MALWARE DETECTION USING MACHINE LEARNING

Project Report

Libraries/modules imported:

- os
- sklearn
- boto3
- sagemaker
- numpy
- pandas
- sklearn.feature_extraction and sklearn.preprocessing
- itertools

Data Analysis, Cleaning and Preprocessing:

After importing all the necessary libraries, pandas' info() function is executed on the data, to see how many columns are there in the data, along with the number of entries in each columns. If the number of entries do not match in the columns, it implies there are Null values present.

```
In [3]:  df_Csv.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50181 entries, 0 to 50180
Data columns (total 28 columns):
BaseOfCode                 50181 non-null int64
BaseOfData                 50181 non-null int64
Characteristics            50181 non-null int64
DllCharacteristics         50181 non-null int64
Entropy                    50181 non-null float64
FileAlignment              50181 non-null int64
FirstSeenDate              50181 non-null object
Identify                   35958 non-null object
ImageBase                  50181 non-null int64
ImportedDlls               50181 non-null object
ImportedSymbols            50181 non-null object
Label                      50181 non-null int64
Machine                    50181 non-null int64
Magic                      50181 non-null int64
NumberOfRvaAndSizes        50181 non-null int64
NumberOfSections           50181 non-null int64
NumberOfSymbols            50181 non-null int64
PE_TYPE                    50181 non-null int64
PointerToSymbolTable       50181 non-null int64
SHA1                       50181 non-null object
Size                       50181 non-null int64
SizeOfCode                 50181 non-null int64
SizeOfHeaders              50181 non-null int64
SizeOfImage                50181 non-null int64
SizeOfInitializedData      50181 non-null int64
SizeOfOptionalHeader       50181 non-null int64
SizeOfUninitializedData    50181 non-null int64
TimeDateStamp              50181 non-null int64
dtypes: float64(1), int64(22), object(5)
memory usage: 10.7+ MB
```

As seen, there are 28 columns, each having a different meaning. Nearly all of them have 50181 entries, but one column: 'Identify'. It has way too less entries (35958). Since it's an Identification type, filling this column's NaNs does not make sense. If we are to drop those rows having the NaN's it would mean a loss of 50,181-35,958 = 14,223 rows, which will lead to too large a loss of relevant data. Hence, that column will be dropped. Let us now have a 'first glance' at the data. (all the columns not visible here)

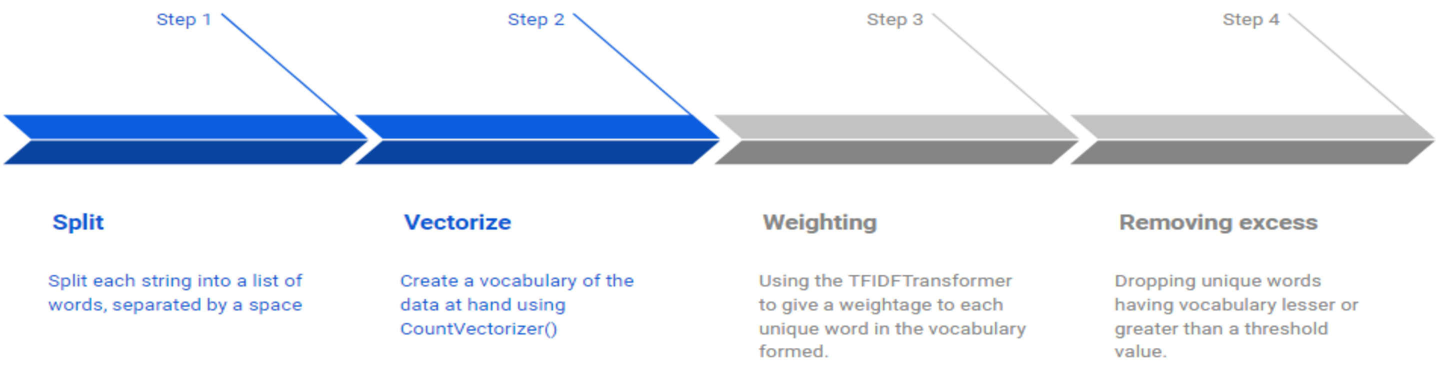
BaseOfCo	BaseOfDat	Characteri	DllCharact	Entropy	FileAlignm	FirstSeenD	Identify	ImageBase	ImportedD	ImportedS	Label	Machine	Magic	NumberOf	NumberOf	NumberOf	PE_TYPE	PointerTo	SHA1	Size	SizeOfCod	SizeOfHea	Siz
4096	69632	783	0	5.981249	512	01-01-70	powerbasi	4194304	comdlg32.	printdlga g	0	332	267	16	5	0	267	0	b0068836a	76288	64855	1024	
4096	1851392	783	0	6.081747	512	01-01-70		4194304	comctl32.d	imagelist_	0	332	267	16	6	0	267	0	5741708cc	2558464	1843888	1024	2
4096	40960	783	0	5.586422	512	01-01-70		4194304	comdlg32.	getopenfil	0	332	267	16	9	0	267	0	507fe5d82	178688	33792	1024	33
1359872	2138112	783	0	7.969464	512	01-01-70	upx 2.93 -	4194304	kernel32.d	loadlibrary	0	332	267	16	3	0	267	0	e51a7811	806816	778240	4096	2
4096	40960	783	32768	7.9999	512	01-01-70		4194304	advapi32.c	regcloseke	0	332	267	16	7	0	267	0	0e046d99	50689096	35840	1024	
192512	245760	783	0	7.328245	512	01-01-70	upx v0.89.	4194304	kernel32.d	loadlibrary	0	332	267	16	3	0	267	0	19de4645	76800	53248	4096	
8192	61440	33166	0	6.257786	512	01-01-70		4194304	kernel32.d	getstringty	0	332	267	16	5	0	267	0	f6e045232	69660	52224	1024	
4096	40960	775	0	5.308237	512	03-01-70		4194304	lua53.dll k	lual_callm	0	332	267	16	9	1574	267	75264	56e362bc3	110146	34304	1024	
4096	40960	775	0	5.256822	512	03-01-70		4194304	lua53.dll k	lual_callm	0	332	267	16	9	1542	267	73728	01a8b85f2	107962	34304	1024	
4096	131072	775	0	5.909817	512	04-01-70		4194304	kernel32.d	deletecriti	0	332	267	16	9	2226	267	194560	1bad4f17d	246187	123392	1024	
4096	262144	783	0	6.233885	512	04-01-70		4194304	kernel32.d	closehandl	0	332	267	16	7	0	267	0	8a1b8284	315904	257536	1024	
524288	921600	259	0	7.994843	512	06-01-70	upx -> ww	4194304	kernel32.d	loadlibrary	0	332	267	16	3	0	267	0	d5a4e36e	5767600	397312	4096	1
532480	937984	259	0	7.993068	512	06-01-70	upx -> ww	4194304	kernel32.d	loadlibrary	0	332	267	16	3	0	267	0	41081871	4934008	405504	4096	1
614400	1093632	259	0	7.950715	512	07-01-70	upx -> ww	4194304	kernel32.d	loadlibrary	0	332	267	16	3	0	267	0	12a5489db	3188880	479232	4096	1
614400	1093632	259	0	7.95329	512	07-01-70	upx -> ww	4194304	kernel32.d	loadlibrary	0	332	267	16	3	0	267	0	f5867990b	3309048	479232	4096	1
65536	131072	33166	0	7.956131	512	15-02-76	borland c+	4194304	kernel32.d	lstrlena lst	0	332	267	16	6	0	267	0	7bd53b19	778356	23040	1024	
4096	34975744	782	0	6.478057	512	28-05-80		4194304	advapi32.c	copysid cr	0	332	267	16	12	0	267	0	f15fa12bb	49366808	34969088	1024	49
4096	40960	783	32768	7.992968	512	16-07-84		4194304	advapi32.c	regcloseke	0	332	267	16	7	0	267	0	0ff72230a	2253360	34816	1024	
4096	45056	33167	32768	7.995115	512	19-06-92	borland de	4194304	kernel32.d	deletecriti	0	332	267	16	8	0	267	0	eaef116aa	11415344	37888	1024	
4096	45056	33167	32768	7.998887	512	19-06-92	borland de	4194304	kernel32.d	deletecriti	0	332	267	16	8	0	267	0	7cb4c3c0e	5122240	40448	1024	
4096	45056	33167	32768	7.992212	512	19-06-92	borland de	4194304	kernel32.d	deletecriti	0	332	267	16	8	0	267	0	62d99928	1704035	40448	1024	
4096	45056	33167	32768	7.999957	512	19-06-92	borland de	4194304	kernel32.d	deletecriti	0	332	267	16	8	0	267	0	587e1e47	48232863	37888	1024	
4096	45056	33167	32768	7.957969	512	19-06-92	borland de	4194304	kernel32.d	deletecriti	0	332	267	16	8	0	267	0	30a0d3b3	657424	40448	1024	
4096	45056	33167	32768	7.999077	512	19-06-92	borland de	4194304	kernel32.d	deletecriti	0	332	267	16	8	0	267	0	99ab5e41	30941112	40448	1024	
4096	45056	33167	32768	7.988488	512	19-06-92	borland de	4194304	kernel32.d	deletecriti	0	332	267	16	8	0	267	0	ac7417f36	5125196	37888	1024	
4096	45056	33167	32768	7.985504	512	19-06-92	borland de	4194304	kernel32.d	deletecriti	0	332	267	16	8	0	267	0	d9cd7474	3515824	37888	1024	
4096	45056	33167	32768	7.974753	512	19-06-92	borland de	4194304	kernel32.d	deletecriti	0	332	267	16	8	0	267	0	d8e8d4d4	856481	40448	1024	
4096	45056	33167	32768	7.999804	512	19-06-92	borland de	4194304	kernel32.d	deletecriti	0	332	267	16	8	0	267	0	d8ef34bce	14462573	40448	1024	
4096	45056	33167	32768	7.99999	512	19-06-92	borland de	4194304	kernel32.d	deletecriti	0	332	267	16	8	0	267	0	0b41190d	66382626	40448	1024	
4096	45056	33167	32768	7.957352	512	19-06-92	borland de	4194304	kernel32.d	deletecriti	0	332	267	16	8	0	267	0	573e03b1	854558	37888	1024	

Though one cannot comprehend the data at first glance, it can be seen that there are a few columns with redundant values, all along. (except 'Label' of course, that's important). Columns like Magic, File Alignment, SizeOfOptionalHeader, PE_TYPE are some. If they have only one value, they will not help our model in understanding the generality of the data better. It is better to have these columns removed.
Also, the column SHA1 has serial-like values, i.e. they are a mixture of numbers and characters, and it makes no sense to find out any pattern in that data. It is better to have this column removed as well.

```
df_Csv = df_Csv.drop(columns=['SHA1', 'FirstSeenDate', 'Identify', 'PE_TYPE', 'SizeOfOptionalHeader', 'Magic'])
```

Now that the columns which would hinder the data analytics have been removed, let's move on to processing and analysing the remaining data. Taking a look at the output of the info() function once again, it is observed that:
Though most of the remaining columns now have numerical values, columns 'ImportedDlls' and 'ImportedSymbols' having data type 'object', meaning they contain strings.
DLLs: DLL is a dynamic link library file format used for holding multiple codes and procedures for Windows programs. DLL files were created so that multiple programs could use their information at the same time, aiding memory conservation. It also allows the user to edit the coding of multiple applications at once, without changing the applications themselves. [https://whatis.techtarget.com/fileformat/DLL-Dynamic-link-library-file]

These columns hold data which will significantly contribute to the analysis of the data, as seen by the definition of DLLs. Hence it is necessary to convert the data stored here to a form which easily understood by the model. The following plan will be used for conversion for both columns:



To decide the threshold value in step 4, we use the sparsity of the data as a decision parameter as well.
Sparsity: In a database, sparsity and density describe the number of cells in a table that are empty (sparsity) and that contain information (density), though sparse cells are not always technically empty—they often contain a "0" digit. Tables and databases are the sum total of their sparse and dense cells.[https://www.quora.com/What-is-a-clear-explanation-of-data-sparsity]

Mathematically, sparsity is calculated as:

$$Sparsity(S) = \frac{number\ of\ non-zero\ entries\ in\ data}{number\ of\ rows\ of\ data * number\ of\ columns\ of\ data}$$

```
In [40]:
cvec = CountVectorizer(stop_words='english', min_df=0.001, max_df=0.8, ngram_range=(1,1), token_pattern = '[a-zA-Z0-9$&+:;=?
cvec.fit(df_Csv.ImportedDlls)
list(islice(cvec.vocabulary_.items(), 20))
len(cvec.vocabulary_)

Out[40]: 235
```

```
In [41]:
cvec_counts = cvec.transform(df_Csv.ImportedDlls)
print('sparse matrix shape:', cvec_counts.shape)
print('nonzero count:', cvec_counts.nnz)
print('sparsity: %.2f%%' % (100.0 * cvec_counts.nnz / (cvec_counts.shape[0] * cvec_counts.shape[1])))

sparse matrix shape: (50143, 235)
nonzero count: 407391
sparsity: 3.46%
```

For column 'ImportedDLLs'

In [45]:
```
cvec = CountVectorizer(stop_words='english', min_df=0.005, max_df=0.9, ngram_range=(1,1))
cvec.fit(df_Csv.ImportedSymbols)
list(islice(cvec.vocabulary_.items(), 20))
len(cvec.vocabulary_)
```
Out[45]: 1935

In [46]:
```
cvec_counts = cvec.transform(df_Csv.ImportedSymbols)
print('sparse matrix shape:', cvec_counts.shape)
print('nonzero count:', cvec_counts.nnz)
print('sparsity: %.2f%%' % (100.0 * cvec_counts.nnz / (cvec_counts.shape[0] * cvec_counts.shape[1])))
```
```
sparse matrix shape: (50143, 1935)
nonzero count: 8174793
sparsity: 8.43%
```
For column 'ImportedSymbols'

It is important to keep the sparsity of the dataset as low as possible, in order to maintain a good dataset. Keeping the sparsity and the amount of data to be maintained later in mind, we set the limits of the Vectorizer as (0.001,0.8) for Imported DLLs, and (0.05,0.9) for Imported Symbols. The arranged words in the vocabulary of each column is seen in the notebook.

Once separation of the string into words is done, we need to convert it into a form that the model understands, ie in numerical format. In order to do that,

- Create a new dataframe, with the columns as the words of the vocabulary.
- For every row entry in the original dataframe:
    - See each element of the list in the column 'ImportedDLLs'
    - If the element matches with a column in the new dataframe (<element> == <column name>), then under that column put 1 (in the new dataframe), else put a
- Repeat the same for column 'ImportedSymbols'

Thus, we would have successfully converted the data into a numerical form. Currently, we have 3 dataframes:

- df_Csv, the original dataframe
- df2, the dataframe created for 'ImportedDLLs'
- df3, the dataframe created for 'ImportedSymbols'

The three dataframes are to be combined as one as the last step of pre-processing the data for our machine learning models.

Using pandas' concat function, we can combine the three dataframes, and convert the resultant dataframe into a new csv, called 'final.csv', which has the pre-processed data, ready to be sent to the model.

Note: Due to some internal problem in AWS, the pre-processing was done locally, and the Jupyter Notebook's code was copied into this notebook, and 'final.csv' was physically uploaded to the S3 bucket and the notebook instance. Since the final part of the pre-processing takes really long, re-executing that code cell would be impractical. The file, 'final.csv' is of size 400+Mb. All further coding has been continued in the AWS notebook instance, the second notebook.

Now that we have completed our pre-processing, we can move on to the model selection, training and tuning.

Dataset Splitting:

As the data is now processed, it should be split up into its training and testing sets. A threshold of 30% was decided to separate the training and the testing set, i.e. the testing set will be 30% of the original dataset, and the training set will be the remaining, i.e. 70% of the original dataset. Of this 70%, we will again split the data into training and validation sets. We will use 20% of the training set obtained above, as the validation set. Then, we will upload the sets our S3 bucket.

Model Selection:

In the second Jupyter Notebook, 4 models were chosen for comparison:

- Decision Tree Classifier
- AdaBoost Classifier
- Random Forest Classifier
- XGBoost Classifier

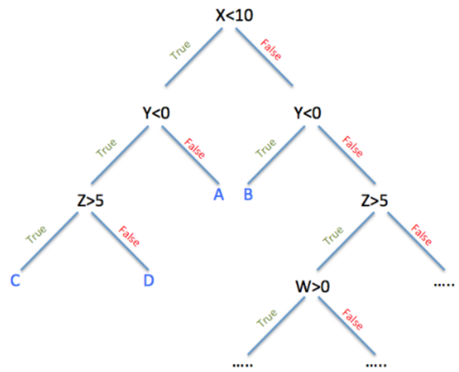
The above 4 models will be compared to one another on a part of the pre-processed data.

About the models:

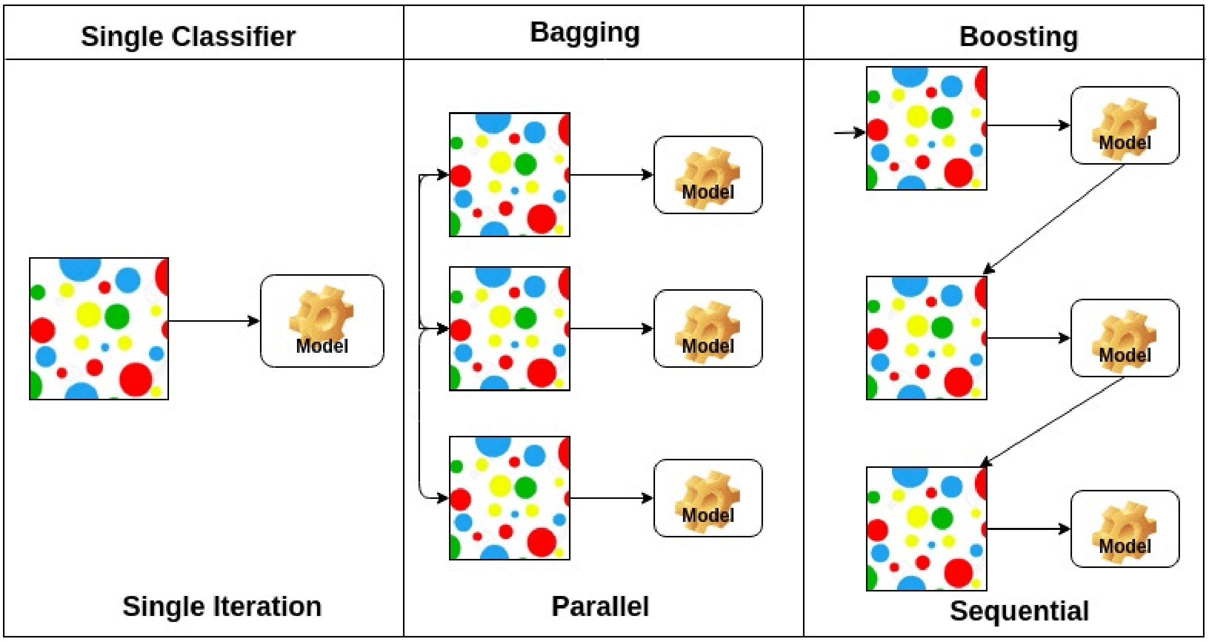
A **Decision Tree** is a simple representation for classifying examples. It is a Supervised Machine Learning where the data is continuously split according to a certain parameter.

Decision Tree consists of:

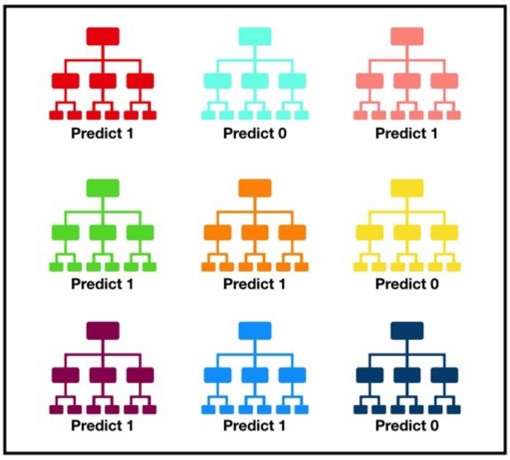
- Nodes: Test for the value of a certain attribute.
- Edges/ Branch: Correspond to the outcome of a test and connect to the next node or leaf.
- Leaf nodes: Terminal nodes that predict the outcome (represent class labels or class distribution).



An **AdaBoost Classifier** is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.



A **random forest** is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is always the same as the original input sample size but the samples are drawn with replacement if bootstrap=True



XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting (also known as GBDT, GBM) that solve many data science problems in a fast and accurate way. The same code runs on major distributed environment (Hadoop, SGE, MPI) and can solve problems beyond billions of examples

The Comparison:

The comparison could not be done on the Sagemaker Notebook, as the 'xgboost' module could not be imported for the comparison the way it was done locally. Hence, an html report of the local notebook has been uploaded along with the files, so that one may see the output of the execution done locally. A pipeline was made to train the models, one by one on 3 parts of the dataset, having 1%, 10% and 100% of the data.

Naturally, the data had been split into training and testing sets, with 30% of the data being kept as the testing data. The models were allowed to fit on the data, with certain factors being noted and stored in a dictionary named 'result'. We then compare the different values among all 4 models in the 'result', and decide which model will be further tuned and hyperparamaters will be set.

In [ ]:

```python
from sklearn.metrics import fbeta_score, accuracy_score
from time import time
def train_predict(learner, sample_size, X_train, y_train, X_test, y_test):
    '''
    inputs:
       - learner: the learning algorithm to be trained and predicted on
       - sample_size: the size of samples (number) to be drawn from training set
       - X_train: features training set
       - y_train: income training set
       - X_test: features testing set
       - y_test: income testing set
    '''

    results = {}

    start = time() # Get start time
    learner.fit(X_train[:sample_size], y_train[:sample_size])
    end = time() # Get end time
    results['train_time'] = end-start
    start = time() # Get start time
    predictions_test = learner.predict(X_test)
    predictions_train = learner.predict(X_train[:300])
    end = time() # Get end time
    results['pred_time'] = end-start
    results['acc_train'] = accuracy_score(y_train[:300],predictions_train)
    results['acc_test'] = accuracy_score(y_test, predictions_test)
    results['f_train'] = fbeta_score(y_train[:300],predictions_train,beta=0.01)
    results['f_test'] = fbeta_score(y_test, predictions_test,beta=0.01)
    return results
```

For each model, the following factors have been taken into consideration:

- Prediction time: The amount of time taken by the model to generate an output, measured from the moment the model was created.
- Accuracy on training set: Shows how well the model has learnt from the training set
- Accuracy on testing set: Shows how well the model can apply its knowledge to the testing set.
- F-score on training: Shows the precision of the model on the training set
- F-score on testing: Shows the precision of the model on the testing set

It should be noted, that the accuracy of the training set and the f-score on the training being more than their corresponding scores on the testing set indicate overfitting of the model. The model should ideally perform better on the testing set, which means that it has got the patterns of the data, rather than learning the data itself.

The following is the result of the comparison. Cases 1,2,3 are on 1%, 10% and 100% of the data respectively:

```
----------------------------case 1------
DecisionTreeClassifier
Training time: 0.2520310878753662
Prediction time: 0.21492242813110352
Accuracy Score: train 1.0
Accuracy Score: test 0.8974853645556147
Fscore : train 1.0
Fscore : test 0.915014269776282
AdaBoostClassifier
Training time: 0.6650574207305908
Prediction time: 8.689386367797852
Accuracy Score: train 0.9833333333333333
Accuracy Score: test 0.8929616817456094
Fscore : train 0.98224910186443
Fscore : test 0.8984199975638543
RandomForestClassifier
Training time: 0.11994600296020508
Prediction time: 0.17587995529174805
Accuracy Score: train 0.9933333333333333
Accuracy Score: test 0.9098589675359233
Fscore : train 0.9940476190476191
Fscore : test 0.9243721074950858
XGBClassifier
Training time: 5.440302848815918
Prediction time: 2.8927595615386963
Accuracy Score: train 0.9966666666666667
Accuracy Score: test 0.9348722724853645
Fscore : train 0.994083428392917
Fscore : test 0.9389723835449842
```

```
----------------------------case 2--------
DecisionTreeClassifier
Training time: 0.27184438705444336
Prediction time: 0.12193489074707031
Accuracy Score: train 1.0
Accuracy Score: test 0.9439861628525812
Fscore : train 1.0
Fscore : test 0.9496265932613658
AdaBoostClassifier
Training time: 1.8499319553375244
Prediction time: 1.6690373420715332
Accuracy Score: train 0.94
Accuracy Score: test 0.9337413517828632
Fscore : train 0.9518060822760106
Fscore : test 0.9441414971664508
RandomForestClassifier
Training time: 0.19388937950134277
Prediction time: 0.1739184856414795
Accuracy Score: train 1.0
Accuracy Score: test 0.939196381053752
Fscore : train 1.0
Fscore : test 0.9521408953858418
XGBClassifier
Training time: 34.03155255317688
Prediction time: 2.4855659008026123
Accuracy Score: train 0.9566666666666667
Accuracy Score: test 0.9589542309739223
Fscore : train 0.9640712790569571
Fscore : test 0.9659403613360804
```

```
----------------------------case 3------
DecisionTreeClassifier
Training time: 7.098905086517334
Prediction time: 0.15091276168823242
Accuracy Score: train 1.0
Accuracy Score: test 0.9753858435337945
Fscore : train 1.0
Fscore : test 0.9787504104711272
AdaBoostClassifier
Training time: 28.6019926071167
Prediction time: 1.6130716800689697
Accuracy Score: train 0.92
Accuracy Score: test 0.9456492815327302
Fscore : train 0.933733814891023
Fscore : test 0.9572152081963373
RandomForestClassifier
Training time: 2.7414186000823975
Prediction time: 0.17987632751464844
Accuracy Score: train 1.0
Accuracy Score: test 0.9700638637573177
Fscore : train 1.0
Fscore : test 0.975558585510146
XGBClassifier
Training time: 359.3747401237488
Prediction time: 2.603517770767212
Accuracy Score: train 0.9533333333333334
Accuracy Score: test 0.966138903672166
Fscore : train 0.9529422974670738
Fscore : test 0.9705610825498041
```

Observing the outputs obtained above, we can see that the Random Forest Classifier would prove a really good fit, as suggested in papers. However, currently it's a little overfitting. The XGBClassifier, on the other hand, takes significantly more time in training, but does a better job than the Random Forest Classifier at preventing overfitting. Observing the Adaboost Classifier, we may see that it also does not overfit. However, it gets beat by the XGBoost Classifier at all the scores. The Decision Tree Classifier heavily overfits.
Thus, we will use the XGBoost classifier as our model to train and tune.

Since we are using Sagemaker, we will need to define the model in a different style than earlier. We will need to use Sagemaker's estimator function to define the model, setting relevant parameters.

First, let us split the data into training, testing and validation sets, and save them in 3 different csv's, 'train.csv', 'test.csv', 'validation.csv'

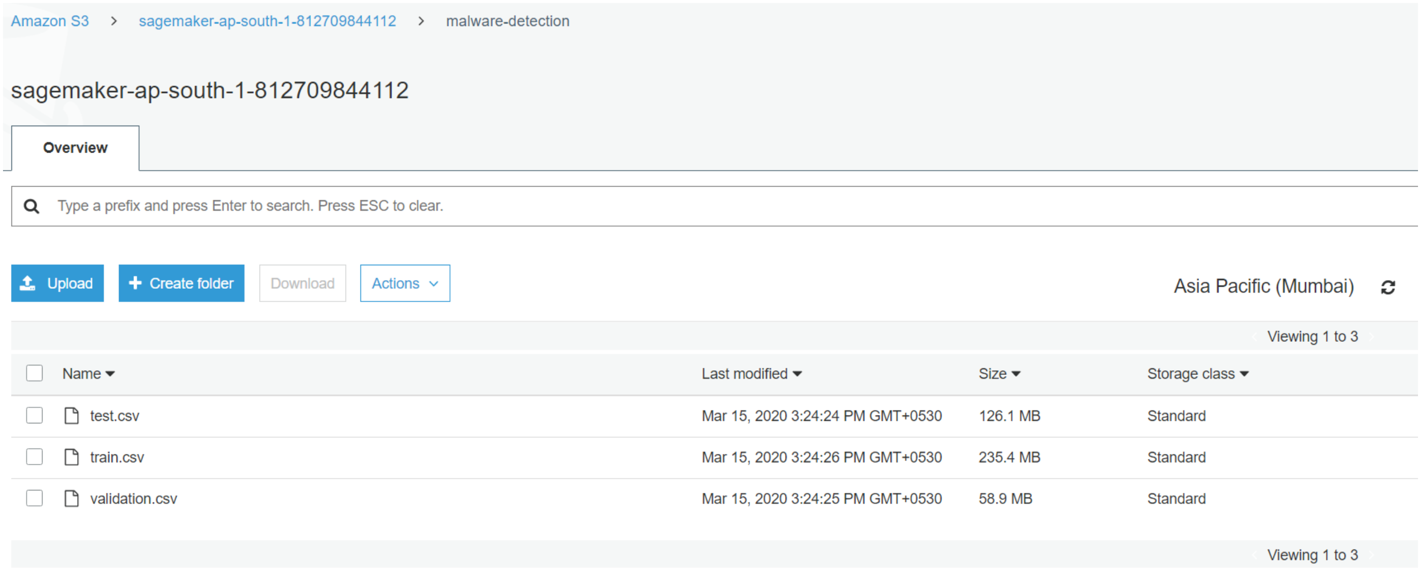
In [5]:
```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X, y, test_size=0.3, random_state=42)

print("Testing set has {} samples.".format(X_test.shape[0]))
X_train,X_val,y_train,y_val = train_test_split(X_train, y_train, test_size=0.2, random_state=42)
print("Training set has {} samples.".format(X_train.shape[0]))
print("Validation set has {} samples.".format(X_val.shape[0]))
```

```
Testing set has 15032 samples.
Training set has 28058 samples.
Validation set has 7015 samples.
```

In [6]:
```
X_test.to_csv('test.csv',header = False, index = False)
pd.concat([y_val,X_val],axis =1).to_csv('validation.csv',header=False, index=False)
pd.concat([y_train,X_train],axis =1).to_csv('train.csv',header = False, index = False)
```

In [8]:
```
prefix = 'malware-detection'
test_location = session.upload_data('test.csv',key_prefix = prefix)
val_location = session.upload_data('validation.csv',key_prefix = prefix)
train_location = session.upload_data('train.csv',key_prefix = prefix)
```

After saving them as 3 different csv's, we upload the data into the s3 bucket, in the folder 'malware-detection'.



Now, to define the XGBoost model. Before we define it, let us understand the meaning of the parameters. According to the AWS documentation (attached with the notebook):

- **max_depth**: Maximum depth of a tree. Increasing this value makes the model more complex and likely to be overfit. 0 indicates no limit. A limit is required when grow_policy=depth-wise.
- **eta:** Step size shrinkage used in updates to prevent overfitting. After each boosting step, you can directly get the weights of new features. The eta parameter actually shrinks the feature weights to make the boosting process more conservative.
- **min_child_weight:** Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, the building process gives up further partitioning. In linear regression models, this simply corresponds to a minimum number of instances needed in each node. The larger the algorithm, the more conservative it is.

- **subsample:** Subsample ratio of the training instance. Setting it to 0.5 means that XGBoost randomly collects half of the data instances to grow trees. This prevents overfitting.
- **gamma**: Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger, the more conservative the algorithm is.
- **early_stopping_rounds**: The model trains until the validation score stops improving. Validation error needs to decrease at least every early_stopping_rounds to continue training. Amazon SageMaker hosting uses the best model for inference.

The XGBoost Classifier has a lot more hyperparameters, we will use only the ones mentioned above.

```
container = get_image_uri(session.boto_region_name,'xgboost')
xgb = sagemaker.estimator.Estimator(container,
                                    role,
                                    train_instance_count = 1,
                                    train_instance_type = 'ml.m4.xlarge',
                                    output_path = 's3://{}/{}/output'.format(session.default_bucket(),prefix),
                                    sagemaker_session = session)
```

```
WARNING:root:There is a more up to date SageMaker XGBoost image. To use the newer image, please set 'repo_version'='0.90-1'. Fo
r example:
        get_image_uri(region, 'xgboost', '0.90-1').
```

```
xgb.set_hyperparameters(max_depth = 5,
                        eta = 0.2,
                        gamma = 4,
                        min_chile_weight = 6,
                        subsample = 0.8,
                        objective = 'reg:linear',
                        early_stopping_rounds = 10,
                        num_round = 200)
```

Now, we define the Hyperparameter Tuner, and start finding the best model, using the training and validation set. We will now get the data to be fed into the models using sagemaker's s3_input() function.

```
s3_input_train = sagemaker.s3_input(s3_data=train_location, content_type = 'csv')
s3_input_validation = sagemaker.s3_input(s3_data=val_location, content_type = 'csv')
```

```
#xgb.fit({'train':s3_input_train,'validation':s3_input_validation})
from sagemaker.tuner import IntegerParameter, ContinuousParameter, HyperparameterTuner
xgb_hyperparameter_tuner = HyperparameterTuner(estimator = xgb,
                                               objective_metric_name = 'validation:rmse',#precision
                                               objective_type = 'Minimize',
                                               max_jobs = 30,
                                               max_parallel_jobs = 3,
                                               hyperparameter_ranges = {
                                                    'max_depth':IntegerParameter(3,12),
                                                    'eta':ContinuousParameter(0.05,0.5),
                                                    'min_child_weight':IntegerParameter(2,8),
                                                    'subsample':ContinuousParameter(0.5,0.9),
                                                    'gamma':ContinuousParameter(0,10)
                                               }
                                               )
```

```
xgb_hyperparameter_tuner.fit({'train':s3_input_train,'validation':s3_input_validation})
```

```
xgb_hyperparameter_tuner.wait()
```

Then, we define our Hyperparameter Tuner, we set the ranges for:

- max_depth as a set of Integers, from 3 to 12
- eta as a set of floating numbers from 0.05 to 0.5
- min_child_weight as a det of Integers from 2 to 8
- subsample as a set of floating numbers from 0.5 to 0.9
- gamma as a set of floating numbers from 0 to 10

We have set 30 models to train, 3 executing at a time. We set the objective of the model to decrease as much 'root mean squared error' as possible, using the validation set as reference. Once we set them to fit, we wait for the 30 training jobs to finish. Since we have that many models, it takes quite some time to finish. Once it finishes, we attach it to the estimator, and generate the optimised model's predictions.

```
: #seeing the best tuner:
  xgb_hyperparameter_tuner.best_training_job()

: 'xgboost-200315-1213-019-3ee3ccc4'
```

This is the best out of 30 different models we had set to run. Now, to use this estimator to predict.

Using a bit of Jupyter notebook magic, we'll fetch a local copy the output of the optimized estimator, and compare with the labels of the testing set.
Then, well see the accuracy score, and find out if the XGBoost Classifier can beat the said record made by the Random Forest Classifier (97% accuracy).

```
In [47]: !aws s3 cp --recursive $xgb_transform.output_path $'../output'

download: s3://sagemaker-ap-south-1-812709844112/xgboost-200315-1213-019-3ee3ccc4-2020-03-15-13-57-45-224/test.csv.out to ../ou
tput/test.csv.out

In [53]: y_pred = pd.read_csv('../output/test.csv.out',header = None)

In [61]: for i in range(len(y_pred)):
             y_pred[0][i] = np.abs(np.round(y_pred[0][i]))

In [64]: from sklearn.metrics import accuracy_score
         print("Accuracy of the model: ",accuracy_score(y_test,y_pred))

Accuracy of the model:  0.9860298030867483
```

We have achieved an accuracy of **98.6%** using XGBoost Classifier.

Inference:
Thus, the tuned XGBoost Classifier seems a good fit as a model for spam prediction, better than the Random Forest Classifier. However, the following factors must be taken into account:

- The columns dropped right before analysis might have held relevant data, which might affect the model's decisions.
- The hyperparameter tuning jobs could have produced a better model than the one achieved.
- The columns we have right now, might not be enough data to predict on new files. Unless we have enough info on a new file, we cannot make an accurate prediction.

In the future, this model could be deployed using AWS, or could be integrated into a website and deployed, as a malware detector. However, it might not be ready for a real-world application, where data unknown to us could be provided, which might prove an essential factor in deciding whether a file contains malware or not. But, if we get as much data as in the way the model has been trained, it would do a good job at correctly classifying files. The vocabulary of the DLLs and Imported Symbols is currently limited. However, that preprocessing can be easily done-over and the model re-trained, making it possible to use in the future.