Malware Detection

Data Mining

Assignment 1: MTL782

Vivek Muskan	Aayush Somani	Harsh Kumar
2017MT10755	2017MT10722	2017MT10729

Overview

Our task is to predict whether a PC has Malware or not based on certain given information. We have a labelled dataset which contained around 9 million training and 8 million data, so we selected a subset of it. We selected 1 million as test data, 0.5 million as validation data and 0.5 million as test data. We used EDA techniques for data preprocessing. We also used feature engineering to engineer some new features and remove redundant features. Finally, we used various methods and models to train and then tuned the parameters for better test results.

EDA

We divided EDA into five Major categories.

1. Basic information and Distribution of Data
2. Missing Value problem
3. Feature Engineering
4. Dimensionality Reduction based on Skewness and Correlation

Basic Info and Distribution of Data

Training Data: 1 million rows, 83 column

Test Data: 1 million rows, 83 column

The basic info about our traindata

[-]

```
traindata.info()
```

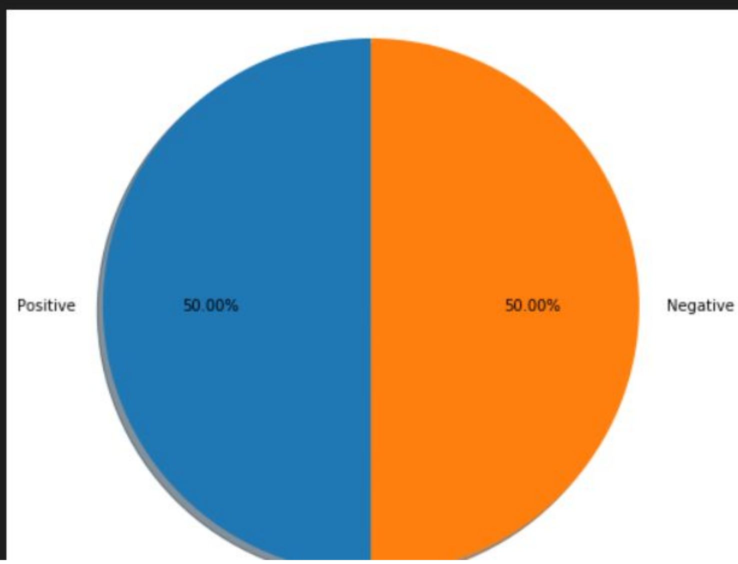
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2000000 entries, 0 to 1999999
Data columns (total 83 columns):
MachineIdentifier                                     object
ProductName                                           object
EngineVersion                                         object
AppVersion                                            object
AvSigVersion                                          object
IsBeta                                                int64
RtpStateBitfield                                      float64
IsSxsPassiveMode                                      int64
DefaultBrowsersIdentifier                             float64
AVProductStatesIdentifier                             float64
AVProductsInstalled                                   float64
AVProductsEnabled                                     float64
HasTpm                                                int64
CountryIdentifier                                     int64
CityIdentifier                                        float64
OrganizationIdentifier                                float64
GeoNameIdentifier                                     float64
LocaleEnglishNameIdentifier                           int64
Platform                                              object
Processor                                             object
OsVer                                                 object
```

Columns contained four different data type out of which around 30 were categorical.
We verified the balance of positive and negative cases in our training data and it turned out to be a balanced dataset.

Distribution of Data

```python
positives = (traindata["HasDetections"]==1).sum()
negatives = (traindata["HasDetections"]==0).sum()
print("Positive detections: ", positives)
print("Negative detections: ", negatives)

labelsForPie1 = ['Positive','Negative']
sizesForPie1 = [positives,negatives]
fig1, ax1 = plt.subplots(figsize=(7,7))
ax1.pie(sizesForPie1, labels=labelsForPie1, autopct='%1.2f%%',shadow=True, startangle=90)
#To add percentages to each of the constitutents of the pie chart, we add in the line, autopct
#1.2f for getting percentages upto hundredths place
ax1.axis('equal')  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.show()
```



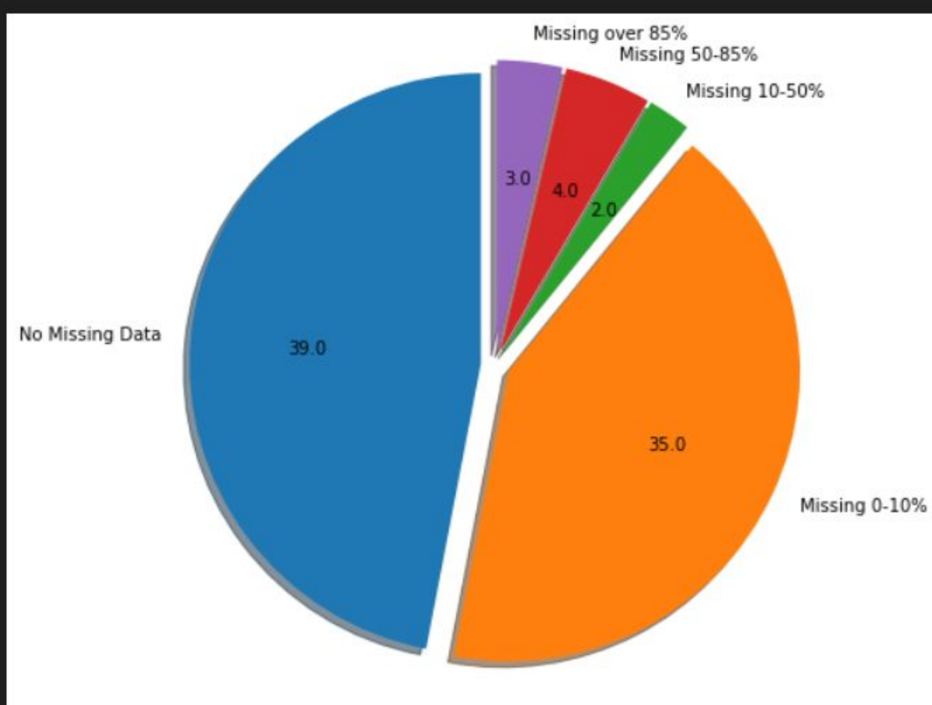
Missing Value Problem

We first listed the missing percentage value for each column. Then we made a pie chart to classify the number of columns based on their missing value percentage. Columns which had missing value more than 85% has been dropped and the rest of them we filled using the following statistic:

1. Zero or Unknown for Categorical data with high Missing Values
2. Mean for Continuous Ordinal Data with significant Missing Values
3. Mode for Categorical Data with few Missing Values

	Attribute Name	Missing Count	Missing Percent
10	PuaMode	1999470	99.97350
21	Census_ProcessorClass	1991850	99.59250
1	DefaultBrowsersIdentifier	1902812	95.14060
34	Census_IsFlightingInternal	1660397	83.01985
31	Census_InternalBatteryType	1420282	71.01410
36	Census_ThresholdOptIn	1269767	63.48835
39	Census_IsWIMBootEnabled	1268044	63.40220
13	SmartScreen	712378	35.61890
6	OrganizationIdentifier	617581	30.87905
11	SMode	120108	6.00540
5	CityIdentifier	73095	3.65475
43	Wdft_RegionIdentifier	67887	3.39435
42	Wdft_IsGamer	67887	3.39435
32	Census_InternalBatteryNumberOfCharges	60183	3.00915

```
plt.show()
```



Finding the droppable attributes

```python
print("The Attributes with more than 85% missing values are:\n\n")
miss85 = missingInfo[missingInfo['Missing Percent']>85]

miss85
```

The Attributes with more than 85% missing values are:

	Attribute Name	Missing Count	Missing Percent
8	DefaultBrowsersIdentifier	1902812	95.1406
28	PuaMode	1999470	99.9735
41	Census_ProcessorClass	1991850	99.5925

Finding the list of good columns, which have less than 15% of their values missing

Feature Engineering

There were few columns which showed versions of different hardware items so we split them into multiple columns i.e.

OS_Version : 1.10.345.18933 has been splitted as:

OS_Version1 : 1

OS_Version2 : 10

OS_Version : 345

OS_Version : 18933

Census_OSBranch1	Census_OSVersion1	Census_OSVersion2	Census_OSVersion3	Census_OSVersion4	OsBuildLab1	OsBuildLab2	OsBuildLab3	OsBuildLab4	OsBuildLab5_1	OsBuildLab5_2	OsVer1	OsVer2	OsVer3	OsVer4
rs4	10	0	17134	165	17134	1	amd64fre	rs4_release	180410	1804	10	0	0	0
rs4	10	0	17134	1	17134	1	amd64fre	rs4_release	180410	1804	10	0	0	0
rs4	10	0	17134	165	17134	1	amd64fre	rs4_release	180410	1804	10	0	0	0
rs4	10	0	17134	228	17134	1	amd64fre	rs4_release	180410	1804	10	0	0	0
rs4	10	0	17134	191	17134	1	amd64fre	rs4_release	180410	1804	10	0	0	0

After this, we assigned label encoding for each Categorical Data.

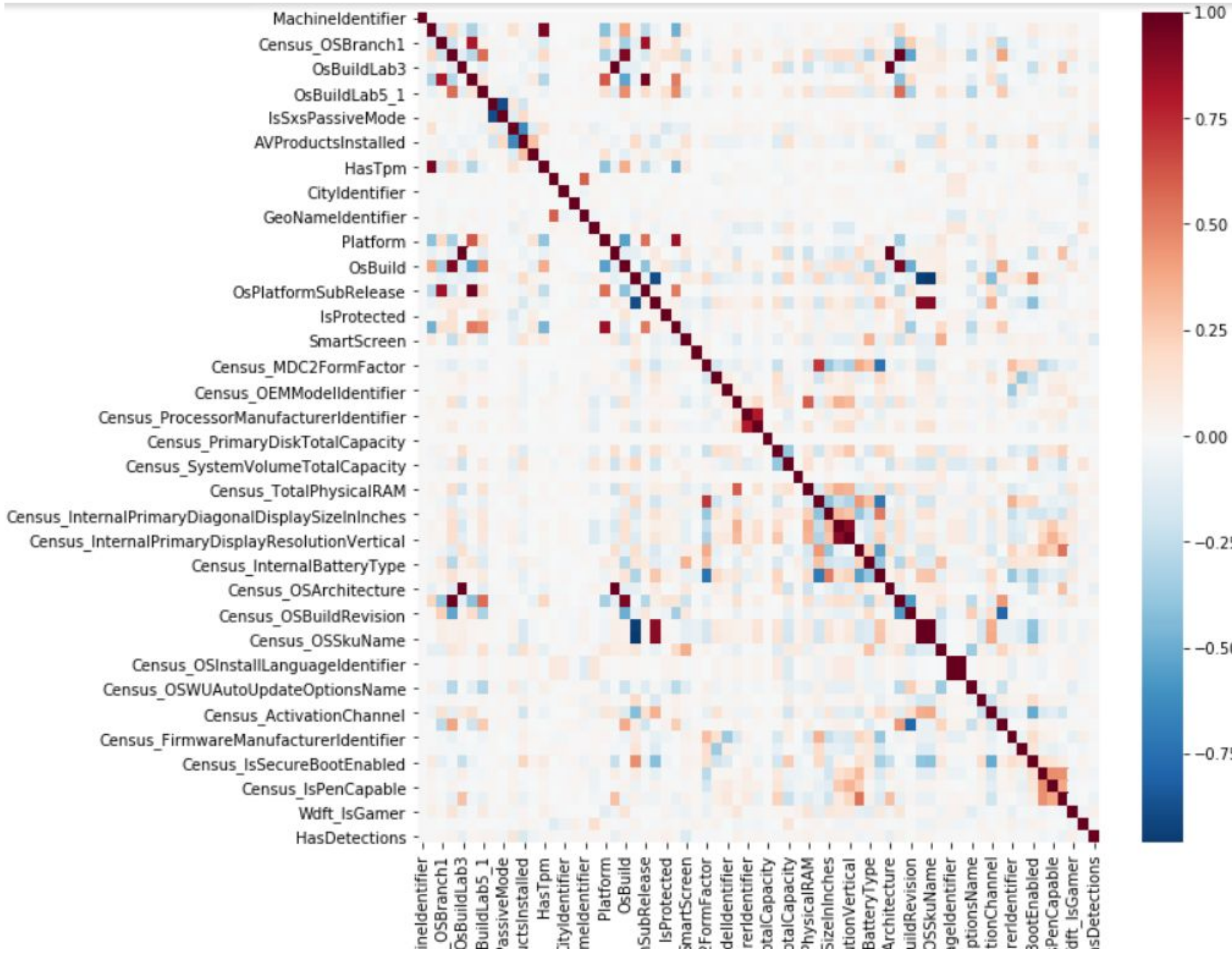
Census_OSBranch1	Census_OSVersion3	Census_OSVersion4	OsBuildLab1	OsBuildLab2	OsBuildLab3	OsBuildLab4	OsBuildLab5_1	OsBuildLab5_2	OsVer1	OsVer2	OsVer3	SigVersion2	SigVersion3	AppVersion2	AppVersion3
4	50	165	17134	1	0	13	175	1804	10	0	0	273	1735	18	1807
4	50	1	17134	1	0	13	175	1804	10	0	0	263	48	13	17134
4	50	165	17134	1	0	13	175	1804	10	0	0	273	1341	18	1807
4	50	228	17134	1	0	13	175	1804	10	0	0	273	1527	18	1807
4	50	191	17134	1	0	13	175	1804	10	0	0	273	1379	18	1807

Dimensionality Reduction based on Skewness and Correlation

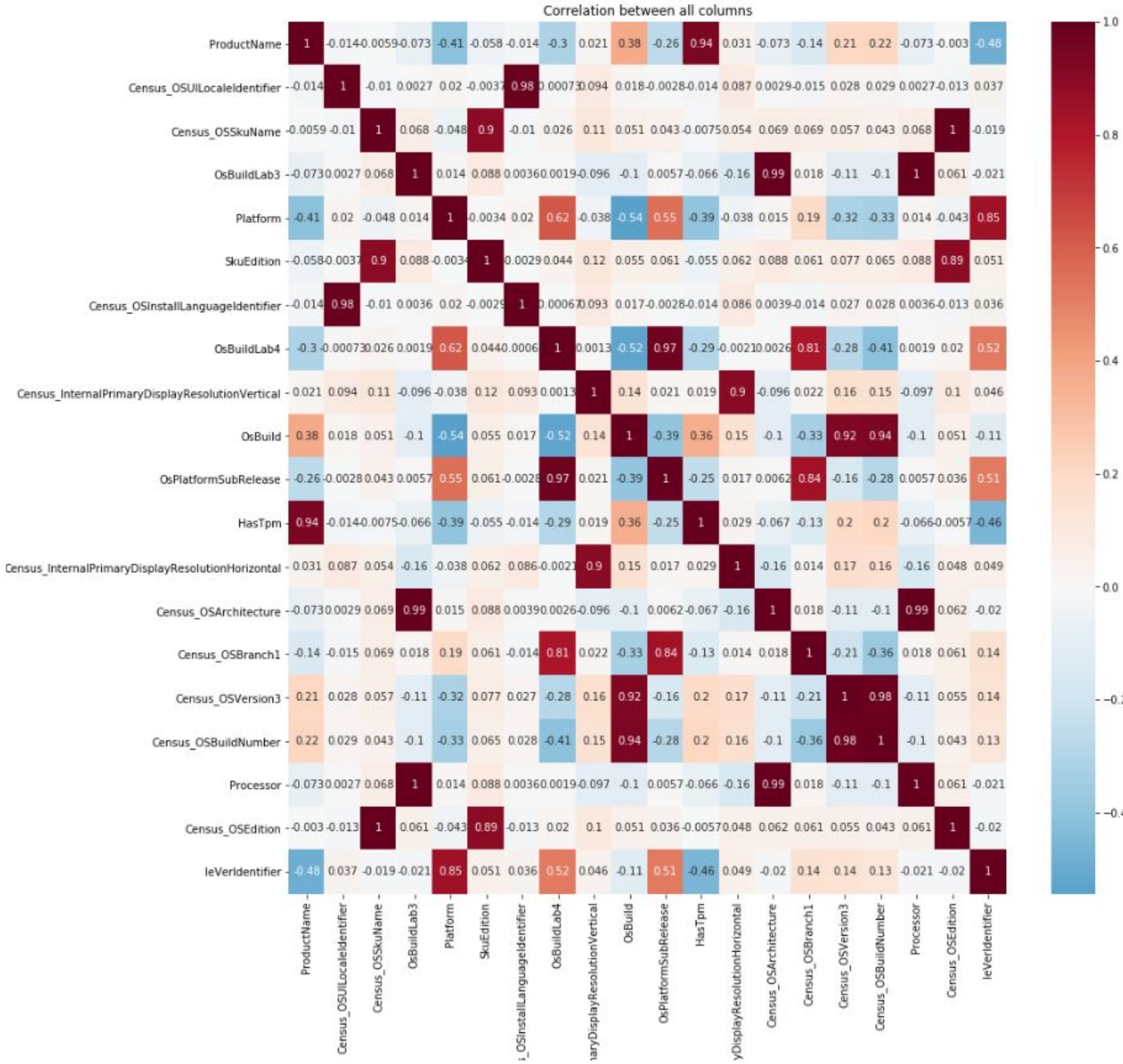
We first tabulated the skewness of each column and removed column with very high skewness (> 99.9 %)

	index	Attribute Name	# of Unique values	% in the biggest category	type
0	13	OsVer3	13	98.94950	object
1	1	ProductName	6	98.93855	int64
2	27	HasTpm	2	98.79585	int64
3	23	IsSxsPassiveMode	2	98.26990	int64
4	42	Firewall	2	97.87905	float64
5	26	AVProductsEnabled	6	97.39405	float64
6	22	RtpStateBitfield	6	97.33365	float64
7	11	OsVer1	2	96.76295	object
8	12	OsVer2	3	96.76295	object
9	33	Platform	4	96.60025	int64
10	77	Census_IsPenCapable	2	96.20510	int64
11	39	IsProtected	2	94.58000	float64
12	78	Census_IsAlwaysOnAlwaysConnectedCapable	2	94.31290	float64
13	72	Census_FlightRing	8	93.67500	int64

Then we plotted the Correlation heat map among all left columns.



We first listed some columns which had a correlation value more than 0.8 and then based on their correlation with class labels, we discarded one of the columns from each pair.



After all these refinements our data was ready to be trained by different models.
Finally, data contained 69 columns.

Application of Machine Learning Algorithms on the refined dataset:
We applied the following ML algorithms to our dataset:

1. Naive Bayes Classifier
2. Artificial Neural network
3. Decision Trees (with pruning)
4. Random Forest
5. Bagging
6. KMeans
7. LightGBM

Naive Bayes Classifier

We applied the Gaussian Naive Bayes and Bernoulli Naive Bayes algorithms on our dataset.

```
from sklearn.naive_bayes import GaussianNB
model = GaussianNB()
model.fit(X_train, X_label)
model.score(X_test, X_test_label)
```

```
0.524392
```

```
from sklearn.naive_bayes import BernoulliNB
model = BernoulliNB()
model.fit(X_train, X_label)
model.score(X_test, X_test_label)
```

```
0.542764
```

As we can see, both the algorithms couldn’t give good accuracies on the dataset. This shows that the “naive” assumption of conditional independence between every pair of features given the value of the class variable does not hold in our dataset, therefore the model did not perform nicely on our dataset.

Artificial Neural Network

We applied 3 different architectures of Neural Network on our dataset:

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 32)                2208
_________________________________________________________________
batch_normalization_1 (Batch (None, 32)                128
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 33
=================================================================
Total params: 2,369
Trainable params: 2,305
Non-trainable params: 64
_________________________________________________________________
```

```
1000000/1000000 [==============================] - 15s 15us/step - loss: 0.6915 - acc: 0.5288 - val_loss: 0.6928 - val_acc: 0.5146
Epoch 9/20
1000000/1000000 [==============================] - 15s 15us/step - loss: 0.6913 - acc: 0.5306 - val_loss: 0.6927 - val_acc: 0.5222
Epoch 10/20
1000000/1000000 [==============================] - 15s 15us/step - loss: 0.6911 - acc: 0.5323 - val_loss: 0.7031 - val_acc: 0.5120
Epoch 11/20
1000000/1000000 [==============================] - 16s 16us/step - loss: 0.6909 - acc: 0.5337 - val_loss: 0.6927 - val_acc: 0.5069
Epoch 12/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6907 - acc: 0.5357 - val_loss: 0.6963 - val_acc: 0.5156
Epoch 13/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6905 - acc: 0.5370 - val_loss: 0.6932 - val_acc: 0.5123
Epoch 14/20
1000000/1000000 [==============================] - 15s 15us/step - loss: 0.6902 - acc: 0.5391 - val_loss: 0.7096 - val_acc: 0.5015
Epoch 15/20
1000000/1000000 [==============================] - 15s 15us/step - loss: 0.6900 - acc: 0.5403 - val_loss: 0.6953 - val_acc: 0.5306
Epoch 16/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6897 - acc: 0.5410 - val_loss: 0.6984 - val_acc: 0.5045
Epoch 17/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6894 - acc: 0.5433 - val_loss: 0.6976 - val_acc: 0.5057
Epoch 18/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6891 - acc: 0.5442 - val_loss: 0.6934 - val_acc: 0.5504
Epoch 19/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6888 - acc: 0.5455 - val_loss: 0.6987 - val_acc: 0.5202
Epoch 20/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6885 - acc: 0.5469 - val_loss: 0.7089 - val_acc: 0.5006
```

```
model.summary()

Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 32)                2208
_________________________________________________________________
batch_normalization_1 (Batch (None, 32)                128
_________________________________________________________________
dense_2 (Dense)              (None, 16)                528
_________________________________________________________________
batch_normalization_2 (Batch (None, 16)                64
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 17
=================================================================
Total params: 2,945
Trainable params: 2,849
Non-trainable params: 96
_________________________________________________________________
```

```
Epoch 9/20
1000000/1000000 [==============================] - 19s 19us/step - loss: 0.6863 - acc: 0.5618 - val_loss: 0.7017 - val_acc: 0.5001
Epoch 10/20
1000000/1000000 [==============================] - 19s 19us/step - loss: 0.6896 - acc: 0.5564 - val_loss: 0.6849 - val_acc: 0.5451
Epoch 11/20
1000000/1000000 [==============================] - 19s 19us/step - loss: 0.6870 - acc: 0.5583 - val_loss: 1.7080 - val_acc: 0.5000
Epoch 12/20
1000000/1000000 [==============================] - 19s 19us/step - loss: 0.6873 - acc: 0.5645 - val_loss: 2.9252 - val_acc: 0.5000
Epoch 13/20
1000000/1000000 [==============================] - 19s 19us/step - loss: 0.6855 - acc: 0.5648 - val_loss: 1.6347 - val_acc: 0.5000
Epoch 14/20
1000000/1000000 [==============================] - 19s 19us/step - loss: 0.6847 - acc: 0.5656 - val_loss: 1.4892 - val_acc: 0.5079
Epoch 15/20
1000000/1000000 [==============================] - 19s 19us/step - loss: 0.6840 - acc: 0.5654 - val_loss: 0.9120 - val_acc: 0.4999
Epoch 16/20
1000000/1000000 [==============================] - 19s 19us/step - loss: 0.6826 - acc: 0.5668 - val_loss: 1.7324 - val_acc: 0.5079
Epoch 17/20
1000000/1000000 [==============================] - 19s 19us/step - loss: 0.6823 - acc: 0.5668 - val_loss: 1.4363 - val_acc: 0.5093
Epoch 18/20
1000000/1000000 [==============================] - 19s 19us/step - loss: 0.6826 - acc: 0.5664 - val_loss: 1.0600 - val_acc: 0.5275
Epoch 19/20
1000000/1000000 [==============================] - 19s 19us/step - loss: 0.6818 - acc: 0.5685 - val_loss: 1.2423 - val_acc: 0.5014
Epoch 20/20
1000000/1000000 [==============================] - 19s 19us/step - loss: 0.6812 - acc: 0.5696 - val_loss: 1.2713 - val_acc: 0.5089
```

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 16)                1104
_________________________________________________________________
batch_normalization_1 (Batch (None, 16)                64
_________________________________________________________________
dense_2 (Dense)              (None, 32)                544
_________________________________________________________________
batch_normalization_2 (Batch (None, 32)                128
_________________________________________________________________
dense_3 (Dense)              (None, 8)                 264
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 9
=================================================================
Total params: 2,113
Trainable params: 2,017
Non-trainable params: 96
_________________________________________________________________
```

```
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6865 - acc: 0.5514 - val_loss: 0.6871 - val_acc: 0.5395
Epoch 9/20
1000000/1000000 [==============================] - 15s 15us/step - loss: 0.6857 - acc: 0.5514 - val_loss: 0.7135 - val_acc: 0.5049
Epoch 10/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6845 - acc: 0.5565 - val_loss: 0.7021 - val_acc: 0.5218
Epoch 11/20
1000000/1000000 [==============================] - 15s 15us/step - loss: 0.6840 - acc: 0.5587 - val_loss: 0.7112 - val_acc: 0.4994
Epoch 12/20
1000000/1000000 [==============================] - 15s 15us/step - loss: 0.6829 - acc: 0.5614 - val_loss: 0.6953 - val_acc: 0.5268
Epoch 13/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6829 - acc: 0.5604 - val_loss: 0.7861 - val_acc: 0.5004
Epoch 14/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6828 - acc: 0.5585 - val_loss: 0.7361 - val_acc: 0.5028
Epoch 15/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6825 - acc: 0.5603 - val_loss: 0.7225 - val_acc: 0.4921
Epoch 16/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6819 - acc: 0.5623 - val_loss: 0.7305 - val_acc: 0.5032
Epoch 17/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6816 - acc: 0.5635 - val_loss: 0.7357 - val_acc: 0.5000
Epoch 18/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6818 - acc: 0.5631 - val_loss: 0.7049 - val_acc: 0.4812
Epoch 19/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6803 - acc: 0.5666 - val_loss: 0.7599 - val_acc: 0.5000
Epoch 20/20
1000000/1000000 [==============================] - 14s 14us/step - loss: 0.6821 - acc: 0.5580 - val_loss: 0.6982 - val_acc: 0.5202
```

As we can see, for any architecture, we do not get any good accuracy, because 50% for a binary classification has no significance. We had normalized the data and even added batch normalization layers in between the data, still, it showed no significant improvement.

Decision Trees

When applied the Decision tree classifier with the default parameters, just varying the performance parameter between Gini Index and Entropy, we found that the accuracy was significantly higher than in the earlier algorithms: (Here we have printed the mean error, accuracy is 1 - error)

```
from sklearn import tree
clf = tree.DecisionTreeClassifier()
clf = clf.fit(train_features, train_labels)
# preds = clf.predict(test_features)
predictions = clf.predict(test_features)
# # Calculate the absolute errors
errors = abs(predictions - test_labels)
print(errors.sum()/len(test_labels))
```

```
0.428259
```

```
clf = tree.DecisionTreeClassifier(criterion='entropy')
clf = clf.fit(train_features, train_labels)
predictions = clf.predict(test_features)
# # Calculate the absolute errors
errors = abs(predictions - test_labels)
print(errors.sum()/len(test_labels))
```

```
0.427454
```

So we figured out that the decision tree classifier works for our dataset. Now we began refining the decision trees. We took 2 methods to prune the decision tree: **min_samples_split** and **min_samples_leaf**.

min_samples_split specifies the minimum number of samples required to split an internal node, while **min_samples_leaf** specifies the minimum number of samples required to be at a leaf node.

When randomly set the values of min_samples_split and min_samples_leaf as 4000 and 1000, we find that the error decreased significantly:

```
clf = tree.DecisionTreeClassifier(criterion='entropy', min_samples_split=4000, min_samples_leaf = 1000)
clf = clf.fit(train_features, train_labels)
predictions = clf.predict(test_features)
# # Calculate the absolute errors
errors = abs(predictions - test_labels)
print(errors.sum()/len(test_labels))
```

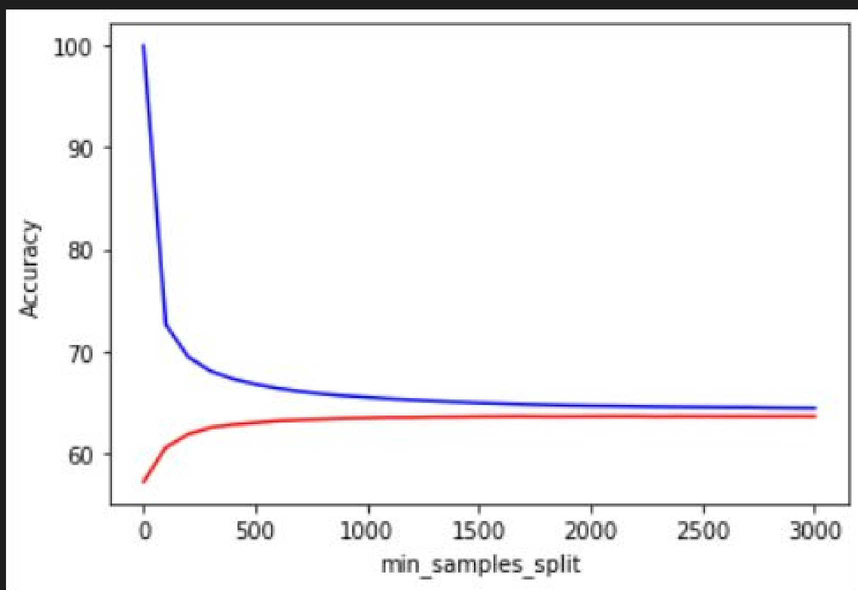
```
0.365574
```

After this, we ran for loops to find the best value for the two parameters:

```
i=2
dic = {}
while i < 2000:
  start = time.time()
  print("Started for i = ",i)
  clf = tree.DecisionTreeClassifier( min_samples_split=i)
  clf = clf.fit(train_features, train_labels)
  print("Train done")
  predictions = clf.predict(train_features)
  # # Calculate the absolute errors
  errors = abs(predictions - train_labels)
```

We found the following variation of train and test accuracy with min_samples_split(blue line for train accuracy, a red line for test accuracy):

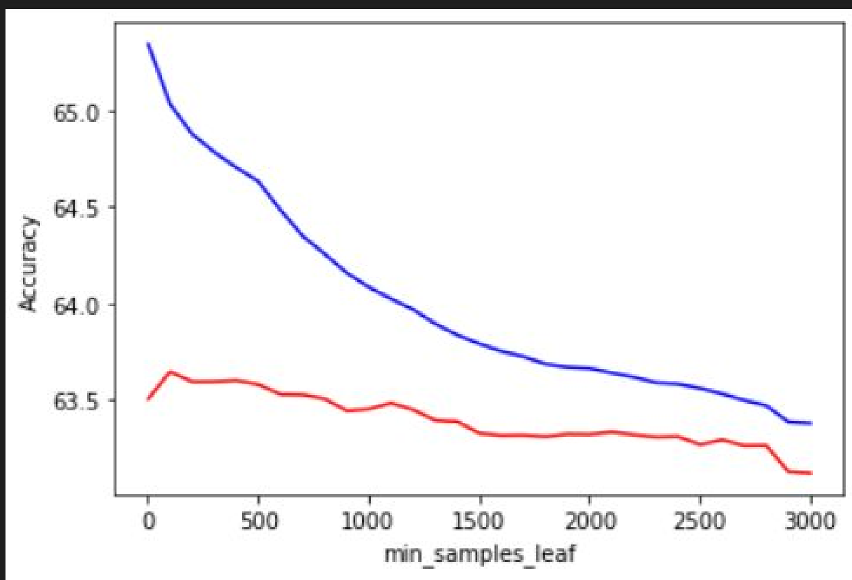
```
plt.plot(x_train_acc, y_train_acc, 'b-', label="Train Accuracy")
plt.plot(x_train_acc, y_test_acc, 'r-', label="Test Accuracy")
plt.xlabel('min_samples_split')
plt.ylabel('Accuracy')
plt.show()
```



We figured out that the best value for min_samples_split will be around 1000.

We did the same procedure for min_samples_leaf and found this distribution:

```
plt.plot(x_train_acc, train_acc_leaf, 'b-', label="Train Accuracy")
plt.plot(x_train_acc, test_acc_leaf, 'r-', label="Test Accuracy")
plt.xlabel('min_samples_leaf')
plt.ylabel('Accuracy')
plt.show()
```



We found that the best value for min_samples_leaf is around 100, so the Best parameters are min_samples_leaf = 100, min_samples_split = 1100 and entropy as the criterion for split. We achieved the highest test accuracy as 63.643%.

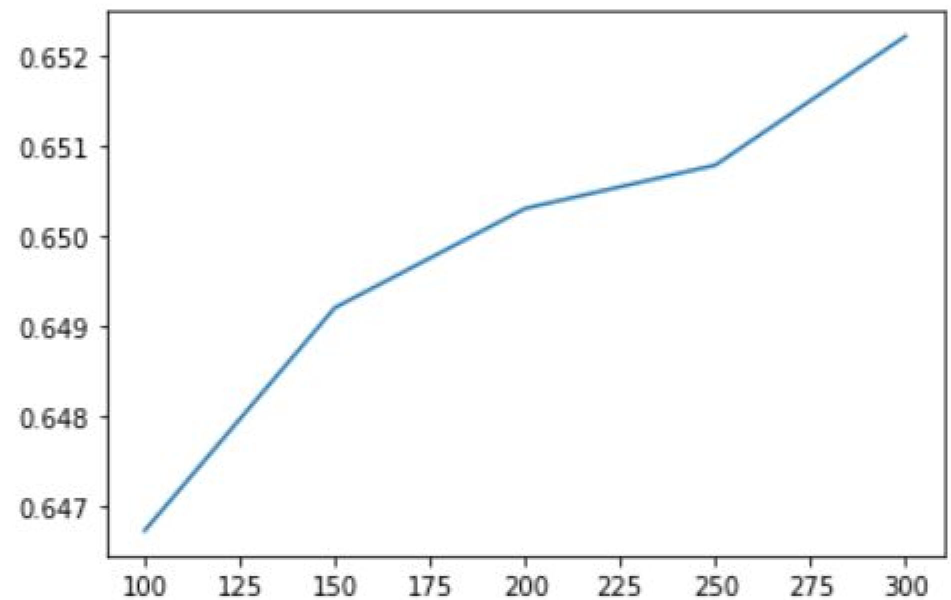
Random Forest Classifier

We applied the Random Forest Classifier on our dataset, with default parameters and number of estimators, i.e., number of decision trees ranging from 100 to 300. Below is the code snippet.

```
from sklearn.ensemble import RandomForestClassifier
for i in range(100, 300, 50):
    model = RandomForestClassifier(n_estimators = i)
    model.fit(X_train, X_label)
    accu = model.score(X_test, X_test_label)
    print("n_estimators ->", i, "Accuracy ->", accu)
    accuracy.append((i, accu))
```

The accuracy that we got for different estimators is as follows:

```
n_estimators -> 100 Accuracy -> 0.646726
n_estimators -> 150 Accuracy -> 0.649198
n_estimators -> 200 Accuracy -> 0.650298
n_estimators -> 250 Accuracy -> 0.650778
n_estimators -> 300 Accuracy -> 0.652208
```



As we can see the Accuracy of the Random Forest Classifier is greater than that of the Decision Tree Classifier. The maximum accuracy we got from random Forest Classifier is 0.652208

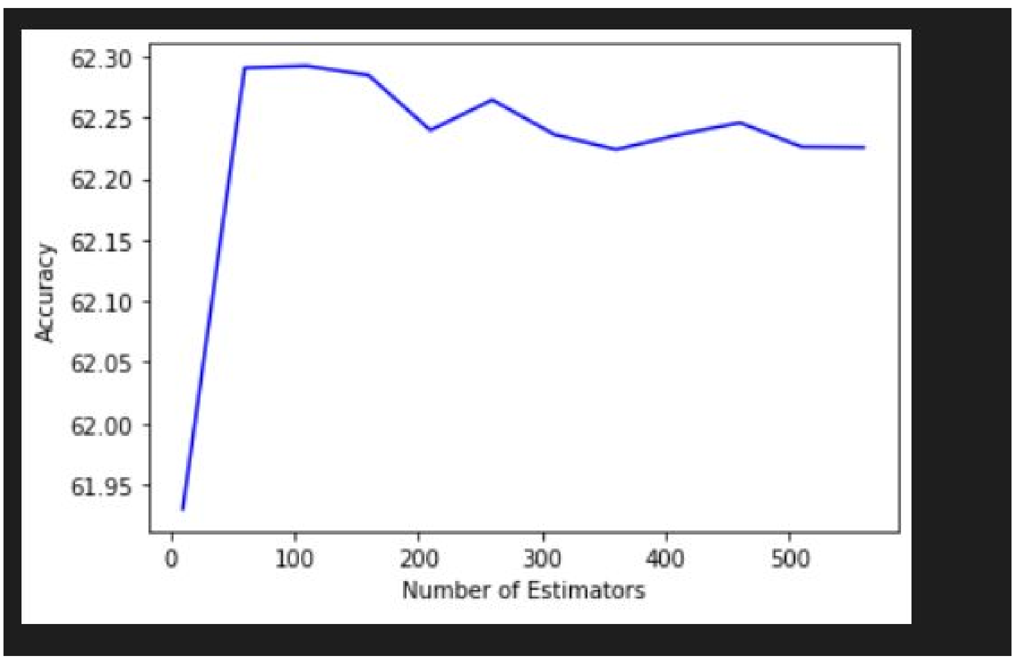
Bagging

A Bagging classifier is an ensemble meta-estimator that fits base classifiers each on random subsets of the original dataset and then aggregate their individual predictions (either by voting or by averaging) to form a final prediction. Such a meta-estimator can typically be used as a way to reduce the variance of a black-box estimator (e.g., a decision tree), by introducing randomization into its construction procedure and then making an ensemble out of it. Since we had achieved good accuracies using Decision trees, we applied bagging on Decision tree itself:

```
dt = tree.DecisionTreeClassifier(criterion='entropy', min_samples_split=1100, min_samples_leaf=100)
for i in range(10,1000,50):
  print("i = ",i)
  start = time.time()
  clf_dt = BaggingClassifier(base_estimator=dt,n_estimators=i, random_state=0,verbose=0).fit(train_features, train_labels)
  clf_dt_pred = clf_dt.predict(test_features)
  errors = abs(clf_dt_pred - test_labels)
  accuracy=100*errors.sum()/len(test_labels)
  accuracy = 100 - accuracy
  acc.append(accuracy)
  print("Accuracy of DecisionTrees bagging is ",accuracy)
  end=time.time()
  print("Time taken is: ",end-start,"\n")
```

Here, n_estimators is the number of independent decision trees made, who’s predictions we will aggregate to get the actual prediction.

This was the distribution of n_estimators with the accuracy achieved:



As we can see, it did not improve the accuracy of the individual Decision Tree. This shows that the Tree got trained better with more data compared to combining multiple trees trained on subsets of original data.

K Means

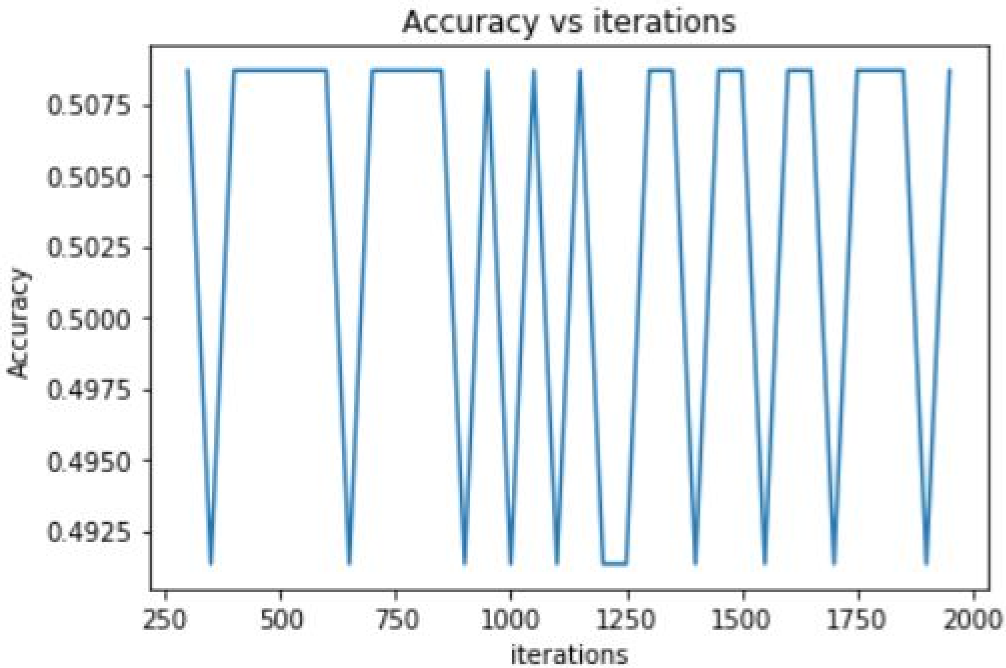
In this model, first, we removed the labels of our dataset and treated it as an unsupervised model. Then using the below K-means Algorithm we trained our model with the removed label dataset. We have classified our dataset into 2 labels by using the default parameters and keeping the number of iterations variable so that we can find that at that number of iterations we are getting our maximum accuracy.

```python
from sklearn.cluster import KMeans
for i in range(300, 2000, 50):
  kmeans = KMeans(n_clusters = 2, max_iter = i)
  kmeans.fit(X_train)
  labels = kmeans.predict(X_train)
  acuu = 1-(abs(labels-X_label).sum()/len(X_train))
  print("iterations->", i, "Accuracy->", acuu)
  accuracy.append((i, acuu))
```

Below code, snippet represents the accuracy achieved at different number of iterations

```
iterations-> 300 Accuracy-> 0.508666000000001
iterations-> 350 Accuracy-> 0.49133400000000005
iterations-> 400 Accuracy-> 0.508666000000001
iterations-> 450 Accuracy-> 0.508666000000001
iterations-> 500 Accuracy-> 0.508666000000001
iterations-> 550 Accuracy-> 0.508666000000001
iterations-> 600 Accuracy-> 0.508666000000001
iterations-> 650 Accuracy-> 0.49133400000000005
iterations-> 700 Accuracy-> 0.508666000000001
iterations-> 750 Accuracy-> 0.508666000000001
iterations-> 800 Accuracy-> 0.508666000000001
iterations-> 850 Accuracy-> 0.508666000000001
iterations-> 900 Accuracy-> 0.49133400000000005
iterations-> 950 Accuracy-> 0.508666000000001
iterations-> 1000 Accuracy-> 0.49133400000000005
iterations-> 1050 Accuracy-> 0.508666000000001
iterations-> 1100 Accuracy-> 0.49133400000000005
iterations-> 1150 Accuracy-> 0.508666000000001
iterations-> 1200 Accuracy-> 0.49133400000000005
iterations-> 1250 Accuracy-> 0.49133400000000005
iterations-> 1300 Accuracy-> 0.508666000000001
iterations-> 1350 Accuracy-> 0.508666000000001
iterations-> 1400 Accuracy-> 0.49133400000000005
iterations-> 1450 Accuracy-> 0.508666000000001
iterations-> 1500 Accuracy-> 0.508666000000001
iterations-> 1550 Accuracy-> 0.49133400000000005
iterations-> 1600 Accuracy-> 0.508666000000001
iterations-> 1650 Accuracy-> 0.508666000000001
iterations-> 1700 Accuracy-> 0.49133400000000005
iterations-> 1750 Accuracy-> 0.508666000000001
iterations-> 1800 Accuracy-> 0.508666000000001
iterations-> 1850 Accuracy-> 0.508666000000001
iterations-> 1900 Accuracy-> 0.49133400000000005
iterations-> 1950 Accuracy-> 0.508666000000001
```

Graph of the number of iterations vs Accuracy of the model is shown below.



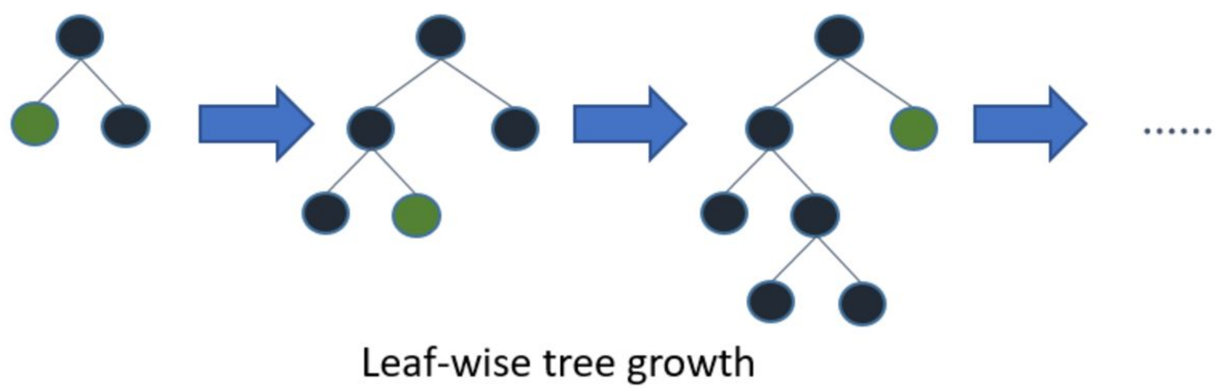
Maximum Accuracy achieved by kMeans Classifier is 51% which is very less as compared to our other models.

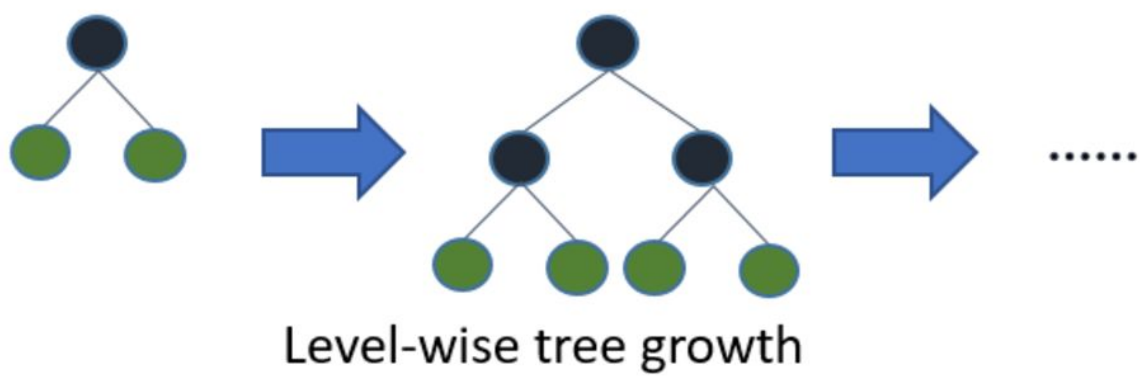
Light GBM

This is a very new algorithm, which is a gradient boosting framework that uses tree-based learning algorithms.

Light GBM grows tree vertically while other algorithms grow trees horizontally meaning that Light GBM grows tree leaf-wise while other algorithms grow level-wise. It will choose the leaf with max delta loss to grow. When growing the same leaf, Leaf-wise algorithm can reduce more loss than a level-wise algorithm.

Below diagrams explain the implementation of LightGBM and other boosting algorithms.





The LightGBM had multiple parameters to be tuned, in which the best was found to be the following:

```
params = {}
params['learning_rate'] = 0.05
params['boosting_type'] = 'gbdt'
params['objective'] = 'binary'
params['device_type '] = 'gpu'
params['metric'] = 'binary_logloss'
params['num_leaves'] = 256
params['min_data'] = 128
params['max_depth'] = 64
params['bagging_freq'] = 8
params['bagging_seed'] = 16
```

params['boosting_type'] = 'gbdt': this is the traditional Gradient Boosting Decision Tree method
params['objective'] = 'binary': our objective function is binary classification
params['device_type '] = 'gpu': We used GPUs for faster parallel growth of decision trees
params['metric'] = 'binary_logloss': binary logloss is our metric for judging trees
params['num_leaves'] = 256: this is the max number of leaves in a tree, used to control overfitting
params['min_data'] = 128: this is the min number of data required to make a split
params['max_depth'] = 64: this is the max depth possible of a tree
params['bagging_freq'] = 8: means perform bagging at every 8th iteration
params['bagging_seed'] = 16: the random seed for bagging

We performed 1024 iterations for the best model, and the best accuracy achieved was 66.456%

```
Train start
Train done, time taken to train =  247.5084035396576
Accuracy of lightgbm bagging is  66.4564
```