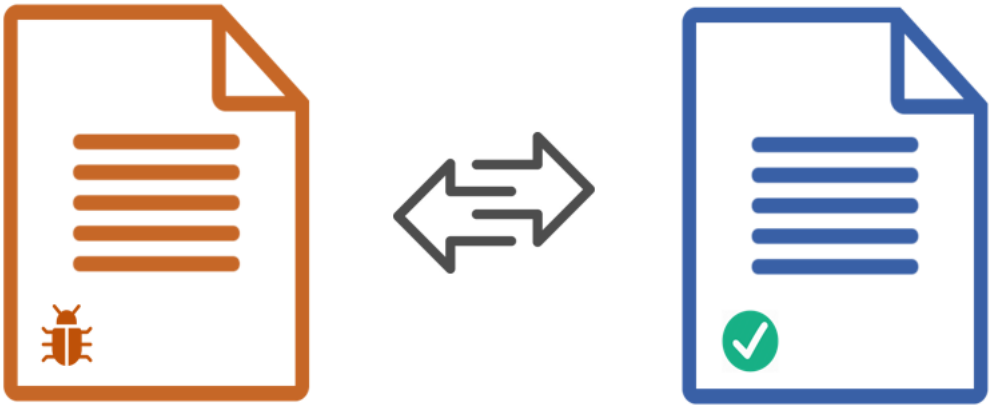
Final Project – Introduction to Machine Learning

-Malicious File Prediction -

Lecturer – Dor Bank

Teaching Assistant – Ilan Vasilevsky

Date of Submission - 30.06.2023



Project Team:

Eitan & Yuval Bakirov

Machine Learning – Final Project

Project Summary

Our task in the project is to solve a binary problem, with the help of static analysis of the files, in which we had to classify records into two categories - a malicious file (malicious 1)) or a non-malicious file (benign 0) based on several features in the data set. Some of the features are known and some are anonymous.

To carry out the project, we were given 60,000 observations classified into malicious and non-malicious files.

First, we explored the existing data, researched it and came to general conclusions about the nature of the features' distribution, their correlative behavior and their statistical data.

After the in-depth visualization, we divided the data into data and validation and began the pre-processing procedure in which we applied the conclusions that emerged from the data study. Among other things, we completed missing values, dealt with unusual values and added relevant features.

After we explored the data and processed it, it was time to run the machine learning models we learned throughout the course. We ran four models, two initial and two advanced to apply the data set to them and found the optimal hyperparameters.

To evaluate the quality of the models we used the confusion matrix and K-Fold cross validation, and with the help of the AUC score we finally chose the Random Forest model.

We are done with making a prediction on the data of the unknown test file.

Part One - Exploration

In this part we will explore the data, we will focus on the distribution of the features, their correlative behavior and their statistical data.

At first glance, it appears that the data set has 60,000 observations (rows) and 24 characteristics of the file (columns) - 23 features that are actually the characteristics of the file, and another column - a label that tells us whether the file is a malicious file or a non-malicious file. We will separate all 23 features from the label column, and this will be the training data we will work with from now on.

First, we'll want to make sure that each observation in the data is unique, so we'll remove duplicates to ensure that each row appears once in the file. Looking at the data we can see that there are exactly 30,000 malicious files and 30,000 non-malicious files.

When examining the types of features, we observed that some of the features are objects, some are represented as float64 decimal numbers, and others as int64 integers.

The sha256, file_type_trid, C features are an object. The features vsize, imports, exports, has_debug, has_relocations, has_resources, has_signature, has_tls, symbols, numstrings, paths, urls, registry, MZ, printables, avlength, file_type_prob_trid, A, B are decimal numbers. The size feature is an integer. It also seems that there are numbers whose values are 0/1, i.e. binary, and there are numbers that are spread over wide ranges of numbers. For the convenience of working with the data, we will define all the numbers to be of type 'float64', the booleans to be of type 'bool' and the objects to be of type 'category'.

By looking at the statistics of the various features, we were able to understand a lot about each feature, the range of values in it, the amount of missing values in it, the average of the values in it, the variance, the maximum and minimum value and the percentages of the values in it. With the help of all these we will be able to work in a smarter and more convenient way in the future.

From looking at the number of different values in each feature we learned mainly about the categorical features. The 'sha256' feature has 60,000 different values, which can indicate it as a representative and unique feature for each file. We also saw that the 'file_type_trid' feature has 89 different categories. Feature 'C' has 7 different categories. We will consider these later.

The distribution of features

First we will draw a histogram for each numerical feature to see how it is distributed (Appendix 1.1). At first glance, it appears that most of the features lack a clear peak at the 0 value on the x-axis, which suggests that outliers in the data significantly affect the histograms, making it challenging to discern the true distribution of these features. We will deal with the outliers later. We will use a log transformation on the numerical features to shrink the values so that we can see the different distributions more closely. Looking after the transformation (appendices 1.2.1-.1.2.15) it seems that only the graph of feature 'A' behaves like a normal distribution. Perhaps our subsequent dealings with abnormal values will help to better see the distribution of the additional features.

Correlation between features

We would like to see if there is a correlation between the various features. We will use the VIF index (Variance Inflation Factor) which will help us look at all the features together. A high VIF score indicates high multicollinearity between the feature and all other features. The VIF for each variable is calculated as the ratio between the variance of the estimated regression coefficient, and the variance of the coefficient if this feature had no correlation with the other features. A VIF score less than 1 indicates a weak correlation. A score between 1-5 indicates a moderate correlation, while a score higher than 5 indicates a high correlation! (Appendix - VIF - 11) The features that the VIF index defines as having high multicollinearity are: 'size', 'has_resources', 'numstrings', 'file_type_prob_trid' and - 'B'. After a little research we realized that this way is not necessarily enough to determine that a feature may harm the model and should be removed.

Another way we adopted to test the correlation between the features is the correlation matrix (Appendix 1.3) which tests a linear correlation between any two features. A higher number in the matrix box indicates a higher correlation between the two features. (+ appendices 1.4.1-1.4.2)

We discovered that there is a high correlation of 0.9 between 'size' and 'numstrings', a high correlation of 0.7 between 'size' and 'MZ' and a correlation of 0.6 between 'numstrings' and 'MZ'. Due to the appearance of the numbered features 'size' and 'numstrings' both in the VIF index and in the correlation matrix, and 'MZ' in the correlation matrix, as having high correlations, we will consider in the future to get rid of one or more of them in order to reduce the amount of dimensions and minimize multicollinearity for the sake of improvement Model performance.

We made further observations on the correlation between the various features and the label to try to learn something about the influence of the features on the maliciousness of the file, but without any special conclusions. (Appendix 1.5)

Dealing with missing values

From a preliminary look it seems that there are about 34,810 missing values in the data set. The features 'sha256', 'size', 'file_type_trid' and 'file_type_prob_trid' have no missing values at all. At most 7 percent of the cells in each feature have missing values. Since missing values can impact subsequent analysis and modeling, we need to fill them in thoughtfully during the preprocessing phase.

Dealing with categorical features

The analyzes will be done on the categorical features 'file_type_trid' and 'C' only (appendices 1.6.1 - 1.6.2), since the feature 'sha256' is a representative and unique feature for each observation and therefore will not be able to help us particularly. Before we start, we will try to understand how the categorical features "behave". First, we will notice that the 'file_type_trid' feature has many categories that appear relatively few times (Appendix 1.7). In dealing with the categorical features later, we will want to break down the features into demi variables.

The additional categorical feature 'C' is a confidential feature ((anonymous) that will interest us to understand how it behaves and whether it can help us learn something about the label that we will have to predict later. We checked whether one of the categories in it mainly characterizes one type of files (malicious or not malicious) But we were present to discover that each category features a fairly similar amount of malicious files and non-malicious files (Appendix 1.8).

Similar to the categorical features, we thought it would be appropriate to check on the Boolean features whether one of the two categories in it (1/0) can characterize a malicious file or not (Appendix 1.9). With the help of graphs, we saw that if the features 'has_debug' and 'has_signature' are 1 (that is, there is a part of debug in the file and/or there is a signature in the file), most of the files are not malicious, something that can help us a lot in classification. We will discuss all of these in the following sections.

In addition, looking at numerical features helped to understand that there are features whose large or small amount can predict the type of file (malicious/non-malicious). For example: file size, the amount of imported and exported functions and the amount of symbols in the file (appendices 1.10.1 - 1.10.4).

From testing whether the average file size, or the average number of imported functions can characterize a malicious file, we had difficulty drawing clear conclusions (appendices 1.10.4 and 1.10.2). Contrary to this, we were present to discover

that the average of the exported functions is particularly high for non-malicious files compared to malicious files (Appendix 1.10.3). The same goes for the average number of symbols (Appendix 1.10.1). Important conclusions that we will express our opinion on later.

Further to our conclusions here, we will later create a new feature that will be a proportion between features and will also be able to help us classify the files.

Looking at exceptions

In this section we will take another look at the distribution of the various numerical features with the help of boxplots. First, most of the features have unusual values (Appendix 1.11). In addition, similar to the conclusions about the distribution of the features that we made earlier, it can be seen that the feature 'A' is distributed normally. Here too, similar to the previous one, we performed a log transformation on the data to try to see the distributions better (Appendix 1.12). It can be seen that after the transformation, the features 'size', 'vsize', 'numstrings' and 'printables' are also distributed normally due to the distribution of the boxplot in such a way that the median of the data is in the center and the outliers on both sides of the box are quite similar.

In order to strengthen our hypotheses, we used the QQ-Plot method in this part, as a method that was not taught in the course, in addition to the VIF index mentioned before. The QQ graph allows you to check if the distribution of values is like another distribution, and in our case, like a normal distribution. If a diagonal straight line is formed rising from left to right then it can be determined that the feature is indeed distributed normally (Appendix 1.13). And really, after testing with QQ-Plot we can conclude and say that these features are distributed normally as well. Finally, we used some different visualizations to see if there are unusual data that might be worth downloading later (Appendix 1.14). In the second part we will refer to the types of features separately and treat them accordingly.

Part Two - Pre-Processing

In this part we will apply the conclusions we reached in the previous part. Let's start by dividing the data into training and validation (train and validation) so that 80 percent of the data will be training and 20 percent of the data will be validation.

We will start by filling in missing values so that the data is complete. We thought of several approaches to fill in missing values: median, majority classification and correlation between features.

The first approach we adopted on our data is the correlation method. Following the conclusions in the previous part, we saw that there are 3 features that have a relatively high correlation between them and they are 'size', 'numstrings', 'MZ'. In this approach, we will fill in missing values only for these three features. Further to what was said above, there is a hierarchy between these three features according to the size of the correlation between them, and exactly according to this hierarchy we will fill in the missing values. (correlation 0.9 between 'size', 'numstrings') > (correlation 0.7 between 'size', 'MZ') > (correlation 0.6 between 'MZ', 'numstrings').

That is, in the case where size is an empty cell, we use the numstrings in the same line to calculate the relative value that should have been in size. In the case where numstrings is also missing in the current line, we will use the correlation of size with MZ to complete the missing value in size. If the three values in the row are empty, we will use the second approach that we will detail immediately. How to complete the missing cells in this approach will be done as follows:

An empty cell = the value of the correlation matrix (between 0-1, in our case 0.9/0.7/0.6 (* (the value of the correlated feature in the same row according to the hierarchy I described).

In the second approach to filling in missing values, we will use the median and the majority method that we mentioned above. Missing numerical values that were not handled in the first approach will be completed according to the median of the data in that feature. Boolean variables are identified in the pre-processing approach as numeric values for everything (0/1) and classified using the median method as well. Missing values of categorical variables will be completed using the majority method, that is, these values will be completed to be the most common category for that feature. At the end of this part, there are no missing values left in the data set.

Immediately after filling in the missing values, we will create a new feature called 'proportion_imports'. The feature will be a proportion between the number of imported functions (imports) and the total number of functions (imports + exports). Following the conclusions from the previous part, we thought to check the behavior of the imported and exported functions in the files because they can help us in classifying the file as malicious or non-malicious. This proportion feature actually checks the file's reliance on external functions (exports), which can largely indicate a malicious file.

In order to avoid calculation problems, when the denominator is 0 we will set it to be 1 to avoid division by 0.

In the next part of the preprocessing, we will deal with the categorical features that we analyzed in the previous part. One challenge we encountered was the presence of numerous categories in the 'file_type_trid' feature. This problem also created a problem with the test file, that since there are a lot of categories, there is a probability that the test will have categories that we did not know before in the training.

To deal with these problems, we decided that all the categories that appear in less than 1 percent of the observations will be combined into one category called 'other'. Hereby we will greatly reduce the number of categories, and significantly simplify the dimensions of the model.

In the next step, we will use the One-Hot-Encoding method by which we will spread the different categories of the features into dummy variables with values 0/1. This method has several advantages, first, turning categorical features into numerical features allows the machine learning algorithms to access all the information in these features. In addition, it maintains significance and equal treatment among all categories.

After removing the features, we are left with much fewer categories, so that 2883 observations will be classified as the 'other' category.

In total, we are left with 50 features so that they all have numerical values except for the file identifier 'sha256'. Hereby we will complete the dealing with categorical features.

From the feature analysis performed earlier, we discovered that there are features that are normally distributed and some that are not. A log transformation will be made on the normal features which will minimize the effect of the abnormal observations. We will use the IQR method to identify the remaining outliers and update them to be the closest possible extreme limit value to the outlier value of that feature.

On the features that are not normally distributed, we will apply the Isolation Forest algorithm learned in lecture 12. This algorithm trains and learns the limits of each feature on the training set and based on the training is able to give a score to each value in the data set for being abnormal or not. If the Isolation Forest score points to an observation that contains unusual values - we will remove it from the data set. In order to avoid overfitting as much as possible, we preferred to classify only a small part of values as outliers, therefore we chose the contamination hyperparameter to be 0.01.

We will use appropriate graphs to see the various changes (appendices 2.1 - 2.2).

After dealing with the abnormal data, we tried to understand whether the problem of dimensionality is too great. Before answering this question, let's understand how to estimate the problem of dimensionality.

There are several methods: feature-to-sample index which calculates the ratio between the number of features and the number of lines. In addition, checking the percentage of zero/close to zero values that exist in features that may indicate unnecessary dimensions that can be downloaded. Furthermore, it is possible to check the amount of contribution of each feature, so for example features with a high correlation do not contribute to the model and may be worth removing. In addition to these, a large running time/ large resource allocation for running can indicate a dimensionality problem. And finally, the contribution of the features to the model.

After examining each of the ways, we came to a sweeping conclusion that indeed the problem of dimensionality is not significant here: the number of features in relation to the number of samples is very small, only two features have a high percentage of zero values, and later it will also appear that the models are of high quality and the contribution of most features is significant .

Before we move on to reducing the dimensionality, we will ask - why is large dimensionality a problem?

One of the main reasons presented above is the runtime problem of the models. A large number of features causes a long running time. In addition, sometimes, a large amount of features makes the data sparse - something that makes it difficult to find significant patterns and may create an overfitting problem.

Just before reducing the dimensionality, we asked ourselves if the information is normalized. No feature is normalized. It was important for us to normalize the data for several reasons: Comparison - this way you can compare the different values in the features, and eliminate the anomalies that may still exist in the data. Prevention of bias - if the features are on a different scale, bias may occur in the data. Interpretation - normalization allows us and the various models to understand the relationships between the features in a better way.

The only normalization we will use will be the standard scaler, in order to maintain consistency in the data and not create a bias in only part of it.

To avoid the problem of dimensionality, we used two approaches:

Downloading features manually: throughout the project we reached three important conclusions about our features: the percentage of zero values in the symbols and registry features is over 0.9%. In addition, there is a large relationship between symbols and the average of the non-malicious files. Also, there is a large correlation between size, numstrings and MZ. In light of the conclusions, we decided to omit the registry and MZ features.

Another approach is the PCA. This approach will help us reduce the dimensionality of the problem even more and obtain 99% of the data variance (Appendix 2.3).

Despite the advantages of the PCA approach, we decided not to use it in the final model and this is because, despite its great advantage, we lose the meaning of the features in the project.

Just before running the models, we applied all the final decisions we made during the exploration and preprocessing to the Preprocess function, which allows all operations to be run at once.

Part Three - Running Models

The initial models we chose are KNN and logistic regression.

- **KNN** – Test AUC – 0.955: (Appendix 3.1)

n_neighbors = 15 - for this model we chose a relatively large amount of neighbors in order to allow the model to be durable and to capture general and local patterns in data of moderate complexity.
distance = weights - we chose distance because the data is normalized and in this situation it will work better than uniform-
power - We used Grid Search CV in order to find the best hyperparameter among 1 / 2.

- **Logistic Regression** – Test AUC – 0.874: (Appendix 3.2) (+ Appendix 10 – Hyper parameters)

C = 10 - Since the data is not too complex, the regularization can be reduced so that 10 is an index that offers a balance between variability and bias.
penalty - we let Grid Search choose between two regularization techniques - L1\L2.
solver - we allowed GS to choose between the two algorithms - Liblinear\lbfgs.
max_iter = 100 - since we would like to give time for the model to converge, we chose the maximum iterations - 100.

The complex models we chose are ANN and Random Forest:

- **ANN** – Test AUC – 0.957: (Appendix 3.3)

hidden_layer_sizes = 64 - We preferred to increase the ability to learn complex patterns.
activation - we let GS choose between the two priority functions - logistic\relu.
solver = adam - we chose adam because it is a good algorithm for medium to large datasets.

- **Random Fores**t - Test AUC - 0.977: (Appendix 3.4) Here we chose not to use GS due to the large running time that is added with the selection of additional hyperparameters and therefore:

n_estimators = 300 - we preferred to improve the model as much as possible and since each subtree is trained on a different data set we chose 300 decision trees.
max_depth = None - in order to allow the model to capture more complex relationships, we preferred not to limit the depth of the tree (Default).
min_samples_split = 10 - for the same reasons we mentioned before, we chose a relatively large split - 10.
min_samples_leaf = 4 - the same reasons here too - 4.

After running the models, we came to the conclusion that Random Forest is the more preferred model for several reasons:

Its AUC score was the highest and without overfitting. In addition, this model is known for its good abilities to learn data and predict correctly.

After choosing the model, we continued to find the features that contribute the most to the selected model. For this we used the Feature Importance of the model and saw that the most influential features are (Appendix 3.5):

1. Avlength – 10.65% - the average length of the words in the file can give good insights into the complexity or structure of the files. Malicious files can contain complex patterns that can be caught by this feature.

2. B – 10.50% - Although we were unable to understand the essence of this feature, the decision to keep it looks promising.

3. imports - 9.56% - the amount of functions imported to the file can indicate its dependence on external files or paths that may be malicious. Malicious files can contain patterns that are relatively different from those of non-malicious files, so this feature can contribute a lot.

4. Urls - 7.53% - similar to the imported functions, the presence of links to external sources can indicate a link to malicious files.

5. File_type_prob_trid – 6.9% – the probability of the file type allows you to get a lot of information about the nature of the file. Certain types of files may already be classified as dangerous.

Part Four - Evaluation of the Models

Confusion matrix - looking at the table you can see that the model manages to classify malicious files well (Appendix 4.1). We can see this in the high values that exist in TN and TP. However, there are still misclassifications of the model, although not significantly so. In terms of precision, we got 0.93, meaning 93% of the samples classified as malicious were indeed malicious. In addition, for the weakness of the model, (Recall/Sensitivity) the model identified 91% of the possible anomalies. That means 9% of the malicious files were not detected. (You can read in detail about all the conclusions from the model in the notebook).

Next, we moved on to evaluate our models by KFold Cross Validation and building ROC output for each model with its 5 different runs (Appendices 4.2 – 4.6). In conclusion, all the models perform reasonably well, with the Random Forest model emerging as the preferred choice, especially when considering the AUC score.

It can be seen that the model has a small difference between AUC Train and AUC Test (0.03), which indicates that the model is not overfitted. In addition, the differences between the training and validation results between the models themselves are also not significant, another way to know that the models do not have overfitting.

In order to increase the generality of the model, we can realize all the steps performed in the preprocessing step: normalization, removal of outliers and dimensionality reduction, and this is what we performed.

Also, with the different runs of the model, we saw that the gap between the train and test results grew, and this is because we handled the training set excessively, and not in a way that matches the validation set. Therefore, we considered it appropriate to try and minimize such gaps by reducing the removal of exceptions from the training set for example.

Part Five - Making a Prediction

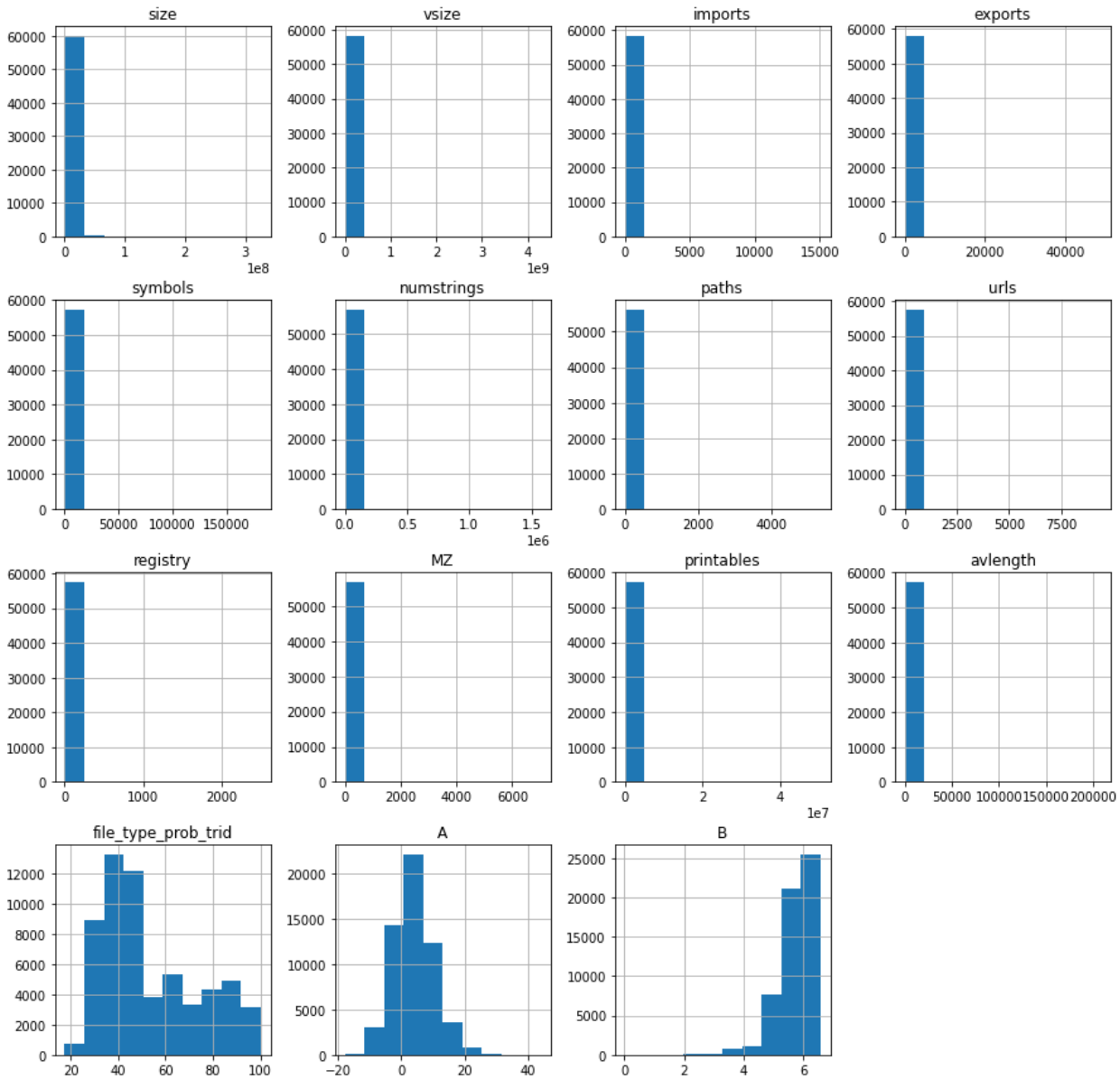
In this section, we have combined all the final operations into one Pipeline function which enables the restoration of the results of the selected model in a fast and clean way. Following model training, we inputed the probabilities assigned by the model to each file to determine their labels (malicious/non-malicious) in the test set (test.csv).

Appendices

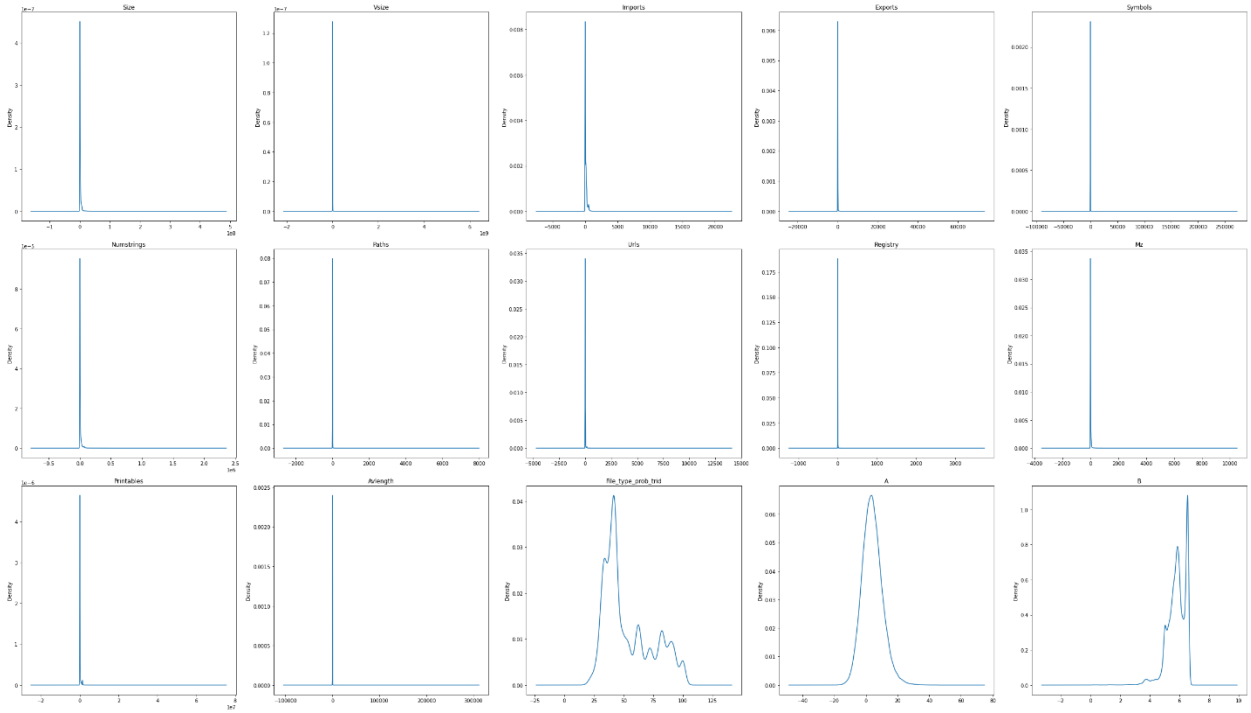
Hyper parameters given to Grid Search in logistic regression: (Appendix 10)

```
# param_grid = {
#     'C': [10.0],  # Inverse of regularization strength
#     'penalty': ['l1', 'l2'],  # Regularization penalty (L1 or L2)
#     'solver': ['liblinear', 'lbfgs'],  # Solver algorithm
#     'max_iter': [100],  # Maximum number of iterations
#     'n_jobs': [-1]
# }
```

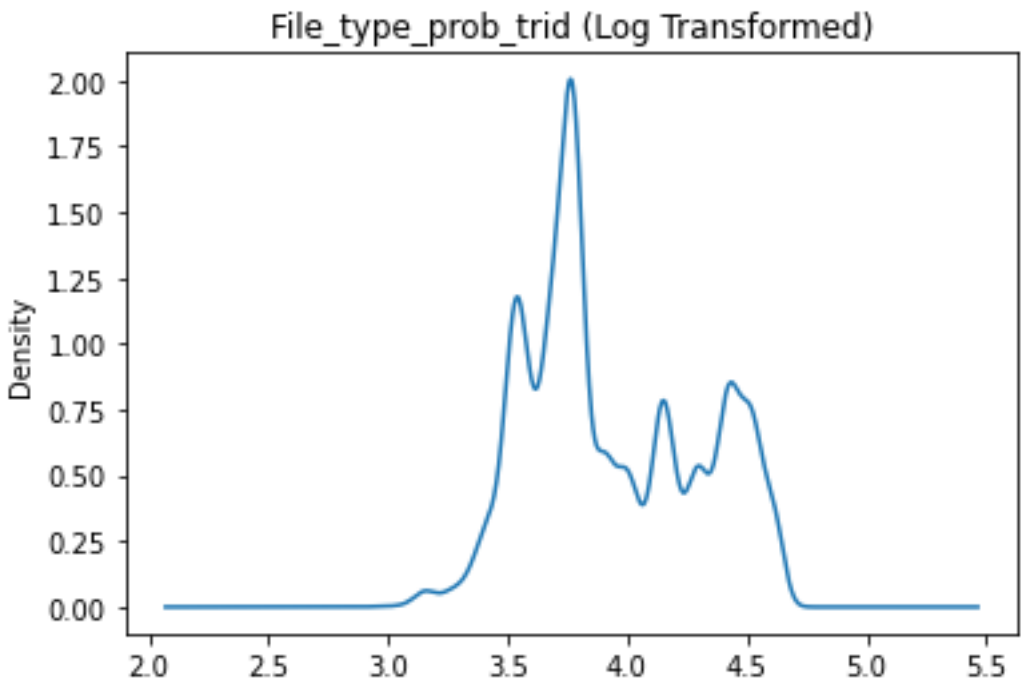
Appendix 1.1

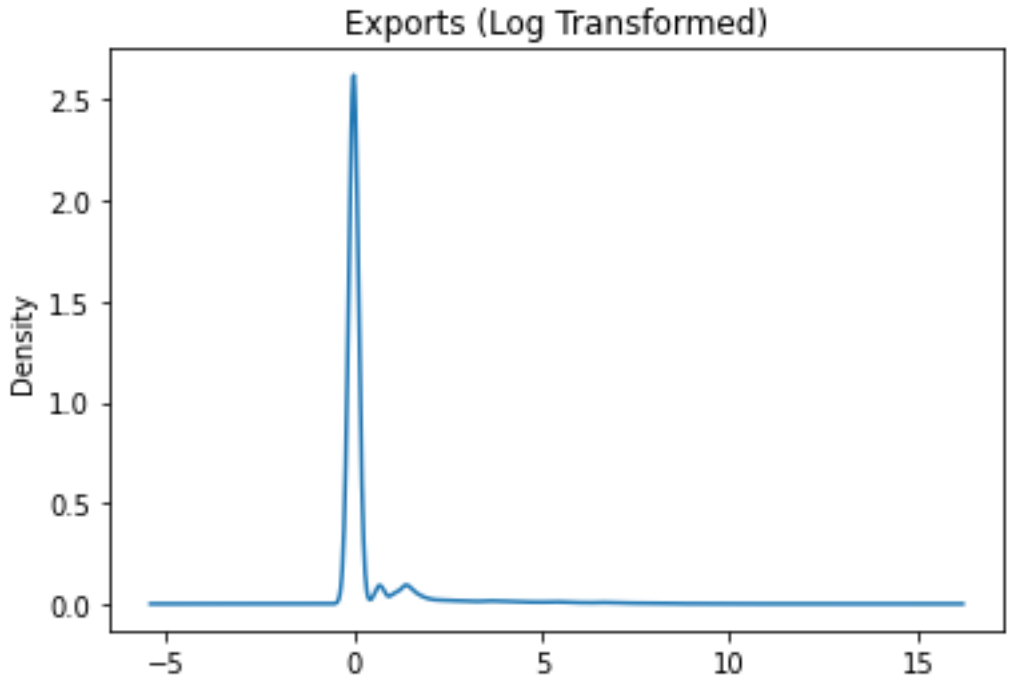


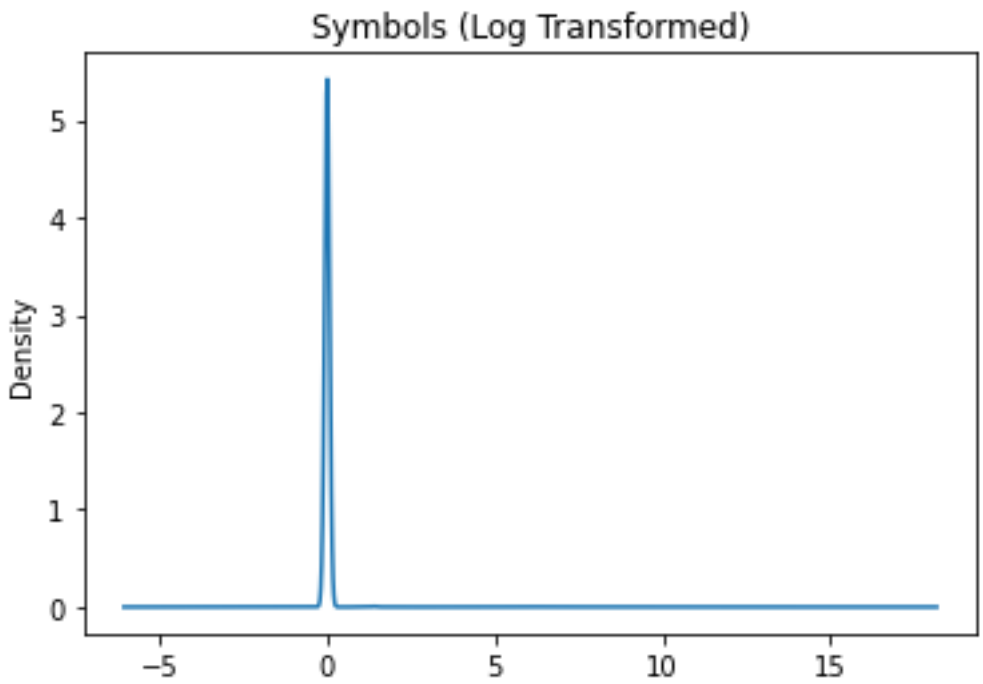
Appendix 1.2

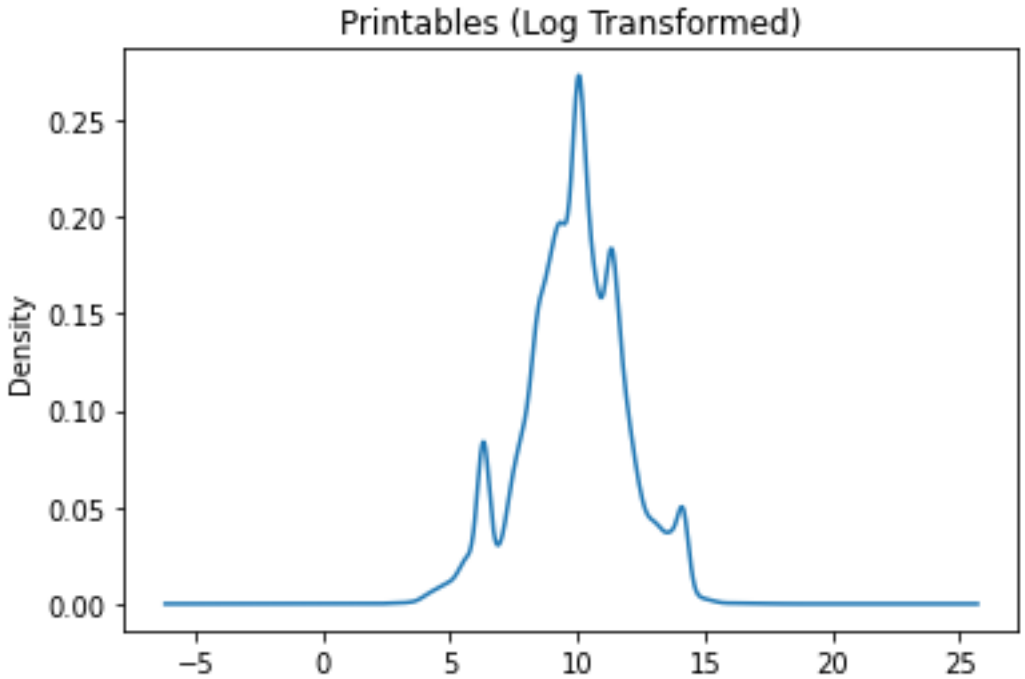


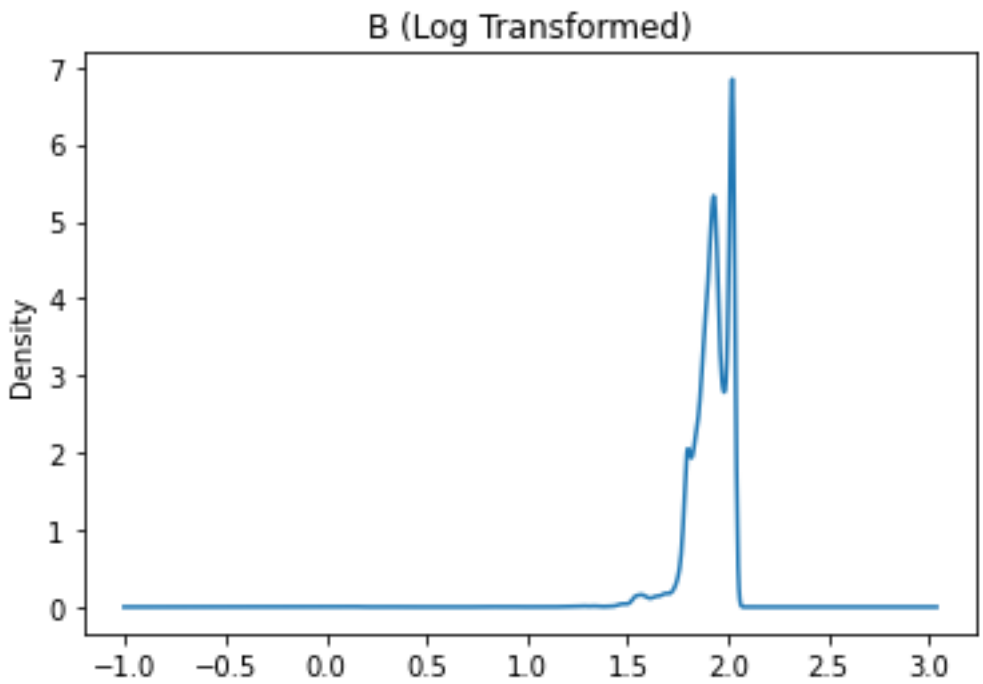
Appendix 1.2.15 – 1.2.1

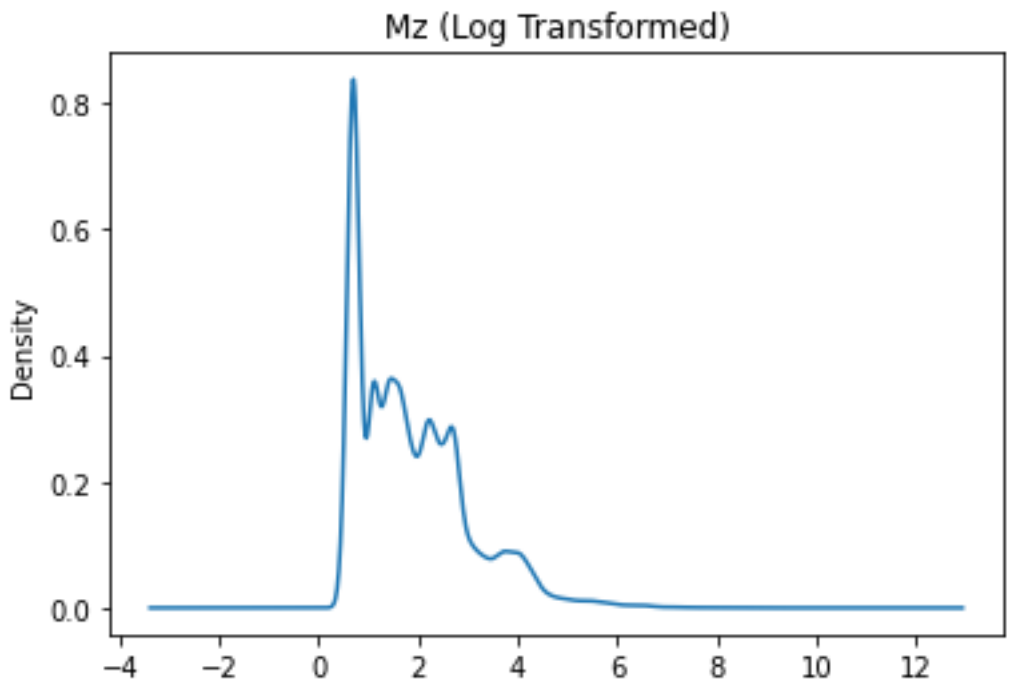

File_type_prob_trid (Log Transformed)
Density
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
2.0
2.5
3.0
3.5
4.0
4.5
5.0
5.5

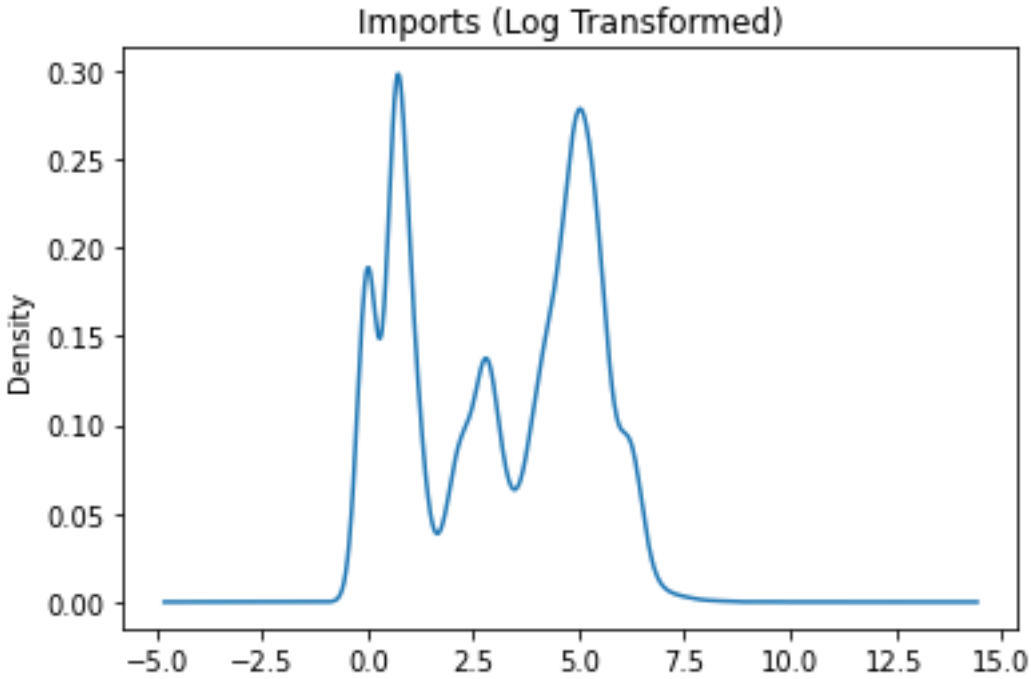

Exports (Log Transformed)
Density
2.5
2.0
1.5
1.0
0.5
0.0
−5
0
5
10
15

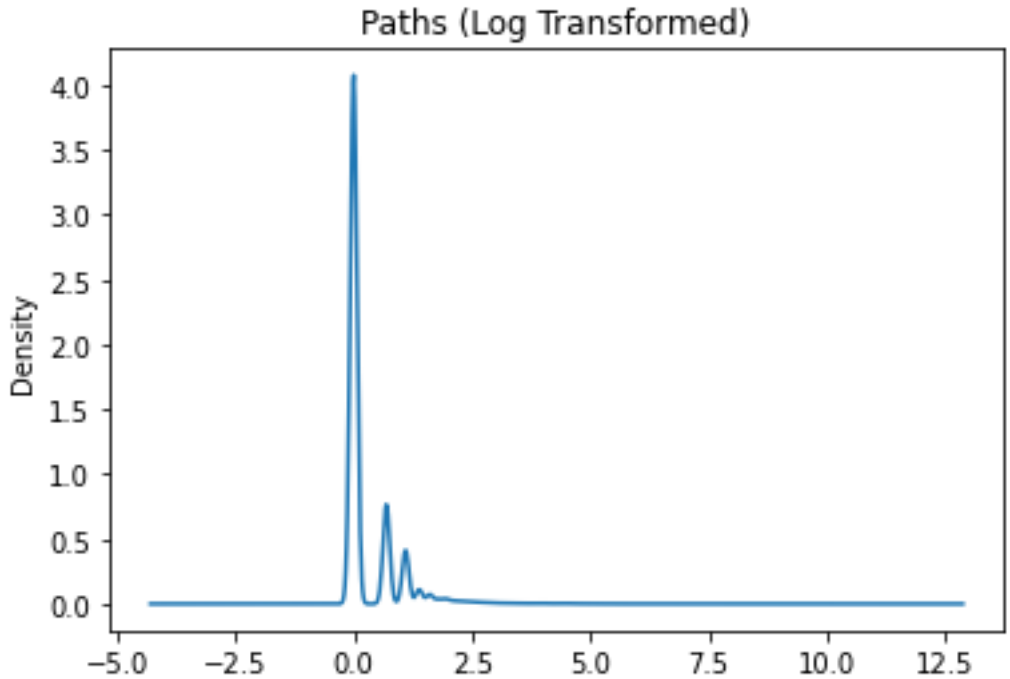

Symbols (Log Transformed)
Density
0
1
2
3
4
5
−5
0
5
10
15

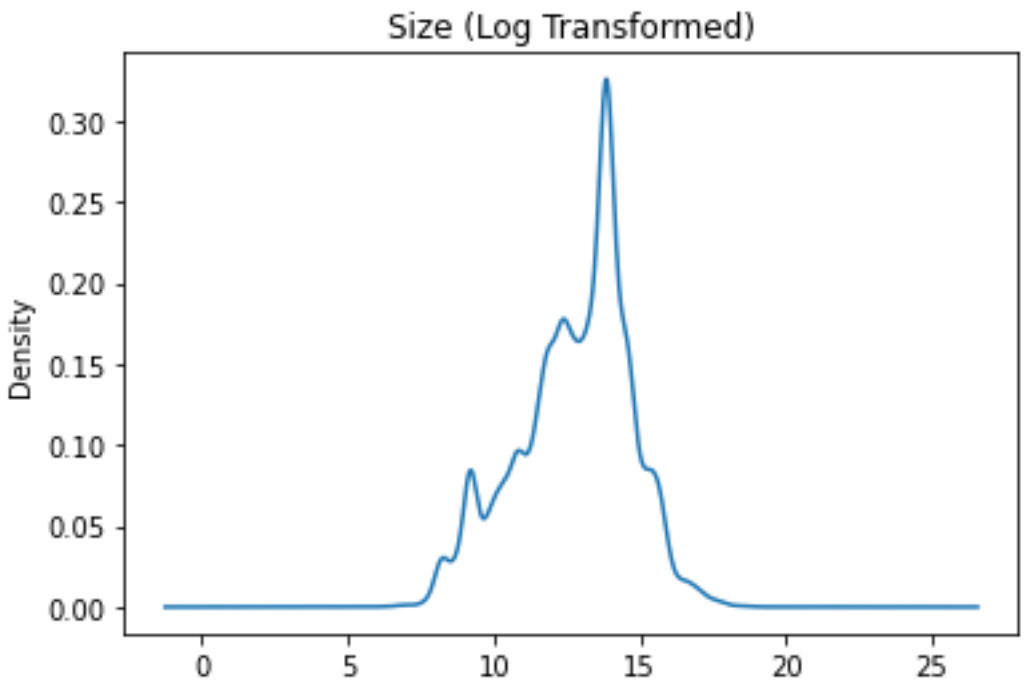

Printables (Log Transformed)
Density
0.00
0.05
0.10
0.15
0.20
0.25
−5
0
5
10
15
20
25

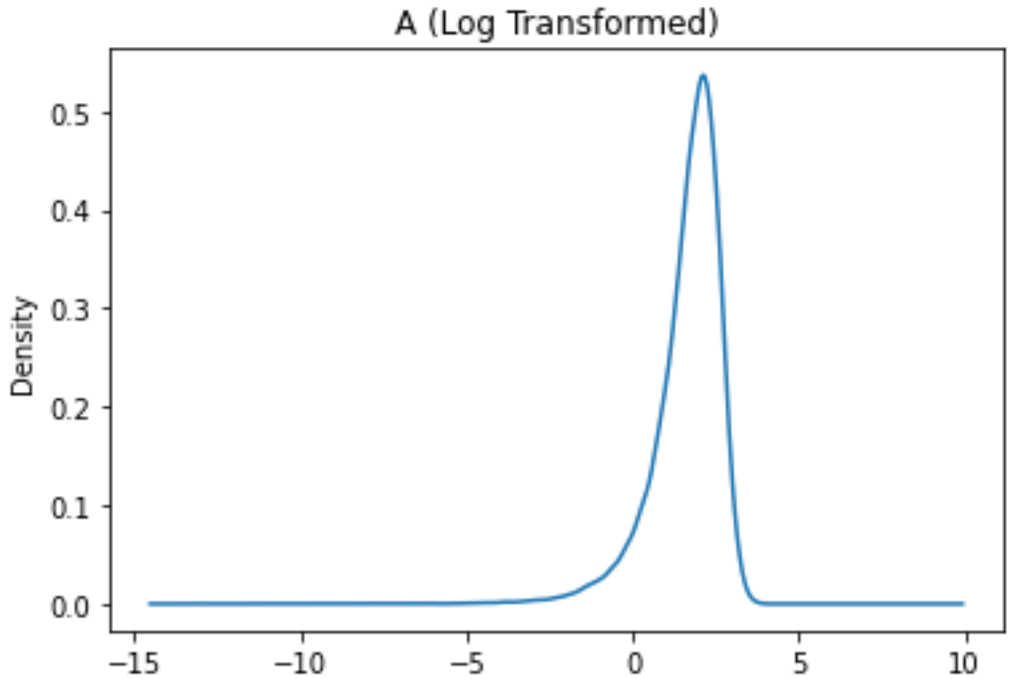

B (Log Transformed)
Density
7
6
5
4
3
2
1
0
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0

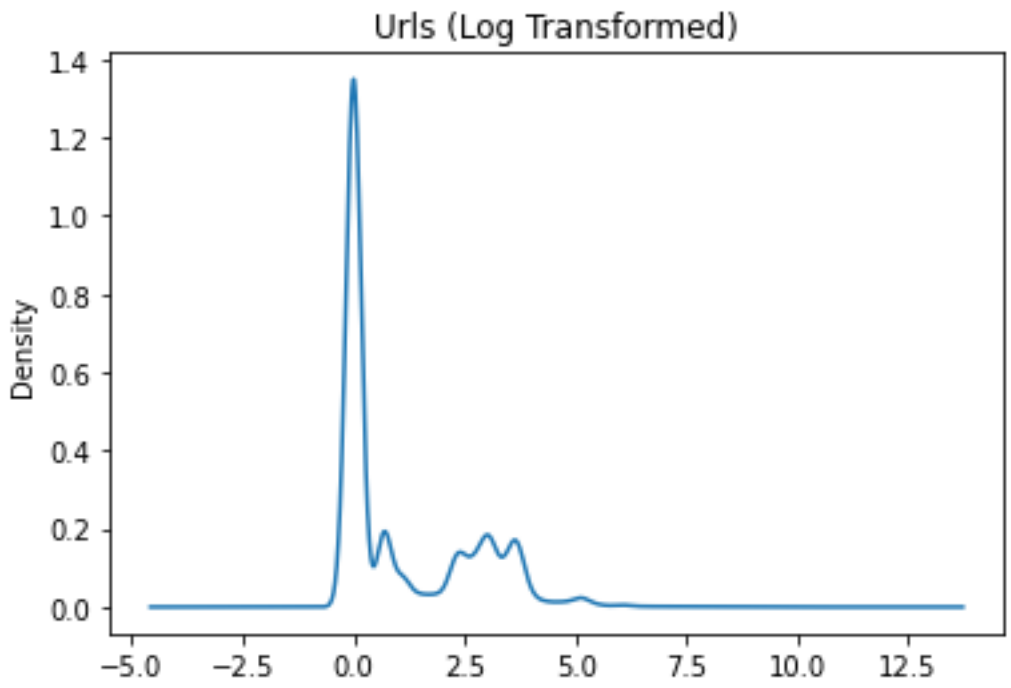

Mz (Log Transformed)
Density
0.8
0.6
0.4
0.2
0.0
−4
−2
0
2
4
6
8
10
12

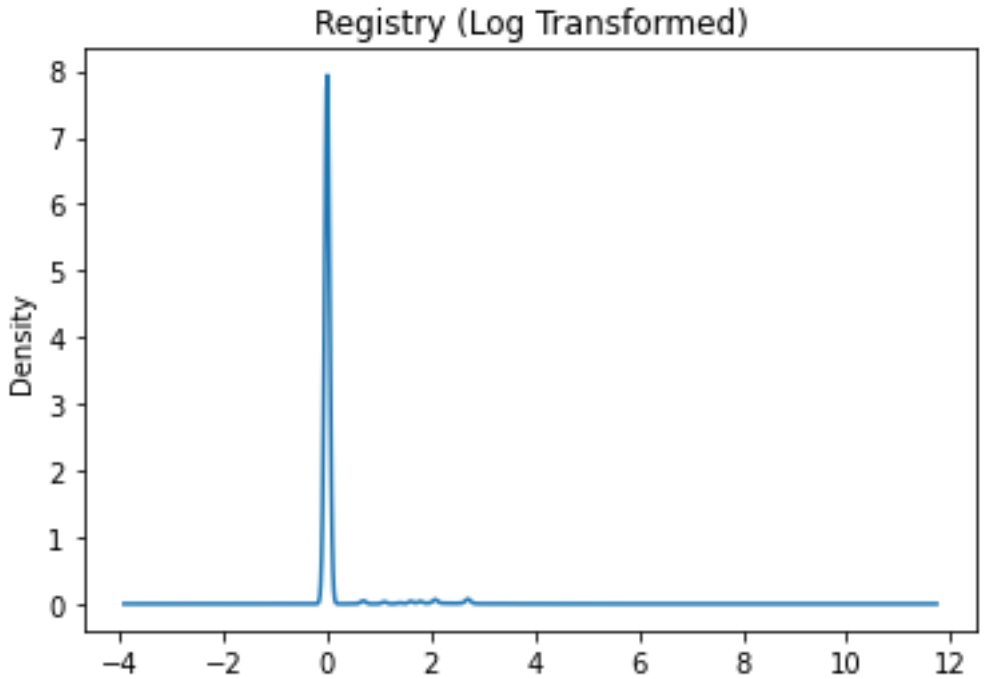

Imports (Log Transformed)
Density
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0

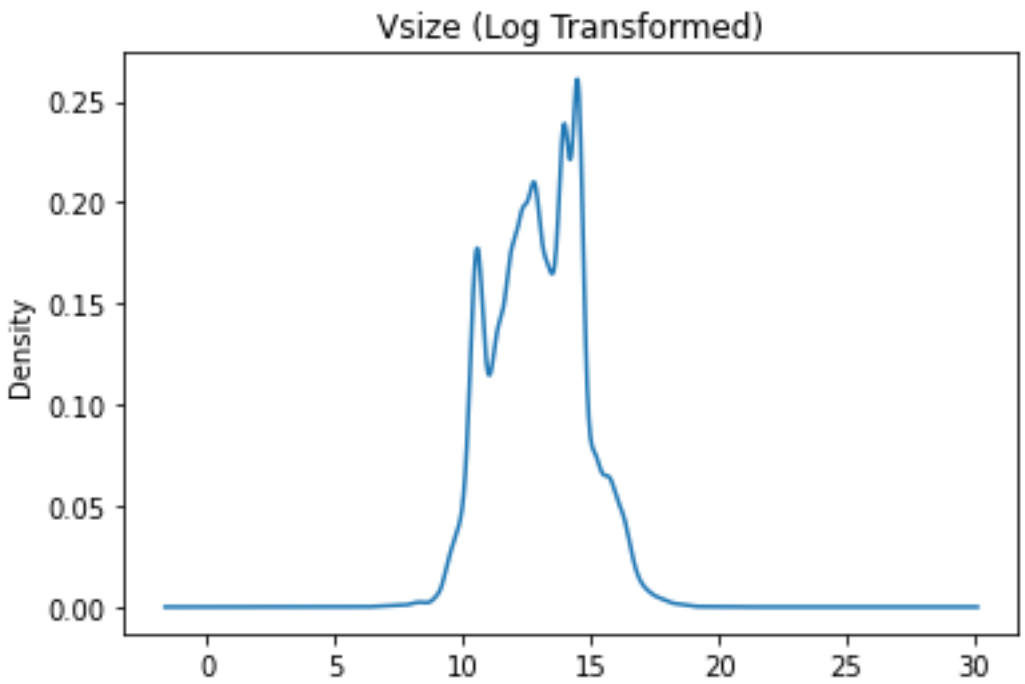

Paths (Log Transformed)
Density
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5

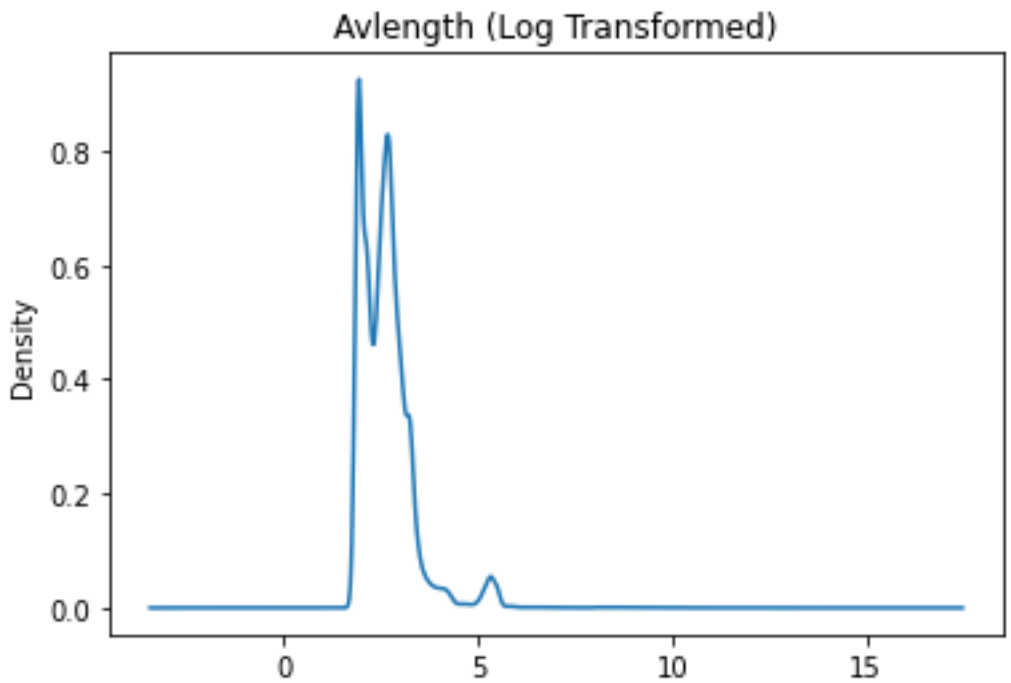

Size (Log Transformed)
Density
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
5
10
15
20
25

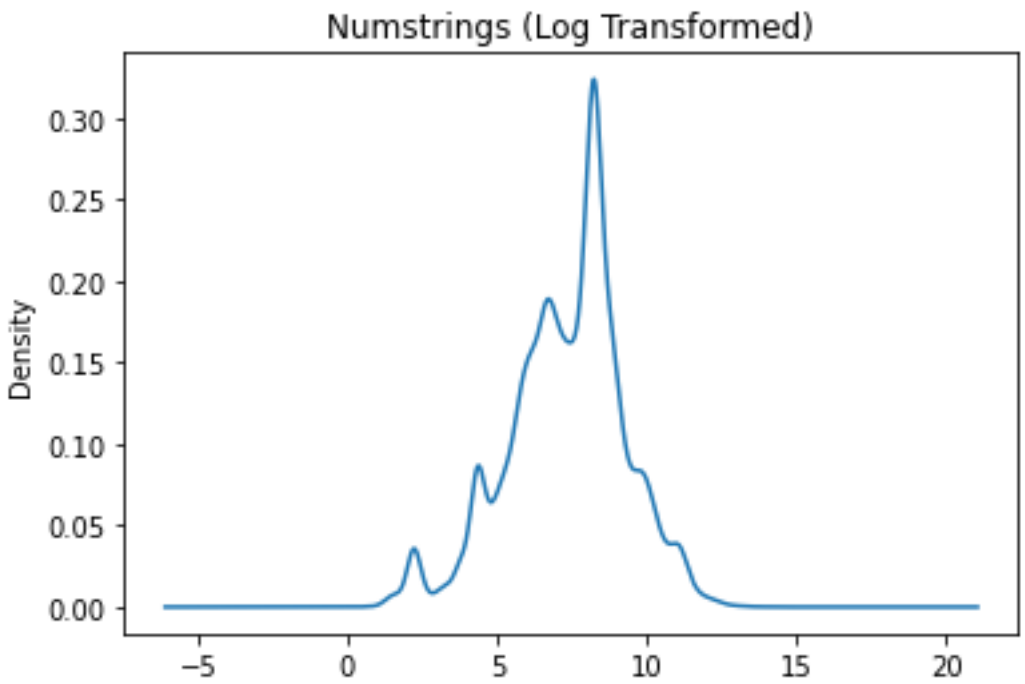

A (Log Transformed)
Density
0.5
0.4
0.3
0.2
0.1
0.0
−15
−10
−5
0
5
10


Urls (Log Transformed)
Density
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
12.5

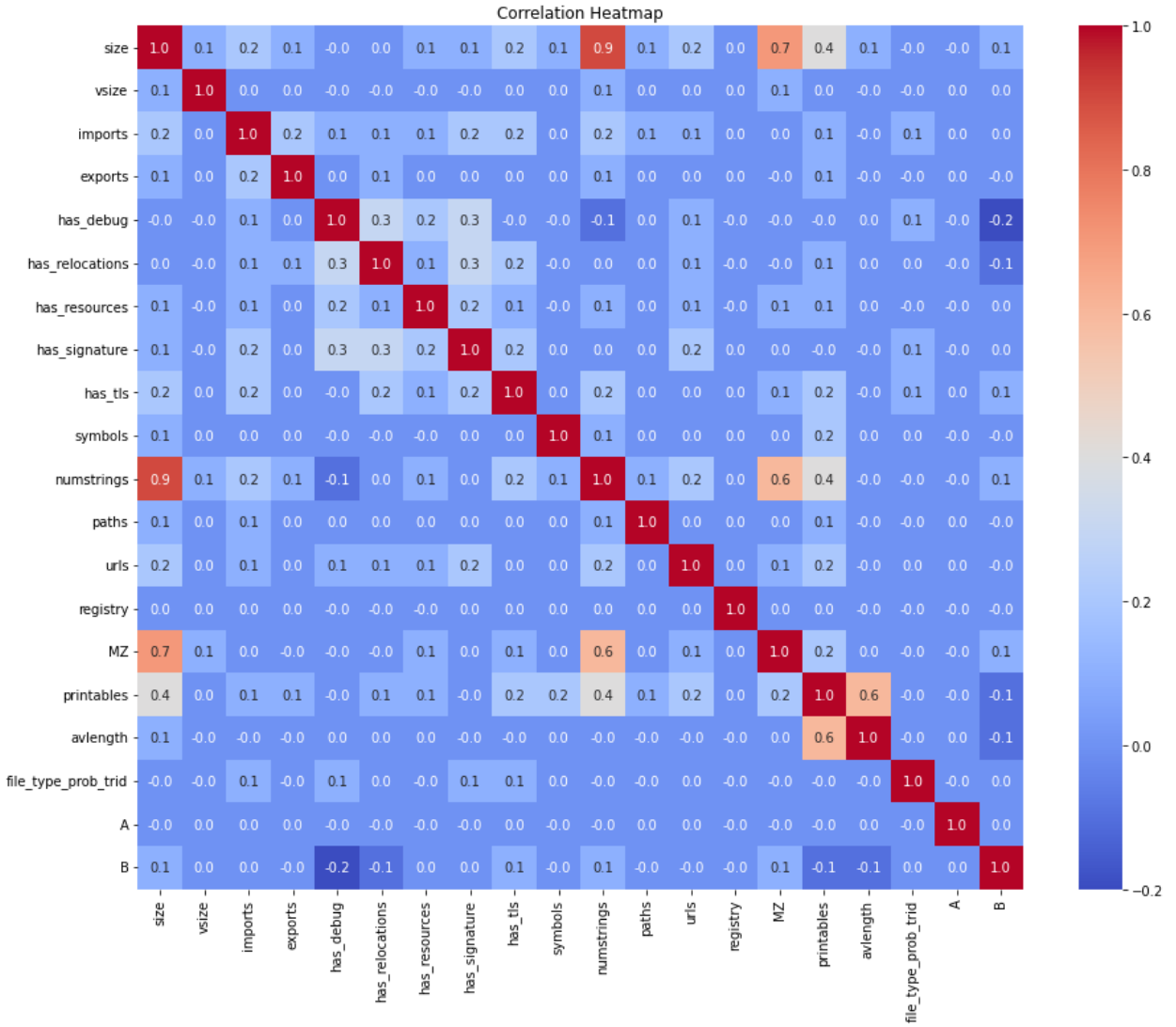

Registry (Log Transformed)
Density
8
7
6
5
4
3
2
1
0
−4
−2
0
2
4
6
8
10
12


Vsize (Log Transformed)
Density
0.25
0.20
0.15
0.10
0.05
0.00
0
5
10
15
20
25
30

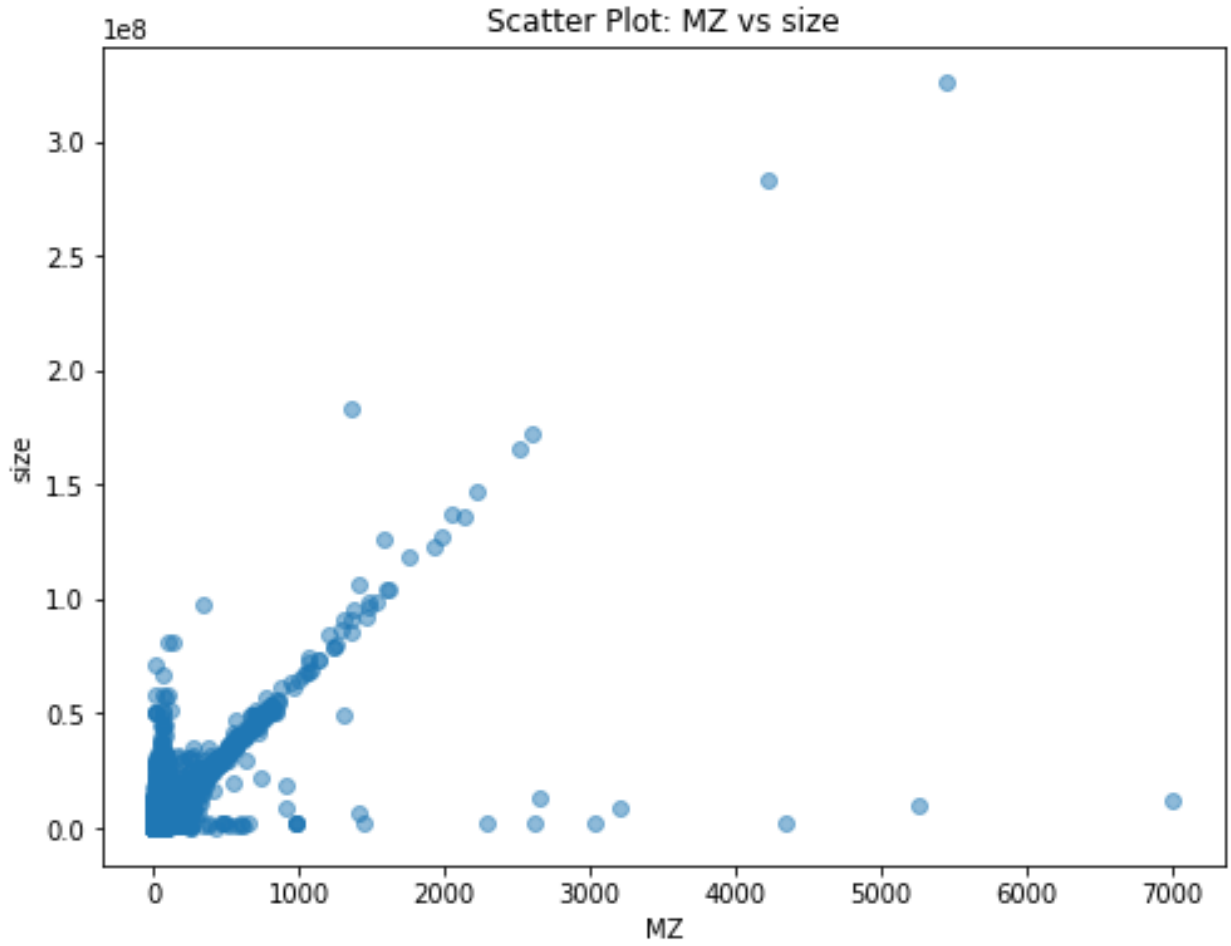

Avlength (Log Transformed)
Density
0.8
0.6
0.4
0.2
0.0
0
5
10
15

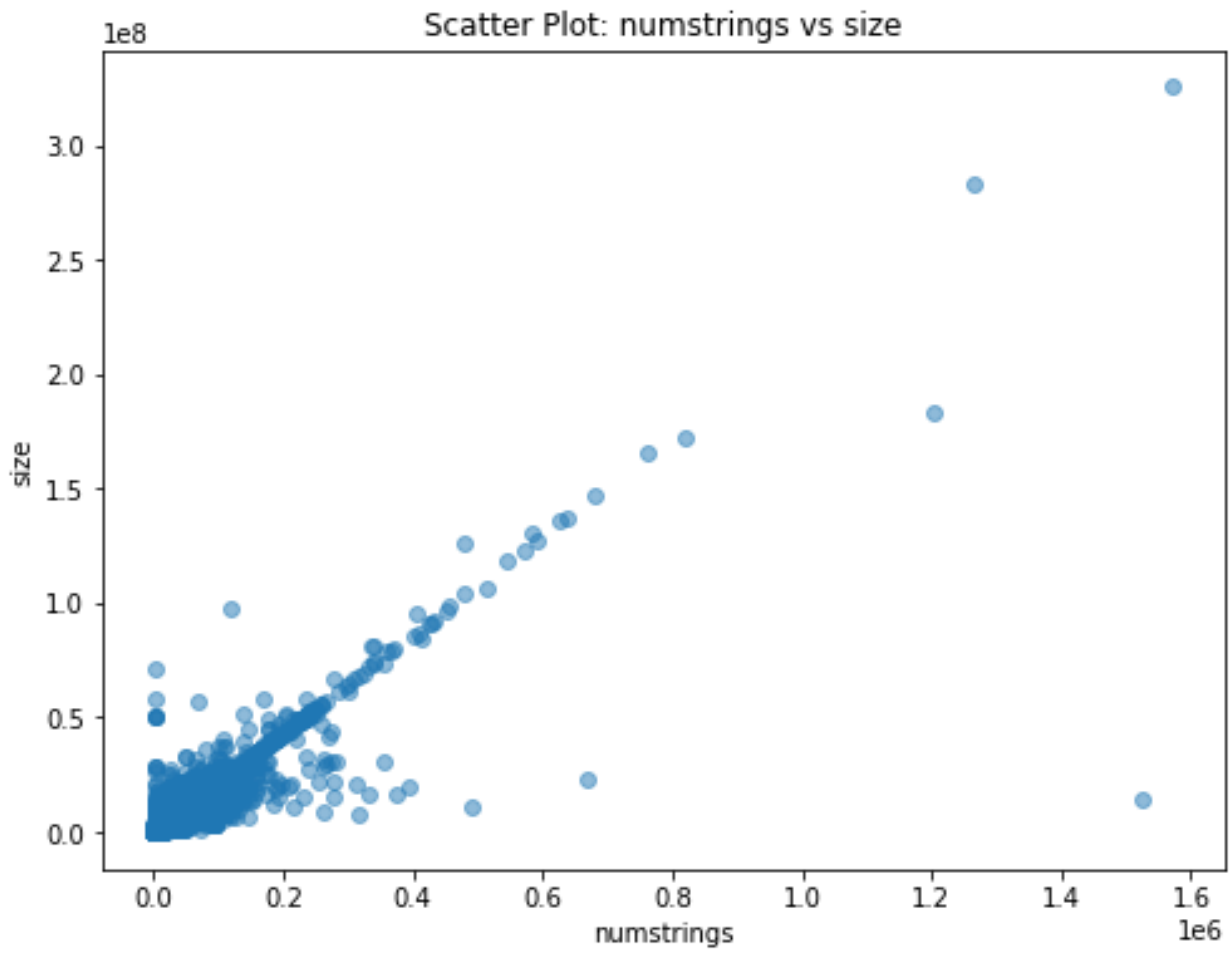

Numstrings (Log Transformed)
Density
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−5
0
5
10
15
20

Appendix 1.3

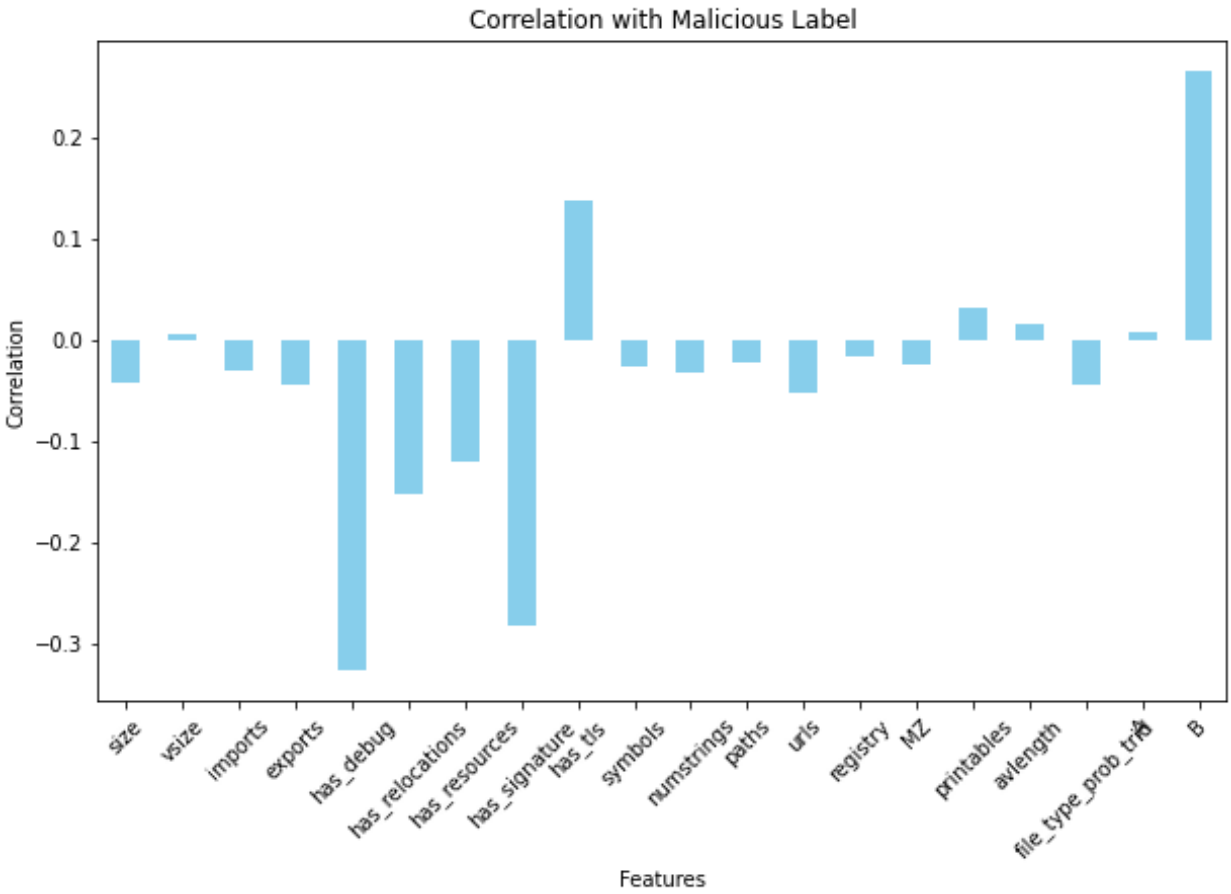

Correlation Heatmap
size 1.0 0.1 0.2 0.1 -0.0 0.0 0.1 0.1 0.2 0.1 0.9 0.1 0.2 0.0 0.7 0.4 0.1 -0.0 -0.0 0.1
vsize 0.1 1.0 0.0 0.0 -0.0 -0.0 -0.0 -0.0 0.0 0.0 0.1 0.0 0.0 0.0 0.1 0.0 -0.0 -0.0 0.0 0.0
imports 0.2 0.0 1.0 0.2 0.1 0.1 0.1 0.2 0.2 0.0 0.2 0.1 0.1 0.0 0.0 0.1 -0.0 0.1 0.0 0.0
exports 0.1 0.0 0.2 1.0 0.0 0.1 0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.0 -0.0 0.1 -0.0 -0.0 0.0 -0.0
has_debug -0.0 -0.0 0.1 0.0 1.0 0.3 0.2 0.3 -0.0 -0.0 -0.1 0.0 0.1 -0.0 -0.0 -0.0 0.0 0.1 -0.0 -0.2
has_relocations 0.0 -0.0 0.1 0.1 0.3 1.0 0.1 0.3 0.2 -0.0 0.0 0.0 0.1 -0.0 -0.0 0.1 0.0 0.0 -0.0 -0.1
has_resources 0.1 -0.0 0.1 0.0 0.2 0.1 1.0 0.2 0.1 -0.0 0.1 0.0 0.1 -0.0 0.1 0.1 0.0 -0.0 -0.0 0.0
has_signature 0.1 -0.0 0.2 0.0 0.3 0.3 0.2 1.0 0.2 0.0 0.0 0.0 0.2 0.0 0.0 -0.0 -0.0 0.1 -0.0 0.0
has_tls 0.2 0.0 0.2 0.0 -0.0 0.2 0.1 0.2 1.0 0.0 0.2 0.0 0.0 0.0 0.1 0.2 -0.0 0.1 0.0 0.1
symbols 0.1 0.0 0.0 0.0 -0.0 -0.0 -0.0 0.0 0.0 1.0 0.1 0.0 0.0 0.0 0.0 0.2 0.0 0.0 -0.0 -0.0
numstrings 0.9 0.1 0.2 0.1 -0.1 0.0 0.1 0.0 0.2 0.1 1.0 0.1 0.2 0.0 0.6 0.4 -0.0 -0.0 -0.0 0.1
paths 0.1 0.0 0.1 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.1 1.0 0.0 0.0 0.0 0.1 -0.0 -0.0 0.0 -0.0
urls 0.2 0.0 0.1 0.0 0.1 0.1 0.1 0.2 0.0 0.0 0.2 0.0 1.0 0.0 0.1 0.2 -0.0 0.0 0.0 -0.0
registry 0.0 0.0 0.0 0.0 -0.0 -0.0 -0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 -0.0 -0.0 -0.0 -0.0
MZ 0.7 0.1 0.0 -0.0 -0.0 -0.0 0.1 0.0 0.1 0.0 0.6 0.0 0.1 0.0 1.0 0.2 0.0 -0.0 -0.0 0.1
printables 0.4 0.0 0.1 0.1 -0.0 0.1 0.1 -0.0 0.2 0.2 0.4 0.1 0.2 0.0 0.2 1.0 0.6 -0.0 -0.0 -0.1
avlength 0.1 -0.0 -0.0 -0.0 0.0 0.0 0.0 -0.0 -0.0 0.0 -0.0 -0.0 -0.0 -0.0 0.0 0.6 1.0 -0.0 0.0 -0.1
file_type_prob_trid -0.0 -0.0 0.1 -0.0 0.1 0.0 -0.0 0.1 0.1 0.0 -0.0 -0.0 0.0 -0.0 -0.0 -0.0 -0.0 1.0 -0.0 0.0
A -0.0 0.0 0.0 0.0 -0.0 -0.0 -0.0 -0.0 0.0 -0.0 -0.0 0.0 0.0 -0.0 -0.0 -0.0 0.0 -0.0 1.0 0.0
B 0.1 0.0 0.0 -0.0 -0.2 -0.1 0.0 0.0 0.1 -0.0 0.1 -0.0 -0.0 -0.0 0.1 -0.1 -0.1 0.0 0.0 1.0
size vsize imports exports has_debug has_relocations has_resources has_signature has_tls symbols numstrings paths urls registry MZ printables avlength file_type_prob_trid A B
1.0 0.8 0.6 0.4 0.2 0.0 −0.2

Appendix 1.4.2 – 1.4.1

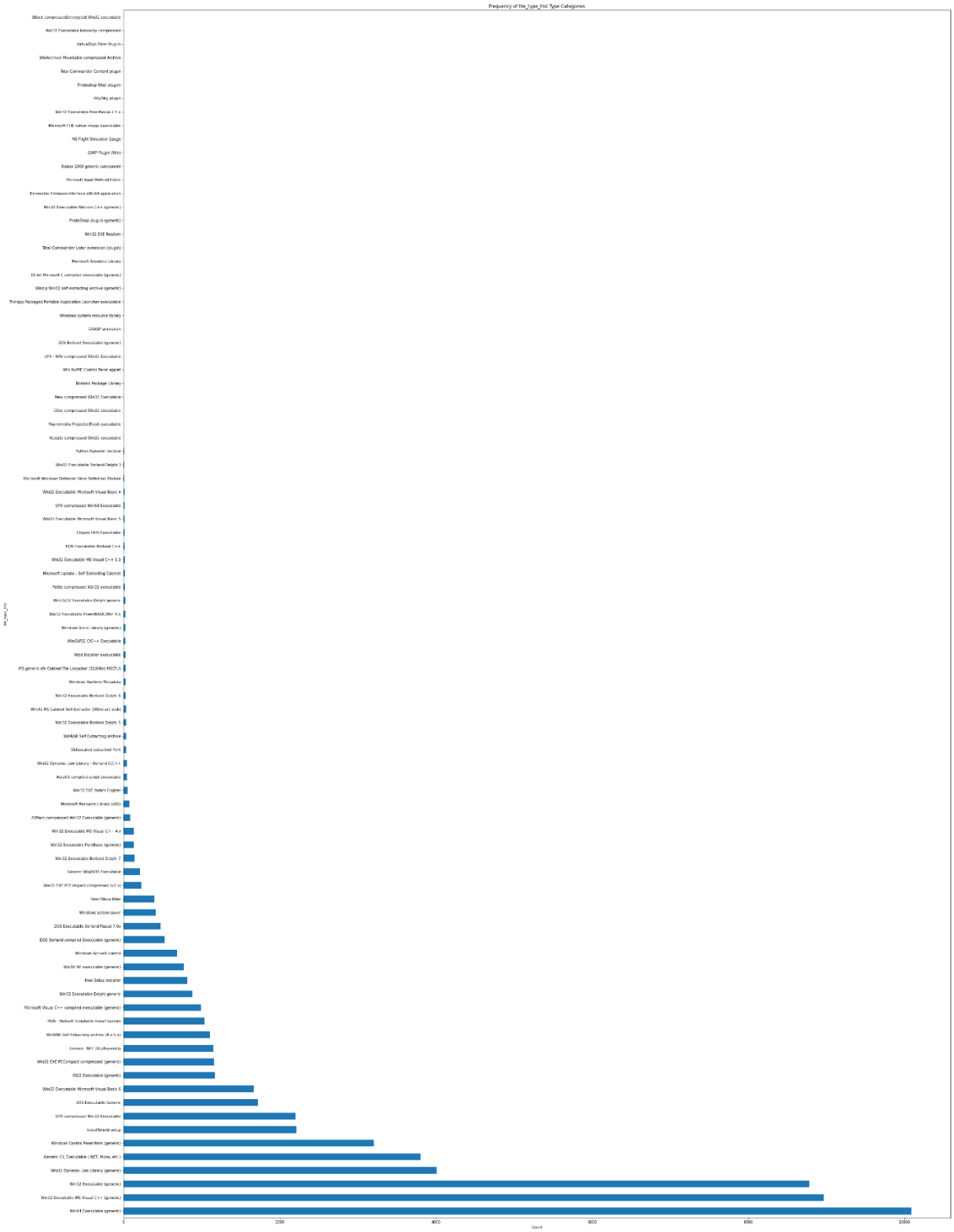



Scatter Plot: numstrings vs size
1e8
3.0
2.5
2.0
1.5
1.0
0.5
0.0
size
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
numstrings
1e6

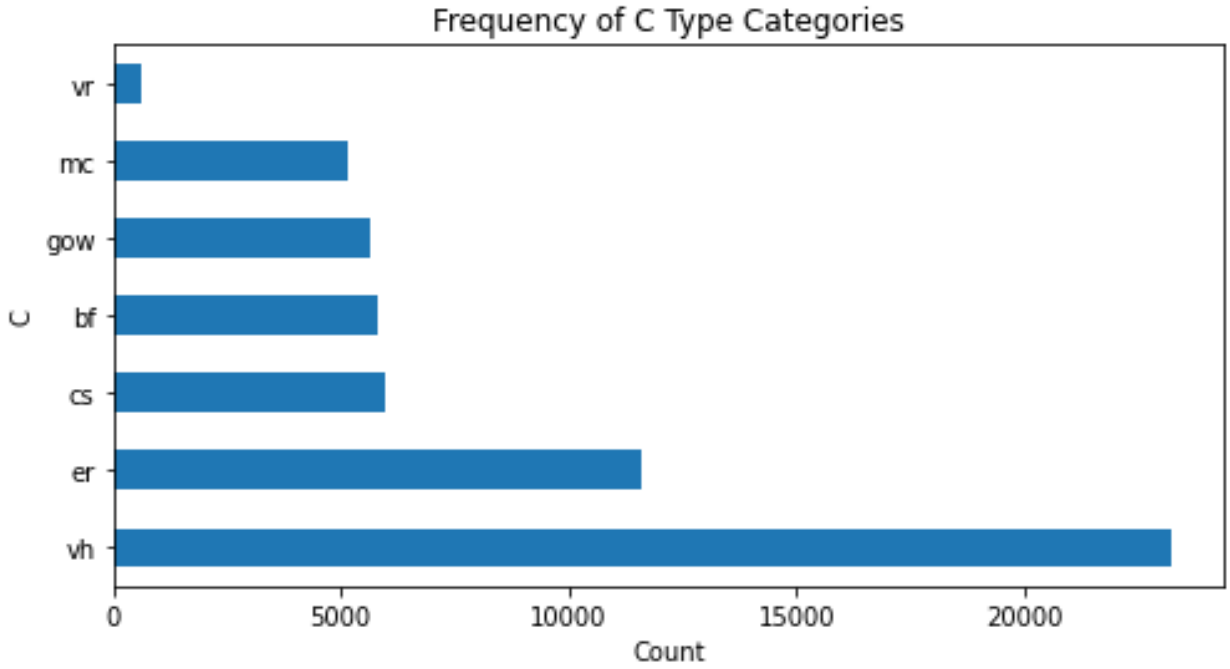
Appendix 1.5


Correlation with Malicious Label
Correlation
0.2
0.1
0.0
−0.1
−0.2
−0.3
size
vsize
imports
exports
has_debug
has_relocations
has_resources
has_signature
has_tls
symbols
numstrings
paths
urls
registry
MZ
printables
avlength
file_type_prob_trid
B
Features

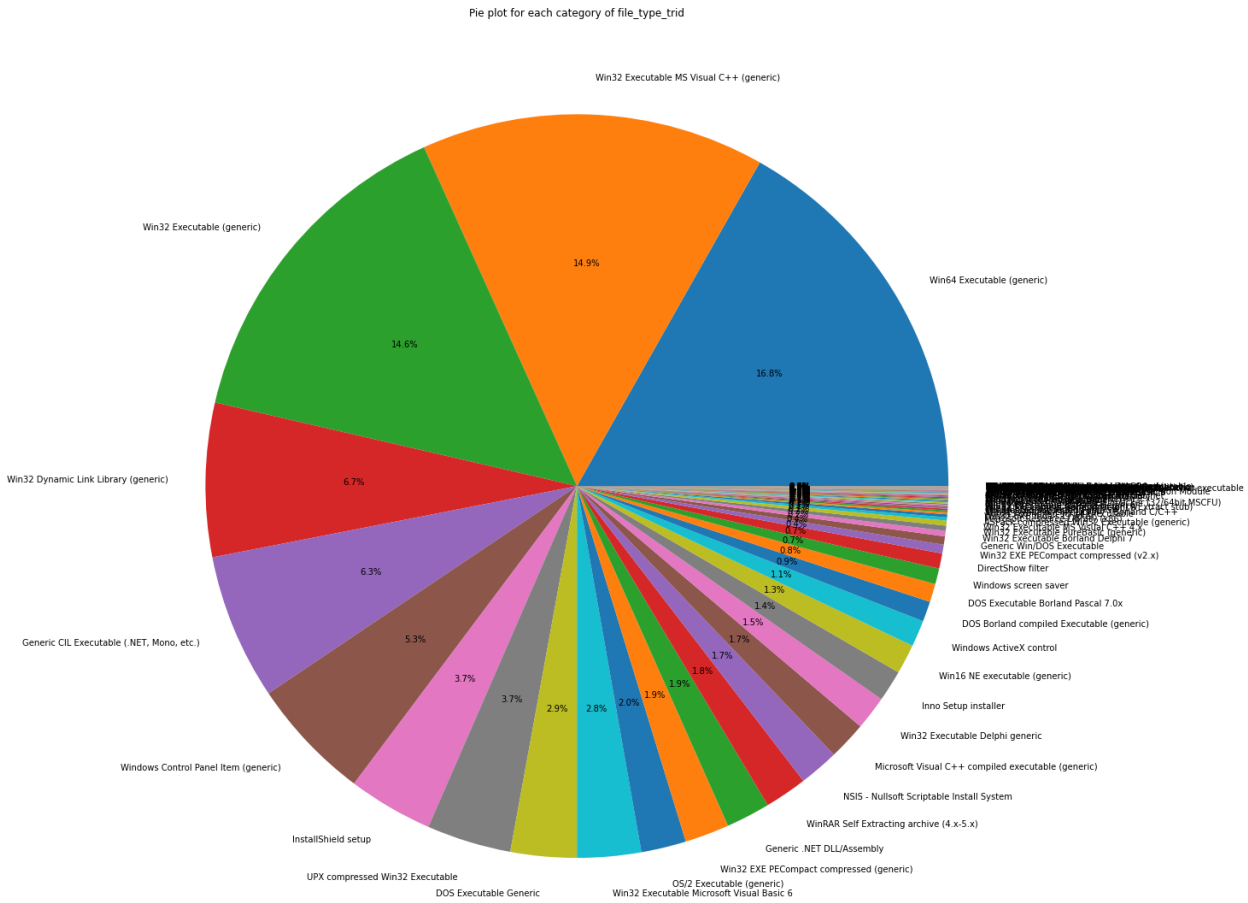
Appendix 1.6.1


Frequency of file_type_trid Type Categories
EBlock compressed/encrypted Win32 executable
Win32 Executable Blinuxhy compressed
VirtualDub filter Plug-in
WinArchiver Mountable compressed Archive
Total Commander Content plugin
Photoshop filter plug-in
OllyDbg plugin
Win32 Executable Free Pascal 2.4 x
Microsoft CLR native image executable
MS Flight Simulator Gauge
GIMP Plug-in (Win)
Rieder 2000 generic component
Microsoft Input Method Editor
Extensible Firmware Interface x86-64 application
Win32 Executable Watcom C++ (generic)
PhotoShop plug-in (generic)
Win32 EXE ResArm
Total Commander Lister extension (plugin)
Microsoft Resource Library
16-bit Microsoft C compiled executable (generic)
Winzip Win32 self-extracting archive (generic)
Thinapp Packaged Portable Application Launcher executable
Windows system resource library
GRASP animation
DCS Borland Executable (generic)
UPX - NRV compressed Win32 Executable
Win 9x/NE Control Panel applet
Borland Package Library
Neo compressed Win32 Executable
CDec compressed Win32 executable
Macromedia Projector/Flash executable
Nsis/lzma compressed Win32 executable
Python Dynamic module
Win32 Executable Borland Delphi 5
Microsoft Windows Defender Virus Definition Module
Win32 Executable Microsoft Visual Basic 4
UPX compressed Win64 Executable
Win32 Executable Microsoft Visual Basic 5
Clipper DOS Executable
DOS Executable Borland C++
Win32 Executable MS Visual C++ 5.0
Microsoft Update - Self Extracting Cabinet
Petite compressed Win32 executable
Win16/32 Executable Delphi generic
Win32 Executable PowerBASIC/Win 9.x
Windows Icons Library (generic)
MinGW32 C/C++ Executable
Wise Installer executable
MS generic cfb Cabinet File Unpacker (32/64bit MSCF.h)
Windows Runtime Metadata
Win32 Executable Borland Delphi 6
Win32 MS Cabinet Self-Extractor (Wibo/cat stub)
Win32 Executable Borland Delphi 5
WinRAR Self Extracting archive
Obfuscated subscribed Perl
Win32 Dynamic Link Library - Borland C/C++
AutoIt3 compiled script executable
Win32 EXE Yoda's Crypter
Microsoft Resource Library (x86)
ASPack compressed Win32 Executable (generic)
Win32 Executable MS Visual C++ 4.x
Win32 Executable PureBasic (generic)
Win32 Executable Borland Delphi 7
Generic MacOS/X Executable
Win32 EXE PECompact compressed (v2 x)
SmartWave filter
Windows screen saver
DOS Executable Borland Pascal 7.0x
DOS Borland compiled Executable (generic)
Windows Activeh control
Win16 NE executable (generic)
Inno Setup Installer
Win32 Executable Delphi generic
Microsoft Visual C++ compiled executable (generic)
NSIS - Nullsoft Scriptable Install System
WinRAR Self Extracting archive (4 x-5 x)
Generic .NET DLL/Assembly
Win32 EXE PECompact compressed (generic)
OS/2 Executable (generic)
Win32 Executable Microsoft Visual Basic 6
DOS Executable Generic
UPX compressed Win32 Executable
InstallShield setup
Windows Control Panel Item (generic)
Generic CIL Executable (.NET, Mono, etc.)
Win32 Dynamic Link Library (generic)
Win32 Executable (generic)
Win32 Executable MS Visual C++ (generic)
Win64 Executable (generic)
file_type_trid
0
2000
4000
6000
8000
10000
Count

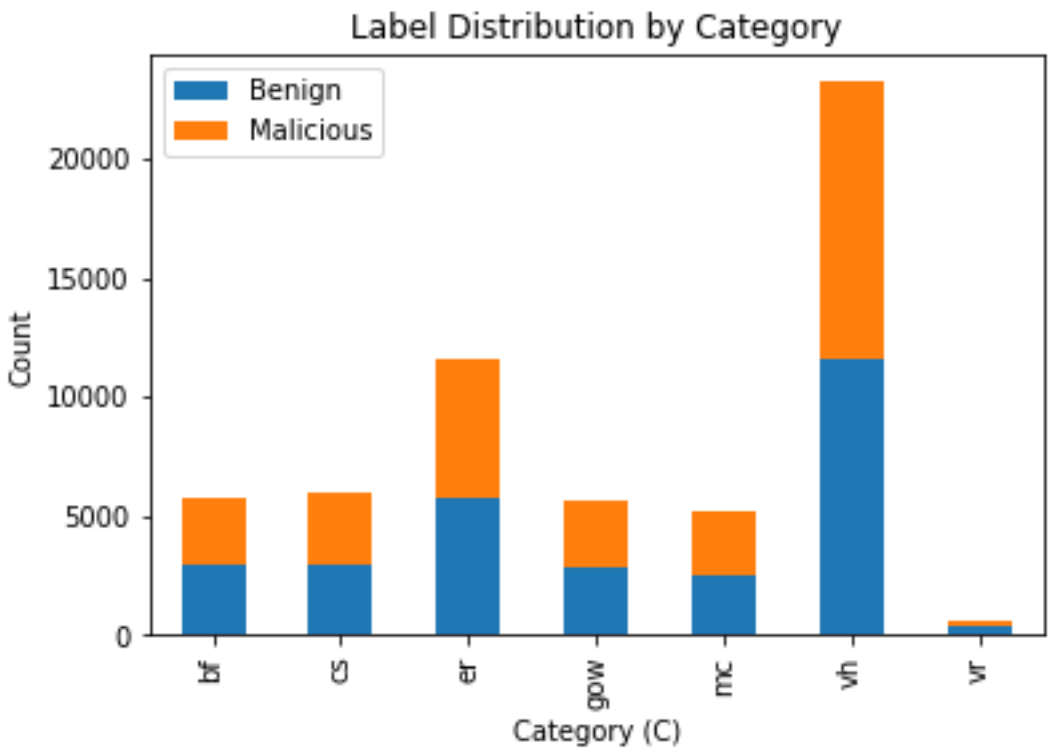
Appendix 1.6.2



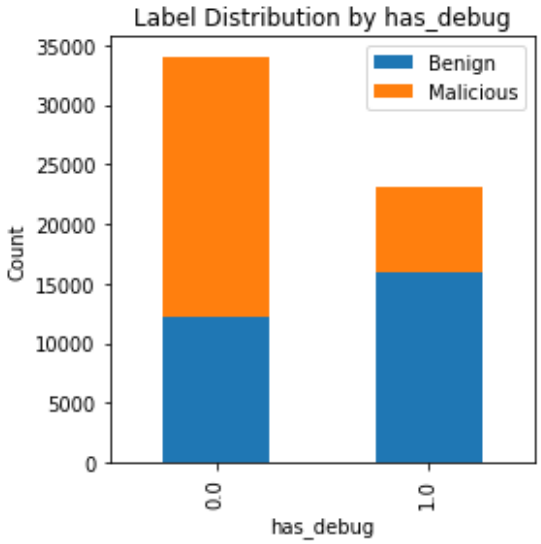
נספח 1.7

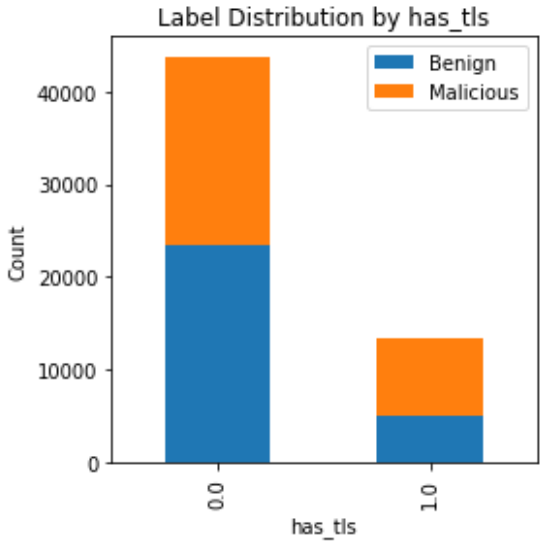


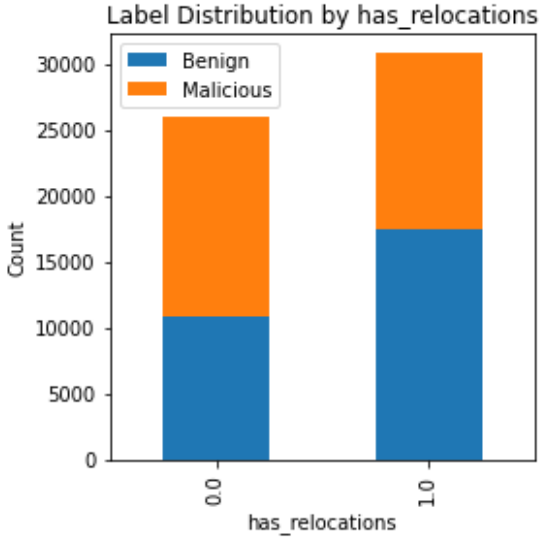
Appendix 1.8

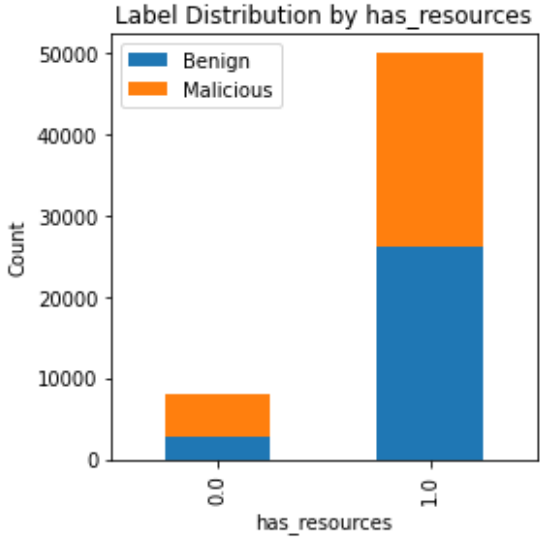

Label Distribution by Category
Benign
Malicious
Count
0
5000
10000
15000
20000
bf
cs
er
gow
mc
vh
vr
Category (C)

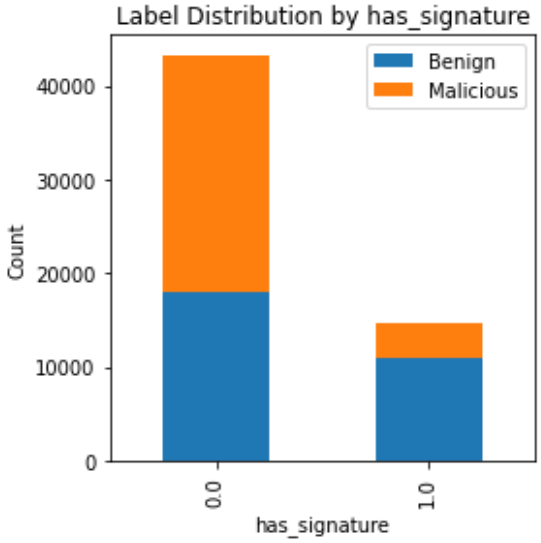
Appendix 1.9



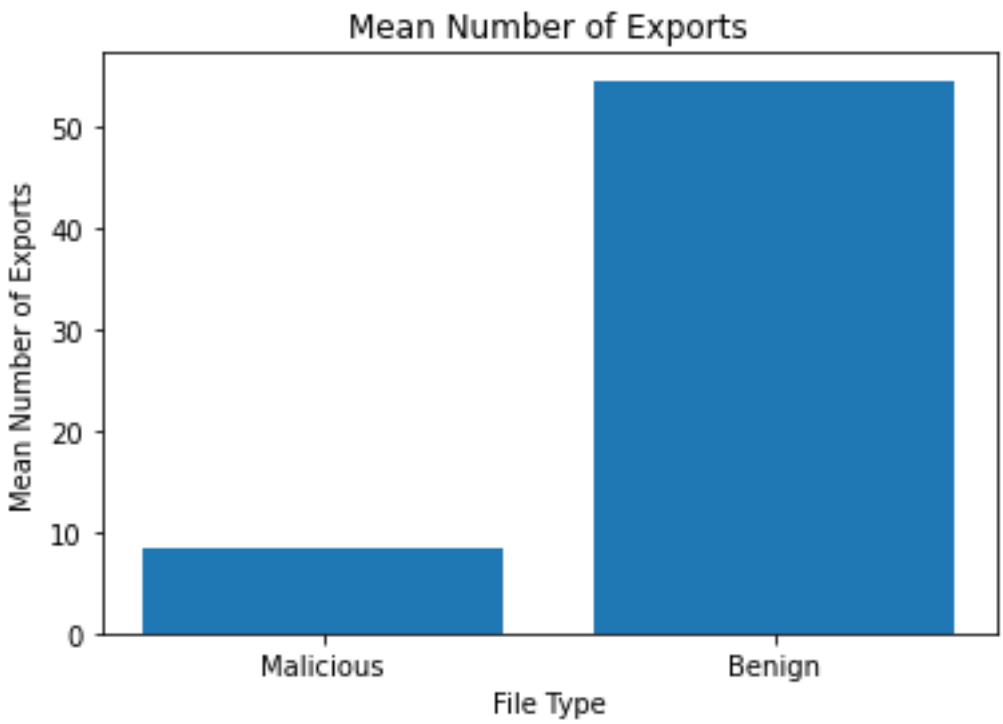


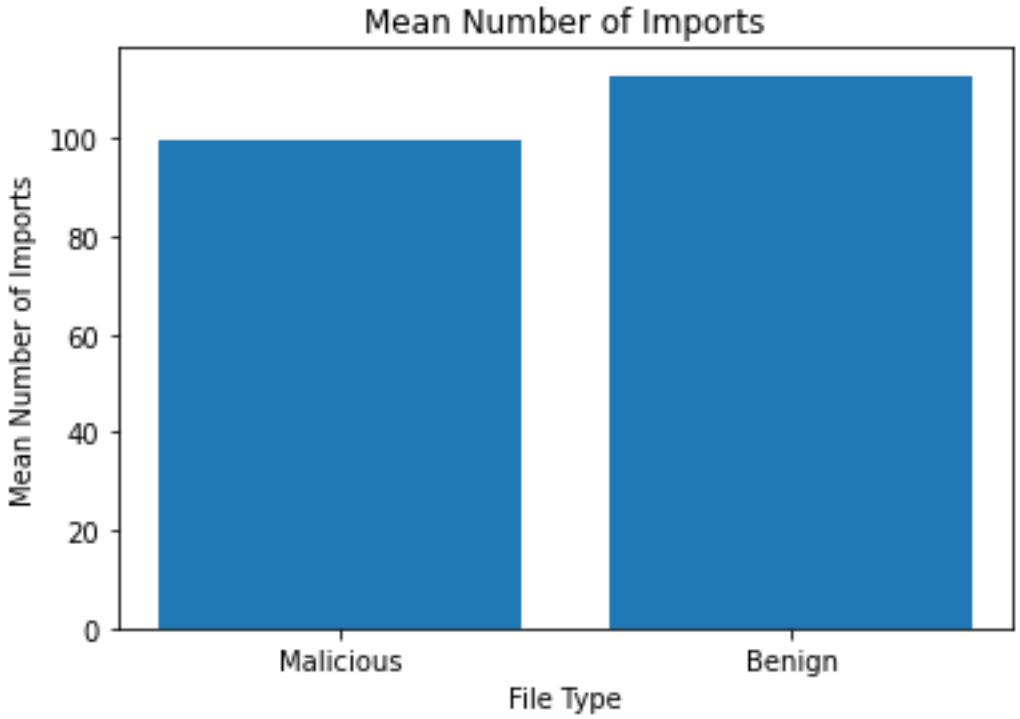


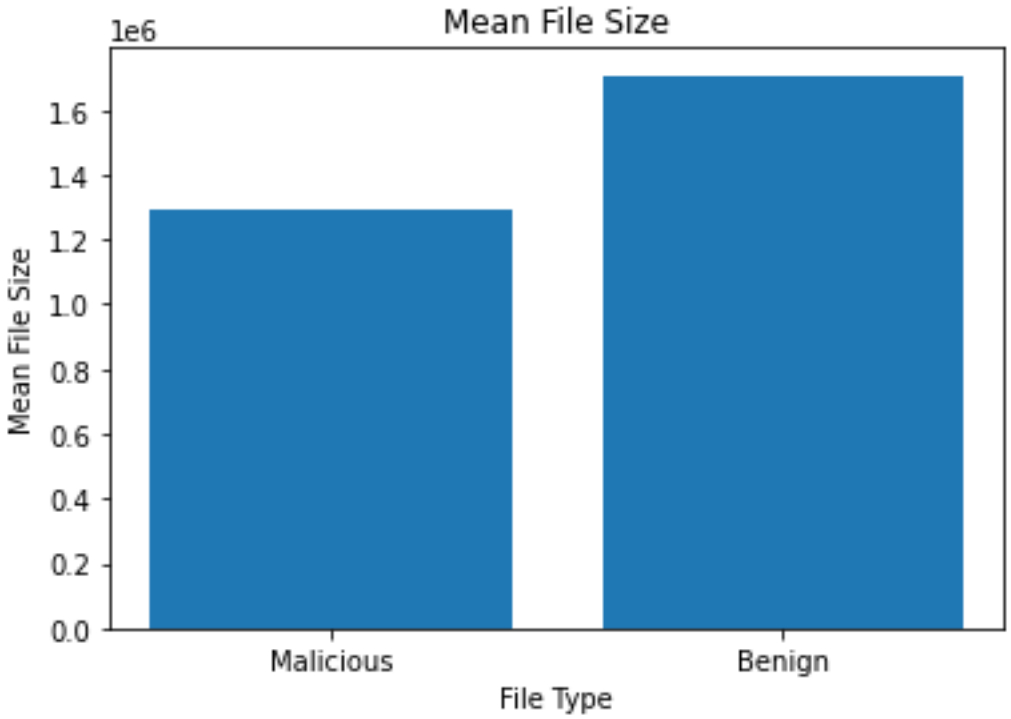


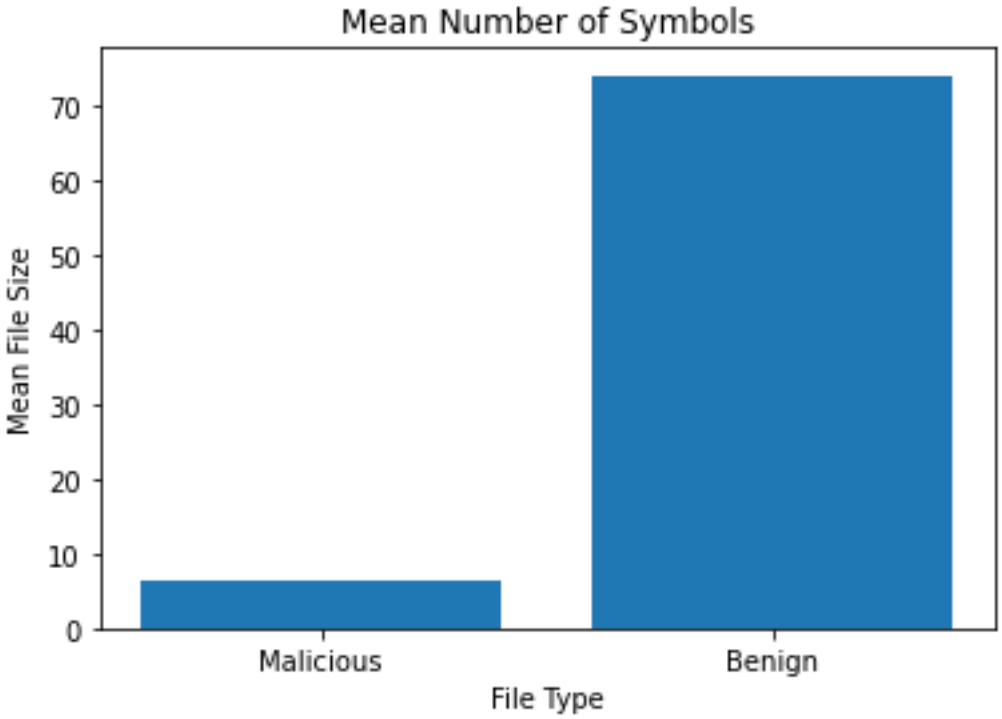


Appendix 1.10.4 – 1.10.1

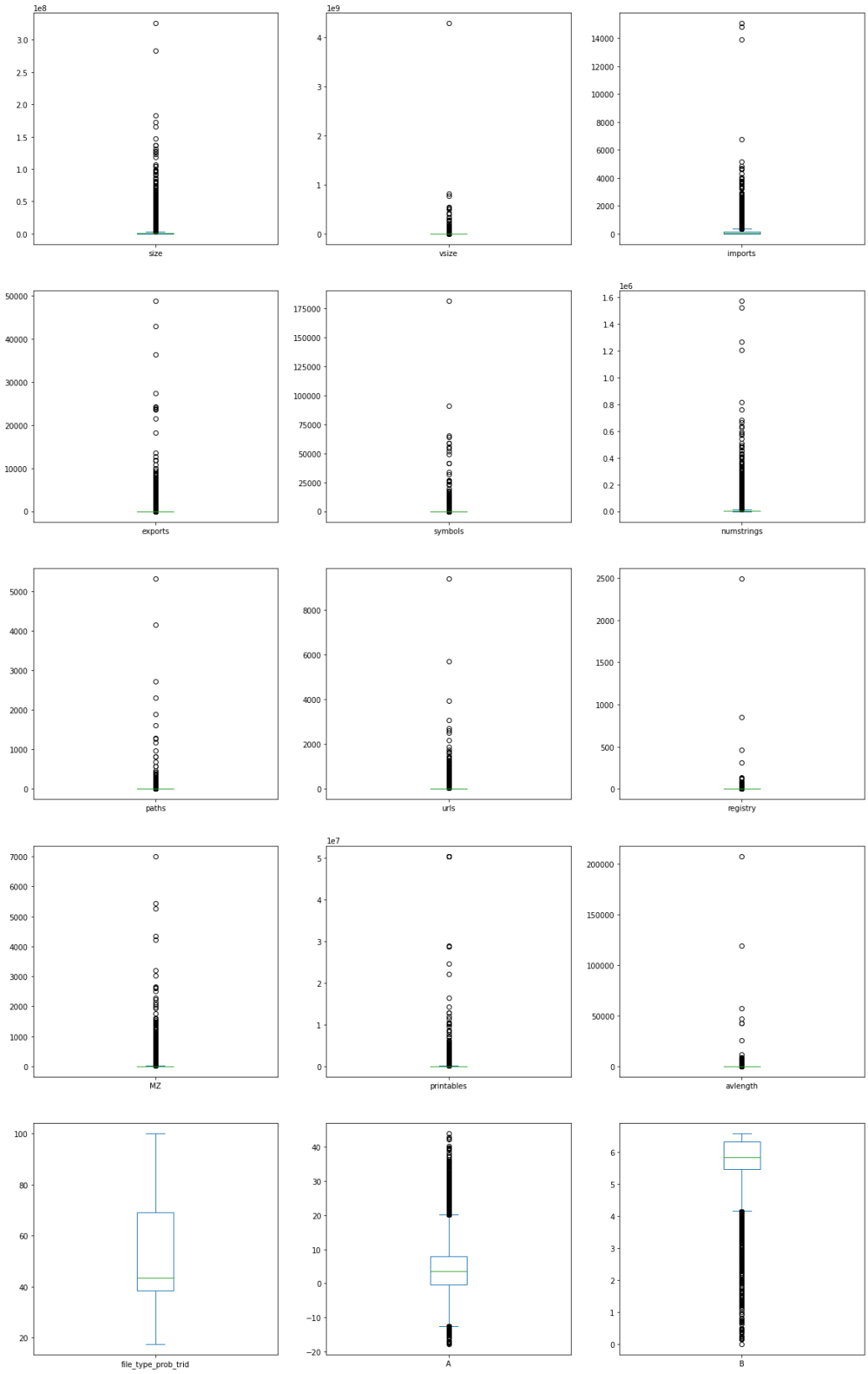

Mean Number of Exports
Mean Number of Exports
0
10
20
30
40
50
Malicious
Benign
File Type


Mean Number of Imports
Mean Number of Imports
0
20
40
60
80
100
Malicious
Benign
File Type

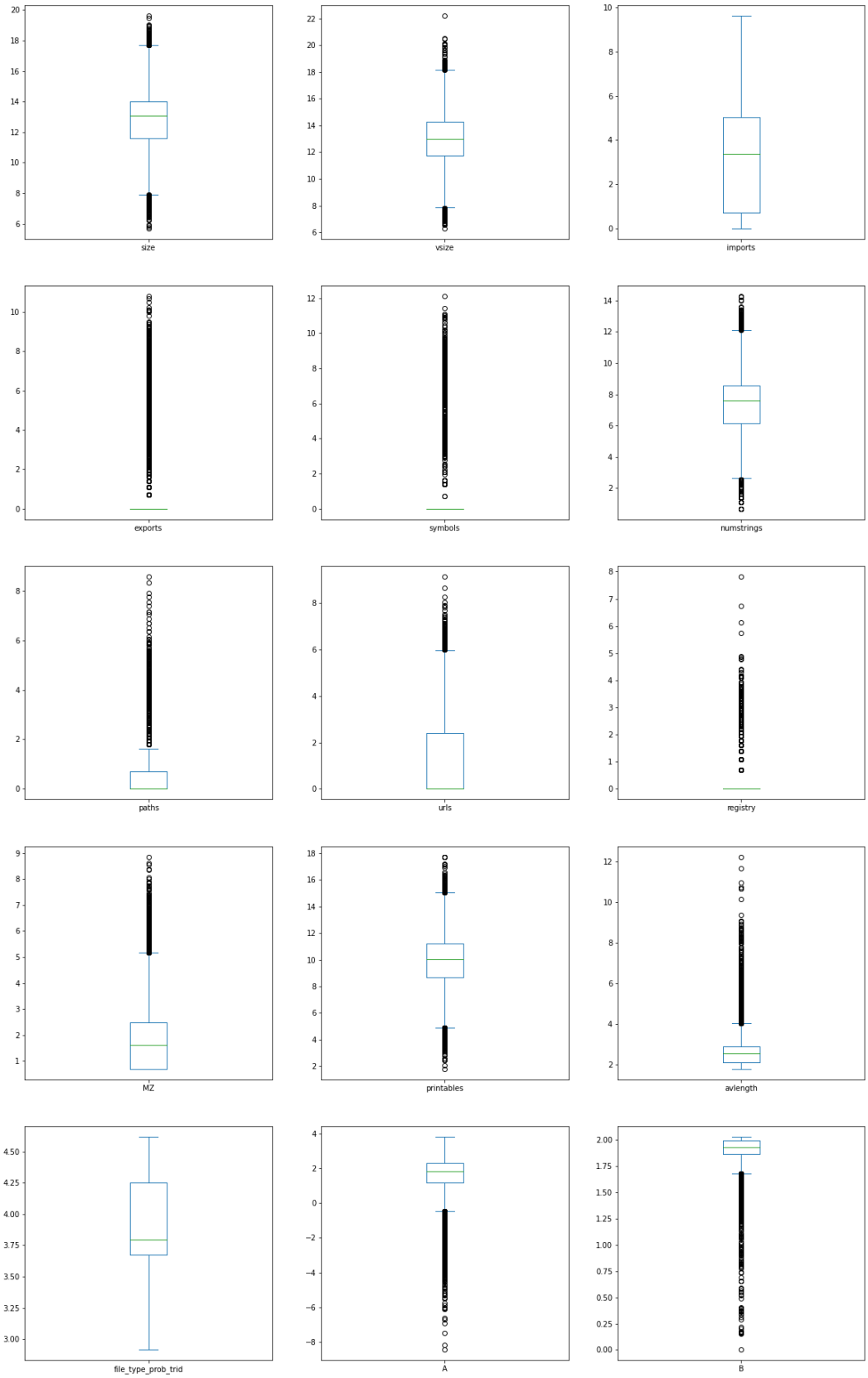

Mean File Size
1e6
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Mean File Size
Malicious
Benign
File Type


Mean Number of Symbols
70
60
50
40
30
20
10
0
Mean File Size
Malicious
Benign
File Type

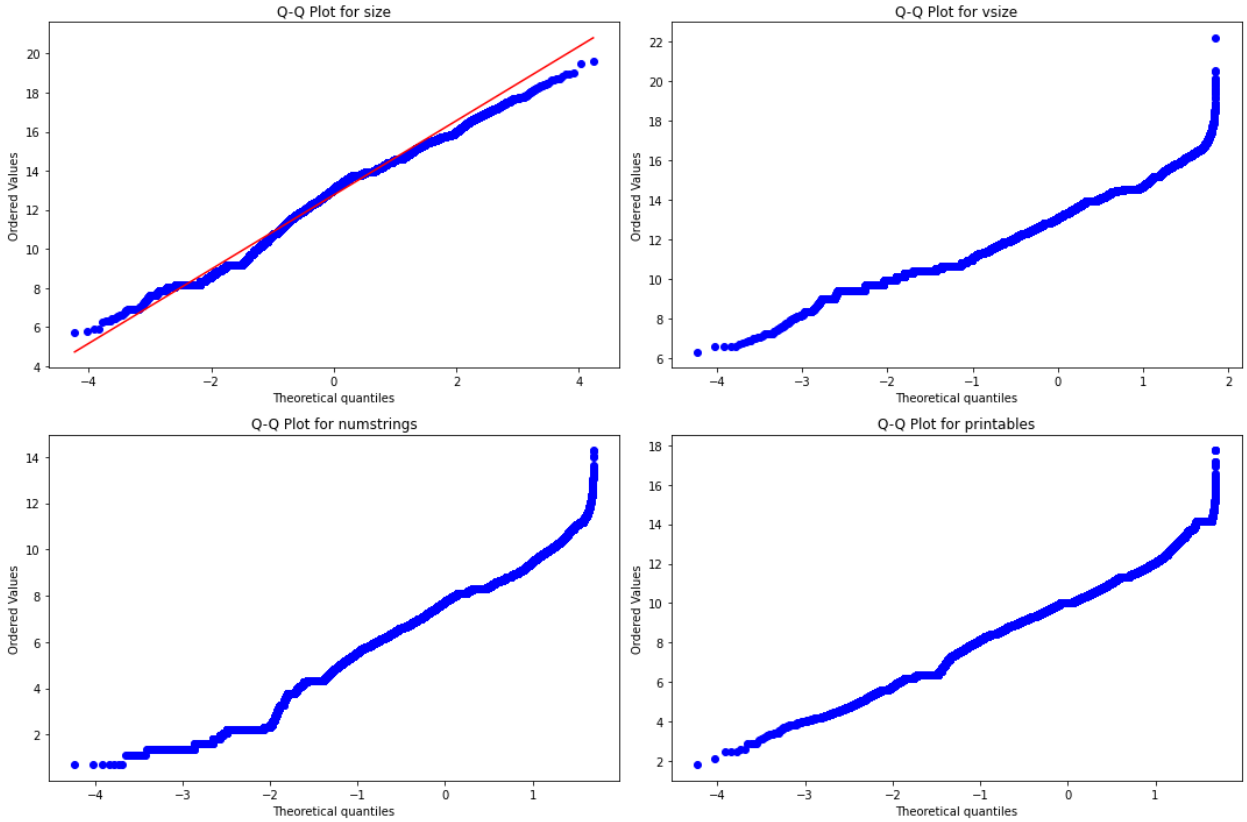
Appendix 1.11



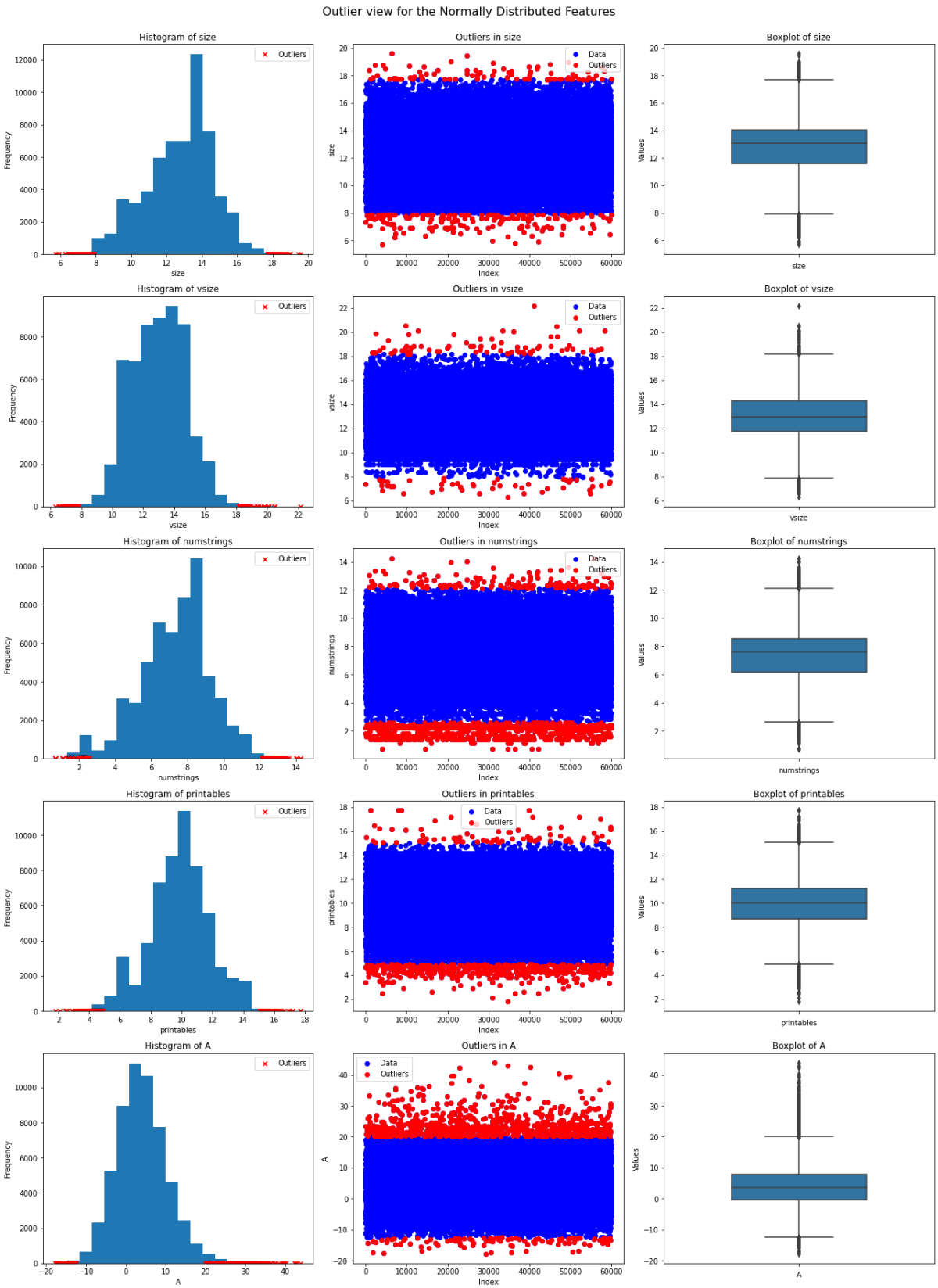
Appendix 1.12


size
vsize
imports
exports
symbols
numstrings
paths
urls
registry
MZ
printables
avlength
file_type_prob_trid
A
B

Appendix 1.13

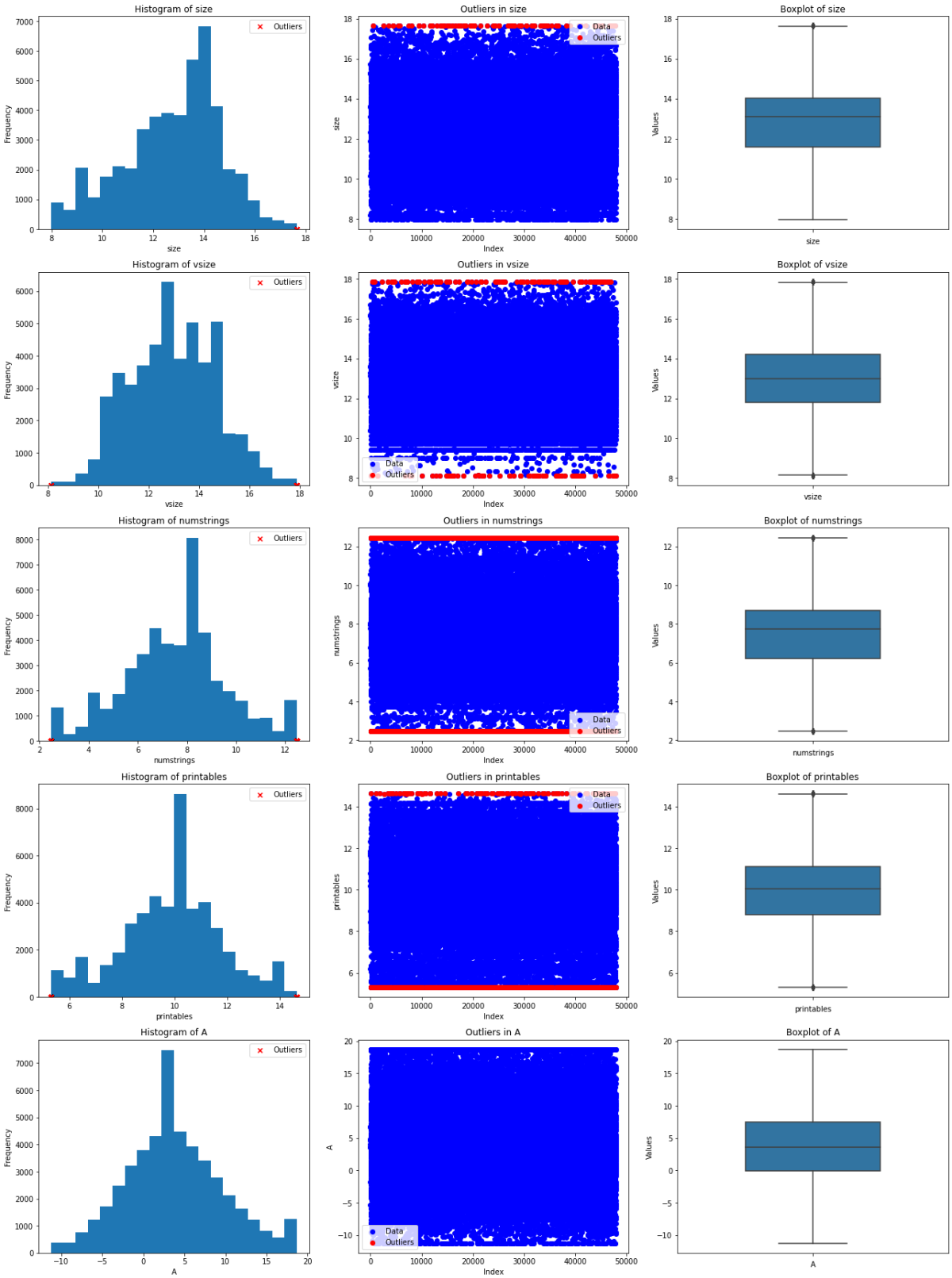


Appendix 1.14

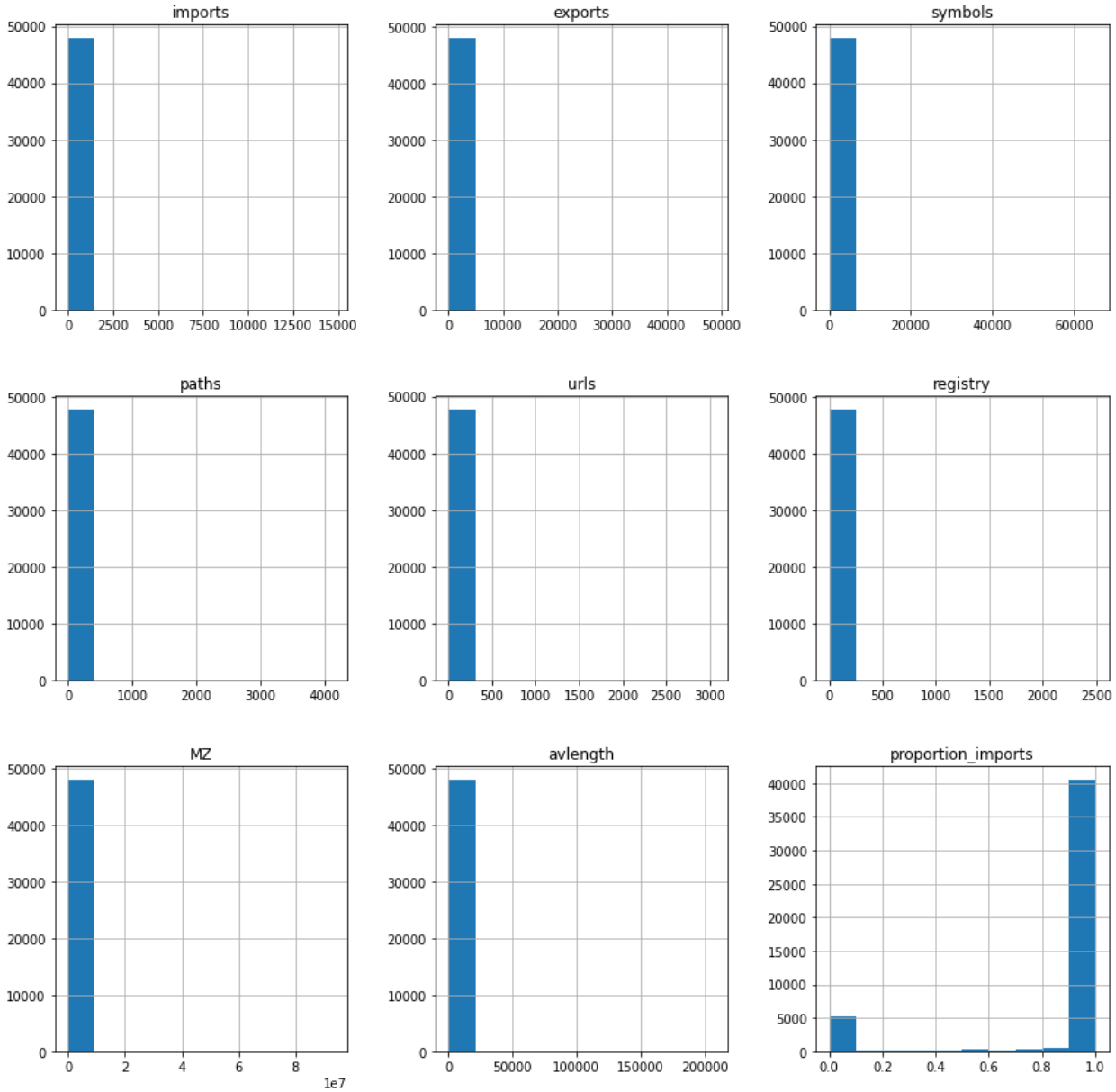

Outlier view for the Normally Distributed Features
Histogram of size
Outliers in size
Boxplot of size
Histogram of vsize
Outliers in vsize
Boxplot of vsize
Histogram of numstrings
Outliers in numstrings
Boxplot of numstrings
Histogram of printables
Outliers in printables
Boxplot of printables
Histogram of A
Outliers in A
Boxplot of A
Outliers
Data
Frequency
Values
Index
size
vsize
numstrings
printables
A

Appendix 2.1

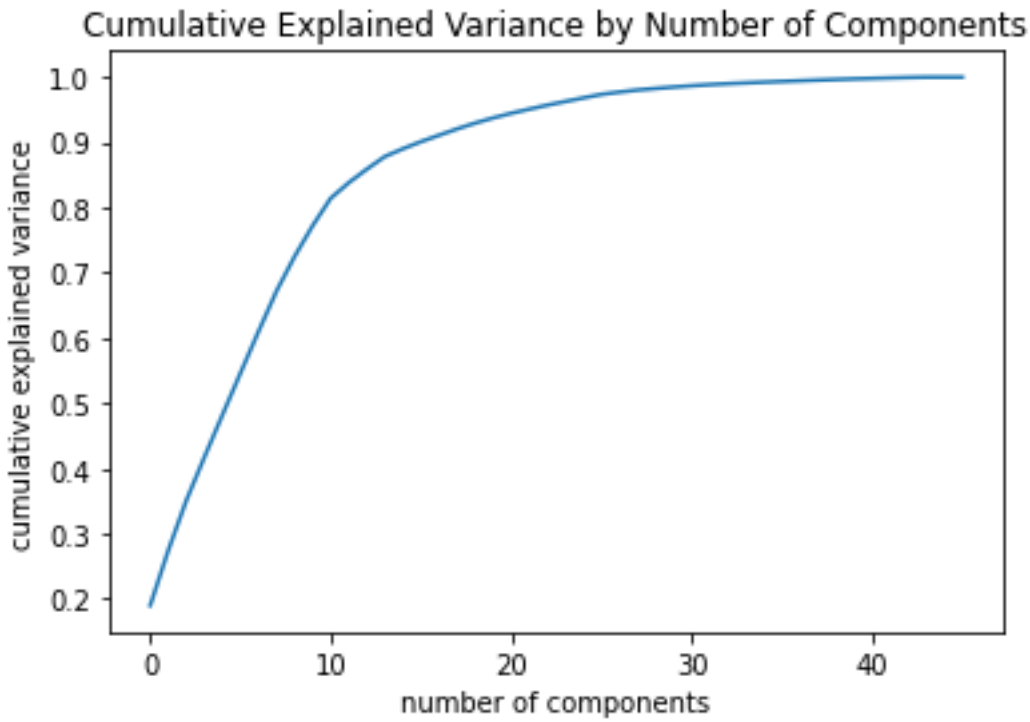
Outlier view for the Normally Distributed Features



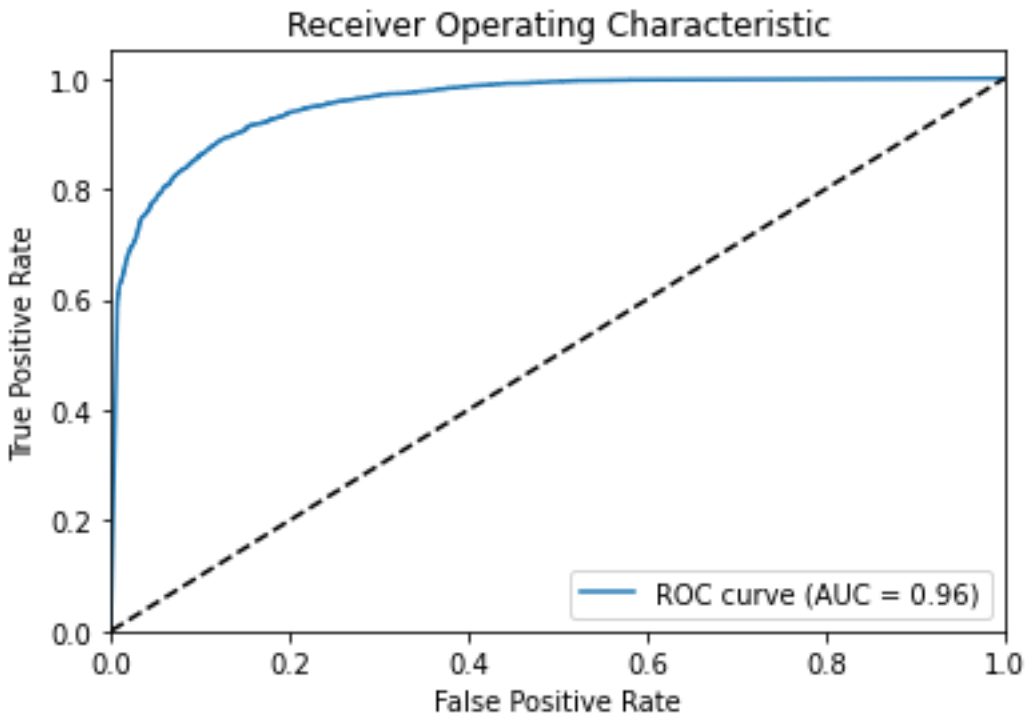
Appendix 2.2


imports
exports
symbols
paths
urls
registry
MZ
1e7
avlength
proportion_imports

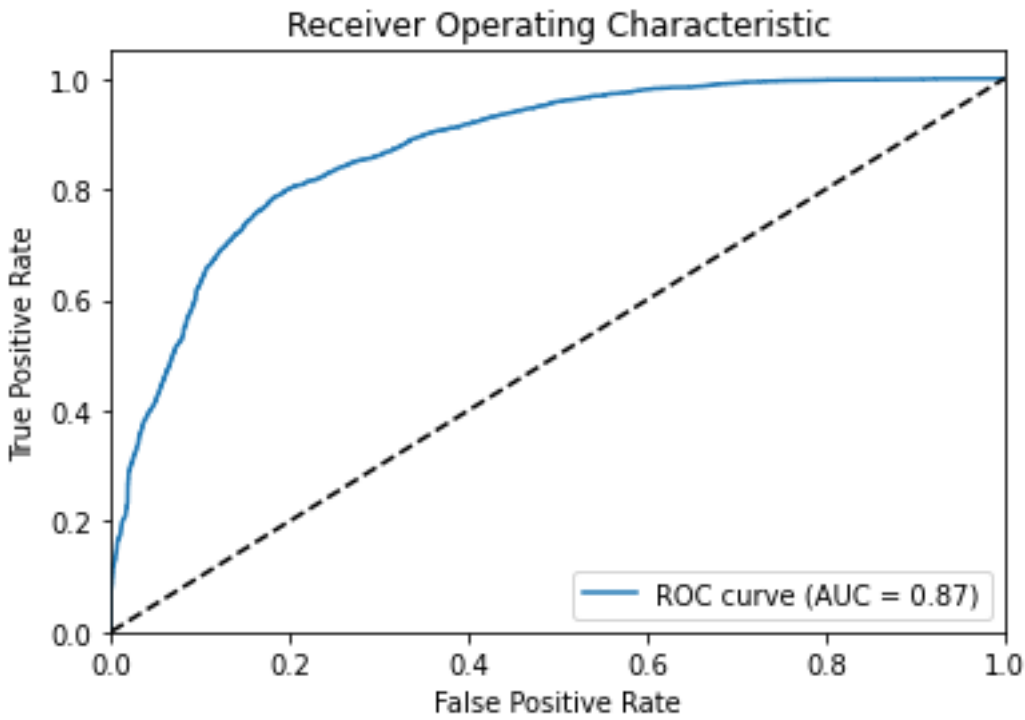
Appendix 2.3



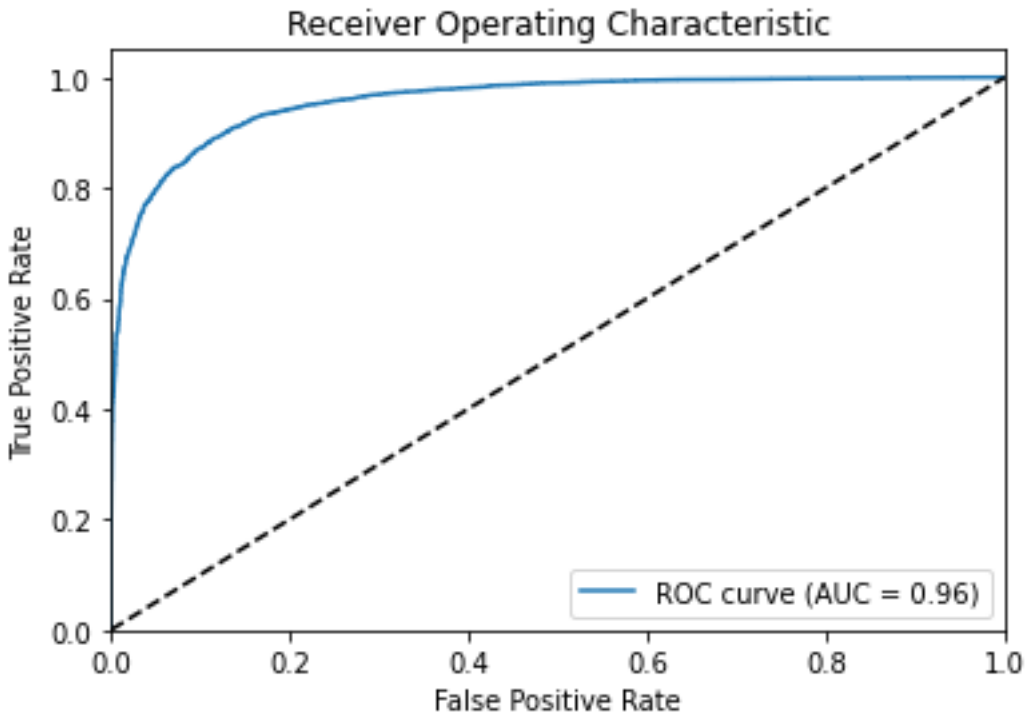
Appendix 3.1



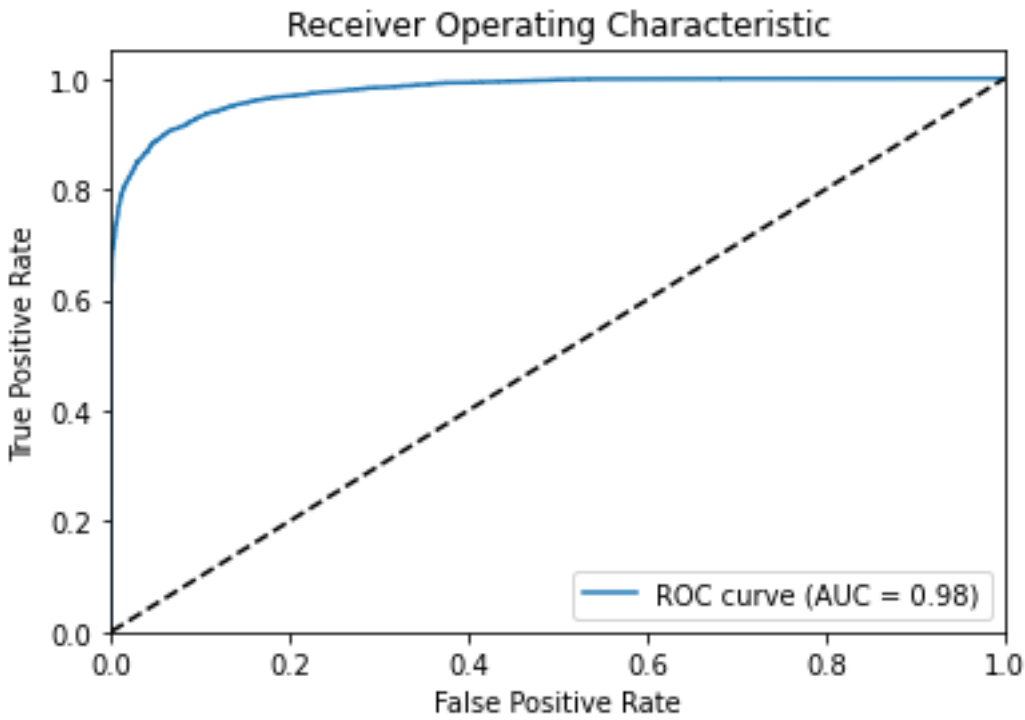
Appendix 3.2



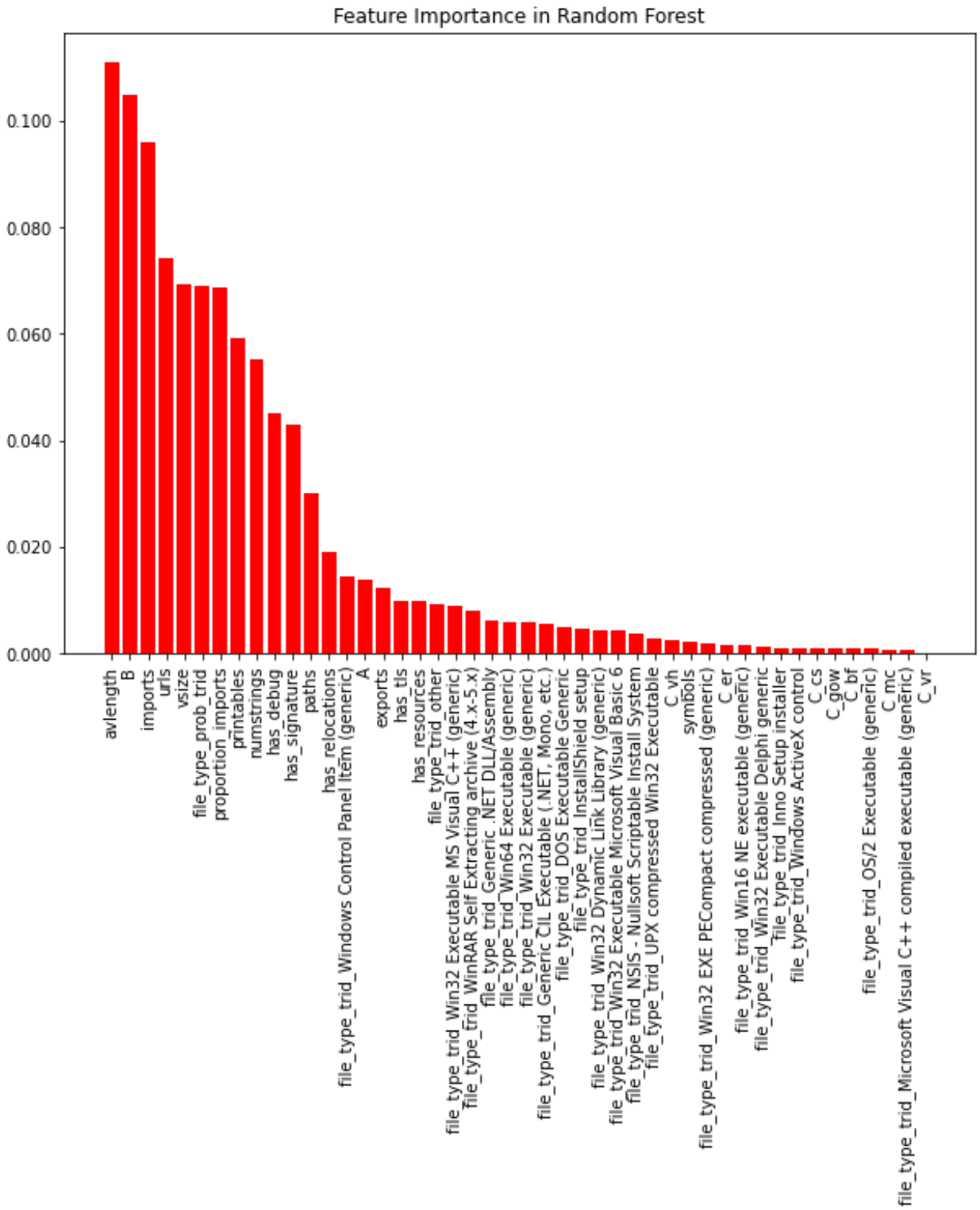
Appendix 3.3



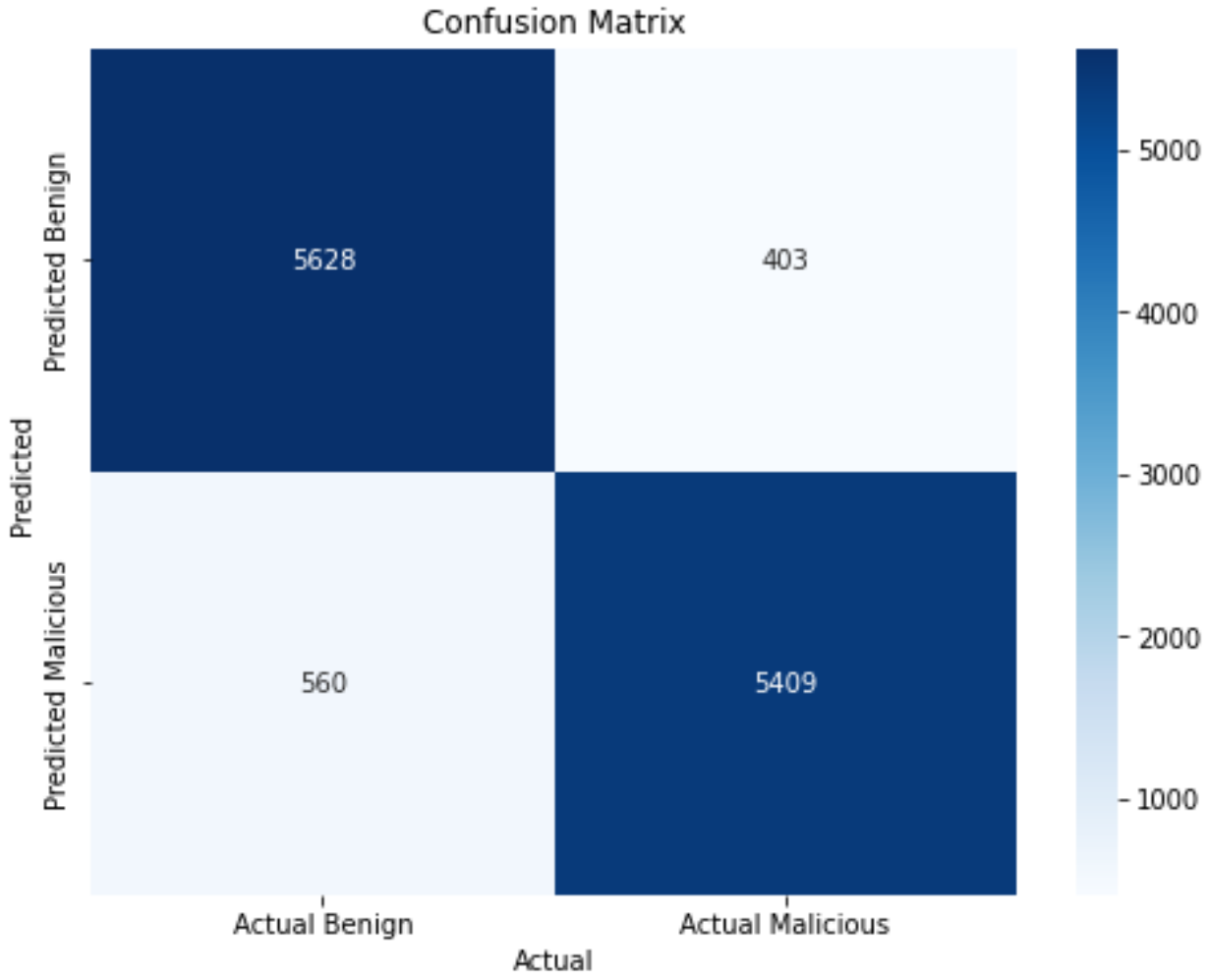
Appendix 3.4


Receiver Operating Characteristic
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (AUC = 0.98)

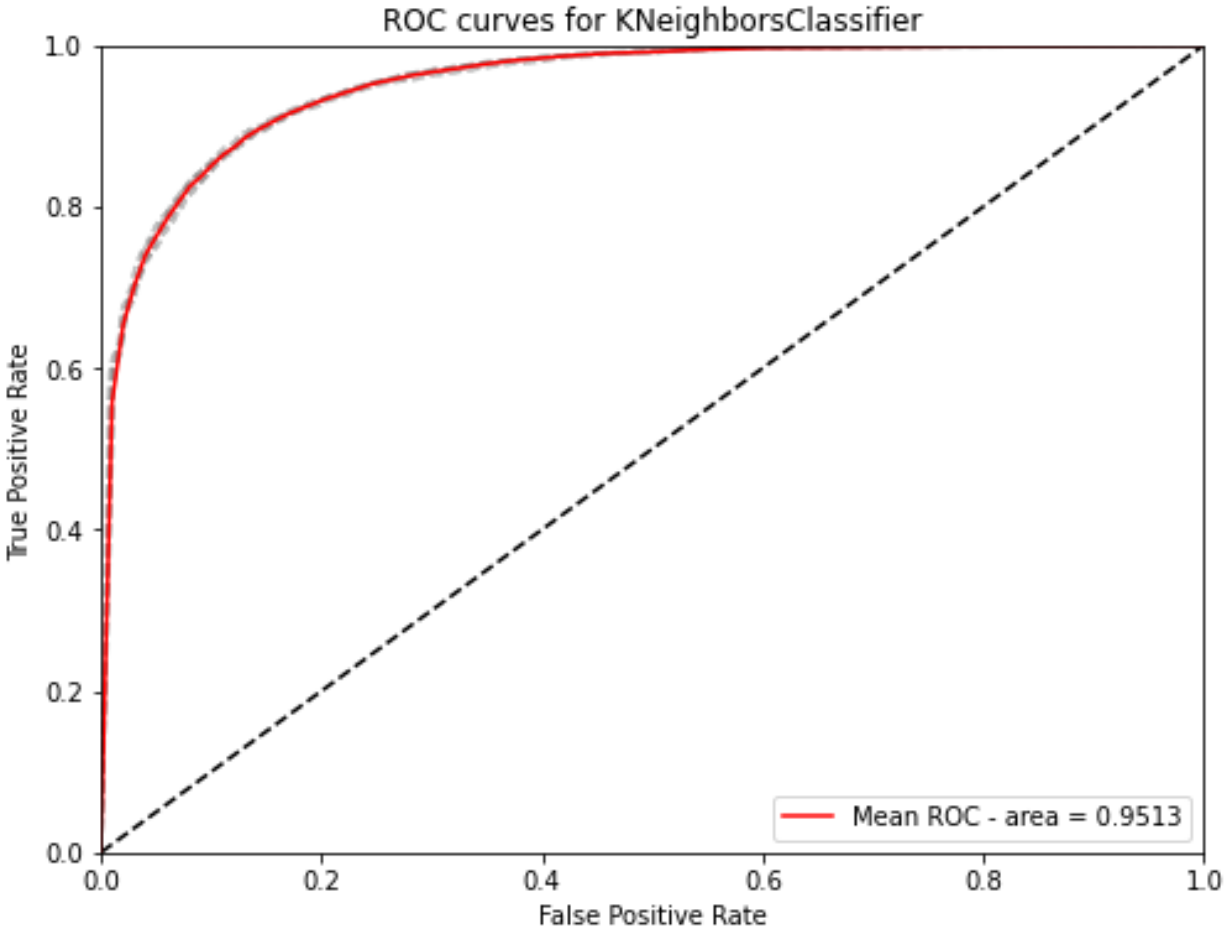
Appendix 3.5


Feature Importance in Random Forest
0.100
0.080
0.060
0.040
0.020
0.000
avlength
B
imports
urls
vsize
file_type_prob_trid
proportion_imports
printables
numstrings
has_debug
has_signature
paths
has_relocations
file_type_trid_Windows Control Panel Item (generic)
A
exports
has_tls
has_resources
file_type_trid_other
file_type_trid_Win32 Executable MS Visual C++ (generic)
file_type_trid_WinRAR Self Extracting archive (4.x-5.x)
file_type_trid_Generic .NET DLL/Assembly
file_type_trid_Win64 Executable (generic)
file_type_trid_Win32 Executable (generic)
file_type_trid_Generic CIL Executable (.NET, Mono, etc.)
file_type_trid_DOS Executable Generic
file_type_trid_InstallShield setup
file_type_trid_Win32 Dynamic Link Library (generic)
file_type_trid_Win32 Executable Microsoft Visual Basic 6
file_type_trid_NSIS - Nullsoft Scriptable Install System
file_type_trid_UPX compressed Win32 Executable
C_vh
symbols
file_type_trid_Win32 EXE PECompact compressed (generic)
C_er
file_type_trid_Win16 NE executable (generic)
file_type_trid_Win32 Executable Delphi generic
file_type_trid_Inno Setup installer
file_type_trid_Windows ActiveX control
C_cs
C_gow
C_bf
file_type_trid_OS/2 Executable (generic)
C_mc
file_type_trid_Microsoft Visual C++ compiled executable (generic)
C_vr

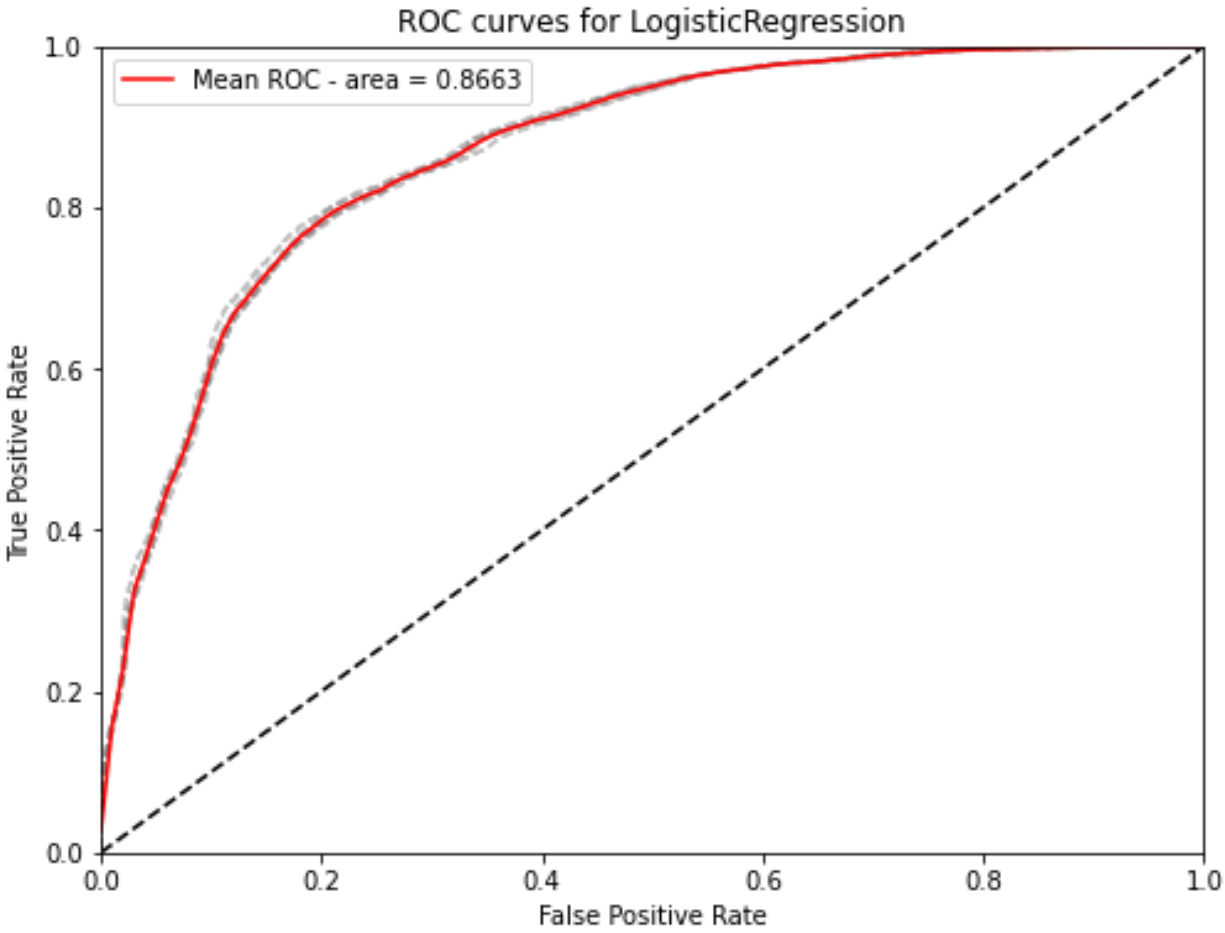
Appendix 4.1


Confusion Matrix
Predicted
Predicted Benign
Predicted Malicious
5628
403
560
5409
Actual Benign
Actual Malicious
Actual
5000
4000
3000
2000
1000

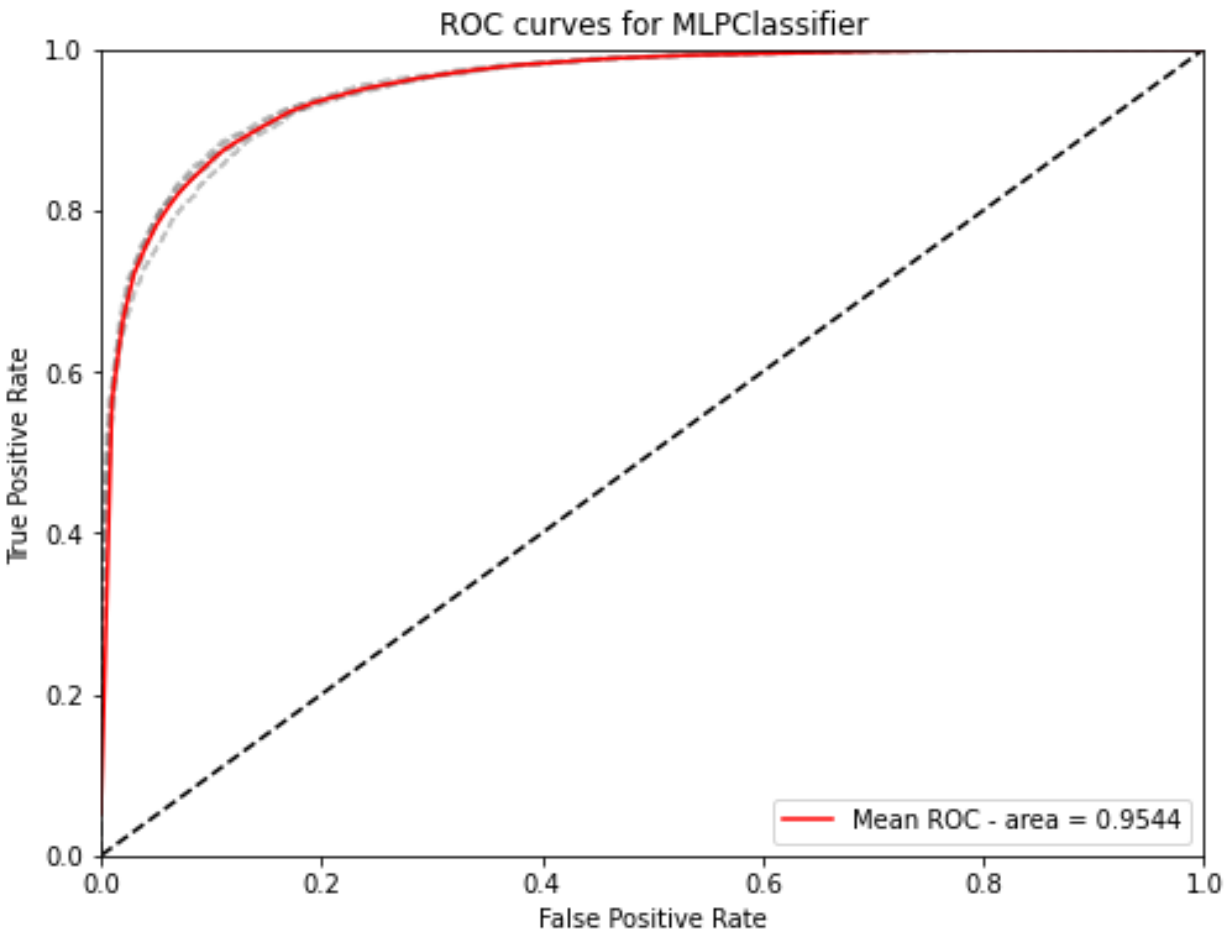
Appendix 4.2


ROC curves for KNeighborsClassifier
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Mean ROC - area = 0.9513

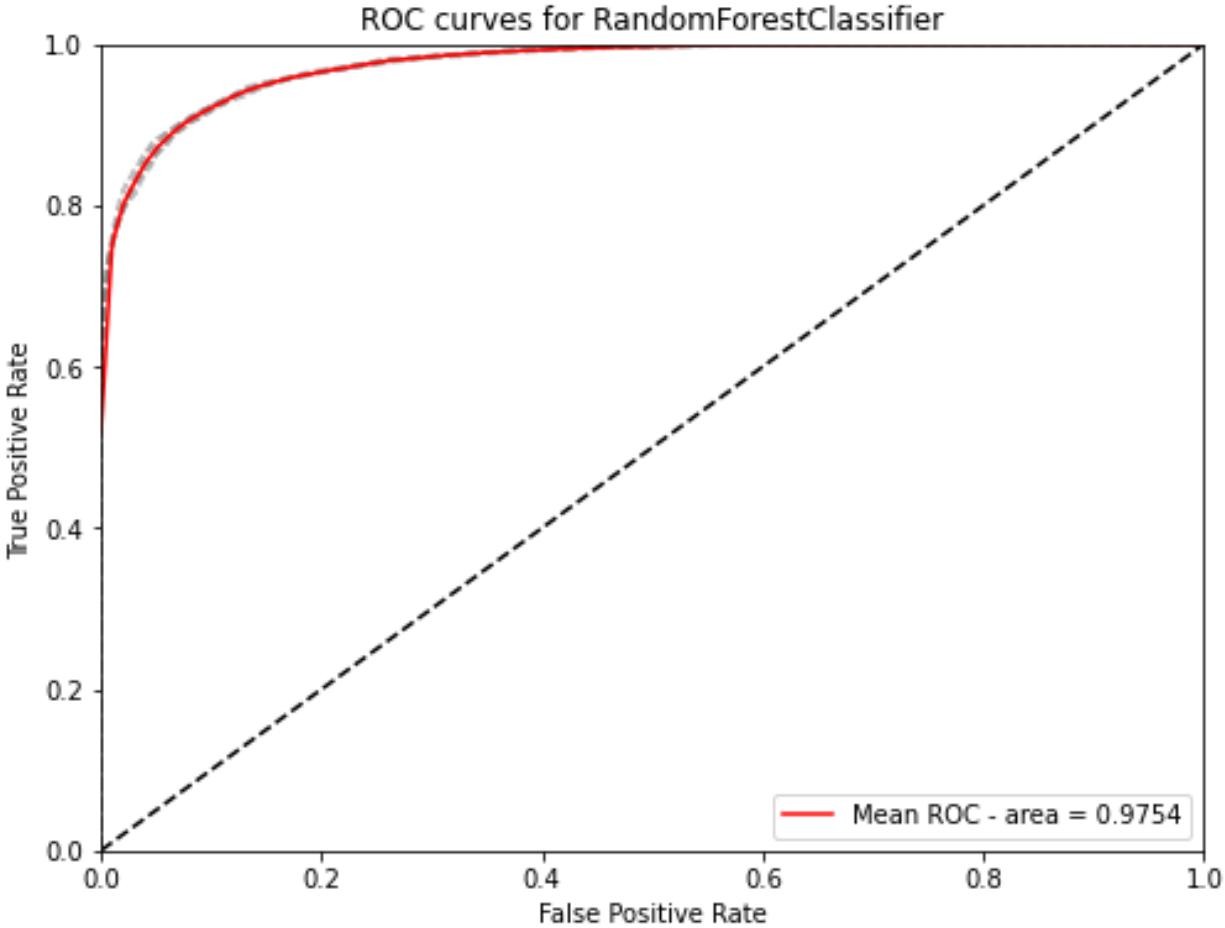
Appendix 4.3


ROC curves for LogisticRegression
Mean ROC - area = 0.8663
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

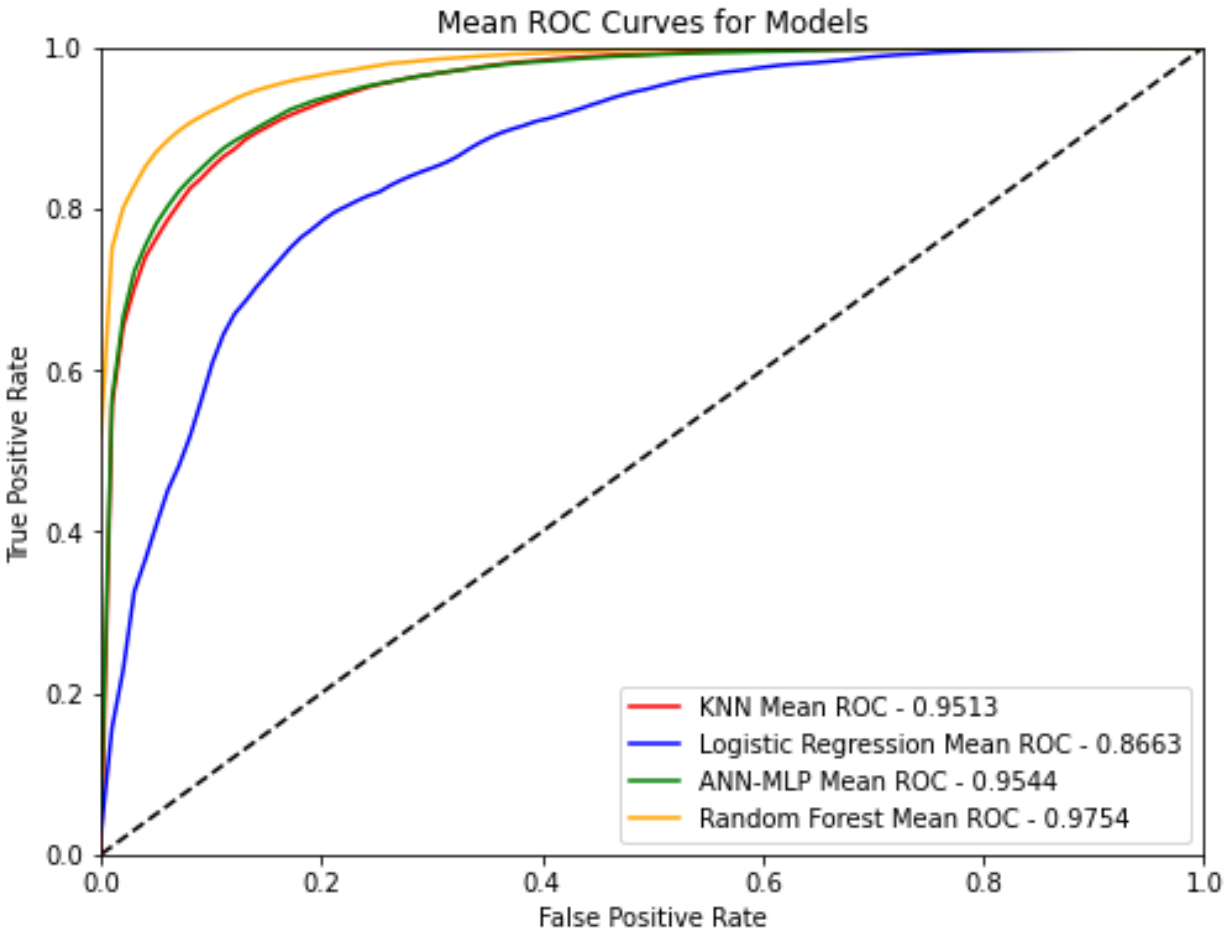
Appendix 4.4


ROC curves for MLPClassifier
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Mean ROC - area = 0.9544

Appendix 4.5


ROC curves for RandomForestClassifier
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
Mean ROC - area = 0.9754

Appendix 4.6


Mean ROC Curves for Models
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
KNN Mean ROC - 0.9513
Logistic Regression Mean ROC - 0.8663
ANN-MLP Mean ROC - 0.9544
Random Forest Mean ROC - 0.9754

VIF for all features except category features:

	Variable	VIF
0	size	12.355045
1	vsize	1.013361
2	imports	1.479693
3	exports	1.059537
4	has_debug	2.164934
5	has_relocations	2.669602
6	has_resources	7.216717
7	has_signature	1.689708
8	has_tls	1.548797
9	symbols	1.172485
10	numstrings	12.103221
11	paths	1.011277
12	urls	1.328562
13	registry	1.007875
14	MZ	1.853792
15	printables	3.450308
16	avlength	2.674772
17	file_type_prob_trid	6.775868
18	A	1.377063
19	B	12.090778

The End :)