Microsoft Malware Prediction

CSE 351 Final Project

By Rediet and Merry

Submitted to:

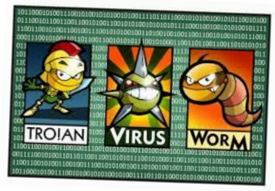
Professor Pravin Pawar

Spring 2020, SUNYK CSE 351

Content

Introduction

Malicious software is abundant in a world of innumerable computer users, who are constantly faced with these threats from various sources like the internet, local networks and portable drives. Malware is potentially low to high risk and can cause systems to function incorrectly, steal data and even crash. Malware may be executable or system library files in the form of viruses, worms, Trojans, all aimed at breaching the security of the system and compromising user privacy. Typically, anti-virus software is based on a signature definition system which keeps updating from the internet and thus keeping track of known viruses. While this may be sufficient for home-users, a security risk from a new virus could threaten an entire enterprise network.



The primary 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. Hence, the data required for malware prediction can be any information about the state of a computer which is hit by a malware attack. As there are various types of malware attacks, machines may behave differently when attacked. Therefore, it is useful to collect a large amount of data about computers that are attacked. Most of the data comes from the system behavior of the machine and the type of the machine. For our case Microsoft has done all the process of capturing the information from a windows defender through a long course of time.

Therefore, we have chosen to use the Kaggle dataset on Microsoft Malware Detection. As the kaggle dataset is highly organized and has all the required (sampleSubmission, training, and testing) data, it is suitable to study and preprocess easily. In addition to that, many Kaggle competitors used the dataset in the past, and under the discussion section many useful tips about preprocessing the dataset are given. Therefore, we decided to use Kaggle as our primary source of data.

Project Objectives

The objective of this project is to analyze the different solutions that the competitors of kaggle community brought and compare their solutions in order to provide the pros and cons of different methods and approaches that were used to tackle this problem and possibly come up with a conclusion on which solution is more effective for predicting the malwares effectively into their respective families. In doing so we hope to be part of the group that is trying to solve the challenge that microsoft is proposing. To explain the objective in great detail, the goal of this competition was to predict a Windows machine's probability of getting infected by various families of malware, based on different properties of that machine. This can be accomplished by the training data that microsoft made available which contains these properties which are the machine's infections that were generated by Windows Defender which combines heartbeat and threat reports collected by Microsoft's endpoint protection solution,
Here are specific objectives:

1, Analyze the codes that were submitted on the competition and understand how different people approached the problem inorder to investigate the accuracy of the model presented and its viability to different sorts of malware detection.

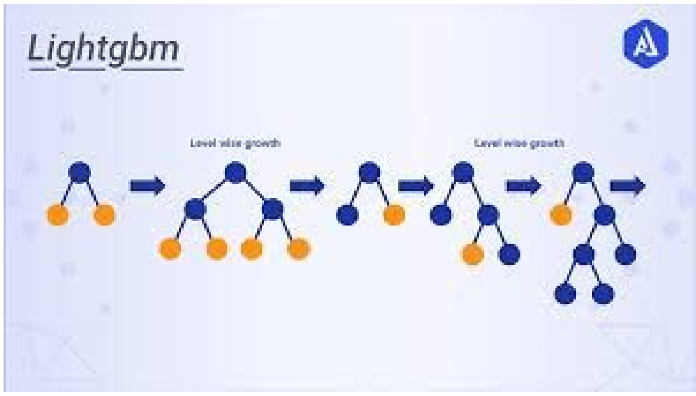
2, Observe the execution information whereby we speculate the amount of time the execution of the code took together with the memory usage, the readability and simplicity of the code and its quality as well. To do so we will be using the execution information presented by kaggle competitors.

3, Examine the significant parameters that affect the model and figure out which ones give better results. This includes scrutinizing the optimizations used and the hyperparameter tuning that were considered.

Machine learning techniques used

Machine Learning technique 1

Light GBM

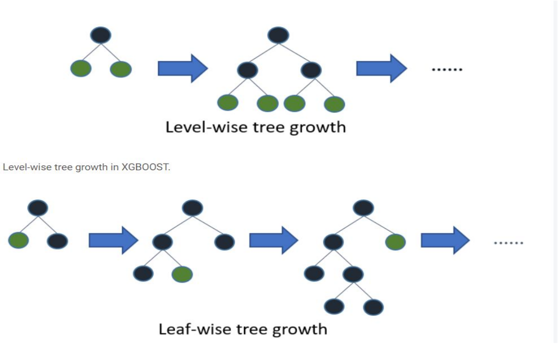


What is Light GBM

- It is a gradient boosting framework that uses tree based learning algorithms.
- It is designed to be distributed and efficient with the following advantages:
    - Support of parallel and GPU learning.
    - Capable of handling large-scale data.

Advantages of Light GBM

1. **Faster training speed and higher efficiency**: Light GBM use histogram based algorithm i.e it buckets continuous feature values into discrete bins which fasten the training procedure.
2. **Lower memory usage:** Replaces continuous values to discrete bins which result in lower memory usage.
3. **Better accuracy than any other boosting algorithm:** It produces much more complex trees by following leaf wise split approach rather than a level-wise approach which is the main factor in achieving higher accuracy. However, it can sometimes lead to overfitting which can be avoided by setting the max_depth parameter.
4. **Compatibility with Large Datasets:** It is capable of performing equally good with large datasets with a significant reduction in training time as compared to XGBOOST.
5. **Parallel learning supported.**



Level-wise tree growth in XGBOOST.

Data preprocessing

The raw data is very high dimensional and has many missing values. The size of data is also comparatively large.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. Using the information and labels in train.csv, we predict the value for HasDetections for each machine in test.csv. A detailed description of the nature of the data will be provided in the next section.

As we all know the big challenge that is related to most machine learning training and testing processes is data. Firstly getting the right data. Second, preparing the data so that it can properly fit the algorithm or whatever predicting and testing architecture that we use. Hence in our project here as well, we have tried preprocessing the data. It was especially so important as the data was too big to even be opened. Hence reducing the size of the data and removing less necessary or unnecessary columns was very crucial. Here we have provided the data cleaning and preprocessing steps with description.

Removing unnecessary columns:

As the data is highly dimensional in this specific kaggle challenge, it is really difficult to do anything with it. So we can reduce the column dimension by eliminating less useful columns which have.

1, Mostly-missing Feature- in the given data set there are around 2 columns that have more than 99% of missing values. Hence removing them wouldn't affect the dataset at all since most of the values in under the column are missing

2, Too-skewed features - these are the columns whose majority categories cover more than 99% of occurrences. Hence there is no need to keep them. Normally highly skewed data give us too little information. When 1% of data with other values have the same distribution of target features.

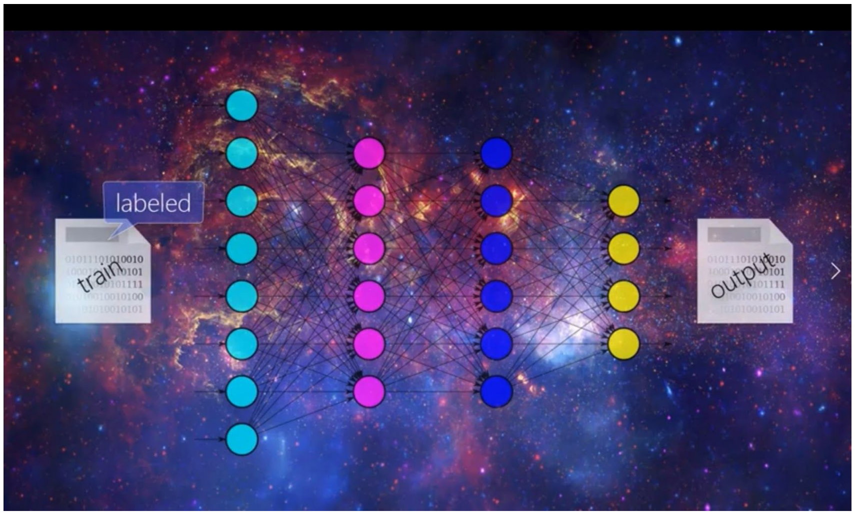
3, Highly-correlated features- correlations between columns show how related the two columns are. Hence, after testing correlations between columns, we picked up pairs whose correlation is greater than 0.99 compared to the distribution of the features in the pairs and also correlation with high detections. Then the minor columns can be eliminated.

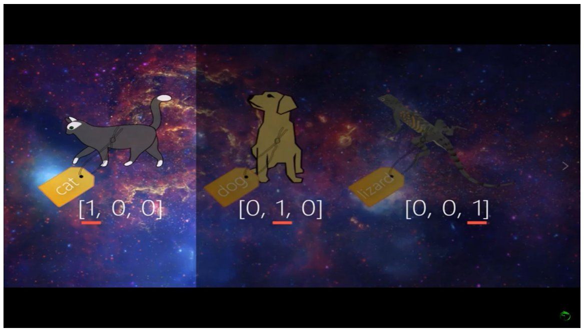
Throughout this process, it is possible to eliminate 17 columns without losing significant information.

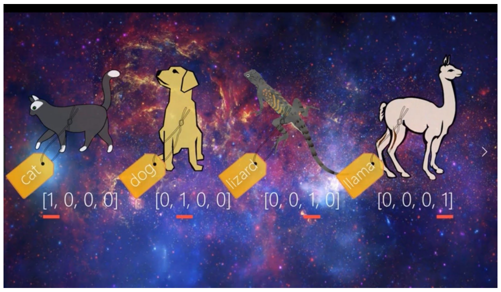
One Hot Encoding

One hot encoding is a process by which categorical variables are converted into a form that could be provided to ML algorithms to do a better job in prediction.

Why is just label encoding not sufficient to provide to the model for training? Why do we need one hot encoding?
Problem with label encoding is that it assumes higher the categorical value, better the category. In addition to that inorder to perform “binarization” of the category and include it as a feature to train the model, one hot encoding was much better. The images below show how we encode using hot encoding through vectors before we pass the category when we had to use a machine learning algorithm.

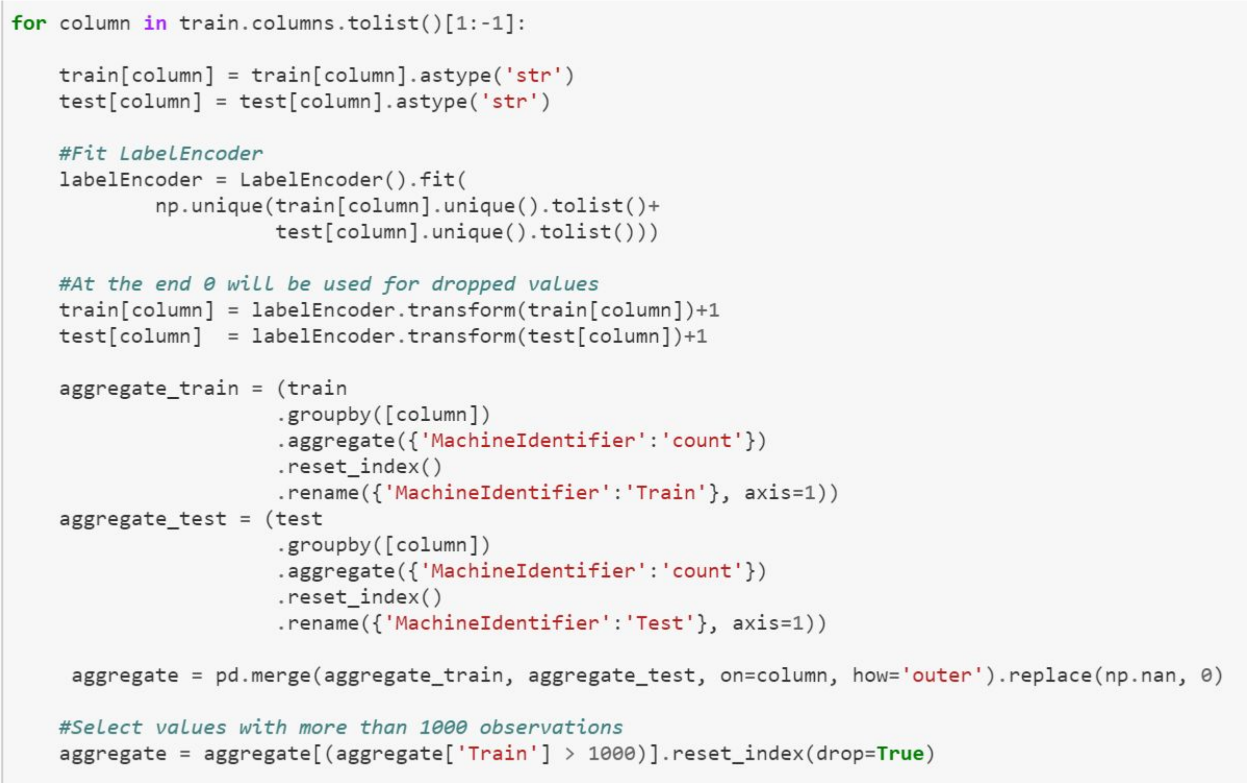






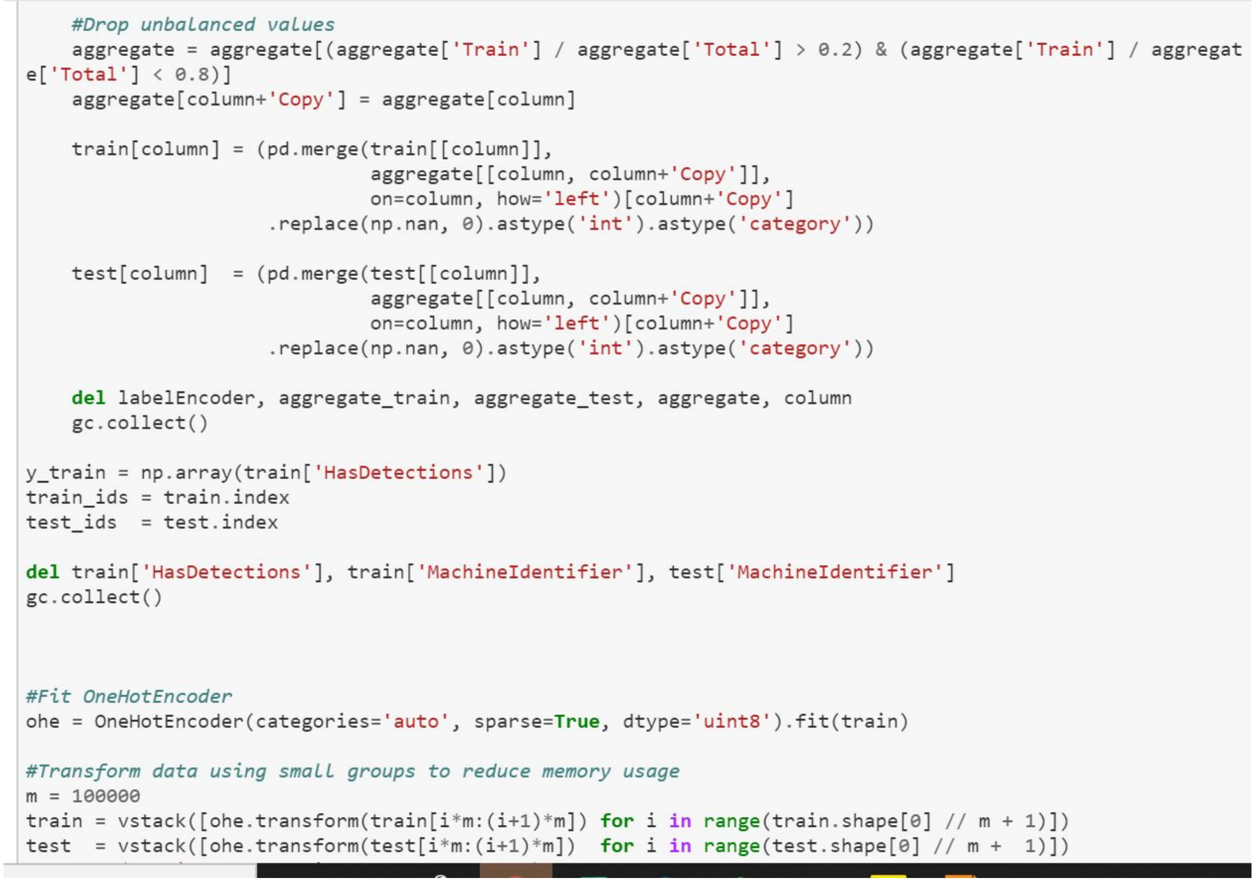
Transforming features and Encoding

The code below shows how the row data was transformed using label encoder and one hot encoder. And then values more than 1000 observations were taken as a sample.



```
for column in train.columns.tolist()[1:-1]:

    train[column] = train[column].astype('str')
    test[column] = test[column].astype('str')

    #Fit LabelEncoder
    labelEncoder = LabelEncoder().fit(
            np.unique(train[column].unique().tolist()+
                      test[column].unique().tolist()))

    #At the end 0 will be used for dropped values
    train[column] = labelEncoder.transform(train[column])+1
    test[column]  = labelEncoder.transform(test[column])+1

    aggregate_train = (train
                      .groupby([column])
                      .aggregate({'MachineIdentifier':'count'})
                      .reset_index()
                      .rename({'MachineIdentifier':'Train'}, axis=1))
    aggregate_test = (test
                      .groupby([column])
                      .aggregate({'MachineIdentifier':'count'})
                      .reset_index()
                      .rename({'MachineIdentifier':'Test'}, axis=1))

     aggregate = pd.merge(aggregate_train, aggregate_test, on=column, how='outer').replace(np.nan, 0)

    #Select values with more than 1000 observations
    aggregate = aggregate[(aggregate['Train'] > 1000)].reset_index(drop=True)
```

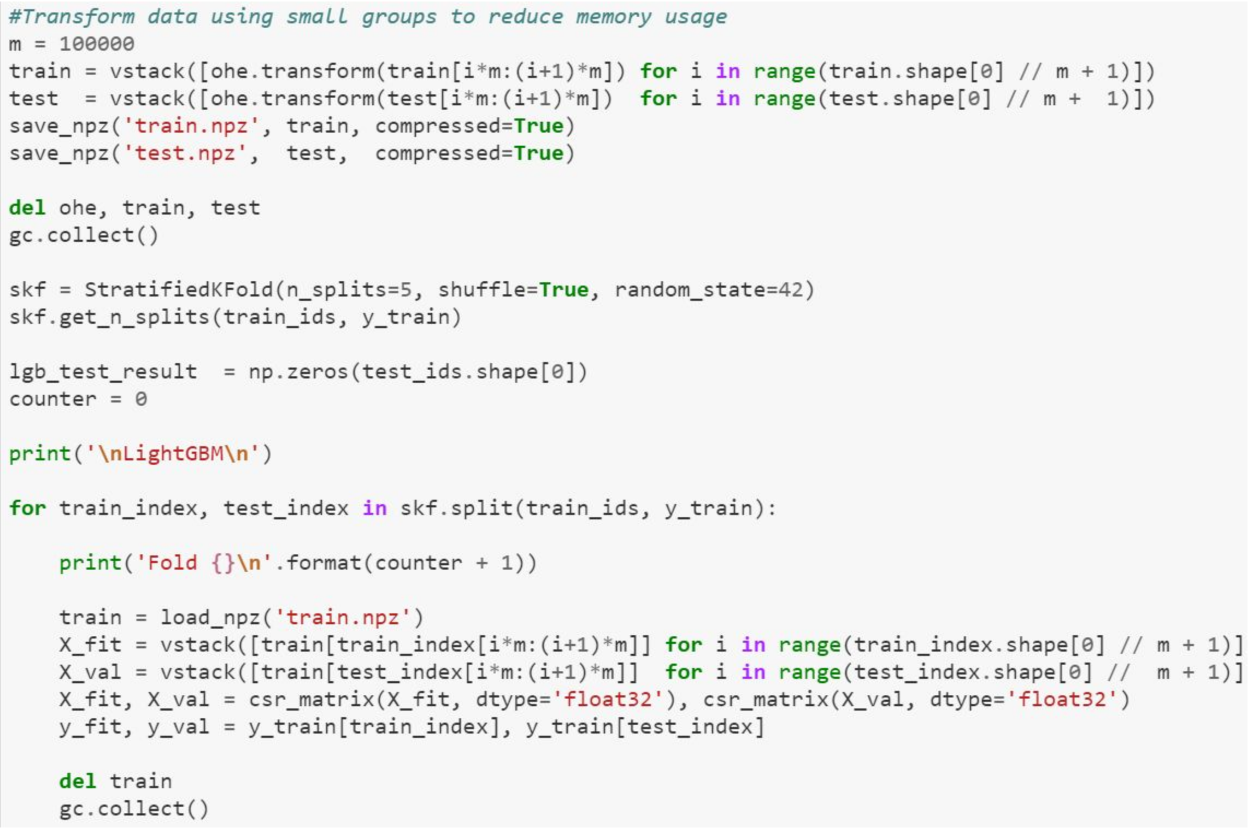
After that the biggest challenge was having unbalanced testing and training values hence we dropped the ones that were unbalanced. That is how we dealt with this challenge as our mentor was mentioning that as a good solution for this kind of case.



```
    #Drop unbalanced values
    aggregate = aggregate[(aggregate['Train'] / aggregate['Total'] > 0.2) & (aggregate['Train'] / aggregat
e['Total'] < 0.8)]
    aggregate[column+'Copy'] = aggregate[column]

    train[column] = (pd.merge(train[[column]],
                              aggregate[[column, column+'Copy']],
                              on=column, how='left')[column+'Copy']
                     .replace(np.nan, 0).astype('int').astype('category'))

    test[column]  = (pd.merge(test[[column]],
                              aggregate[[column, column+'Copy']],
                              on=column, how='left')[column+'Copy']
                     .replace(np.nan, 0).astype('int').astype('category'))

    del labelEncoder, aggregate_train, aggregate_test, aggregate, column
    gc.collect()

y_train = np.array(train['HasDetections'])
train_ids = train.index
test_ids  = test.index

del train['HasDetections'], train['MachineIdentifier'], test['MachineIdentifier']
gc.collect()



#Fit OneHotEncoder
ohe = OneHotEncoder(categories='auto', sparse=True, dtype='uint8').fit(train)

#Transform data using small groups to reduce memory usage
m = 100000
train = vstack([ohe.transform(train[i*m:(i+1)*m]) for i in range(train.shape[0] // m + 1)])
test  = vstack([ohe.transform(test[i*m:(i+1)*m])  for i in range(test.shape[0] // m +  1)])
```

After fitting the one hot encoder we were dealing with memory usage which we will discuss briefly in the next page.

Memory utilization



```
#Transform data using small groups to reduce memory usage
m = 100000
train = vstack([ohe.transform(train[i*m:(i+1)*m]) for i in range(train.shape[0] // m + 1)])
test  = vstack([ohe.transform(test[i*m:(i+1)*m])  for i in range(test.shape[0] // m +  1)])
save_npz('train.npz', train, compressed=True)
save_npz('test.npz',  test,  compressed=True)

del ohe, train, test
gc.collect()

skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
skf.get_n_splits(train_ids, y_train)

lgb_test_result  = np.zeros(test_ids.shape[0])
counter = 0

print('\nLightGBM\n')

for train_index, test_index in skf.split(train_ids, y_train):

    print('Fold {}\n'.format(counter + 1))

    train = load_npz('train.npz')
    X_fit = vstack([train[train_index[i*m:(i+1)*m]] for i in range(train_index.shape[0] // m + 1)]
    X_val = vstack([train[test_index[i*m:(i+1)*m]]  for i in range(test_index.shape[0] //  m + 1)]
    X_fit, X_val = csr_matrix(X_fit, dtype='float32'), csr_matrix(X_val, dtype='float32')
    y_fit, y_val = y_train[train_index], y_train[test_index]

    del train
    gc.collect()
```

Vstack Vs simple approach

```
X_fit = vstack([train[train_index[i*m:(i+1)*m]] for i in range(train_index.shape[0] // m + 1)])
```

Vs

```
X_fit = train[train_index]
```

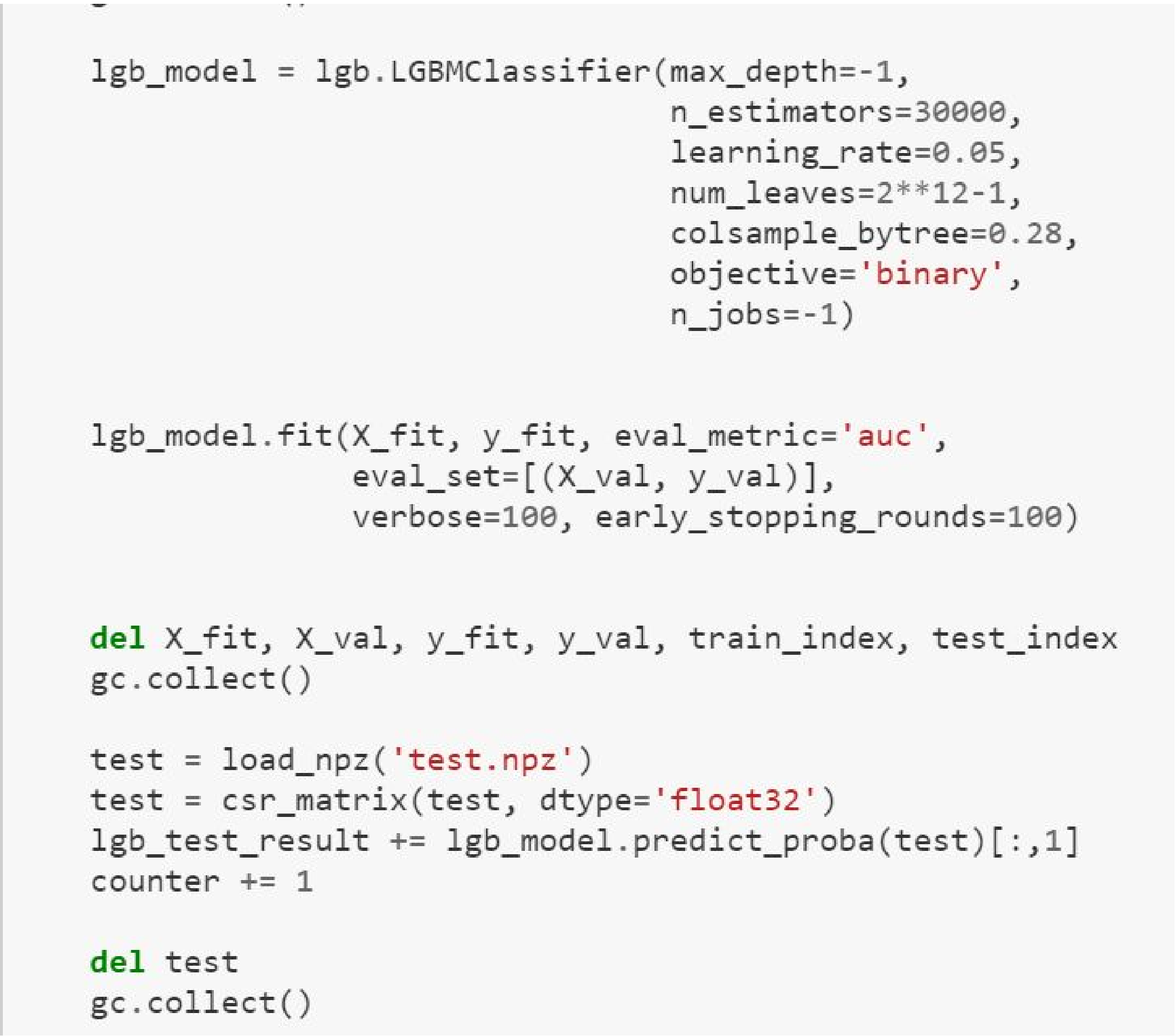
Using vstack helps to fit the kernel inside the memory

Criterias used for Unbalanced Values

The criteria that was used to deal with unbalanced values between train and test was by analyzing the number of observations between the two. This was important because such groups may have different density functions and it can have a negative impact on the model. Some values are temporary and have a limited life cycle.

In production it might be necessary to know what categories are temporary to exclude them from the model or predict the behavior of the categories by the time series row, The main reason was to reduce memory usage. Also small groups have little impact on the final score.

Model Selection



```
    lgb_model = lgb.LGBMClassifier(max_depth=-1,
                                   n_estimators=30000,
                                   learning_rate=0.05,
                                   num_leaves=2**12-1,
                                   colsample_bytree=0.28,
                                   objective='binary',
                                   n_jobs=-1)


    lgb_model.fit(X_fit, y_fit, eval_metric='auc',
                  eval_set=[(X_val, y_val)],
                  verbose=100, early_stopping_rounds=100)


    del X_fit, X_val, y_fit, y_val, train_index, test_index
    gc.collect()

    test = load_npz('test.npz')
    test = csr_matrix(test, dtype='float32')
    lgb_test_result += lgb_model.predict_proba(test)[:,1]
    counter += 1

    del test
    gc.collect()
```

LGBMClassifier uses a scikit-learn style api. That was the reason why we used the classifier right away as our model. But if we use original lgbm, we might need to create LGB.datasets first and then pass it which is a lot of work.

The parameters are initialized as it is shown in the snap above. It is especially known that learning rate affects accuracy and we found that 0.05 performs better. It is a hyper parameter which can be improved later on.

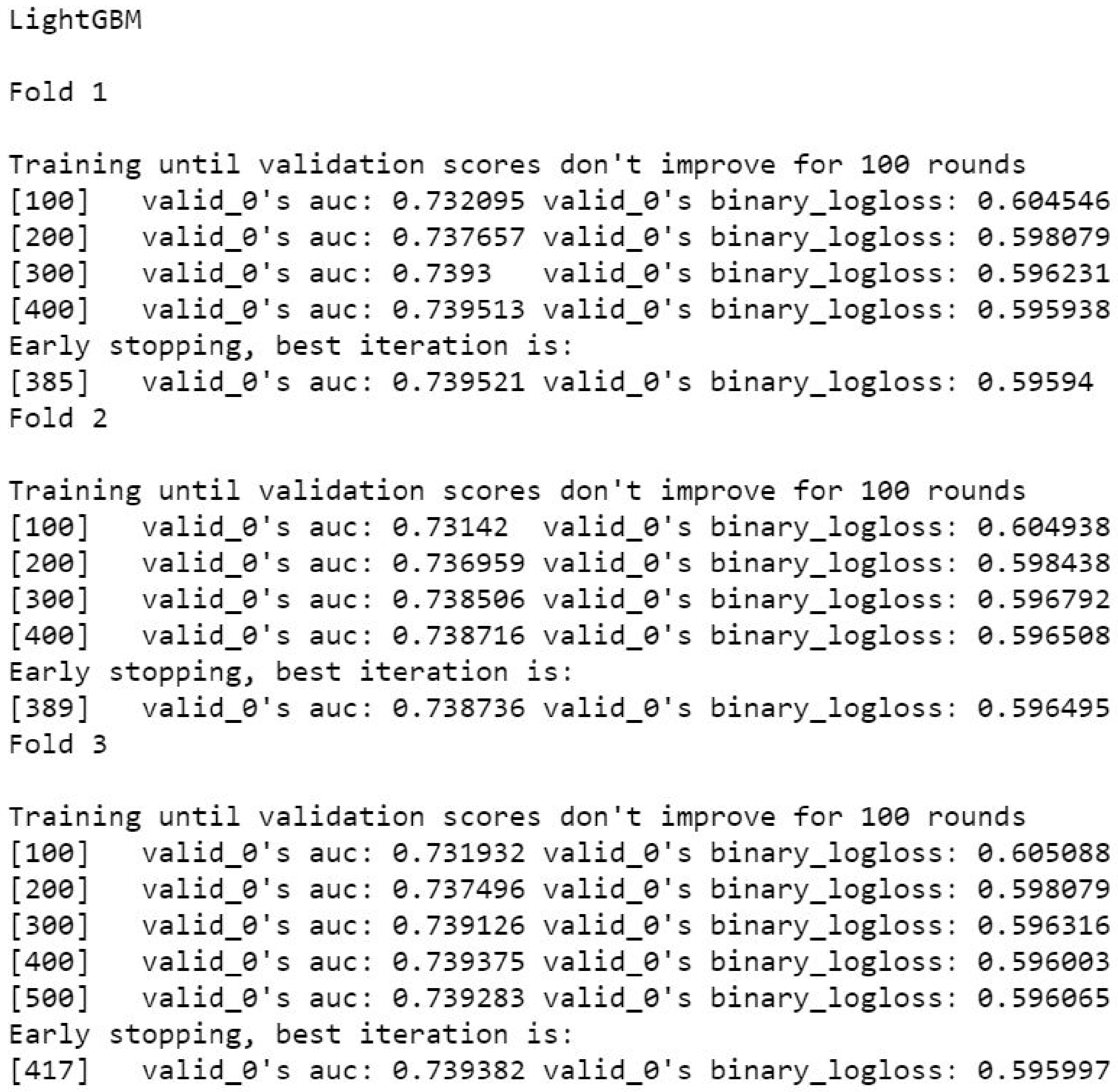
Parameter tuning

colsample_bytree - allows to use all the 82 variables without removing the bad ones. Using small values for this gives better results. This is because now the bad ones get randomly "removed" as new trees are added. (Trees that use bad ones get low weighting.). However, this might change with some optimization.

There were also other hyper parameters that were affecting the algorithm which are like the learning rate, the number of leaves, the depth and the like. Most of them we initialized them all as follows:

```
max_depth=-1,

 n_estimators=30000,

 learning_rate=0.05,

 num_leaves=2**12-1,

 colsample_bytree=0.28,

 objective='binary',

 n_jobs=-1)
```

Accuracy

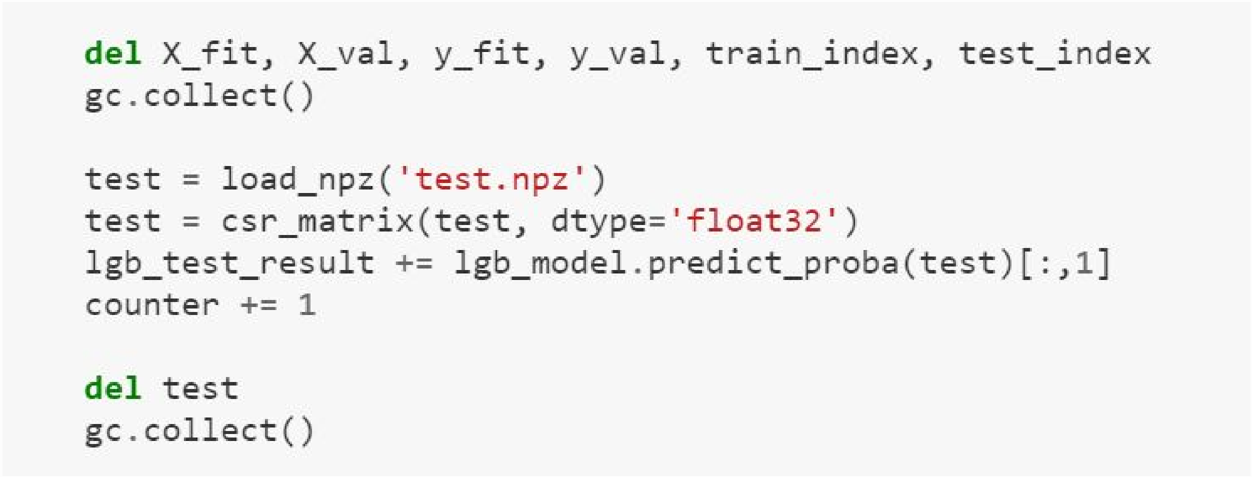


```
LightGBM

Fold 1

Training until validation scores don't improve for 100 rounds
[100]   valid_0's auc: 0.732095 valid_0's binary_logloss: 0.604546
[200]   valid_0's auc: 0.737657 valid_0's binary_logloss: 0.598079
[300]   valid_0's auc: 0.7393   valid_0's binary_logloss: 0.596231
[400]   valid_0's auc: 0.739513 valid_0's binary_logloss: 0.595938
Early stopping, best iteration is:
[385]   valid_0's auc: 0.739521 valid_0's binary_logloss: 0.59594
Fold 2

Training until validation scores don't improve for 100 rounds
[100]   valid_0's auc: 0.73142  valid_0's binary_logloss: 0.604938
[200]   valid_0's auc: 0.736959 valid_0's binary_logloss: 0.598438
[300]   valid_0's auc: 0.738506 valid_0's binary_logloss: 0.596792
[400]   valid_0's auc: 0.738716 valid_0's binary_logloss: 0.596508
Early stopping, best iteration is:
[389]   valid_0's auc: 0.738736 valid_0's binary_logloss: 0.596495
Fold 3

Training until validation scores don't improve for 100 rounds
[100]   valid_0's auc: 0.731932 valid_0's binary_logloss: 0.605088
[200]   valid_0's auc: 0.737496 valid_0's binary_logloss: 0.598079
[300]   valid_0's auc: 0.739126 valid_0's binary_logloss: 0.596316
[400]   valid_0's auc: 0.739375 valid_0's binary_logloss: 0.596003
[500]   valid_0's auc: 0.739283 valid_0's binary_logloss: 0.596065
Early stopping, best iteration is:
[417]   valid_0's auc: 0.739382 valid_0's binary_logloss: 0.595997
```

As we can see from the result on the right side the accuracy has not improved after the second fold. And the binary_logloss also started increasing. However after the 300th iteration of the 3rd fold, the accuracy again started to increase resulting in 73.93%. Here is the detailed information on which fold gabe the best accuracy. The best accuracy for the first fold is **73.8998%** at iteration **402.** The best accuracy for the second fold is **73.9095%** at iteration **390.** The best accuracy for the third fold is **73.939382%** at iteration **417.**

Prediction

Prediction is divided by 5 since it's a 6 fold CV. This is more of like a 6 Models kind of average



```
del X_fit, X_val, y_fit, y_val, train_index, test_index
gc.collect()

test = load_npz('test.npz')
test = csr_matrix(test, dtype='float32')
lgb_test_result += lgb_model.predict_proba(test)[:,1]
counter += 1

del test
gc.collect()
```

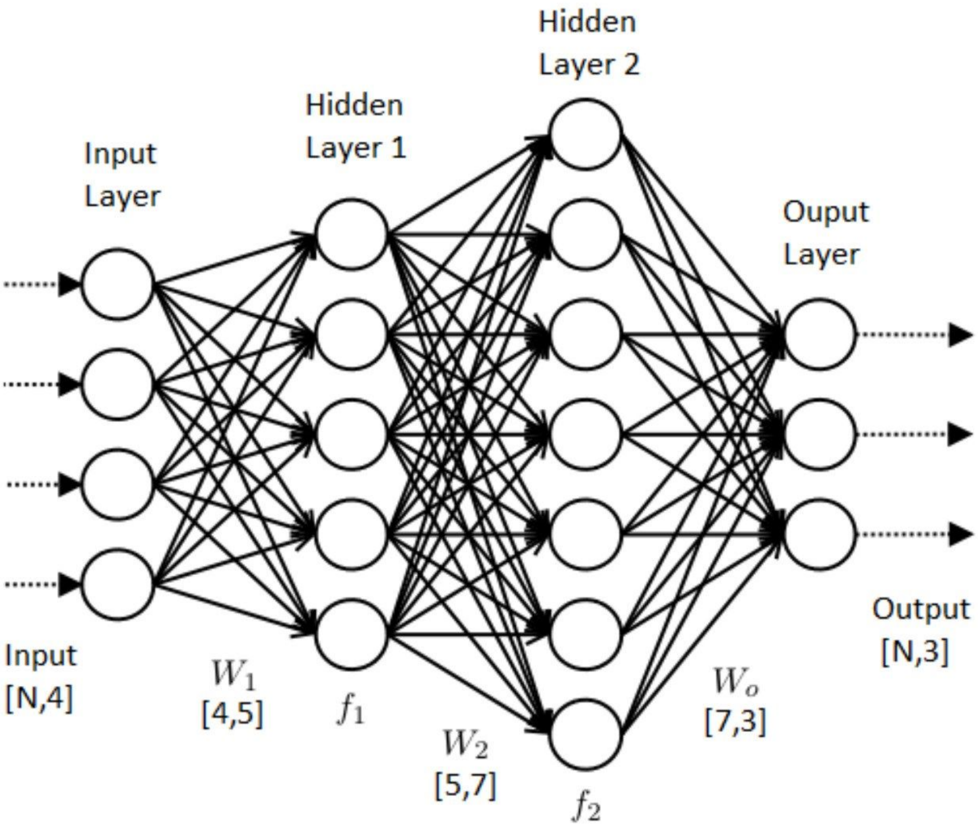
The above code shows how we loaded the test data that we already preprocessed and saved and how we used the model “lgb_model” to predict the test data.

Run time

The algorithm took around **24262.4 seconds** which is around 6 hours for 6 folds each with **100** rounds each round being **100** iterations.

Machine Learning technique 2

Neural Network

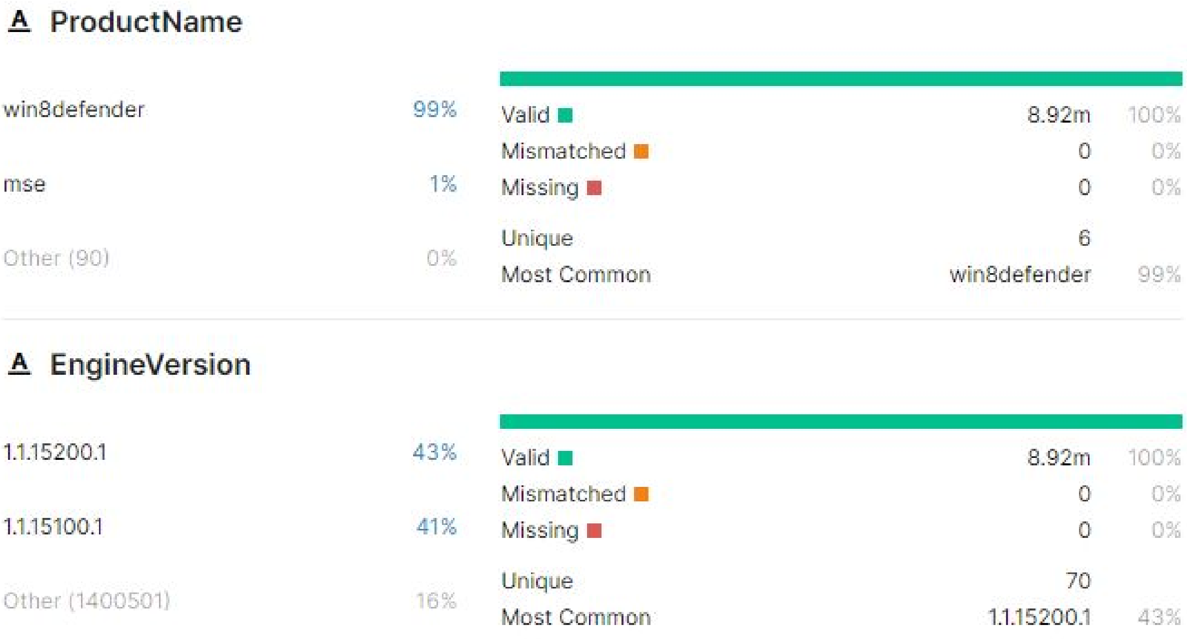


Data Preprocessing

We first studied three categorical features encoding techniques which are Drop categorical variables, Frequency encoding, and One-Hot encoding.

Drop Categorical variables: This is the easiest technique of all.The mechanism is to remove categorical variables which do not contain any useful information. We used this technique to preprocess our dataset on milestone 2, however, it turns out that our data has many categorical variables of which categories' frequency is very important to study. Therefore, we decided not to use this technique.

Frequency encoding:This technique uses the frequency of each category as labels.It helps the model to understand and assign the weight to categories according to the nature of the data. Many of the categorical variables in our dataset had frequently appearing categories and we believed it would be important to study about these categories so we decided to apply Frequency encoding technique.



A ProductName

win8defender	99%
mse	1%
Other (90)	0%

Valid 🟩	8.92m	100%
Mismatched 🟧	0	0%
Missing 🟥	0	0%
Unique	6	
Most Common	win8defender	99%

A EngineVersion

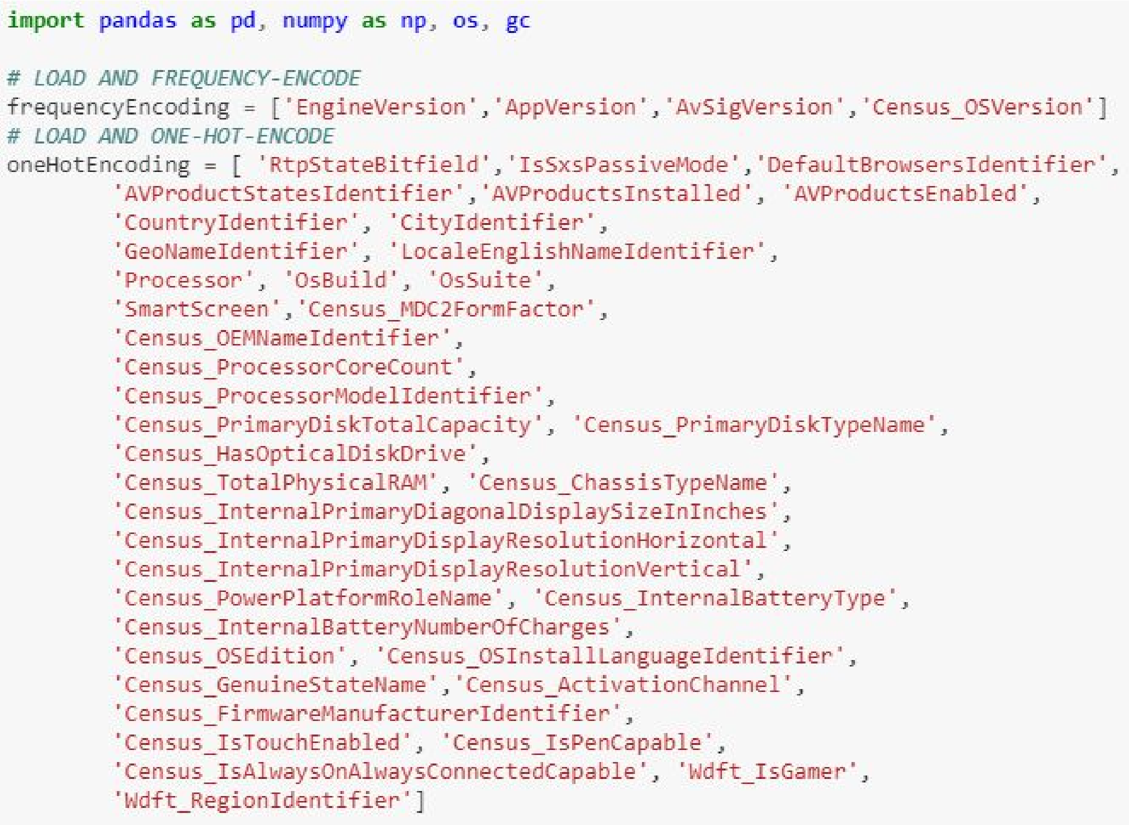
1.1.15200.1	43%
1.1.15100.1	41%
Other (1400501)	16%

Valid 🟩	8.92m	100%
Mismatched 🟧	0	0%
Missing 🟥	0	0%
Unique	70	
Most Common	1.1.15200.1	43%

Snipped image from Kaggle that shows the most common categories from categorical variables.

One-Hot encoding: This technique represents categories in the form of vectors. By creating columns, One-Hot-encoding makes it easy to indicate the presence/absence of each category. This technique also works well if there's no ordering in the categorical data. As our dataset had many categories missing and no clear ordering, using One-Hot-encoding to preprocess our data was the best choice to feed into the Neural Network model.

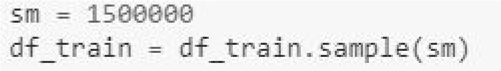
After carefully studying the aforementioned encoding techniques, we decided to identify which categories should be encoded with which encoding technique. We identified the encoding techniques for the categories according to the nature of the categories we studied on kaggle. For the categorical variables which have frequently appearing categories, we decided to encode them using frequency encoding; for categorical variables which have many missing categories and not many categories, we decided to encode them using one-hot encoding technique.



```
import pandas as pd, numpy as np, os, gc

# LOAD AND FREQUENCY-ENCODE
frequencyEncoding = ['EngineVersion','AppVersion','AvSigVersion','Census_OSVersion']
# LOAD AND ONE-HOT-ENCODE
oneHotEncoding = [ 'RtpStateBitfield','IsSxsPassiveMode','DefaultBrowsersIdentifier',
        'AVProductStatesIdentifier','AVProductsInstalled', 'AVProductsEnabled',
        'CountryIdentifier', 'CityIdentifier',
        'GeoNameIdentifier', 'LocaleEnglishNameIdentifier',
        'Processor', 'OsBuild', 'OsSuite',
        'SmartScreen','Census_MDC2FormFactor',
        'Census_OEMNameIdentifier',
        'Census_ProcessorCoreCount',
        'Census_ProcessorModelIdentifier',
        'Census_PrimaryDiskTotalCapacity', 'Census_PrimaryDiskTypeName',
        'Census_HasOpticalDiskDrive',
        'Census_TotalPhysicalRAM', 'Census_ChassisTypeName',
        'Census_InternalPrimaryDiagonalDisplaySizeInInches',
        'Census_InternalPrimaryDisplayResolutionHorizontal',
        'Census_InternalPrimaryDisplayResolutionVertical',
        'Census_PowerPlatformRoleName', 'Census_InternalBatteryType',
        'Census_InternalBatteryNumberOfCharges',
        'Census_OSEdition', 'Census_OSInstallLanguageIdentifier',
        'Census_GenuineStateName','Census_ActivationChannel',
        'Census_FirmwareManufacturerIdentifier',
        'Census_IsTouchEnabled', 'Census_IsPenCapable',
        'Census_IsAlwaysOnAlwaysConnectedCapable', 'Wdft_IsGamer',
        'Wdft_RegionIdentifier']
```

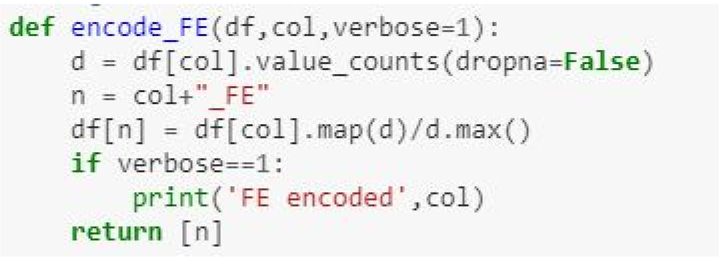
Categorical variables of which categories are going to be encoded either in frequency/one-hot encoding.

Before encoding the categories, we first took a sample of 1500000 rows of the total 8921483 rows using pandas' 'sample(numberOfsample)' method.



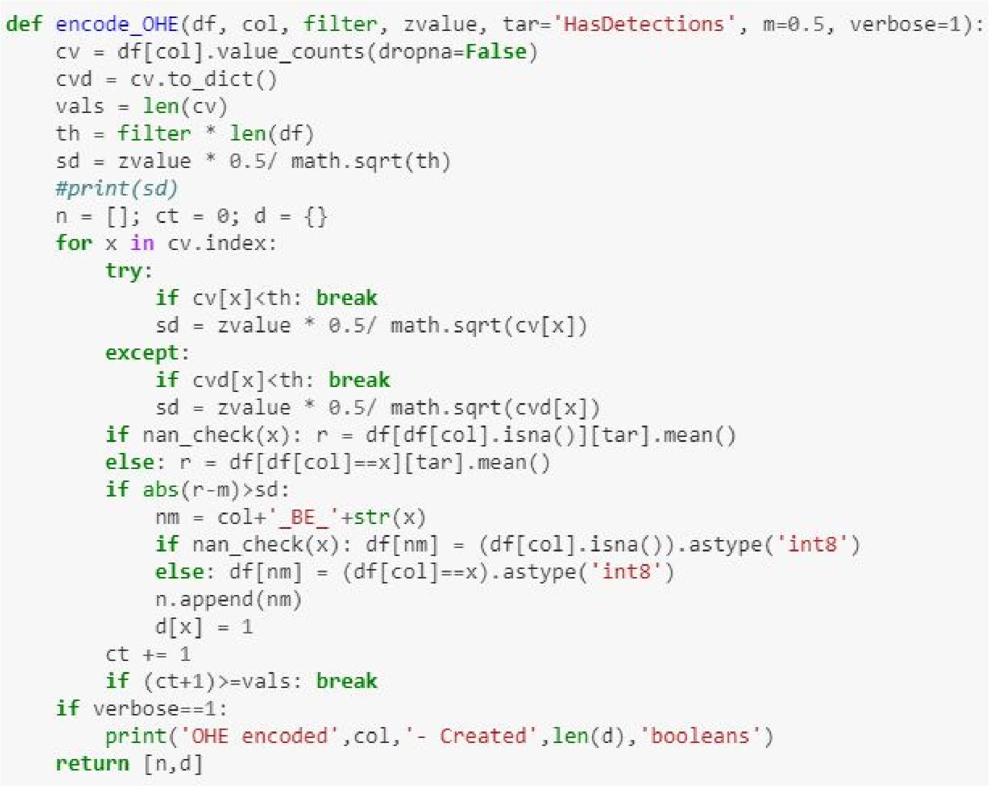
```
sm = 1500000
df_train = df_train.sample(sm)
```

Then for the randomly 1500000 selected rows, we applied the encoding techniques.



```
def encode_FE(df,col,verbose=1):
    d = df[col].value_counts(dropna=False)
    n = col+"_FE"
    df[n] = df[col].map(d)/d.max()
    if verbose==1:
        print('FE encoded',col)
    return [n]
```

Frequency encoding

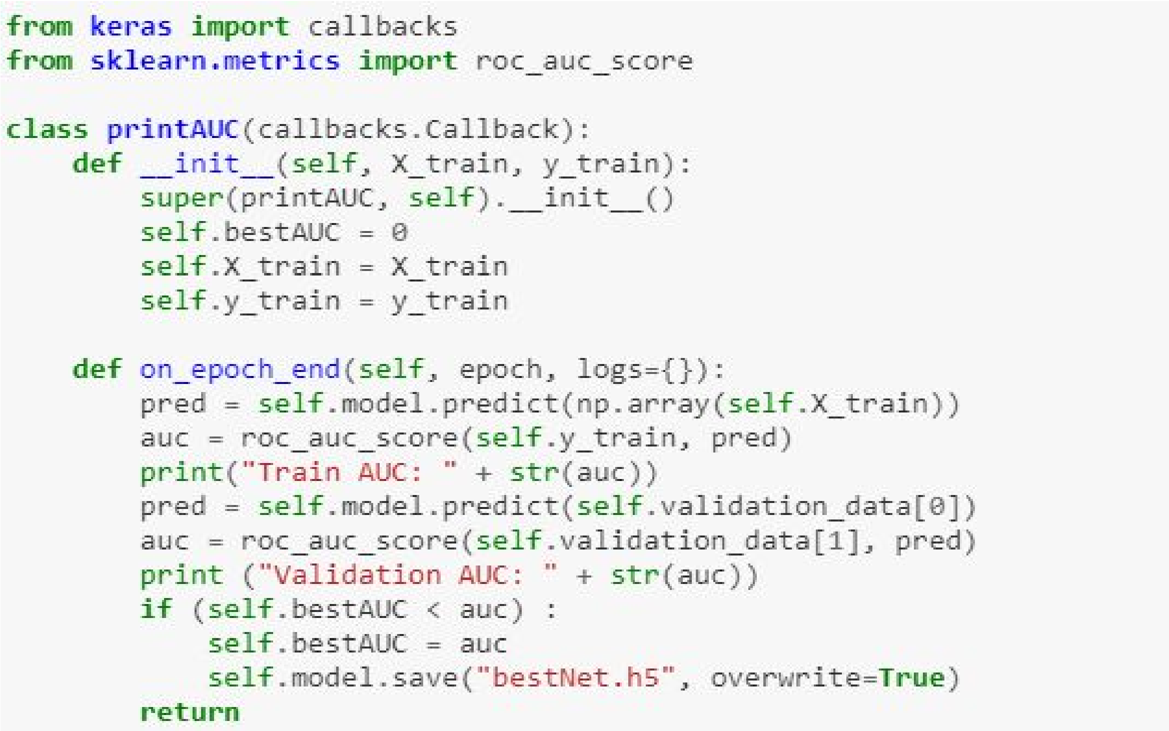


```
def encode_OHE(df, col, filter, zvalue, tar='HasDetections', m=0.5, verbose=1):
    cv = df[col].value_counts(dropna=False)
    cvd = cv.to_dict()
    vals = len(cv)
    th = filter * len(df)
    sd = zvalue * 0.5/ math.sqrt(th)
    #print(sd)
    n = []; ct = 0; d = {}
    for x in cv.index:
        try:
            if cv[x]<th: break
            sd = zvalue * 0.5/ math.sqrt(cv[x])
        except:
            if cvd[x]<th: break
            sd = zvalue * 0.5/ math.sqrt(cvd[x])
        if nan_check(x): r = df[df[col].isna()][tar].mean()
        else: r = df[df[col]==x][tar].mean()
        if abs(r-m)>sd:
            nm = col+'_BE_'+str(x)
            if nan_check(x): df[nm] = (df[col].isna()).astype('int8')
            else: df[nm] = (df[col]==x).astype('int8')
            n.append(nm)
            d[x] = 1
        ct += 1
        if (ct+1)>=vals: break
    if verbose==1:
        print('OHE encoded',col,'- Created',len(d),'booleans')
    return [n,d]
```

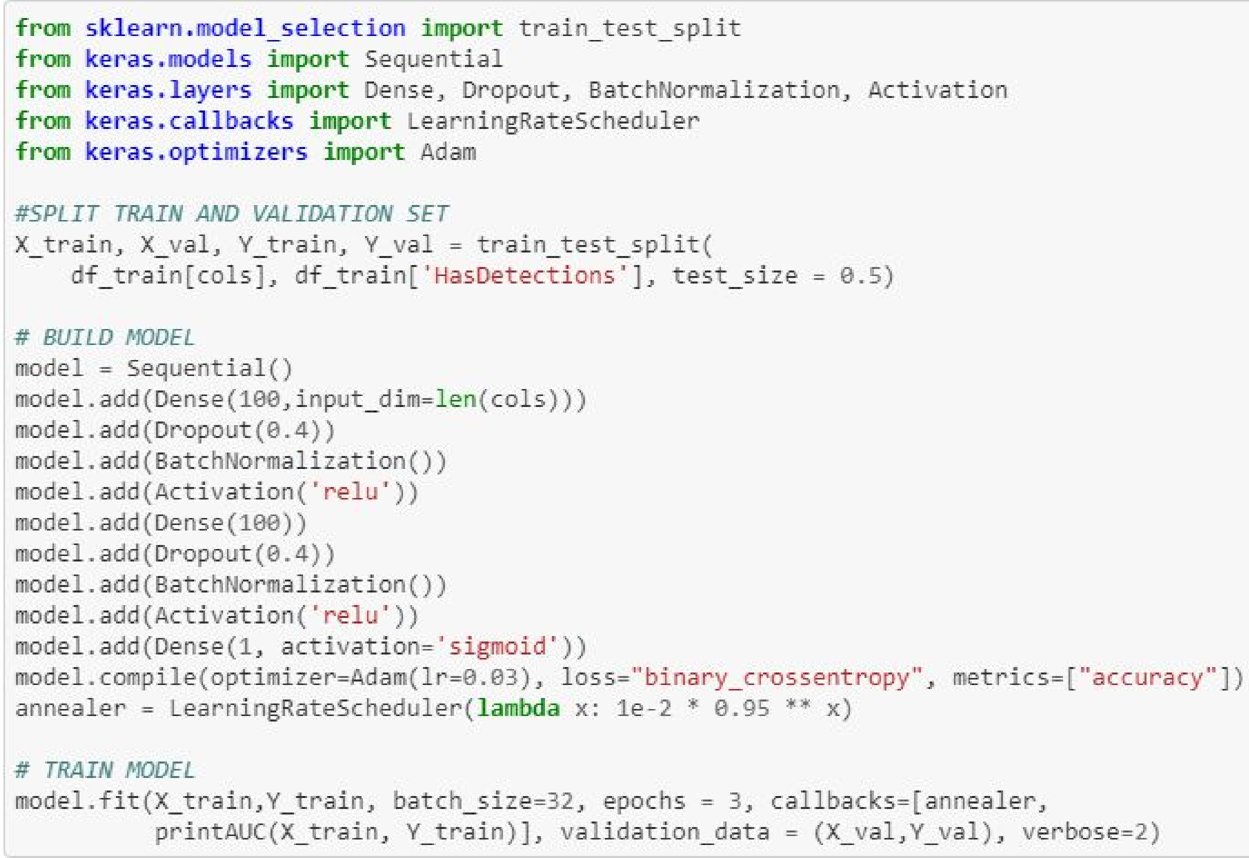
One-Hot encoding

Build and Train Network

We built a 3 layer fully connected network with 100 neurons on each hidden layer. Using ReLU activation, Batch Normalization, 40% Dropout, Adam Optimizer, and Decaying Learning Rate, we tried to increase the accuracy of our neural network. Since we did not have an AUC loss function, so we used Cross Entropy instead. We called a custom Keras callback after each epoch to display the current AUC and continually save the best model.

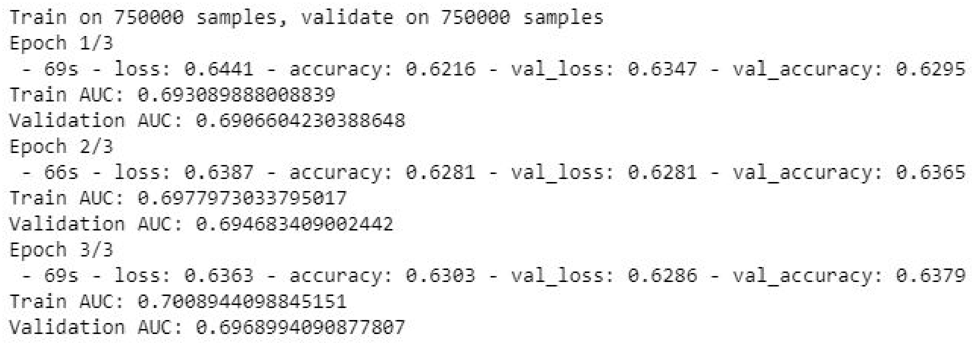


```
from keras import callbacks
from sklearn.metrics import roc_auc_score

class printAUC(callbacks.Callback):
    def __init__(self, X_train, y_train):
        super(printAUC, self).__init__()
        self.bestAUC = 0
        self.X_train = X_train
        self.y_train = y_train

    def on_epoch_end(self, epoch, logs={}):
        pred = self.model.predict(np.array(self.X_train))
        auc = roc_auc_score(self.y_train, pred)
        print("Train AUC: " + str(auc))
        pred = self.model.predict(self.validation_data[0])
        auc = roc_auc_score(self.validation_data[1], pred)
        print ("Validation AUC: " + str(auc))
        if (self.bestAUC < auc) :
            self.bestAUC = auc
            self.model.save("bestNet.h5", overwrite=True)
        return
```



```
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Dense, Dropout, BatchNormalization, Activation
from keras.callbacks import LearningRateScheduler
from keras.optimizers import Adam

#SPLIT TRAIN AND VALIDATION SET
X_train, X_val, Y_train, Y_val = train_test_split(
    df_train[cols], df_train['HasDetections'], test_size = 0.5)

# BUILD MODEL
model = Sequential()
model.add(Dense(100,input_dim=len(cols)))
model.add(Dropout(0.4))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dense(100))
model.add(Dropout(0.4))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer=Adam(lr=0.03), loss="binary_crossentropy", metrics=["accuracy"])
annealer = LearningRateScheduler(lambda x: 1e-2 * 0.95 ** x)

# TRAIN MODEL
model.fit(X_train,Y_train, batch_size=32, epochs = 3, callbacks=[annealer,
          printAUC(X_train, Y_train)], validation_data = (X_val,Y_val), verbose=2)
```

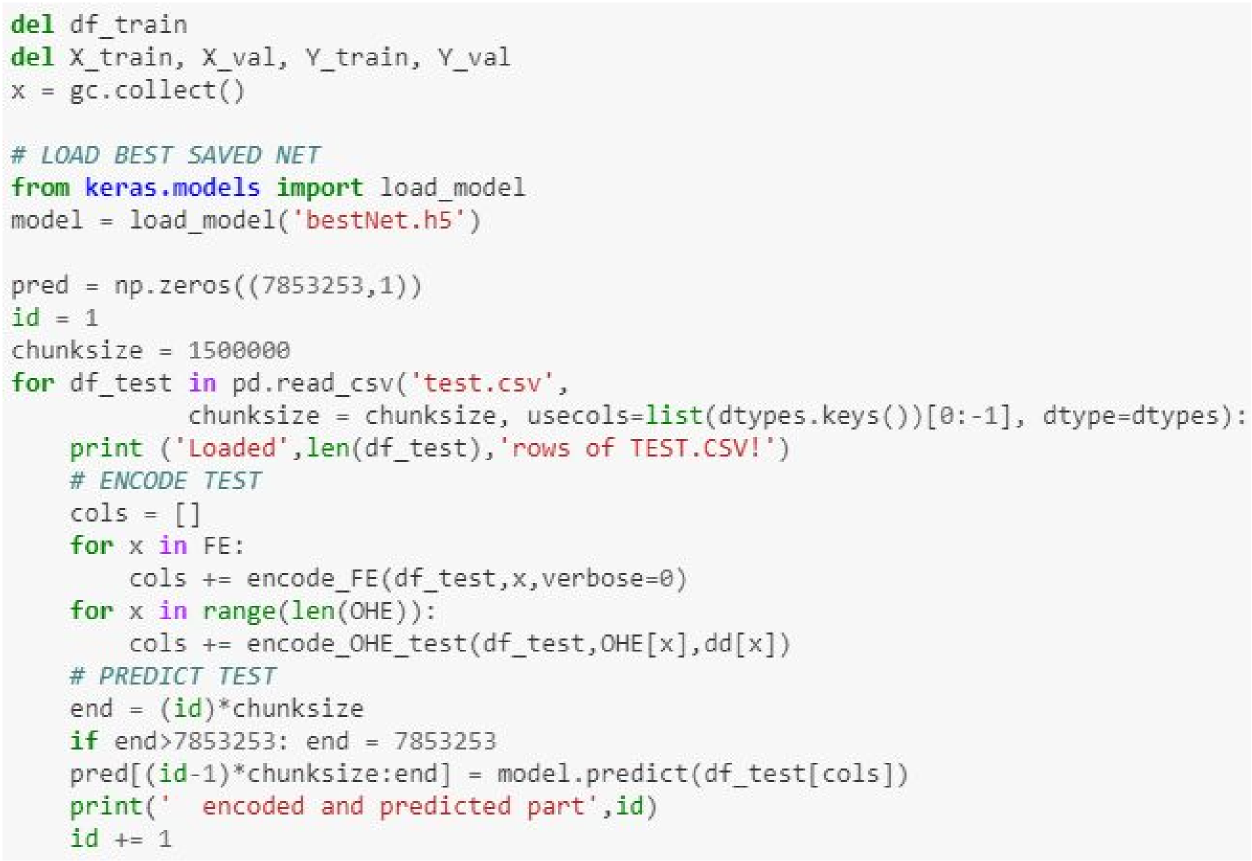
We split the training dataset into a training set and a validation set. We needed to have the validation set because we wanted to find and optimize the best model to solve the challenge. We divided the 1500000 sample rows equally into two for the training set and the validation set. We had 750000 rows and columns for each training set and validation set.Then we fitted the data into the model and began training the model.



```
Train on 750000 samples, validate on 750000 samples
Epoch 1/3
 - 69s - loss: 0.6441 - accuracy: 0.6216 - val_loss: 0.6347 - val_accuracy: 0.6295
Train AUC: 0.693089888008839
Validation AUC: 0.6906604230388648
Epoch 2/3
 - 66s - loss: 0.6387 - accuracy: 0.6281 - val_loss: 0.6281 - val_accuracy: 0.6365
Train AUC: 0.6977973033795017
Validation AUC: 0.694683409002442
Epoch 3/3
 - 69s - loss: 0.6363 - accuracy: 0.6303 - val_loss: 0.6286 - val_accuracy: 0.6379
Train AUC: 0.7008944098845151
Validation AUC: 0.6968994090877807
```

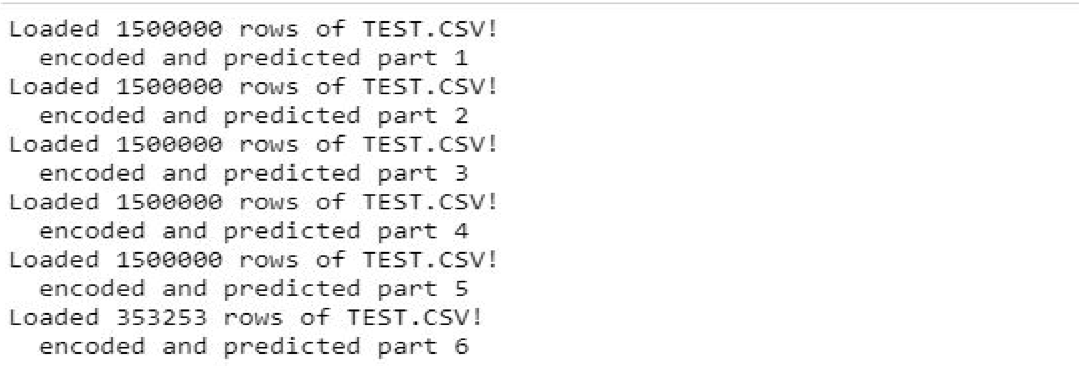
Predict Testing File

Our neural network needed a lot of available RAM even after deleting the training data. Therefore we needed to load the test.csv file by chunks and predict by chunks. we specify a chunk size.



```
del df_train
del X_train, X_val, Y_train, Y_val
x = gc.collect()

# LOAD BEST SAVED NET
from keras.models import load_model
model = load_model('bestNet.h5')

pred = np.zeros((7853253,1))
id = 1
chunksize = 1500000
for df_test in pd.read_csv('test.csv',
            chunksize = chunksize, usecols=list(dtypes.keys())[0:-1], dtype=dtypes):
    print ('Loaded',len(df_test),'rows of TEST.CSV!')
    # ENCODE TEST
    cols = []
    for x in FE:
        cols += encode_FE(df_test,x,verbose=0)
    for x in range(len(OHE)):
        cols += encode_OHE_test(df_test,OHE[x],dd[x])
    # PREDICT TEST
    end = (id)*chunksize
    if end>7853253: end = 7853253
    pred[(id-1)*chunksize:end] = model.predict(df_test[cols])
    print('  encoded and predicted part',id)
    id += 1
```

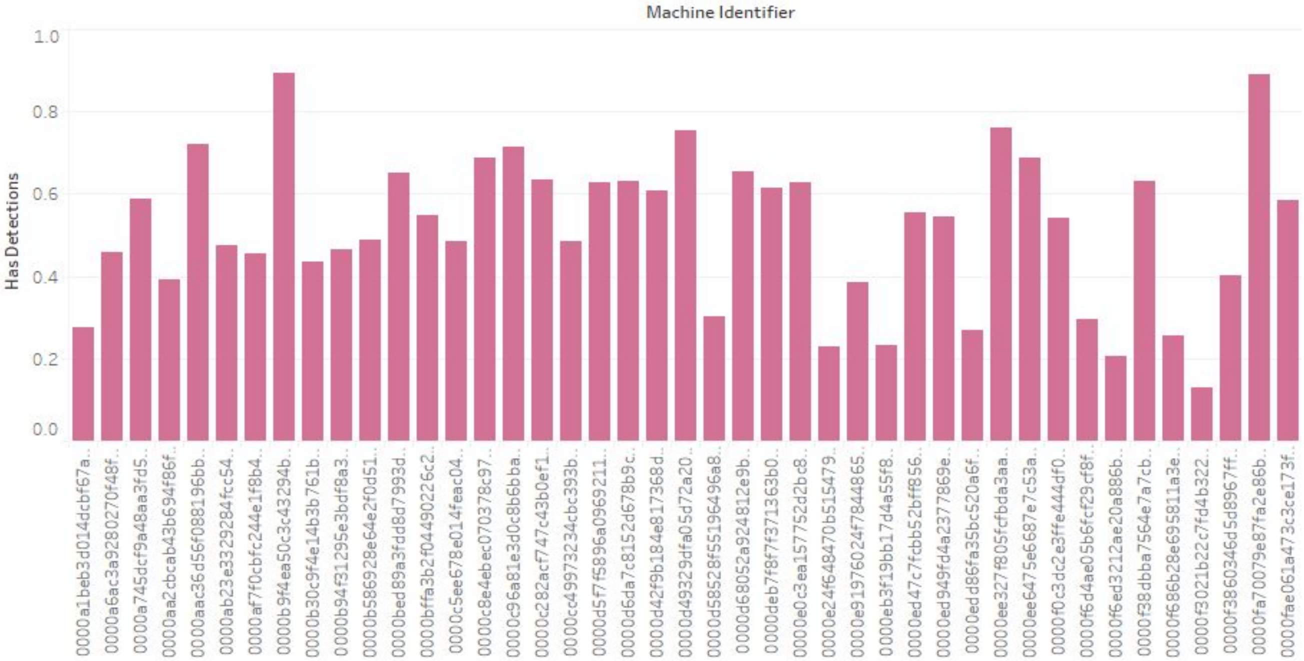
The test.csv file had a total of 7853253 rows. We needed to divide them and predict the results by a chunksize of 1500000 , by the number of rows we trained our model.



```
Loaded 1500000 rows of TEST.CSV!
    encoded and predicted part 1
Loaded 1500000 rows of TEST.CSV!
    encoded and predicted part 2
Loaded 1500000 rows of TEST.CSV!
    encoded and predicted part 3
Loaded 1500000 rows of TEST.CSV!
    encoded and predicted part 4
Loaded 1500000 rows of TEST.CSV!
    encoded and predicted part 5
Loaded 353253 rows of TEST.CSV!
    encoded and predicted part 6
```

Results

Our validation AUC was 0.703.



Result visualization using tableau

Memory Utilization

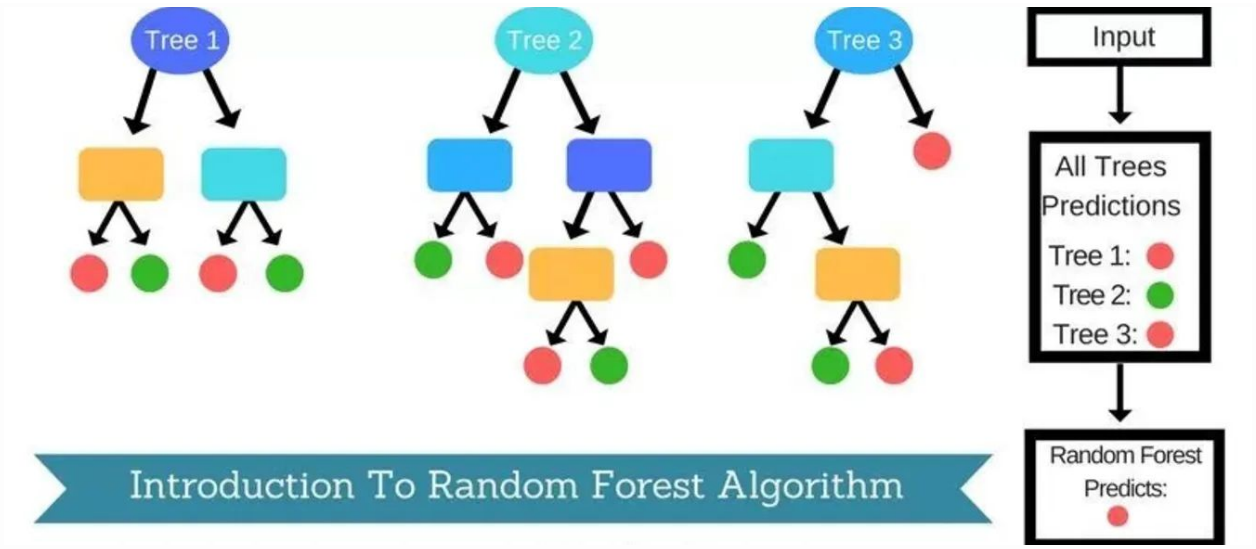
Loading the test.csv file directly and testing heavily used memory and we needed to load and predict by chunks. It still needed a lot of available RAM though.

Run Time

The algorithm took around **4336.7 seconds** which is around 1 hour and 12 minutes.

Machine Learning technique 3

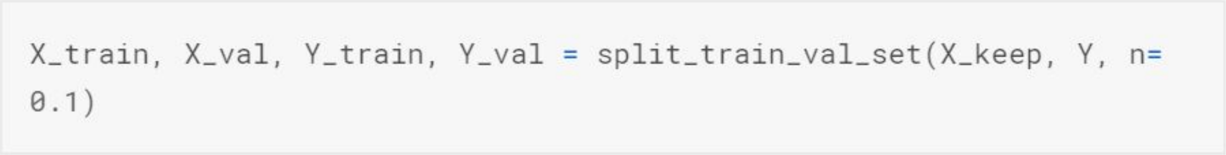
Random Forest



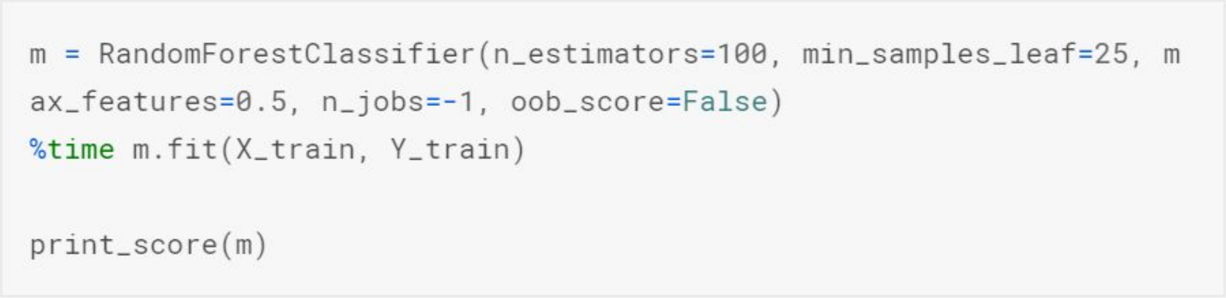
What is random forest

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set.

Here is the process of initializing the classifier with different parameters.

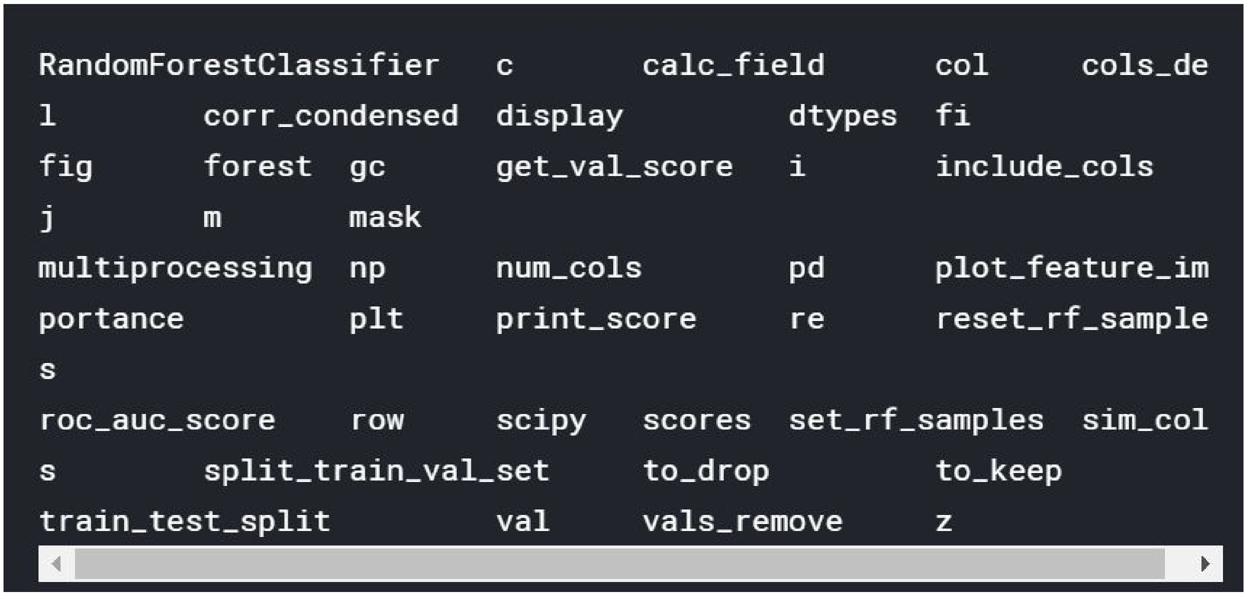


```
X_train, X_val, Y_train, Y_val = split_train_val_set(X_keep, Y, n=
0.1)
```



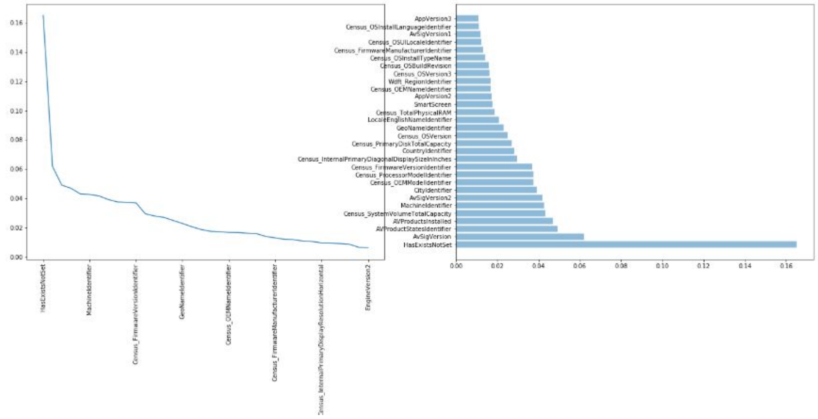
```
m = RandomForestClassifier(n_estimators=100, min_samples_leaf=25, m
ax_features=0.5, n_jobs=-1, oob_score=False)
%time m.fit(X_train, Y_train)

print_score(m)
```

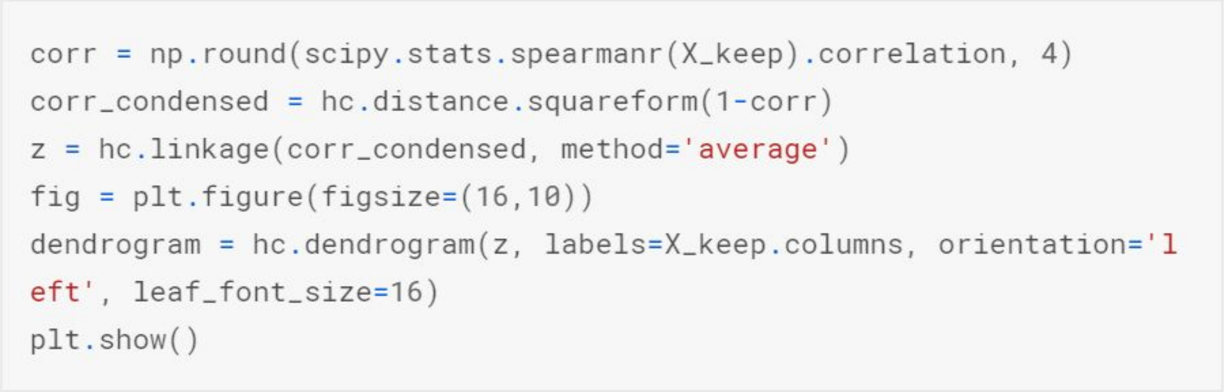
After changing the parameters here is the final model that was used.



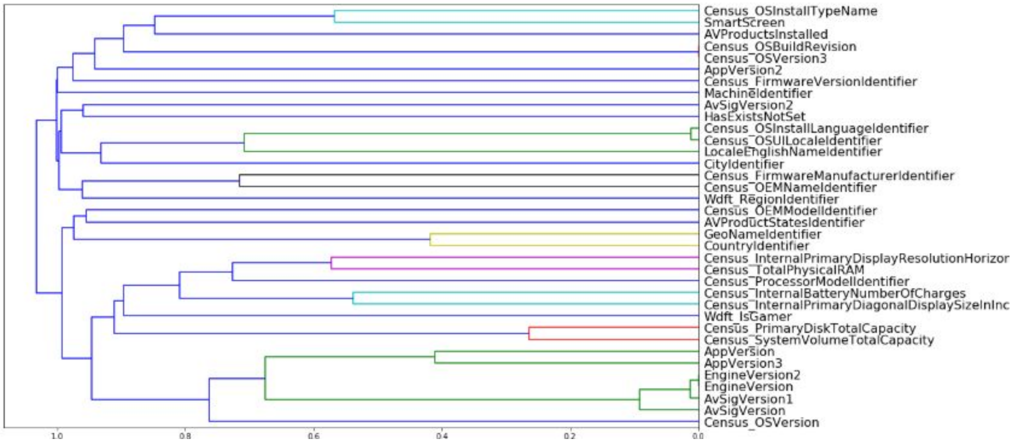
```
RandomForestClassifier   c       calc_field      col     cols_de
l        corr_condensed  display         dtypes  fi
fig      forest  gc      get_val_score   i       include_cols
j        m       mask
multiprocessing  np      num_cols        pd      plot_feature_im
portance         plt     print_score     re      reset_rf_sample
s
roc_auc_score    row     scipy   scores  set_rf_samples  sim_col
s        split_train_val_set     to_drop         to_keep
train_test_split         val     vals_remove     z
```

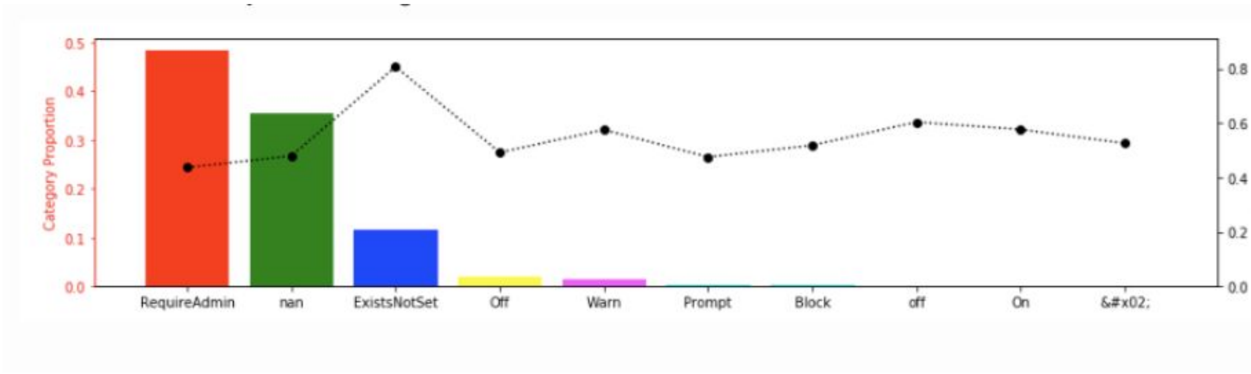
It is obvious that most of the machine learning algorithms require data preprocessing hence, random forest also needs the data to be prepared in certain ways. We have done the typical cleaning using correlation. The following graph shows these relations between the data.

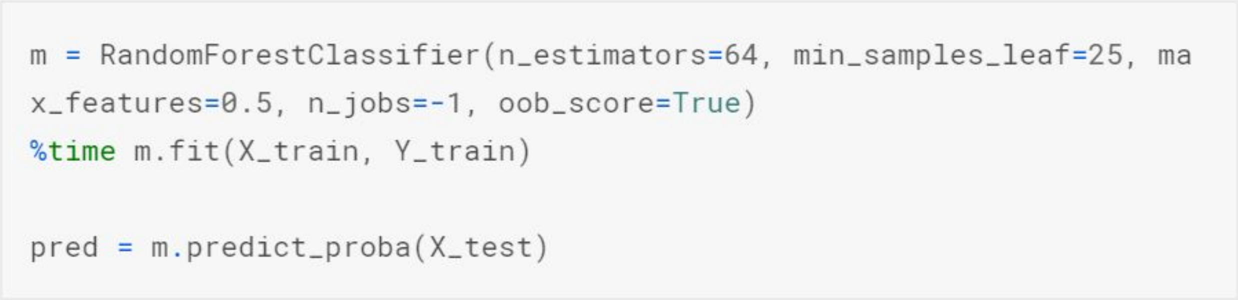




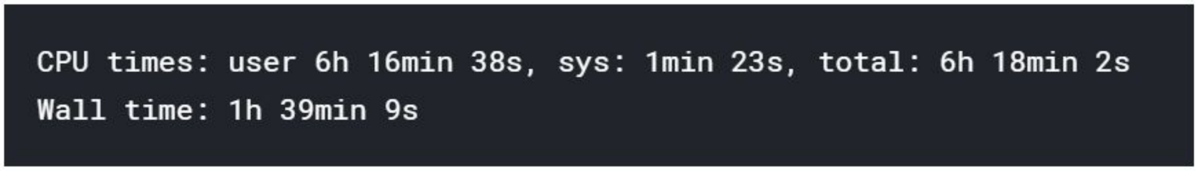
```
corr = np.round(scipy.stats.spearmanr(X_keep).correlation, 4)
corr_condensed = hc.distance.squareform(1-corr)
z = hc.linkage(corr_condensed, method='average')
fig = plt.figure(figsize=(16,10))
dendrogram = hc.dendrogram(z, labels=X_keep.columns, orientation='l
eft', leaf_font_size=16)
plt.show()
```







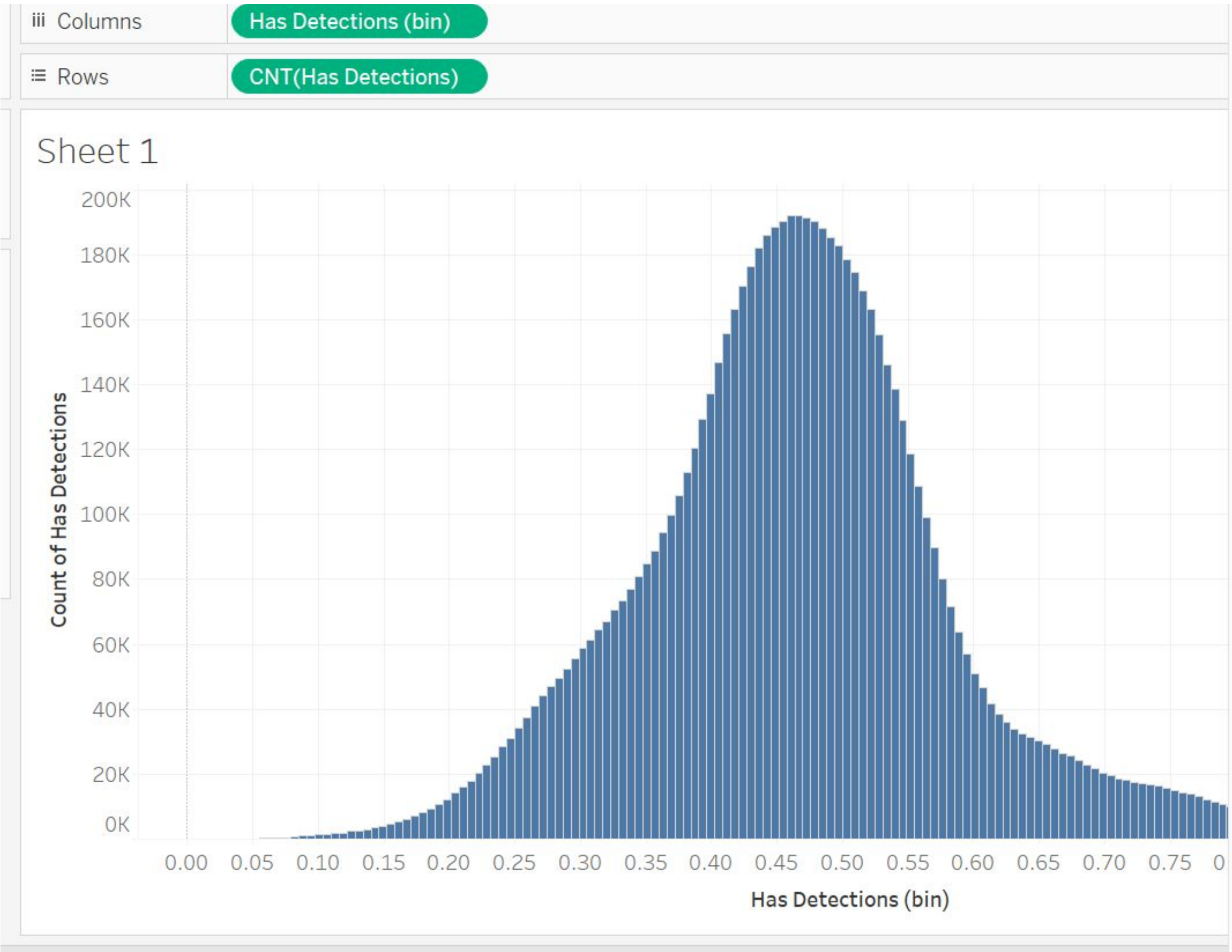
```
m = RandomForestClassifier(n_estimators=64, min_samples_leaf=25, ma
x_features=0.5, n_jobs=-1, oob_score=True)
%time m.fit(X_train, Y_train)

pred = m.predict_proba(X_test)
```

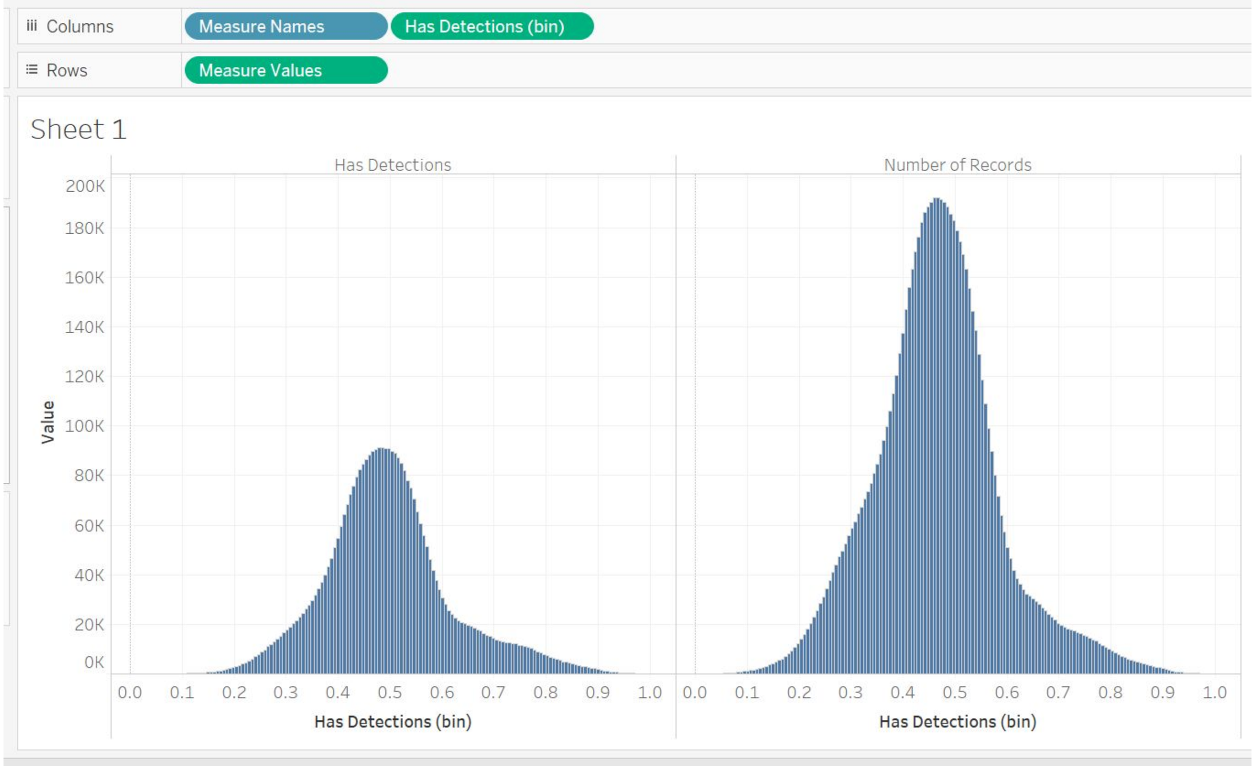


```
CPU times: user 6h 16min 38s, sys: 1min 23s, total: 6h 18min 2s
Wall time: 1h 39min 9s
```

The run time that was taken by this algorithm was 6hrs and 16min 38secs as is shown above.

Data Visualization




Columns
Measure Names
Has Detections (bin)
Rows
Measure Values
Sheet 1
Has Detections
Number of Records
Value
200K
180K
160K
140K
120K
100K
80K
60K
40K
20K
0K
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Has Detections (bin)
Has Detections (bin)

Conclusion

- There is always a tradeoff when we choose different machine learning algorithms in order to carry out our training and testing. It is obvious that there are challenges associated with machine learning algorithms. The main challenges or the so called concerns are memory usage, the time it takes to run and the accuracy that the gives.
- The results of this project also witnesses that. This is because when the first algorithm tries to prioritize memory, the run time has become greater compared to using the later algorithm which uses Neural networks having smaller runtime compared to the first one.
- The third algorithm also consumed almost similar time with the first algorithm. In general we can say that the algorithms that run for a long time were able to perform better.
- One very big conclusion is that all of the algorithms required a way of preprocessing the data therefore, it is very important to know what the data is beforehand and preprocess the data since most of the algorithms would have performed worse if that very crucial step was omitted.

Comparison factors	Light GBM	Neural Network	Random Forest
Run Time	24262.4 seconds	4336.7 seconds	6hr 1min 38sec
Accuracy	73.939382%	70.3%	63%
Memory Utilization	Less memory utilization	High memory utilization	More memory utilization

Future Work

- Now that we have a much better understanding of the commonly used approaches to solve Microsoft Malware Prediction challenge, we will try to apply other machine learning algorithms and improve the scores.
- We will also do hyperparameter tuning to achieve better results
- Apart from that we would see what difference would be made if the iteration increases in terms of increasing the accuracy.
- Takeaways from the project:

    - How to handle categories during preprocessing (Frequency/OH Encoding)

    - How to wisely and efficiently use training and testing datasets.

    - How to apply ML algorithms on problems, such as LBM and NN.

References

https://www.kaggle.com/jiegeng94/everyone-do-this-at-the-beginning/data?select=sample_submission.csv

https://www.kaggle.com/timon88/load-whole-data-without-any-dtypes

https://www.kaggle.com/c/microsoft-malware-prediction/data?select=test.csv

https://www.kaggle.com/harmeggels/random-forest-feature-importances?select=submissionv2.csv