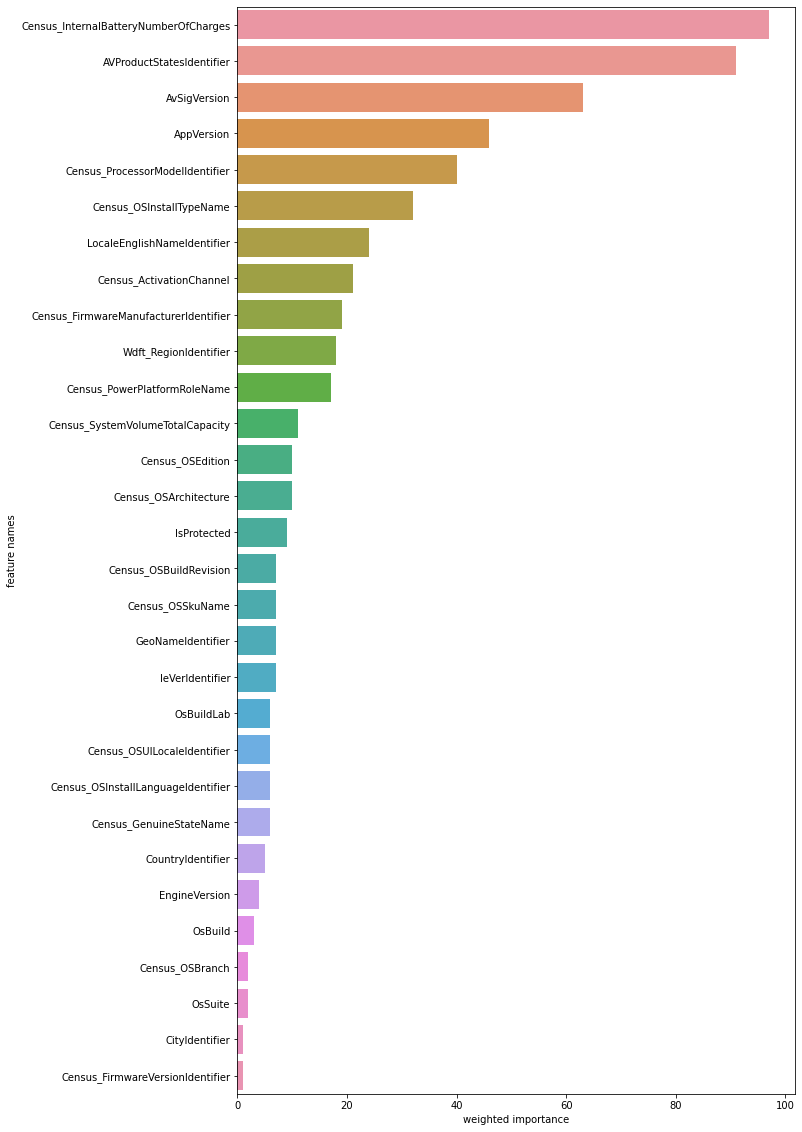
As we were interested in finding out what were the most important features contributing to the classification outcomes, we generated a horizontal bar chart of all features’ weighted importance.

Figure 16. Horizontal bar chart of the weighted importance of features

It was found that the top 5 most important features are ‘Census\_InternalBatteryNumberOfCharges’, ‘AVProductStatesIdentifier’, ‘AVSigVersion’, ‘AppVersion’ and ‘Census\_PrcessorModelIdentifier’.

Looking through *Figure 16*, the second top feature is the ID of the configuration for the machine’s antivirus software. The third and fourth features are AppVersion and AVSigVersion, each representing the defender state information. The fifth feature is the type of OS installed on the machine which would contribute greatly towards a machine being classified as having malware detections or not. All of these features made sense. However, Census\_InternalBatteryNumberOfCharges does not seem to make sense as the top feature in explaining the malware vulnerability. The dataset provided did not document what this feature specifically meant, however, looking at the dataset, it could be due to the skewness of the feature. As such, we did a logarithmic transformation which caused the feature to fall from the most important feature after a re-run of the xgboost results. Although the evaluation metrics were similar, the bar chart showed that Census\_InternalBatteryNumberOfCharges is no longer at the top of the chart.



Figure 17 . Log transformation of features

Post log-transformation - the feature importance graph became this:

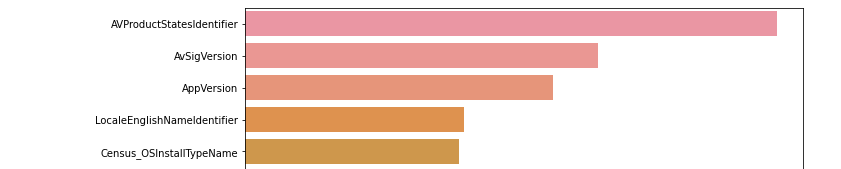


Figure 18 . Feature importance post log transformation

It is evident that the skewness probably contributed to the feature importance and, now, the top 3 features made more sense. Hence, we decided to keep the log transformation of both the system volume capacity and internal battery number of charges given that the results of the model remained similar, but the feature importance made more sense now.

### **5.3.2 CatBoost & CatBoost with Hyperparameter Tuning**

CatBoost is a gradient boosting algorithm that deals with categorical features automatically without the need for extensive data pre-processing and it is well known to be robust (Ray, 2017). As our dataset contained a large number of categorical features, we decided to try implementing this algorithm. Another reason for doing so came from our prior research, where it was said that CatBoost prediction speed’s were up to ten times faster than that of a traditional model. (“Parameter tuning,” n.d.)

An interesting characteristic of CatBoost was that CatBoost makes symmetrical decision trees whereby the same features are used to split data instances into right and left partitions for each level, otherwise known as depth-wise learning (“CatBoost Enables Fast Gradient Boosting on Decision Trees Using GPUs”, 2018)

However, it should be noted that CatBoost was not used in conjunction with OneHotEncoder as that had negatively affected the training speed and quality of the model after having tried One-Hot Encoding. Some of the parameters were also explicitly specified to circumvent computation limitations, such as the learning rate where we made sure to use a value that is not too small. This is so as the smaller the value, the more iterations are required for training the model (“Parameter tuning,” n.d.)

Keeping that in mind, we first generated the categorical features position before setting arbitrary parameters for our baseline model for testing.

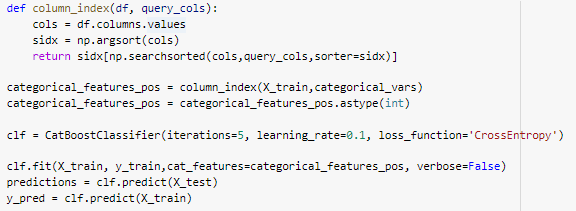


Figure 19. Codes to implement CatBoost algorithm (baseline code)

After which, we performed CatBoost with Hyperparameter Tuning to find the optimal parameters for the CatBoost model using the in-built randomized search. This method searched for the specific subset of hyperparameters exhaustively. Due to the long processing time, we used the in-built randomized search to carry out the hyperparameter tuning to obtain the most optimal results.

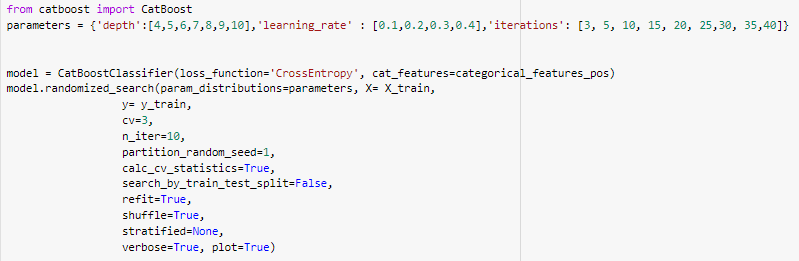


Figure 20. Codes to implement CatBoost with Hyperparameter Tuning

Here are the best hyperparameters for CatBoost after hyperparameter tuning:

'params': {'depth': 8, 'iterations': 40, 'learning\_rate': 0.4}}

**Results**

|  |  |  |
| --- | --- | --- |
|  | CatBoost | CatBoost + Hyperparameter Tuning |
| Accuracy | 61.27% | 63.20% |
| Precision | 59.63% | 62.08% |
| Recall | 70.56% | 69.79% |
| F1-Score | 64.64% | 65.71% |

Figure 21 . Accuracy, Precision, Recall and F1-Score of CatBoost and CatBoost with Hyperparameter Tuning

The accuracy of CatBoost improved by 1.93%, precision improved by 2.45%. However, recall declined by 0.77% and F1-score increased by 1.07 %. It is unexpected that even after hyperparameter tuning, there are slightly more false negatives. More investigation into this issue is required to understand why there is a slight decline in recall score. However, we note that there is a slight improvement in accuracy, precision and F1-score after hyperparameter tuning and will be using the model for evaluation.

### **5.3.3 LightGBM**

LightGBM, another gradient boosting algorithm was implemented. LightGBM was unique to the algorithms as it grows trees vertically (leaf-wise) whereas the other algorithms grow trees horizontally (level-wise) (“*Features”, n.d.)*.

As LightGBM 1) chooses the leaf with maximum delta loss to grow, reducing more loss than level-wise algorithms, and 2) performs at a high speed even with big data, we thought that it would be suitable for our dataset (“*Features”, n.d.)*. .

Initially, we set some arbitrary values for our baseline model with 100 iterations for a start after defining a list of categorical features. (as seen in figure 22)

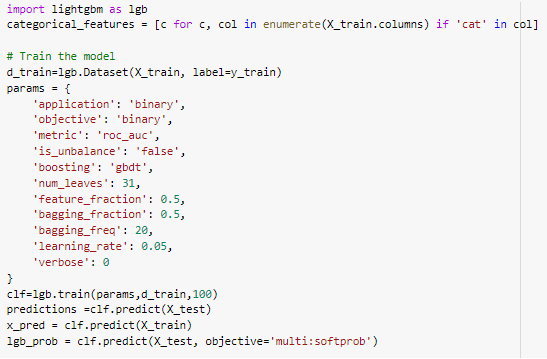


Figure 22. Codes to implement LightGBM (baseline)

We then performed hyperparameter tuning on this model using Randomized Search Cross Validation method to generate optimal parameters. Consequently, the parameter tuned LightGBM model yielded the results shown below (refer to figure 23).

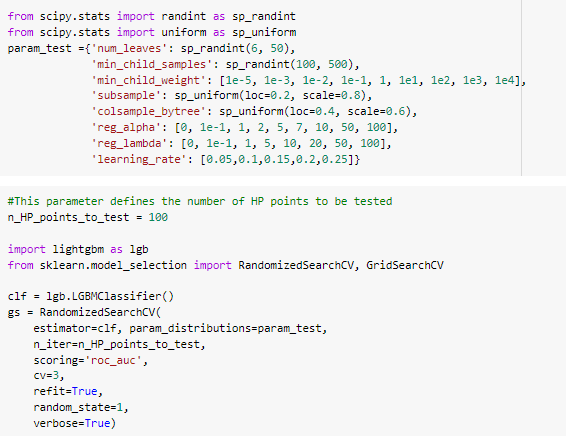


Figure 23: Codes to perform hyperparameter tuning for LightGBM

The best hyperparameters for LightGBM are:

params: {'colsample\_bytree': 0.6470727273814318, 'learning\_rate': 0.2, 'min\_child\_samples': 380, 'min\_child\_weight': 0.001, 'num\_leaves': 49, 'reg\_alpha': 5, 'reg\_lambda': 0, 'subsample': 0.21463062187535345}

|  |  |  |
| --- | --- | --- |
|  | LightGBM | LightGBM + Hyperparameter Tuning |
| Accuracy | 62.32% | 63.48% |
| Precision | 61.23% | 62.65% |
| Recall | 67.81% | 68.60% |
| F1-Score | 64.35% | 65.49% |

Figure 24. Accuracy, Precision, Recall and F1-Score of LightGBM

All the respective metrics improved after hyperparameter tuning. Accuracy increased by 1.16%, precision improved by 1.42%, recall improved by 0.79% and f1-score improved by 1.14%. The performance of LightGBM is one of the best amongst all the models thus far, which is consistent with our literature review, even though the performance of LightGBM is not as good as theirs. This is probably due to differences in our feature engineering steps.

### **5.3.4 Random Forest**

We also implemented Random Forest as we hoped for a better model performance. Given that random forest is made up of small decision trees from random subsets of data which result in a biased classifier, the ensemble of these decision trees results in better model performance than individual ones. A snippet of our implemented code is shown in figure 25 below (baseline model):

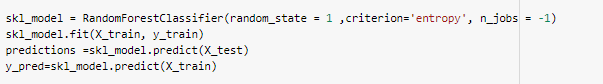


Figure 25: Showing codes of baseline model

Thereafter, we performed hyperparameter tuning using RandomizedSearchCV for faster processing.

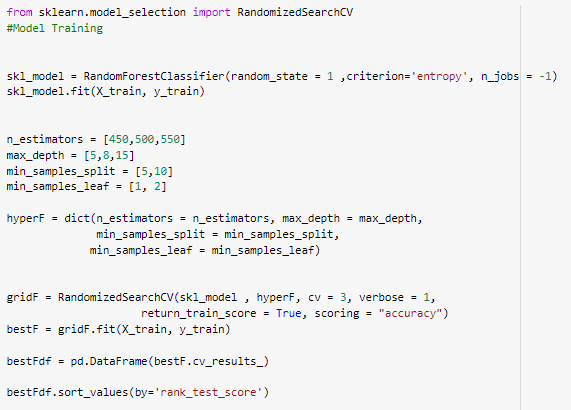


Figure 26. Codes to implement Random Forest hyperparameter tuning



Figure 27: Here are the best hyperparameters for Random Forest after hyperparameter tuning

The results before and after hyperparameters tuning are shown below.

|  |  |  |
| --- | --- | --- |
|  | Random Forest | Random Forest + Hyperparameter Tuning |
| Accuracy | 62.07% | 60.41% |
| Precision | 61.95% | 58.33% |
| Recall | 64.55% | 75.74% |
| F1-Score | 63.23% | 65.91% |

Figure 28 . Accuracy, Precision, Recall and F1-Score of Random Forest

After hyperparameter tuning, the accuracy and precision decreased by 1.66% and 3.62% respectively. However, the recall increased by 11.19% and F1-Score increased by 2.68%. This means that the number of false negatives have decreased by about 2.68% after hyperparameter tuning.

### **5.3.5 Multi-layered Perceptron**

A perceptron is a binary classification algorithm that can solve complex problems. A multilayer perceptron is a perceptron that teams up with additional perceptrons, stacked in several layers - also called Neural Network. Taking the input data, it multiplies them with weights of relative importance. The algorithm learns by discovering better weights that result in more accurate predictions. (“Complete Guide to Artificial Neural Network Concepts & Models,” n.d.)

An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear [activation function](https://en.wikipedia.org/wiki/Activation_function) (“Complete Guide to Artificial Neural Network Concepts & Models,” n.d.). With a non-linear activation function, MLP can classify data points which are not linearly separable. During the hyperparameter tuning, we tried both tanh and relu activation functions. For most neural networks, relu is generally considered one of the best activation functions, but we still tried both tanh and relu. For solvers, we tried Stochastic Gradient Descent (SGD) which seeks for the parameters to optimize the objective function. We also selected Adaptive Moment Estimation (ADAM) to be used as one of the solvers, which is an adaptive learning rate optimisation algorithm (Bushaev, 2018)

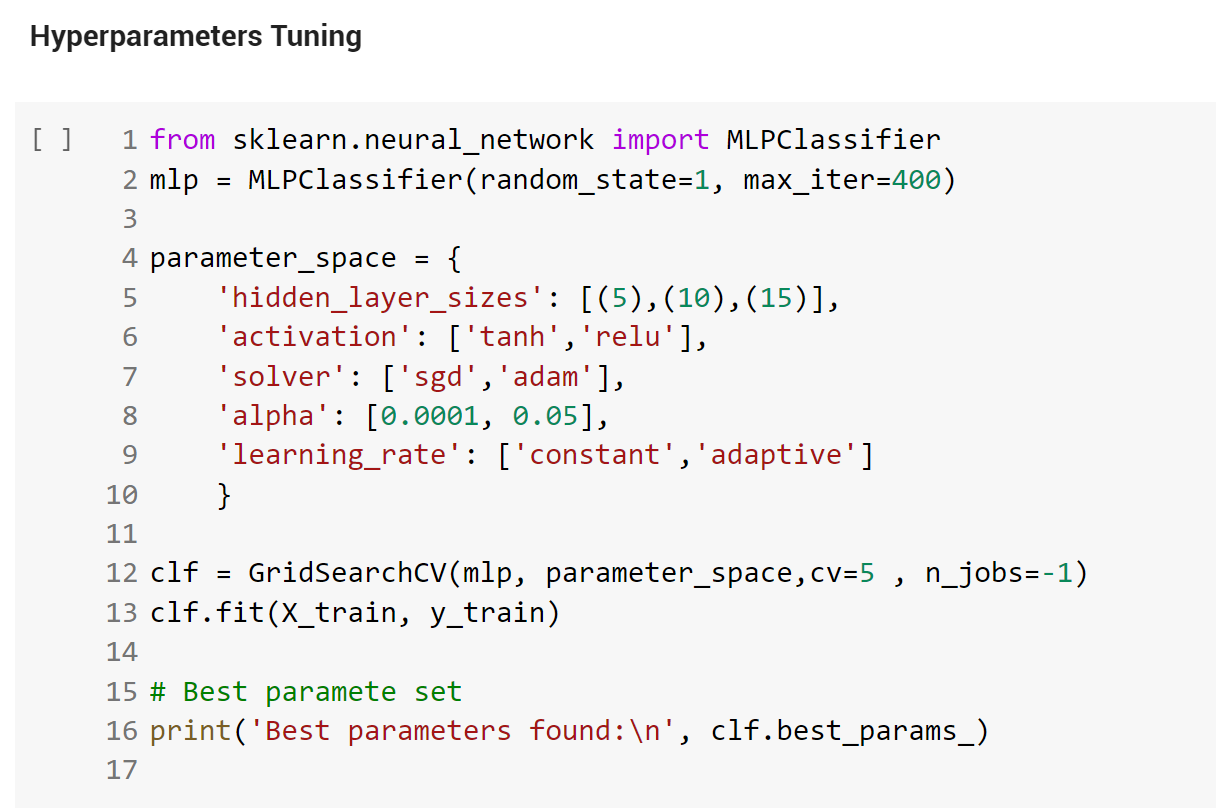


Figure 29 . Codes to implement Multi-Layered Perceptron hyperparameter tuning



Figure 30: Showing the best hyperparameters

Upon implementing this model, we calculated the following performance metrics to evaluate its performance, as seen in the figure below:

|  |  |  |
| --- | --- | --- |
|  | Multi-Layered Perceptron | Multi-Layered Perceptron + Hyperparameter Tuning |
| Accuracy | 55.52% | 51.96% |
| Precision | 61.14% | 51.90% |
| Recall | 32.81% | 73.30% |
| F1-Score | 42.71% | 60.78% |

Figure 31 . Accuracy, Precision, Recall and F1-Score of Multi-Layered Perceptron

While accuracy, precision and F1-score for this MLP model was poorer than our previous implementations of LightGBM, XGBoost and CatBoost, the recall measure improved relative to other models. This indicates that the MLP model was better able to correctly identify true positives. However, because accuracy is a crucial evaluation metric, Multi-layered Perceptron’s performance is considered to be poorer than the other Boosting classifier models even after hyperparameter tuning.

Note that after Multi-Layered Perceptrons, no hyperparameters tuning will be done for the remaining models as their baseline models performed too poorly and it is also time-consuming to run hyperparameters tuning, so we only chose the top 5 models (the previous 5) to run.

**5.3.6 Support Vector Machine (SVM)**

The support vector machine algorithm aims to find a hyperplane in an N-dimensional space, where N is the number of features, which in our case is N=38. This hyperplane distinctly classifies the given data instances (Gandhi, 2018). The optimal hyperplane is where there is maximum distance between data instances of the two distinct classes. Our code below aims to find this SVM model.

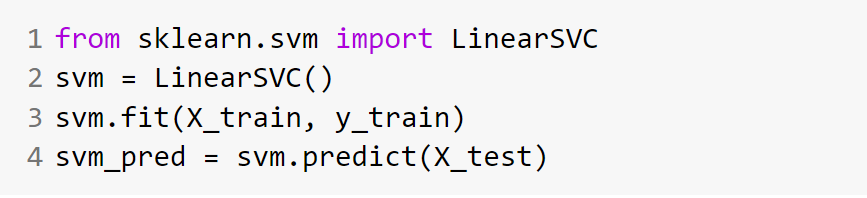


Figure 32. Codes to implement SVM

|  |  |
| --- | --- |
|  | SVM |
| Accuracy | 49.58% |
| Precision | 50.13% |
| Recall | 39.37% |
| F1-Score | 44.10% |

Figure 33. Accuracy, Precision, Recall and F1-Score of SVM

The accuracy, precision, and F1-score for the SVM model was the lowest in comparison to our previously implemented models. Recall improved as compared to XGBoost, CatBoost, and LightGBM but still remained lower than that of MLP and Random Forest algorithms.

### **5.3.7 K-NN**

The K-NN algorithm classifies data instances according to the majority class of the specified number of (K) nearest neighbouring data points. The distance to neighbouring data points is determined by similarity measures. In our implementation below, we specified K=5 nearest neighbours for classification.

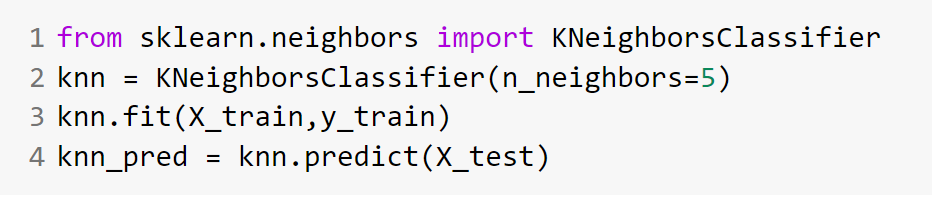


Figure 34. Codes to implement K-NN

|  |  |
| --- | --- |
|  | K-NN |
| Accuracy | 52.95% |
| Precision | 53.12% |
| Recall | 56.05% |
| F1-Score | 54.55% |

Figure 35. Accuracy, Precision, Recall and F1-Score of K-NN

While K-NN performed better than SVM with regards to the performance metrics, it remained lower than all other previously implemented models.

### **5.3.8 Logistic Regression**

Given that our case is a binary classification problem, we decided to implement the logistic regression algorithm as shown in Figure 28 below. Having a large dataset, we set the solver parameter = ‘sag’ which is Stochastic Average Gradient Descent which is known to be faster due to memory of previous gradient values resulting in faster convergence rates (Clement and Yahya,2019).

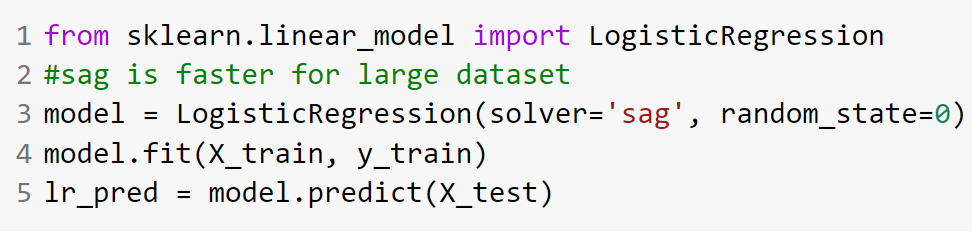


Figure 36. Codes to implement Logistic Regression

|  |  |
| --- | --- |
|  | Logistic Regression |
| Accuracy | 55.82% |
| Precision | 55.07% |
| Recall | 68.20% |
| F1-Score | 60.94% |

Figure 37. Accuracy, Precision, Recall and F1-Score of Logistic Regression

The accuracy, precision, and F1-score measures for Logistic Regression were fairly low in comparison to all other previous classification models.

# Results and discussion

Figure 38 below summarizes the performance measures (Accuracy, Precision, Recall, and F1-score) for all the classification models implemented - XGBoost, CatBoost, LightGBM, Support Vector Machine, Multi-layer Perceptron, Logistic Regression and K-NN.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Accuracy | Precision | Recall | F1-Score |
| LightGBM (after hyperparameters tuning) | 63.48% | 62.65% | 68.60% | 65.49% |
| XGBoost (after hyperparameters tuning) | 62.52% | 61.18% | 70.65% | 65.57% |
| CatBoost (after hyperparameters tuning) | 63.20% | 62.08% | 69.79% | 65.71% |
| RandomForest (after hyperparameters tuning) | 60.41% | 58.33% | 75.74% | 65.91% |
| MLP(after hyperparameters tuning) | 51.96% | 51.90% | 73.30% | 60.78% |
| SVM (baseline) | 49.58% | 50.12% | 39.37% | 44.10% |
| Log Reg (baseline) | 55.82% | 55.07% | 68.20% | 60.94% |
| K-NN (baseline) | 52.95% | 53.12% | 56.05% | 54.55% |

Figure 38. Accuracy, Precision, Recall and F1-Score of the models created

Here, LightGBM is seen to have higher accuracy than the other models. The reason why we can rely on accuracy as an evaluation measure is because our models are binary classifiers with a balanced dataset. To confirm this finding, we went beyond these performance metrics in order to better compare our classification models. We generated an AUC-ROC curve (Area Under The Curve-Receiver Operating Characteristics) as seen in the figure below for the top 5 models:

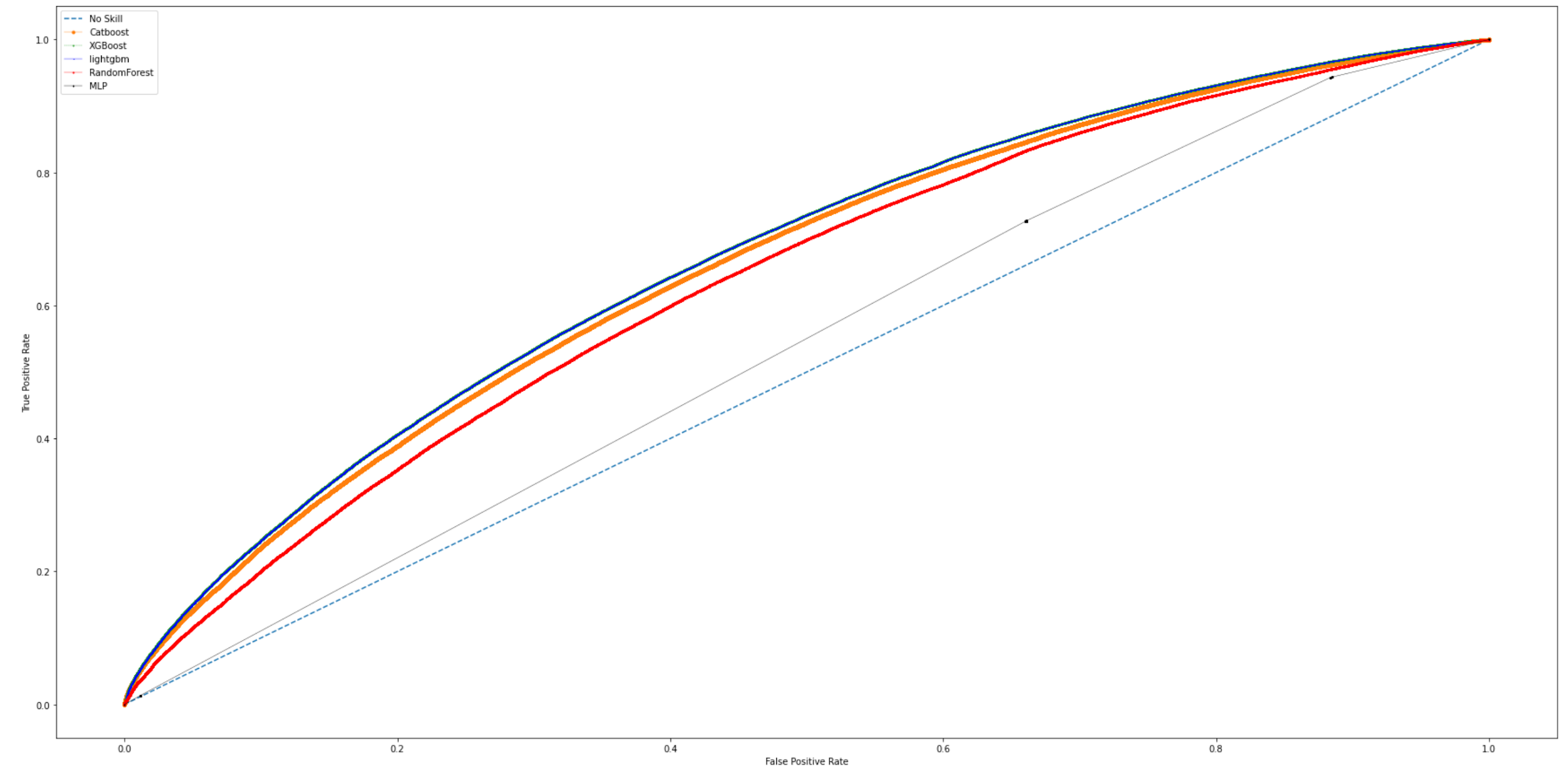


Figure 39. AUC-ROC curve for model comparison

AUC-ROC measures how much each model is able to distinguish between the two classes. This was a suitable model evaluation method to use as our dataset was balanced. When AUC-ROC = 0.5, the model is not able to distinguish between classes. Any value above 0.5 indicates that the model is able to distinguish between classes well. A value below 0.5 indicated that the model actually reverses the class assignment and therefore incorrectly classifies data instances. The following table shows our implemented classification models with their respective AUC-ROC values.

|  |  |
| --- | --- |
|  | AUC-ROC value |
| CatBoost (after hyperparameters tuning) | 0.680 |
| XGBoost(after hyperparameters tuning) | 0.674 |
| LightGBM(after hyperparameters tuning) | 0.686 |
| Random Forest(after hyperparameters tuning) | 0.645 |
| MLP(after hyperparameters tuning) | 0.522 |

Figure 40. AUC-ROC values

From the above results, all models are able to distinguish between the two classes (has malware detections and does not have malware detections) with all AUC-ROC values being above 0.5. It can be seen that LightGBM performs the best with AUC-ROC value of 0.686, as compared to XGBoost, CatBoost, Random Forest, and Multi-layered Perceptron models.

# Conclusion and future work

Among the different classification models implemented - XGBoost, CatBoost, LightGBM, Random Forest, Support Vector Machine, Multi-Layer Perceptron, K-NN and Logistic Regression - LightGBM is the best performing classification model which successfully classifies whether a Windows machine will have malware infections detected based on 38 different properties of the machine with LightGBM being the best performing model with an accuracy of 63.48%.

For future work, there are a few things which we can improve on such as feature encoding, feature engineering and running all the top performing models on K-fold cross-validation. Firstly, on the issue of encoding, from our literature review there are a few ways we can handle the encoding. However, we have only tried Label Encoding and One-Hot Encoding on selected features (with less than 100 unique values), which yielded poorer performance, arguably due to the more sparse matrix. During our literature review, we found that one of the kaggle submissions attempted to overcome the high dimensionality after One-Hot Encoding by using sparse matrix as a solution (Bogorod, 2019). That could be done in the data-preprocessing stage to see if the results improve. In addition, there are other ways of encoding data such as frequency encoding which might improve the model performance. Hence, each of these encoding methods could be attempted to improve the score.

Another future work to try is to conduct more feature engineering. For instance, we found that certain variables such as Primary Disk Capacity can be subtracted from System Volume Capacity to obtain remaining disk capacity. Other features such as ‘Wdft\_IsGamer' can also be combined with Firewall to form a new feature. This is just one of many feature engineering work which could be done. Due to time constraint, we could not try all the possible feature engineering and have only done the basic log transformation for Census\_InternalBatteryNumberOfCharges and 'Census\_SystemVolumeTotalCapacity'. (Jugaloza,2020). Moreover, more data preprocessing such as removing outliers and removing one of the correlated variables can also be done in the future. However, as most features are categorical features, we did not perform correlation analysis and remove outliers. Hence, this area is definitely an area for improvement for future work.

Furthermore, we can also try Stacking Custom Ensembles. Given some of our models have better accuracy and other models have better recall. By trying to stack a few of the selected models, perhaps the overall accuracy and ROC\_AUC curve can be improved (Jugaloza,2020). Last, but not least, if we can acquire better computational resources, we can build our model on a larger dataset which could yield better results.

# References

Aggiwal, R. (2017). Introduction to Random forest. Retrieved from: <https://dimensionless.in/introduction-to-random-forest/>

Arunlal, K. S., & Vijayanand, C. D. (2019, May/June). Impact of Malware in Modern Society. Retrieved November 13, 2020, from <https://www.researchgate.net/publication/334130947_Impact_of_Malware_in_Modern_Society>

Bilge, L., Han, Y., & Dell’Amico, M. (2017, November 30). RiskTeller: Predicting the Risk of Cyber Incidents. Retrieved from: <https://acmccs.github.io/papers/p1299-bilgeA.pdf>

Bogorod.V. (2019, February 17). LightGBM. Baseline Model Using Sparse Matrix. Retrieved November 13, 2020, from <https://www.kaggle.com/bogorodvo/lightgbm-baseline-model-using-sparse-matrix>

Boyle, T. (2019). Hyperparameter Tuning. Retrieved from: <https://towardsdatascience.com/hyperparameter-tuning-c5619e7e6624>

Bushaev, V. (2018, October 24). Adam - latest trends in deep learning optimization. Retrieved November 13, 2020, from <https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c>

CatBoost Tutorial. (2018). CatBoost Enables Fast Gradient Boosting on Decision Trees Using GPUs. Retrieved from: <https://catboost.ai/news/catboost-enables-fast-gradient-boosting-on-decision-trees-using-gpus>

Clement, & Yahya. (2019, October 01). Logistic regression python solvers' definitions. Retrieved November 13, 2020, from <https://stackoverflow.com/questions/38640109/logistic-regression-python-solvers-defintions>

Complete Guide to Artificial Neural Network Concepts & Models. (n.d.). Retrieved November 13, 2020, from <https://missinglink.ai/guides/neural-network-concepts/complete-guide-artificial-neural-networks/>

Features. (n.d.). Retrieved November 13, 2020, from <https://lightgbm.readthedocs.io/en/latest/Features.html>

Gandhi, R. (2018). Support Vector Machine — Introduction to Machine Learning Algorithms. Retrieved from: <https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47>

Jain, A. (2020, October 18). XGBoost Parameters: XGBoost Parameter Tuning. Retrieved November 13, 2020, from <https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/>

Jovanović, B. (2019, November 22). Malware statistics – You’d better get your computer vaccinated. DataProt. Retrieved from: <https://dataprot.net/statistics/malware-statistics/>

Jugaloza. (2020). Microsoft Malware Prediction using Classical Machine Learning Algorithms. Retrieved November 13, 2020, from <https://jugaloza.medium.com/microsoft-malware-prediction-using-classical-machine-learning-algorithms-5ade962ca73a>

Kang, C., Park, N., Prakash, B. A., Serra, E., & Subrahmanian, V.S. (2016, February .). Ensemble Models for Data-driven Prediction of Malware Infections. ACM Digital Library.

Retrieved from: <https://dl.acm.org/doi/abs/10.1145/2835776.2835834>

Kouroupetroglou, P. N. (2019, March 19). Microsoft Malware Detection EDA + XGBoost. Retrieved November 13, 2020, from <https://www.kaggle.com/praxitelisk/microsoft-malware-detection-eda-xgboost>

Mandot, P. (2017). What is LightGBM, How to implement it? How to fine tune the parameters?. Retrieved from: <https://medium.com/@pushkarmandot/https-medium-com-pushkarmandot-what-is-lightgbm-how-to-implement-it-how-to-fine-tune-the-parameters-60347819b7fc>

Microsoft. (2018, December 14). Microsoft Malware Prediction Dataset. Kaggle. Retrieved from: <https://www.kaggle.com/c/microsoft-malware-prediction/data>

Microsoft Malware Classification Challenge (BIG 2015). Kaggle. Retrieved from: <https://www.kaggle.com/c/malware-classification>

Narkhede, S. (2018). Understanding AUC - ROC Curve. Retrieved from: <https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5>

O¨ mer, A., Refik, S., & O¨ mer, O. z. T. (2020). Using a Subtractive Center Behavioral Model to Detect Malware. Malware Analysis and Vulnerability Detection Using Machine Learning, Volume 2020(Article ID 7501894), 17. <https://www.hindawi.com/journals/scn/2020/7501894/>

Nahon, A. (2020, February 10). XGBoost, LightGBM or CatBoost - which boosting algorithm should I use? Retrieved November 13, 2020, from <https://medium.com/riskified-technology/xgboost-lightgbm-or-catboost-which-boosting-algorithm-should-i-use-e7fda7bb36bc>

Parameter tuning. (n.d.). Retrieved November 13, 2020, from <https://catboost.ai/docs/concepts/parameter-tuning.html>

Perceptrons and Multi-Layer Perceptrons: The Artificial Neuron at the Core of Deep Learning. Retrived from: <https://missinglink.ai/guides/neural-network-concepts/perceptrons-and-multi-layer-perceptrons-the-artificial-neuron-at-the-core-of-deep-learning/>

Ray, S. (2017). CatBoost: A machine learning library to handle categorical (CAT) data automatically. Retrieved from: <https://www.analyticsvidhya.com/blog/2017/08/catboost-automated-categorical-data/>

Seif, G. (2019). A Beginner’s guide to XGBoost. Retrieved from: <https://towardsdatascience.com/a-beginners-guide-to-xgboost-87f5d4c30ed7>

Worcester, P. (2019, June 06). A Comparison of Grid Search and Randomized Search Using Scikit Learn. Retrieved November 13, 2020, from <https://blog.usejournal.com/a-comparison-of-grid-search-and-randomized-search-using-scikit-learn-29823179bc85>

Yee (2019). What I've Learned: Microsoft Malware Prediction Competition on Kaggle. Retrieved November 13, 2020, from <https://medium.com/datadriveninvestor/what-ive-learned-microsoft-malware-prediction-competition-on-kaggle-3c8189dcc850>