Machine Learning Engineer Nanodegree
Capstone Project
Menna Essa
March 3rd , 2019

I. Definition

Project Overview

Information has always been a challenge in technology as users always download programs from untrusted sources ; PE format (portable executable) is the main file type for executables on windows operating system ; to fight this , different malicious behavior detection technologies have evolved throughout the years , starting from simple hash comparisons all the way to emulation and sand-boxed environments.
legacy techniques like hash comparisons and regular expressions have proven to be inefficient against modern malware which now can evade them using polymorphic code , so attention was shifted towards dynamic techniques , however these techniques are resource exhaustive and can pose a risk as they involve running the suspicious executable.
Finally, with the evolution of machine learning algorithms , information security research began utilizing them to gain a heuristic insight of the behavior of the target , heuristic analysis appeared as middle ground between static and dynamic analysis and helps cut the cost of needlessly running dynamic analysis on non-suspicious samples.

The PE file format is well documented on MSDN here , a very helpful graph of the PE file file header can be found here.

Problem Statement

The problem is the PE files can be very big and complex , there are also many anti-analysis techniques that are employed by the threat actor that makes recognizing the executable's behavior without running it a challenging and sometimes impossible task.
However , Some times just the Headers of the PE file can have indicators that we are looking at a malicious/suspicious file ; In fact , the PE header is usually the first thing an analyst looks at before analyzing the actual code in the binary.
It would be interesting to analyze a mass of header data , discover relevant fields and eventually use them to predict whether a sample is malicious or not.
A paper was published exploring this here : https://arxiv.org/pdf/1709.01471.pdf .

The goal is to find the optimal model for this classification problem , primary candidates are Ensemble decision trees ,XGBoost ,Naive Bayes and MLP neural network which will be used after data analysis and preprocessing , I also want to explore the effect of PCA (principal component analysis) on the dataset and whether or not it improves performance on this particular problem.

The primary steps to solve this problem are outlined below:

step 1 : data exploration and preprocessing
In the first part of the project , I will explore the data set , look for irrelevant fields and process sparse and skewed fields , this is important to facilitate the work of the algorithms , also some fields are skewed so I may have to apply a log function to them and normalize other columns so stay with [0,1] range.

Step2 : Train the supervised models [ensemble decision trees , XGBoost , Naive Bayes] and iterate to optimize , after making sure the data is clean , I will start training the models on the dataset and optimize them with Grid search cross validation to make sure I found the optimal hyper parameters.

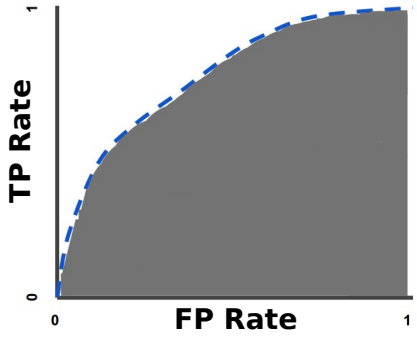
Step3: final accuracy will be calculated on a test set also provided by kaggle.

Step4 : compare results with and without feature selection.

Metrics

For both benchmark and final model Area under the ROC Curve and F-beta Score will be used to measure the performance of the model , this is due to the fact the dataset is not balanced because the malicious label is dominant (74% of the data set) which means that it's very probable that there will be a case of accuracy paradox , in this case using metrics that utilize the precision and recall is more appropriate.

AUC "Area under the ROC Curve." : measures the two-dimensional area underneath the ROC curve (0,0) to (1,1) [resource]



F Beta-Score is the weighted harmonic mean of both precision and recall , A beta value of 0.5 was chose because in our problem precision is what matters the most.

$$FBeta = \frac{(1+B^2)*precision*recall}{B^2*precision+recall}$$

II. Analysis

Data Exploration

The data set covers all the fields in the headers , some fields are repetitive and quite irrelevant so the data will be processed to find fields that seem to capture relevant and irrelevant fields , PCA and further Supervised learning algorithms should help in discovering relevant information that is connected to the maliciousness of a file , there are currently 77 input fields shown in the link above , most of the features are continuous only 4 features are categorical (subsystem , dll_characteristics , LoaderFlags , Machine) The dataset has a total of 19611 entries , **14599** of them are malware and **5012** of the are not.

e_magic	e_cblp	e_cp	e_crlc	e_cparhdr	e_minalloc	e_maxalloc	e_ss	e_sp	e_csum	...	SectionMaxChar	SectionMainChar	DirectoryEntryImport	Direct
23117	144	3	0	4	0	65535	0	184	0	...	3758096608	0	7	

Truncated sample of the a dataset field.

	e_magic	e_cblp	e_cp	e_crlc	e_cparhdr	e_minalloc	e_maxalloc	e_ss	e_sp	e_csum
count	19611.0	19611.000000	19611.000000	19611.000000	19611.000000	19611.000000	19611.000000	19611.000000	19611.00000	19611.000000
mean	23117.0	178.615726	71.660752	49.146958	37.370710	37.032635	64178.739687	10.418490	226.46530	29.689103
std	0.0	987.200729	1445.192977	1212.201919	864.515405	915.833139	9110.755873	637.116265	1249.68033	1015.303419
min	23117.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.00000	0.000000
25%	23117.0	144.000000	3.000000	0.000000	4.000000	0.000000	65535.000000	0.000000	184.00000	0.000000
50%	23117.0	144.000000	3.000000	0.000000	4.000000	0.000000	65535.000000	0.000000	184.00000	0.000000
75%	23117.0	144.000000	3.000000	0.000000	4.000000	0.000000	65535.000000	0.000000	184.00000	0.000000
max	23117.0	59448.000000	63200.000000	64613.000000	43690.000000	43690.000000	65535.000000	61436.000000	65464.00000	63262.000000

Truncated data statistics

Upon exploring the data , it was found that 7 fields are actually just the same across all points , so these feature will simply be omitted

e_magic , SectionMaxEntropy , SectionMaxRawSize , SectionMaxVirtualSize, SectionMinPhysical, SectionMinVirtual, SectionMinPointerData, SectionMainChar

Histogram of the fields is shown below :

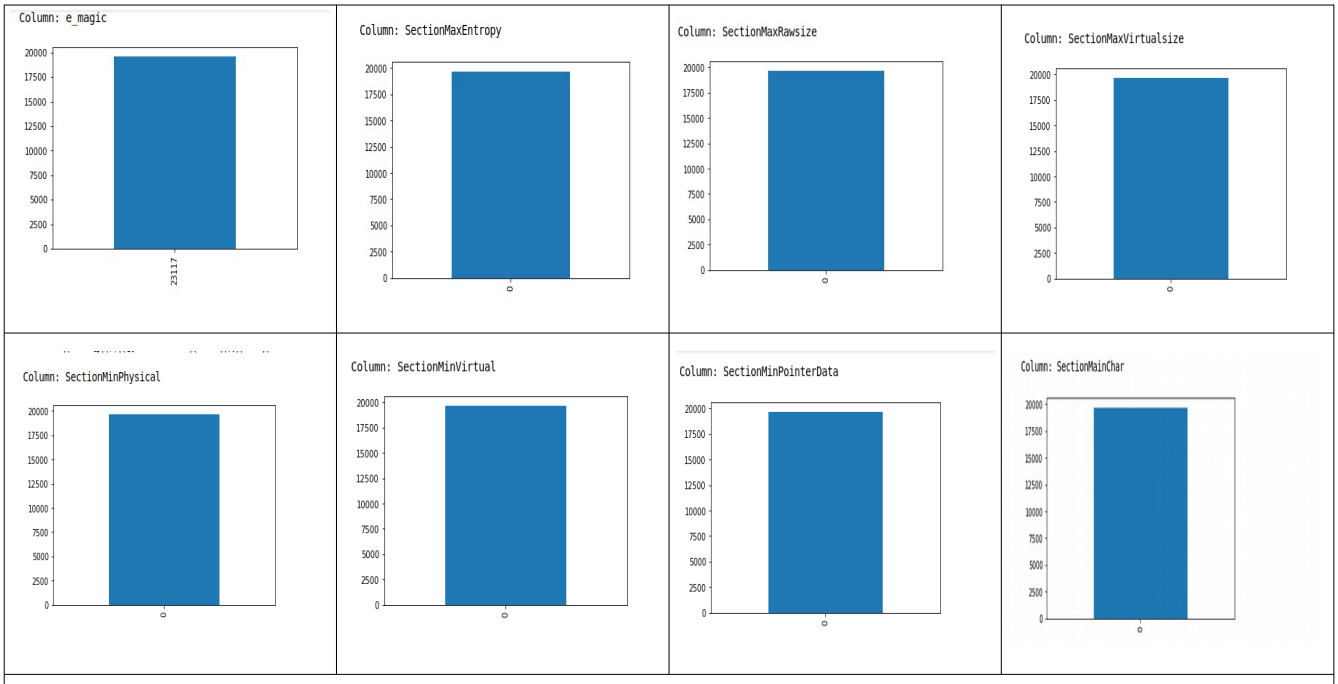


Fig 1 : redundant fields

Another thing to note is that most fields need to be scaled as they represent Memory addresses , sizes and offsets and sometimes flags that are relevant to the operating system.

Finally , although some features were dropped the features are still quite too many (69 features) and will need features selection.

Exploratory Visualization

The features that I assume stand out the most are :

-AddressOfEntryPoint : Where the program will start executing , malicious software usually starts in non traditional locations outside the main (.text) code section to avoid detection.
-NumberOfSections : Malware usually adds extra sections to hide it’s data , packed malwares can also have too few sections.
To emphasize this I created a scatter plot to show if it does indeed create a pattern in the data.

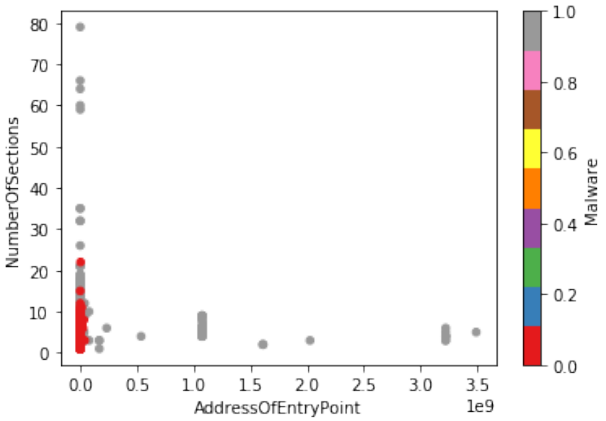


Fig2: Scatter plot of samples using Numberofsections and AddressOfEntryPoint

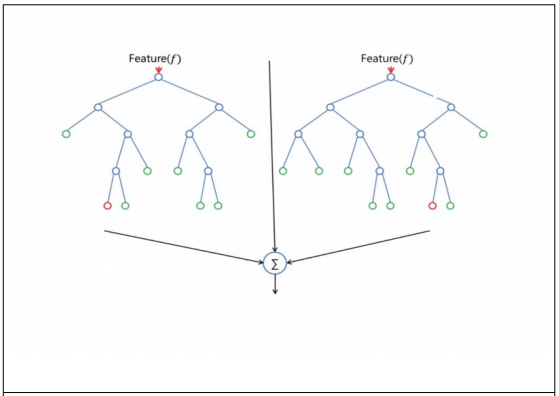
Below are the fields statistics:

count 19611.000000 mean 5.030544 std 2.084433 min 1.000000 25% 4.000000 50% 5.000000 75% 6.000000 max 79.000000	count 1.961100e+04 mean 8.037910e+06 std 1.085113e+08 min 0.000000e+00 25% 7.784000e+03 50% 3.135100e+04 75% 9.892100e+04 max 3.490505e+09
NumberOfSections	AddressOfEntryPoint

Algorithms and Techniques

This is a Supervised learning classification problem , I will first start with a simple Naive Bayes algorithm and observe how it performs , since our data is continuous , Gaussian implementation will be used , Naive bayes was a good candidate as the features are independent.

Since the data is complex and we have a lot of features , ensemble methods seem to be suitable for this problem , therefore a Random forest classifier seemed to be a good option for this problem , Random forest classifier 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 this a method call bagging.

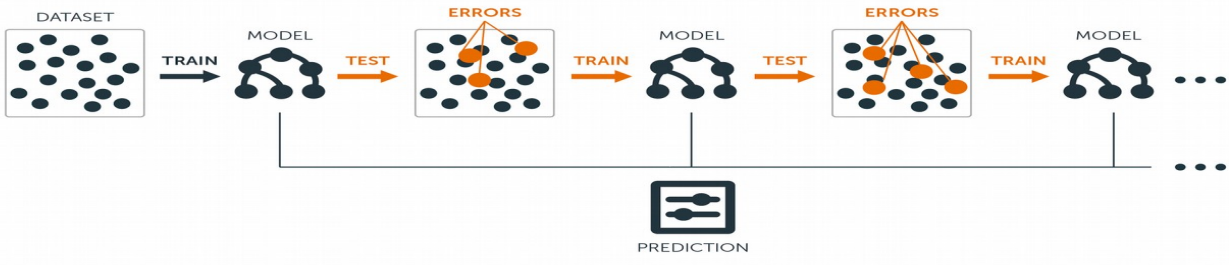


Bagging : combining results of multiple trees to improve overall accuracy.

Hyperparameters of interest:

- n_estimators : The number of trees in the forest.
- max_depth : The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- max_features : The number of features to consider when looking for the best split:

Finally Combining ensembling and boosting in XGBoost to hopefully obtain optimal results , XGBoost is an implementation of gradient boosted decision trees , Gradient boosting is an approach where new models are created that predict the residuals or errors of prior models and then added together to make the final prediction. It is called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models , In contrast to bagging techniques like Random Forest, in which trees are grown to their maximum , boosting makes use of trees with fewer splits



Boosting : training the trees sequentially to improve residual errors and generate an new model.

Hyperparameters of intereset:

- learning_rate: step size shrinkage used to prevent overfitting. Range is [0,1]
- max_depth: determines how deeply each tree is allowed to grow during any boosting round.
- colsample_bytree: percentage of features used per tree. High value can lead to overfitting.

Benchmark

the benchmark model for this problem is logistic regression with default parameters , a simple linear model is a good benchmark model for this dataset , The model achieve FBeta score of 0.83 and and area under ROC (auc) score: 0.638 which is very promising.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import fbeta_score
from sklearn.metrics import roc_auc_score

X_train, X_test, y_train, y_test = train_test_split(features, labels , test_size=0.33, random_state=0 , shuffle=True )
clf = LogisticRegression(random_state=0).fit(X_train, y_train)
predictions = clf.predict(X_test)
print("FBeta score : " , fbeta_score(y_test, predictions, beta = 0.5))
print ("ROC auc score: " , roc_auc_score(y_test, predictions))
```

III. Methodology

Data Preprocessing

1-Irrelevant/Redundant features:
Irrelevant fields were detecting by plotting the frequency histogram per feature.

```
for column in data:
   print("Column:" , column)
   data[column].value_counts().plot(kind='bar')
   plt.show()
```

Then the fields (refer to fig1) were dropped.

```
features = features.drop(['e_magic','SectionMaxEntropy','SectionMaxRawsize','SectionMaxVirtualsize','SectionMinPhysical',\
                'SectionMinVirtual','SectionMinPointerData','SectionMainChar'] , axis=1)
print("malware dataset has {} data points with {} variables each.".format(*features.shape))
display(features.head(n=1))
```

We now have 69 features to process.

2-Huge feature values:
Most of the features deal with memory address , offsets and byte counts , which means that they are very large and inefficient for the algorithms.
All the fields are scaled using Standard scaler to limit them within [-1 , 1] range.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

```
scale_np = scaler.fit_transform(features)
features_scaled = pd.DataFrame(scale_np, columns=features.columns)
display(features_scaled.head(n=1))
```

	e_cblp	e_cp	e_crlc	e_cparhdr	e_minalloc	e_maxalloc	e_ss	e_sp	e_csum	e_ip	...	SectionMaxPointerData	SectionMaxChar	DirectoryEntryImport	I
0	144	3	0	4	0	65535	0	184	0	0	...	245248	3758096608	7	

	e_cblp	e_cp	e_crlc	e_cparhdr	e_minalloc	e_maxalloc	e_ss	e_sp	e_csum	e_ip	...	SectionMaxPointerData	SectionMaxCha
0	-0.035065	-0.047511	-0.040545	-0.038601	-0.040437	0.148867	-0.016353	-0.033982	-0.029242	-0.05893	...	-0.078912	1.01441

Fig3 : Data before and after scaling.

3-Feature selection (Optional):
Although the irrelevant features were dropped , it is still possible that not all 69 features are relevant to our problem , for this I used SelectPercentile with f_regression , f_regression made since because it computes the correlation between each regressor and the target and converts it to an F-score .
I chose to pick 50% of the features so it’s not cutting out too much data

```
from sklearn.feature_selection import SelectPercentile , f_regression
selection = SelectPercentile(f_regression, percentile=50)
features_scaled_= selection.fit_transform(features_scaled,labels)
columns = np.asarray(features_scaled.columns.values)
support = np.asarray(selection.get_support())
selected_features = columns[support]

print("Selected features " , selected_features)
```

Now our dataset is composed of 34 scaled features and ready for our algorithms.

```
'e_maxalloc', 'e_ip' ,'e_oemid' ,'e_oeminfo', 'e_lfanew', 'Machine', 'NumberOfSections', 'TimeDateStamp' ,'NumberOfSymbols',
'SizeOfOptionalHeader', 'Characteristics', 'Magic', 'MajorLinkerVersion', 'MinorLinkerVersion',
'AddressOfEntryPoint' ,'ImageBase' ,'SectionAlignment', 'FileAlignment', 'MajorSubsystemVersion', 'CheckSum' ,'Subsystem',
'DllCharacteristics', 'SizeOfStackReserve', 'SizeOfHeapReserve','SuspiciousImportFunctions' ,'SuspiciousNameSection',
'SectionsLength', 'SectionMinEntropy' ,'SectionMinRawsize' ,'SectionMinVirtualsize',
'SectionMaxPointerData' ,'SectionMaxChar' ,'DirectoryEntryImport','DirectoryEntryExport'
```

Implementation:

The data was split at 70:30 training and testing respectively , with shuffling to ensure randomization.

```
X_train, X_test, y_train, y_test = train_test_split(features_processed, labels , test_size=0.33, random_state=0 , shuffle=True )
```

To train the models , I wrote a generic function to train , predict and return the evaluation results.

```
def train_predict(learner, X_train, y_train, X_test, y_test):
  '''
  inputs:
    - learner: the learning algorithm to be trained and predicted on
    - X_train: features training set
    - y_train: income training set
    - X_test: features testing set
    - y_test: income testing set
  '''
  results = {}
```

```
    learner.fit(X_train,y_train)
    predictions_test =learner.predict(X_test)
    predictions_train = learner.predict(X_train[:300])
    results['auc_train'] = roc_auc_score(y_train[:300], predictions_train)
    results['auc_test'] = roc_auc_score(y_test, predictions_test)
    results['f_train'] = fbeta_score(y_train[:300] , predictions_train , beta=0.5)
    results['f_test'] = fbeta_score(y_test , predictions_test , beta=0.5)
    print("{} trained on samples.".format(learner.__class__.__name__))
    return results
```

Initial results:

```
Premature Models
----------------
GaussianNB
auc_train : 0.6867430441898528
auc_test : 0.6957339039464563
f_train : 0.7624398073836275
f_test : 0.7722553061536113

XGBClassifier
auc_train : 1.0
auc_test : 0.9840549069566313
f_train : 1.0
f_test : 0.9914255091103966

RandomForestClassifier
auc_train : 1.0
auc_test : 0.9871274488102104
f_train : 1.0
f_test : 0.9933267014838763
```

Refinement

Naive bayes did not show promising results , in fact they were much lower than the benchmark (69% auc , 77% f-beta score) so I Chose to focus on the refinement of Random forest classifier and XGBoost.

- **Parameter tuning**

I used GridSearch cross validation to test different hyperparameters settings and choose the optimal estimator.

Random forest:

```
regressor = RandomForestClassifier()
cv_sets = ShuffleSplit(n_splits = 10, test_size = 0.20, random_state = 0)
params = {'n_estimators':[100,150,200] ,
      'max_depth': range(10,20) ,  'max_features' : ['sqrt' , 'auto', 'log2']}
scoring_fnc =  make_scorer(fbeta_score , beta=0.5)
grid = GridSearchCV(regressor,params,scoring_fnc,cv=cv_sets)
grid = grid.fit(X_train, y_train)
```

```
Result:'max_depth': 18, 'max_features': 'sqrt', 'n_estimators': 200
```

Xgboost (Refrence)
Xgboost had a lot of paramters to tune so I used randomized Search cross validation with 100 iterations to gain a better insight and create a smaller parameter list for a faster grid search , obviously the goal parameter is binary:logistic which is logistic regression for binary classification and output probability which is generally better in the information security field because nothing is ever 100%.

```
regressor = XGBClassifier()
cv_sets = ShuffleSplit(n_splits = 10, test_size = 0.20, random_state = 0)
params = {"learning_rate"    : [0.05, 0.10, 0.15, 0.20, 0.25, 0.30 ] ,
 "max_depth"        : [ 3, 4, 5, 6, 8, 10, 12, 15],
 "min_child_weight" : [ 1, 3, 5, 7 ],
 "gamma"            : [ 0.0, 0.1, 0.2 , 0.3, 0.4 ],
 "colsample_bytree" : [ 0.3, 0.4, 0.5 , 0.7 ] }
scoring_fnc =  make_scorer(fbeta_score , beta=0.5)
grid_r =RandomizedSearchCV(regressor, param_distributions=  params,scoring=scoring_fnc,cv=cv_sets , verbose=10 , n_jobs=-1)
grid_r = grid_r.fit(X_train, y_train)
```

```
Best params :  'min_child_weight': 1, 'max_depth': 8, 'learning_rate': 0.25, 'gamma': 0.1, 'colsample_bytree': 0.5'
```

```
params = {"learning_rate"    : [ 0.10, 0.15, 0.20 ] ,
 "max_depth"        : [10, 12, 15],
 "min_child_weight" : [ 1, 3, 5],
 "gamma"            : [ 0.0, 0.1, 0.2 , 0.3, 0.4],
 "colsample_bytree" : [ 0.3, 0.4, 0.5 ] }
scoring_fnc =  make_scorer(fbeta_score , beta=0.5)
grid_r =GridSearchCV(regressor,params,scoring=scoring_fnc,cv=cv_sets , verbose=10 , n_jobs=-1)
grid_r = grid_r.fit(X_train, y_train)
print(grid_r.best_params_)
```

```
Final best parameters : 'colsample_bytree': 0.5, 'gamma': 0.1, 'learning_rate': 0.2, 'max_depth': 12, 'min_child_weight': 1
```

Final solutions

```
clf_RFC = RandomForestClassifier(n_estimators=100, random_state=0,
                oob_score = True,
                max_depth = 19,
                max_features = 'log2')
clf_XGB = XGBClassifier(min_child_weight= 3, max_depth= 10, learning_rate= 0.2, gamma= 0.0,colsample_bytree= 0.5 ,  objective='binary:logistic' )
```

```
XGBClassifier
auc_train : 1.0
auc_test : 0.9885893688671513
f_train : 1.0
f_test : 0.9938837920489297

RandomForestClassifier
auc_train : 1.0
auc_test : 0.9866718066019056
f_train : 1.0
f_test : 0.9928565529771244
```

It can be observed that Parameter tuning did not make a huge difference following the fact that the default parameters were already performing exceptionally (and suspiciously well) , if it weren't for the good test results I would have assumed that this is a case of over-fitting.

- **Feature selection**

I trained the models with and without feature selection and their scores were very close < 0.001 difference and had longer training time , so feature selection is a good performance boost and did not affect the models accuracy much.

With feature selection	No feature selection
XGBClassifier	XGBClassifier

auc_test : 0.9885893688671513 f_test : 0.9938837920489297 RandomForestClassifier auc_test : 0.9866718066019056 f_test : 0.9928565529771244	auc_test : 0.9895052962916672 f_test : 0.9943393108007602 RandomForestClassifier auc_test : 0.9867755841029434 f_test : 0.9928990174221781

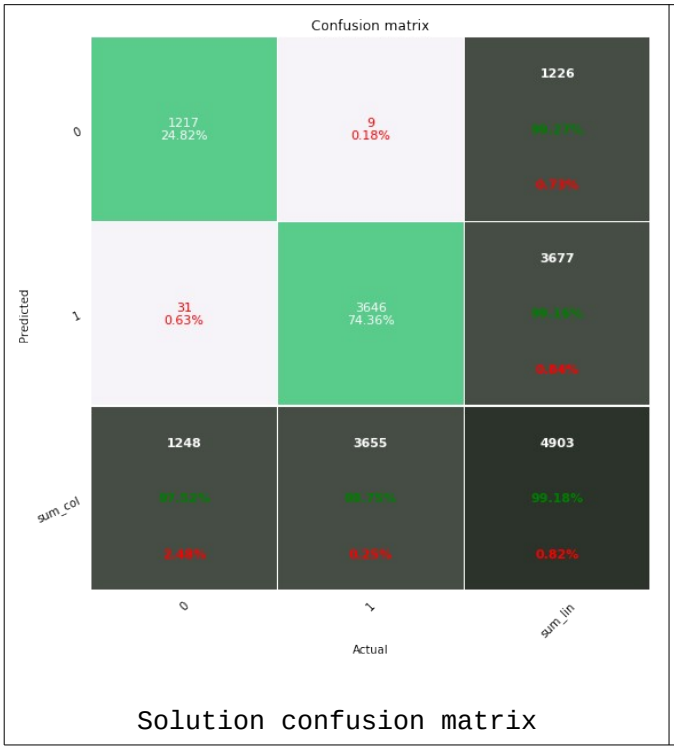
IV. Results

Model Evaluation and Validation

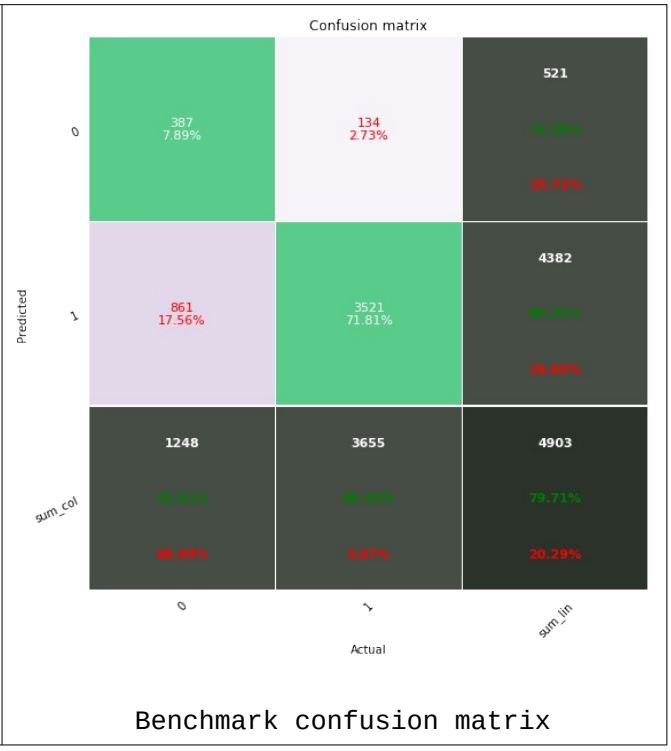
Although Random forest classifier and XGBoost seem to show close and high resutls , XGBoost seemed to be Very sensitive to new data , while Random forest showed less senstivity , in a small test set composed of 16 files , The random forest model classified 12/16 files correctly which is 85% of the real life test set while Xgboost classified 10/16 files which is 62.5% which is very poor , both models showed sensitivity which implies that they need more diverse training data however random forest is more stable and hence is more appropriate for our problem.

Justification

Random forest achieved 99% accuracy on this data set which is much better than the benchmark’s 83%
Below is the confusion matrix of both the benchmark and the final solution , I used this for pretty printing



Solution confusion matrix

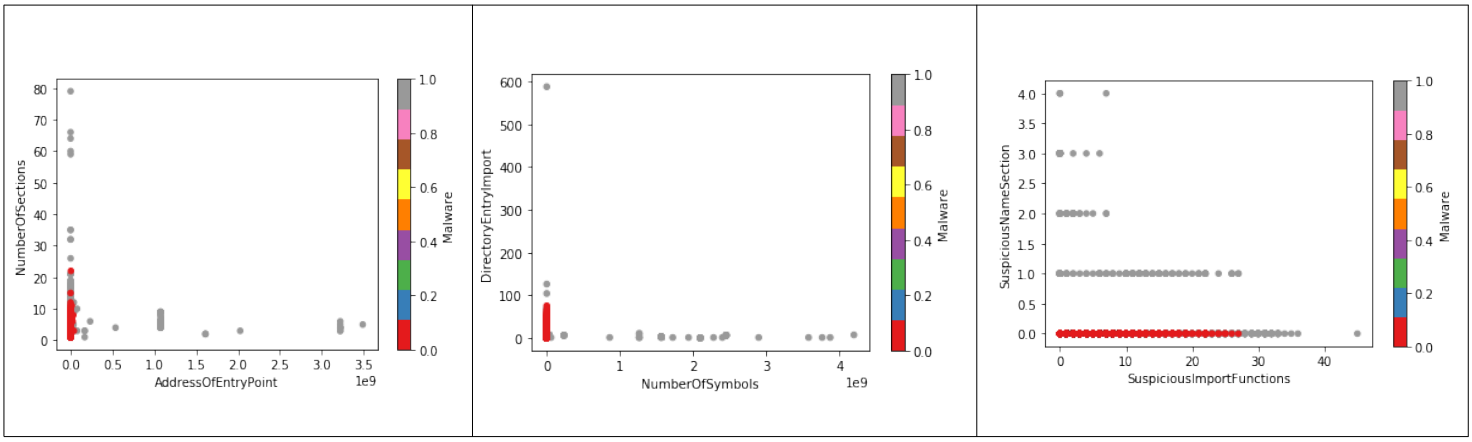


Benchmark confusion matrix

V. Conclusion

Free-Form Visualization

The closer the header data is analyzed the more obvious it is a strong indicator of the maliciousness of a program



Reflection

The problem started with abstract data that did not show much correlation to the labels by just looking at it , however data analysis proved that the data provides great insight to the expected behavior of an executable , data preprocessing improved the quality of data , removed noise and showed very strong features ,

later on training the supervised models and comparing them showed how each model fits the data and sometimes although the model just is not suitable for the data no matter how you tune it evaluating the models and comparing them to the bench mark showed the huge effect good parameter tuning can do to the end result.

I found the data analysis and feature selection fascinating and very interesting as it converted seriously vague data to something very useful.

Tuning the hyper-parameters was troublesome for me as I had a huge parameter list at the begging which gave huge number of fits (38400 for XGBoost) but I learn during the process about Randomized search cross validation and using parallelism improved search time and helped me narrow down the parameter list and reach optimal hyper parameters.

The model exceeded my expectations , however I expect it to show higher false negative rates in the wild as many modern malware are very good at disguising as a normal executable without any header abnormality.

Although I planned in the proposal to use a Neural network as well , after these results it seemed as just over engineering as Random forest truly seems to fit the problem perfectly.

Improvement

The most obvious improvements will be on the data set , balancing the data set with more clean files (0 flagged) and covering more malware classes should improve the overall performance of the model , also offering more insight into the executable code and data sections can do a huge difference and break ties in classification.