Project Report

Introduction to Machine Learning

Daniel Yohan – 207106493
Tamara Lerman – 20766741

We began our journey by trying to understand how our data looks. At first, we attempted to build a histogram of the first column, 'size', without doing any mathematical manipulations on it (Fig 1.01). We realised this doesn't help us in the slightest, so we tried a different approach and applied the log function on the column (Fig 1.02). This helped us understand the data better. From here on out, we performed log on most of our numerical columns, since we saw it made the data much clearer (Figs 1.05, 1.07, 1.10, 1.14, 1.23, 1.26, 1.29, 1.33, 1.37, 1.41, 1.45, 1.48, 1.51). In addition, after doing internet research, we learned that there is a strong connection between the difference between the logs of vsize and size. We created a new column dedicated to that difference (Fig 1.07). Once we were done visualising all of our columns, we reached the following conclusions:

- The columns 'size', 'vsize', 'numstrings', 'printables' and 'A' all distribute normally.
- The columns 'urls', 'MZ', 'B', 'paths', 'exports', 'imports', 'avlength' are all heavy-tail distributions.
- The columns 'has_debug', 'has_relocations', 'has_resources', 'has_signature', 'has_tls' are all binary columns that received a pie chart to show how they look (Figs 1.16-1.20).
- The columns 'symbols' and 'registry' are made up of mostly zeroes, with very little non-zero values (Figs 1.21-1.22, 1.36, 1.38).
- The column 'file_type_prob_trid' distributes in an almost normal way (Figs 1.50-1.51).
- The categorial column 'file_type_trid' contains 89 unique values, some of which repeat themselves thousands of times, while others only appear once (Fig 6.01).
- The categorial column 'C' is divided nearly equally between the labels '1' and '0' in all its unique values (Fig 1.57). We decided to completely drop this column based on its visualization as it was clear that it had no impact on the label whatsoever.
- The final column, 'label' distributes evenly between 1 and 0 (Fig 1.58).

Once we finished this, we decided to add to each column's graph its correlation to the label. We added each feature another graph displaying its relationship with the label (Figs 1.03, 1.06, 1.08, 1.11, 1.15, 1.24, 1.27, 1.31, 1.35, 1.39, 1.43, 1.46, 1.49, 1.52, 1.54, 1.56, 1.57), how some features seem to have a higher chance of being malicious if their numbers are in the higher or lower on the spectrum. For example, we can see that in Fig 1.03, the numbers that are at the higher end of the spectrum are related to the label zero, making them non-malicious.

After we finished visualising our features, we wanted to see how connected they are to each other. To do that, we built a correlation heat-map (Fig 2.01), including all our numerical features. The heat-map can be found in our appendix. Our main takeaways from what we saw were that size is highly correlated with MZ and numstrings.

Before we continued, we decided to create a copy for our train data and do all our actions on it, so it wouldn't affect the original file.

After doing so, we wanted to handle our most complicated column, 'file_type_trid'. We saw that it possesses many unique values (Fig 6.01). We decided to use the OneHotEncoder method which converts categorical variables into binary features, to turn our categorial column ‘file_type_trid’, into numerical columns, so we could work on it as well. We decided to keep the top 3 common 'file_type_trid' values since they have significantly more occurrences compared to the following. Considering that, it's reasonable to assume that the test set will have the same 'file_type_trid' values as the top 3.

Before moving forward with the pre-processing step, we divided our data into train and validation data. Since we wanted to avoid any unwanted complications or disruptions, we decided to create a new copy to maintain the integrity of the original file that we worked on. After doing this, we split this copy into two parts: 80% of the data went to our training set, while the remaining 20% went to our validation set. We shuffled the data randomly.

We attempted to normalize our multiple times. We tried using the Standard Scaler method and the Min-Max method. We placed them in different positions in our code, trying to normalize our data right after the train-validation separation, we also tried to place it directly after the outlier detection part, but no matter what we did, all our scores for the advanced models dropped massively, and their performances deteriorated drastically. We know it is important to normalize the data because when all the data is placed on the same scale (like how in Min-Max all the data is scaled between 0 and 1). This makes it easier to perform distance-based models such as KNN. At the end we decided to not normalize our data at all and leave it as is.

Next up, we went to fill our null values, for our train data and our validation data separately. We divided our columns into groups, based on their distribution and values. For columns that exhibited a smooth numerical progression and had a wide range of values, ranging from zero to over a million, we imputed the missing values with the mean of their respective column (i.e., the nulls of the column ‘vsize’ received the mean of ‘vsize’). The columns that got this treatment were: ‘vsize’, ‘imports’, ‘exports’, ‘numstrings’, ‘MZ’, ‘printables’, ‘avlength’, ‘A’, ‘B’. For the other columns, we used the K-Nearest Neighbours (KNN) method, which fills null values by estimating them based on the values of their nearest neighbours in the feature space. The columns that had their nulls filled with KNN were: ‘has_debug’, ‘has_relocations’, ‘has_resources’, ‘has_signature’, ‘has_tls’, ‘symbols’, ‘paths’, ‘urls’, ‘registry’. We chose the universal hyperparameter for neighbours, which is five. We repeated this in our validation data and in our test data while transforming the data based only on the trains' imputers.

Once we were done, and had all our missing values filled, we began doing mathematical manipulations on our data. We began by doing the log function on the columns: ‘size‘, ‘vsize‘, ‘imports’, ‘exports‘, ‘numstrings‘, ‘printables‘. In the columns ‘imports’ and ‘exports’, we specifically applied the log transformation on the values which were bigger than zero. We repeated this in our validation data and in our test data.

In the next step, we created our new feature, ‘vsize_diff’, which was the difference between vsize and size. We repeated this in our validation data and in our test data. We created a brand-new correlation heatmap (Fig 2.02) in our train data to check if there is a difference after all our changes. We see that size and vsize are highly correlated with numstrings, printables and each other. It may cause overfitting and unnecessary model complexity, so our resolution was to omit both. After doing so, we created a third heatmap (Fig 2.03), and here we saw that ‘numstrings’ and ‘printables’ have a high correlation score.

After completing our mathematical manipulations, we proceeded to check for outliers. We focused on numerical columns, excluding binary variables. To identify outliers, we employed the Z-Score system, even though it wasn't covered in the course materials. This method measures the number of standard deviations a data point deviates from the mean, considering values beyond a specific threshold as potential outliers in our normally distributed column 'A.' We developed a function to automatically detect outliers and implemented it by replacing outlier values with the column mean. For the columns 'registry,' 'urls,' 'B,' 'exports,' 'imports,' and 'vsize_diff,' we utilized the Isolation Forest method. This algorithm constructs random forests and isolates instances that are easier to separate from the rest of the data, effectively identifying and modifying outliers. We created a function to detect outlier values, activated it as needed, and replaced the outliers with the mean of the respective column. The same approach was applied to the validation and test data, with transformations based solely on the fitted train data.

We specifically did not perform outlier detection on ALL the features mainly since some of them have a high connection to being either malicious or non-malicious when in the ‘outlier’ zone. For example, in Fig 1.49, higher values have a stronger connection to the label ‘1’. If we were to remove the outliers here, we would lose valuable information regarding what makes a file malicious. By replacing the outliers with different values instead of removing them entirely, we were able to preserve our test data and validation data completely. In addition, while studying in class, we have talked about deleting outliers, but never about replacing them. This adds to tools we haven’t learned in the course while working on the project.

We noticed that our data contains many dimensions, and this is not a good thing due to several reasons. Working with data in high-dimensional spaces leads to increased complexity, where the model becomes too closely fitted to the training data. Moreover, it can disrupt the bias-variance trade-off. The more dimensions we have, the higher the variance is and the lower the bias is. This is not good as it leads to overfitting and low model flexibility. We want to keep our bias-variance trade-off balanced. High dimensionality can be recognized by several ways: very long run time when attempting to activate Machine Learning models and struggle to achieve the satisfactory AUC of 0.9, mainly because of overfitting.

To handle this case, we attempted to apply PCA, which is a dimensionality reduction technique that transforms high-dimensional data into a lower-dimensional representation while preserving the most important patterns and variability in the data. Our analysis showed that we were able to keep 17 particle components as new features (Fig 3.01). The components were a linear combination of the existing features. We have decided to not use the new features as they do not exhibit an improvement on our best predicting model, and they lose the meaning of the original features.

Our next idea was to perform Forward Selection using Mallows Cp on the features. Our results (Fig 3.02) showed that we should keep 16 features and drop all the one-hot-encoded ones that we received plus 'exports', 'numstrings', 'registry', and 'A'.
After conducting the algorithm, we've decided to keep 19 features instead of 16. It's a greedy algorithm and does not check all the possible feature combinations.
We first dropped all the proposed features and saw a decrease in the models' performances. We tried omitting some other combinations from the list above and found that dropping only the three one-hot-encoded 'file_type_trid' yields us the best performances.

After this, we repeated the pre-processing actions on our validation and test data separately.

And now began the most exciting part of the project: running our selected models. From the simpler models, we chose Naïve Bayes Classifier & Logistic Regression. Unsurprisingly, the two had a low validation accuracy. This is due to them not having the ability to capture more complex relationships and interactions in the data. Once we finished running the simpler models, we moved forward and picked out the more advanced models. We chose Random Forest Classifier and Decision Tree. For these models we received a much better validation accuracy score.

To ensure best results, we conducted grid search on our two advanced models:
Random Forest Classifier:

- n_estimators: The number of decision trees in the random forest. Increasing the number of trees generally improves model accuracy, but also increases computation time.
- max_depth: The maximum depth of the decision trees. Controlling the maximum depth helps prevent overfitting by limiting the complexity of the trees.
- min_samples_split: The minimum number of samples required to split an internal node. Setting a higher value requires a minimum number of samples for a node to split, reducing overfitting.
- min_samples_leaf: The minimum number of samples required to be at a leaf node. Specifying a minimum number of samples for a leaf node helps prevent overfitting by creating simpler trees.
- max_features: The number of features to consider when looking for the best split. Limiting the number of features considered for each split reduces randomness and potential overfitting.

Decision Tree:

- criterion: The choice of criterion affects how the tree measures the quality of splits. "Gini" tends to create smaller, more balanced trees, while "entropy" tends to create more complex trees that can capture intricate relationships in the data.
- splitter: The splitter strategy determines how the best split is chosen at each node. "Best" selects the optimal split based on the criterion, while "random" introduces randomness by considering random splits. "Best" is generally recommended for better performance.

- The hyperparameters max_depth, min_samples_split, min_samples_leaf and max_features have all been stated above and their meaning remains the same in this model as well.

Our best model turned out to be Random Forest Classifier. For it we received a validation accuracy score of 0.903, while Decision Tree received 0.796, Naïve Bayes Classifier achieved 0.706, and our worst model Logistic Regression received a mere score of 0.623.

Later we conducted a Confusion Matrix for our best model, Random Forest (Fig 4.01). A confusion matrix summarizes the accuracy of a classification model by comparing its predicted and actual class labels. The cells in our confusion matrix represent the number of true and false classifications of the label (the label either is 1 or 0). The performances were rather impressive for our model.

Afterward, we designed a K-Fold Cross Validation and a ROC curve function. K-Fold Cross Validation assesses model performance by dividing data into subsets for training and testing, while an ROC curve illustrates the true positive and false positive trade-off of a classification model. We activated it on each of our models and received our AUC scores for each model (Figs 5.01-5.04). The AUC score, or Area Under the Curve score, is a single metric that quantifies the overall performance of a binary classification model by measuring the area under the ROC curve.

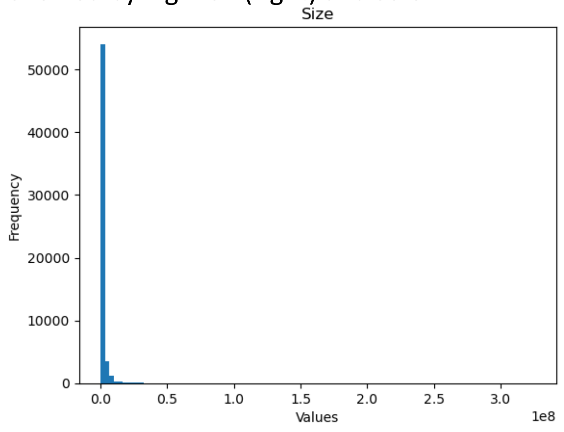
In the next stage, we performed a prediction comparison. We trained our best model, Random Forest Classifier, on our train data and on our validation data and we compared their AUC scores in a different ROC curve graph (Fig 5.05). We can see that there is a difference in the AUC score between the Train and Validation sets, this is due to overfitting to some degree since we trained the Random Forest model on our training set. Thus, we only consider the score of the validation set and preprocess the training and validation separately.

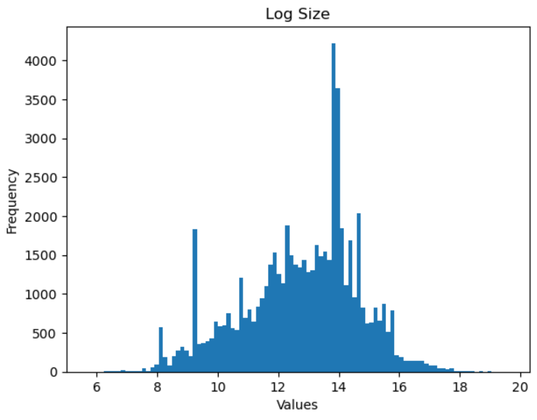
Finally, we predicted the probability of each file being malicious on our test data. We, once again, opted to use our finest model, Random Forest Classifier, to do so. We saved all results in the file called ‘results_32’. The number 32 is because we are group 32.

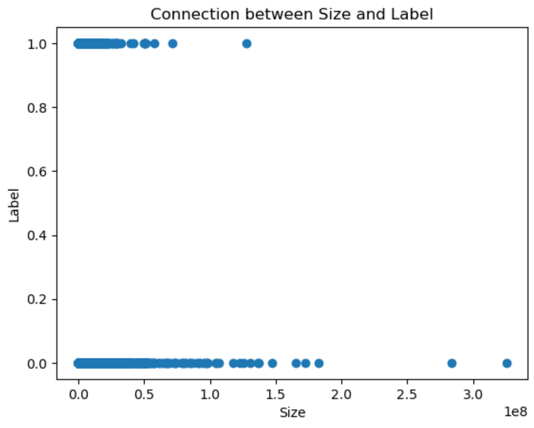
Both of us collaborated on the code, exchanging advice, and working on visualizations together. We divided the preprocessing tasks, with Daniel handling outliers and Tamara managing missing values. This applied to all three datasets: train, validation, and test. We jointly selected models and individually developed two each from scratch. Daniel took care of the confusion matrix, ROC curve functions, prediction comparison, and the pipeline. Meanwhile, Tamara started writing the project report. We encountered challenges along the way, necessitating changes in our models and preprocessing. We realized our error of using the 'fit' method on validation and test data, promptly rectifying it by using 'fit' only on the train data and 'transform' on the others. After that, everything proceeded smoothly.

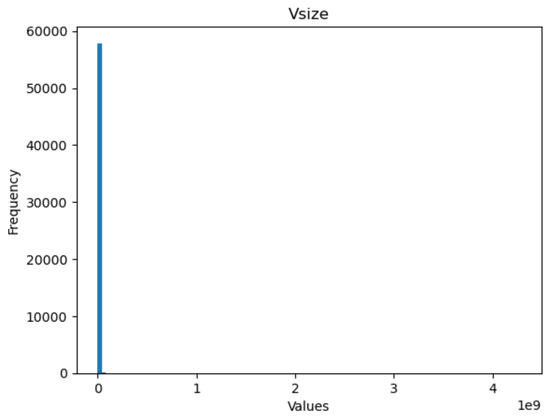
Appendix:

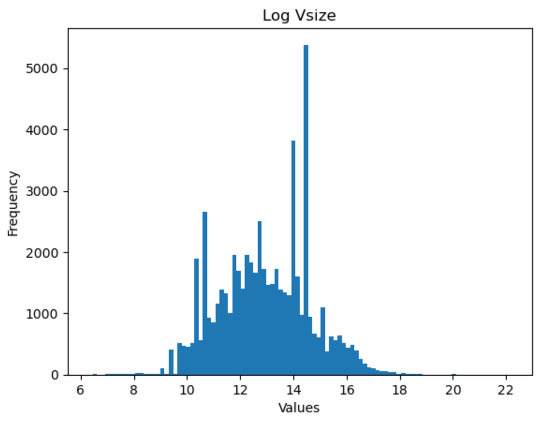
Firstly, the graphs for each one of our features, plus how correlated it is to the label. The numbering will go like this: starting from the first graph (left), it will be called Fig 1.01, followed by Fig 1.02 (right) and so on.

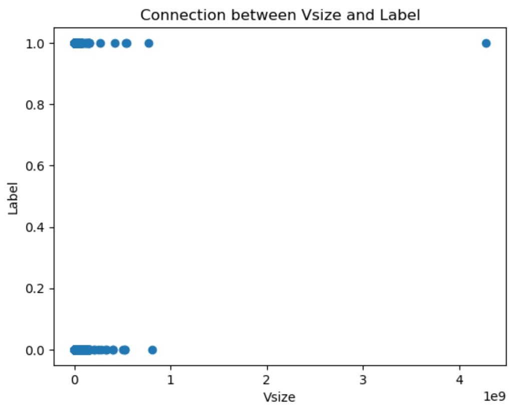


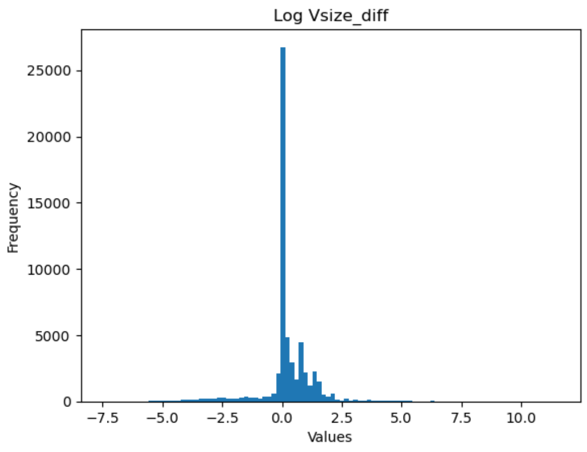


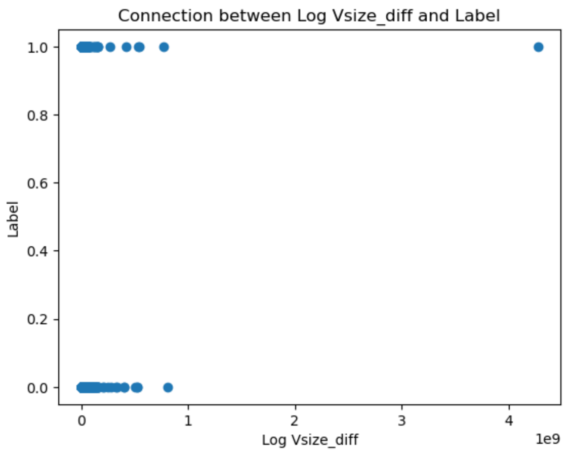


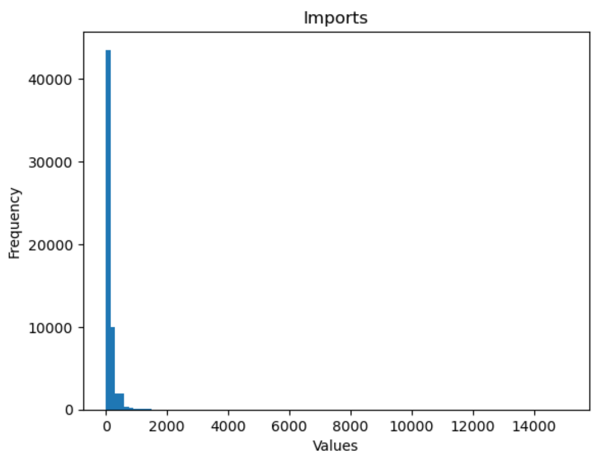
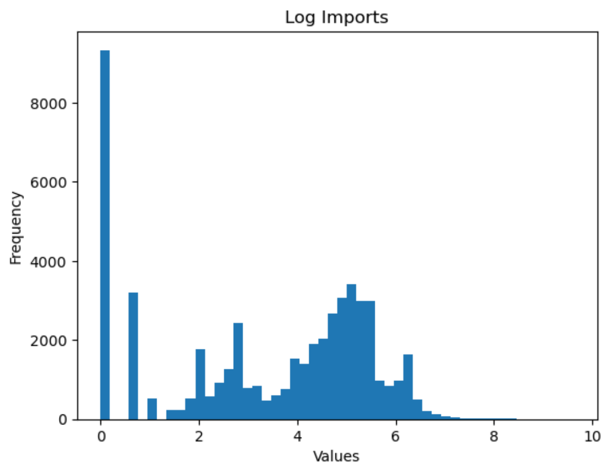
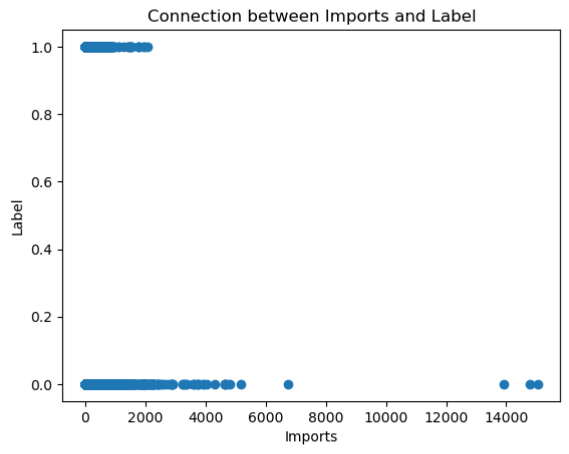
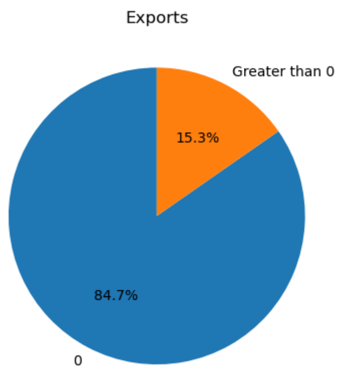
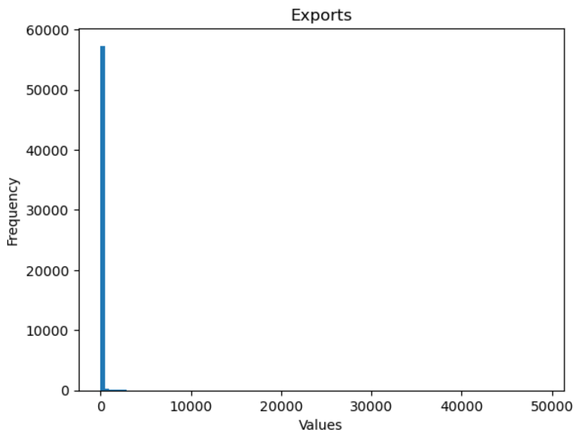
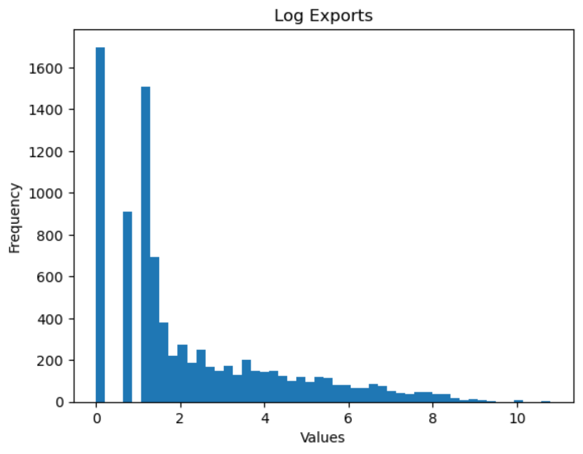
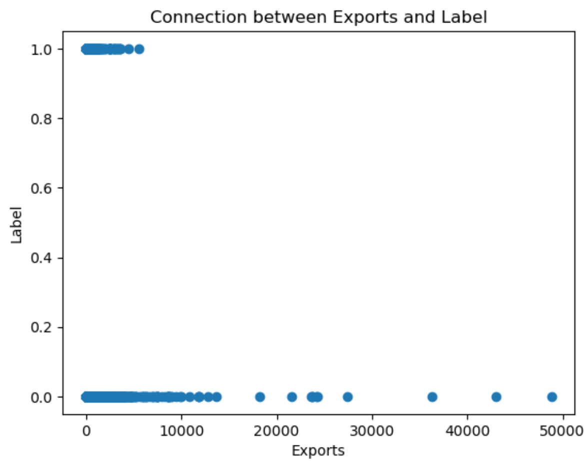
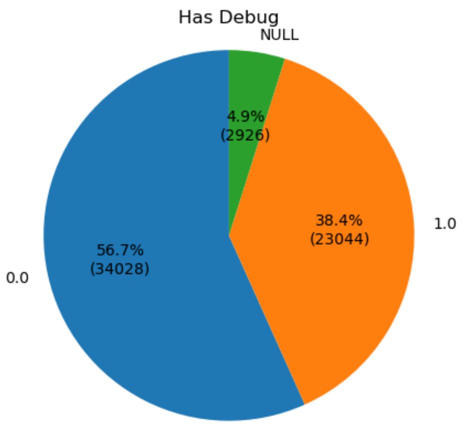


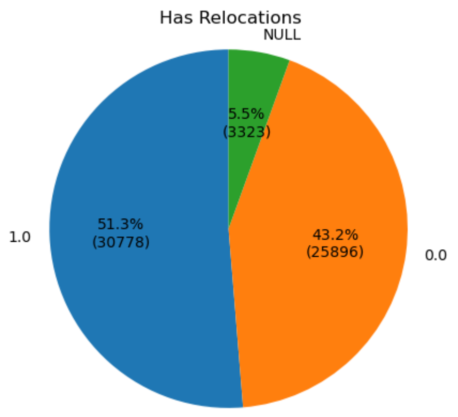


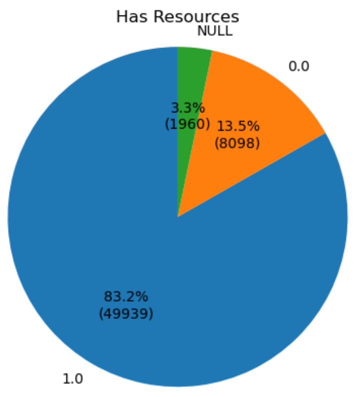


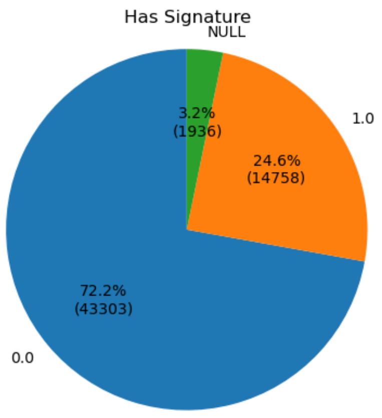


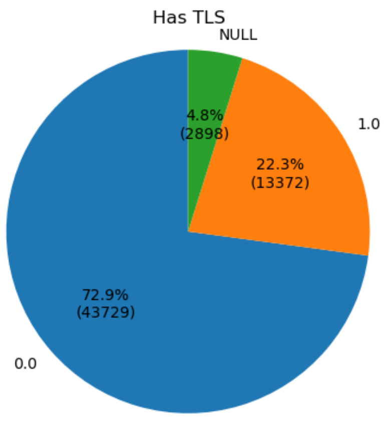


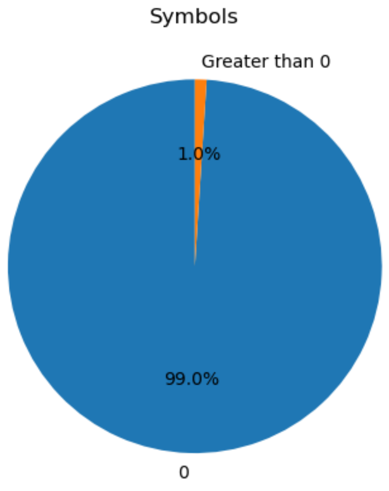

Imports
Frequency
Values

Log Imports
Frequency
Values

Connection between Imports and Label
Label
Imports

Exports
Greater than 0
15.3%
84.7%
0

Exports
Frequency
Values

Log Exports
Frequency
Values

Connection between Exports and Label
Label
Exports

Has Debug
NULL
4.9%
(2926)
38.4%
(23044)
1.0
56.7%
(34028)
0.0

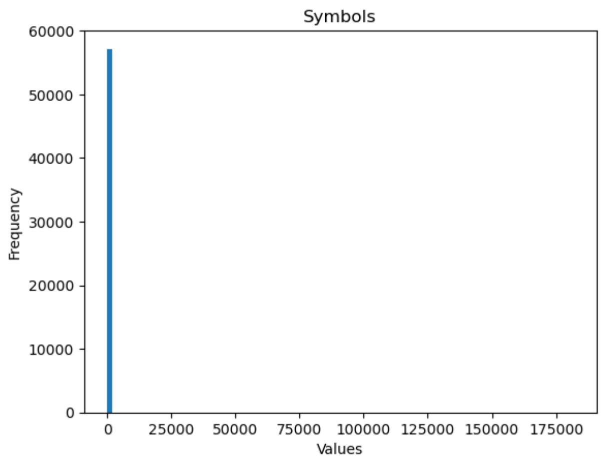

Has Relocations
NULL
5.5%
(3323)
1.0
51.3%
(30778)
43.2%
(25896)
0.0

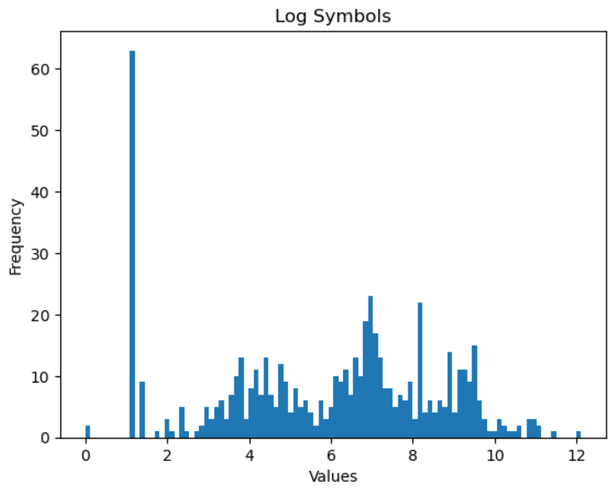

Has Resources
NULL
0.0
3.3%
(1960)
13.5%
(8098)
83.2%
(49939)
1.0

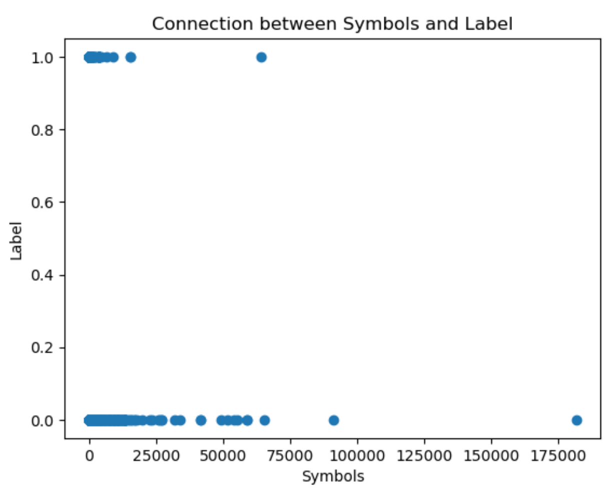

Has Signature
NULL
3.2%
(1936)
1.0
24.6%
(14758)
72.2%
(43303)
0.0

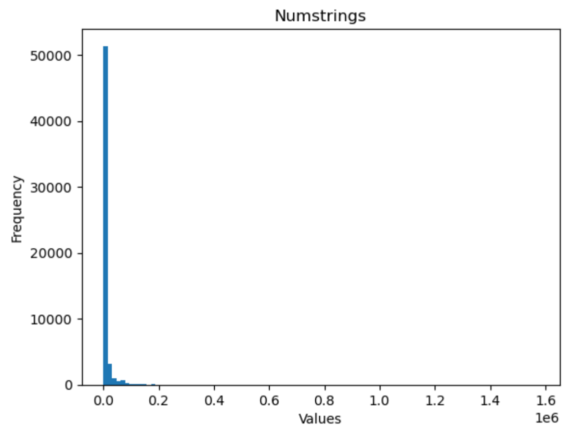
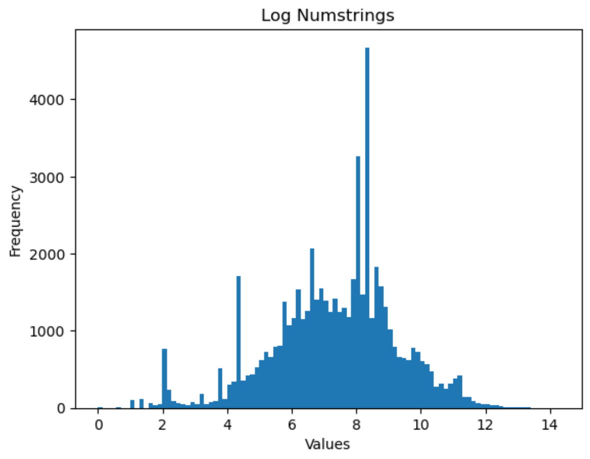
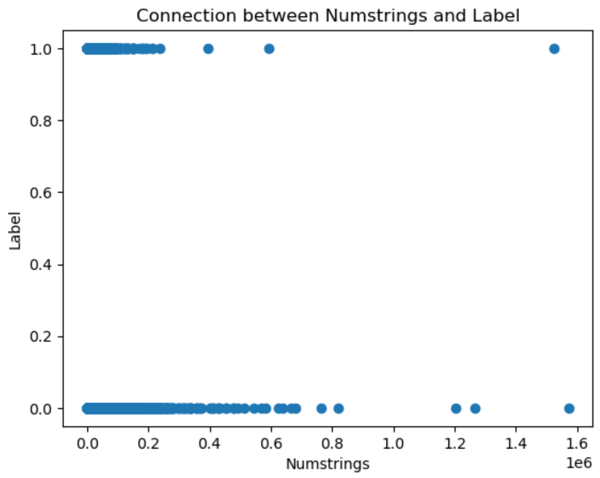
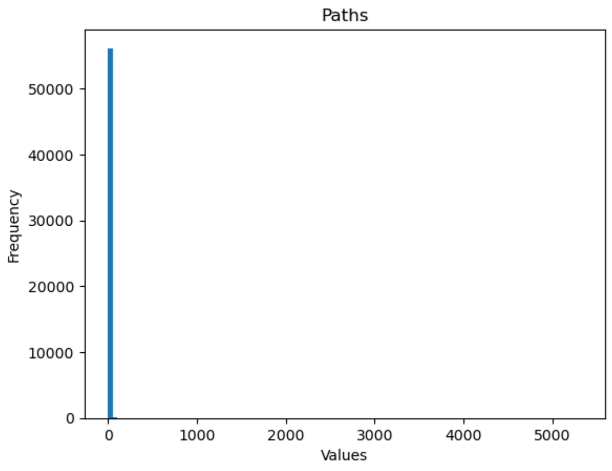
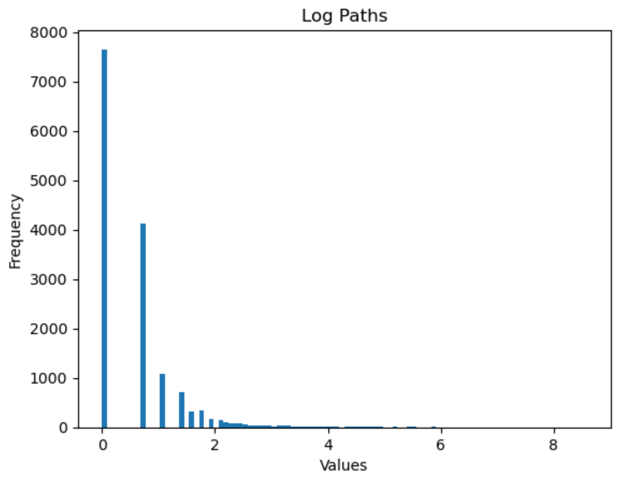
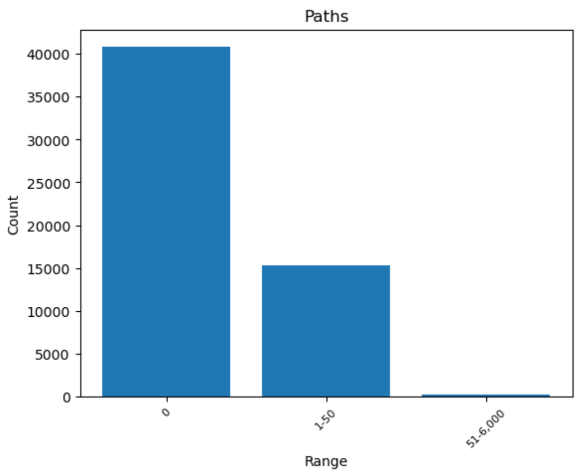
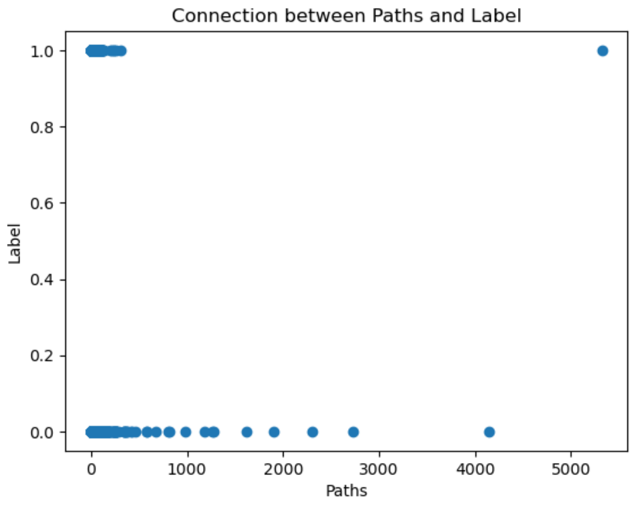
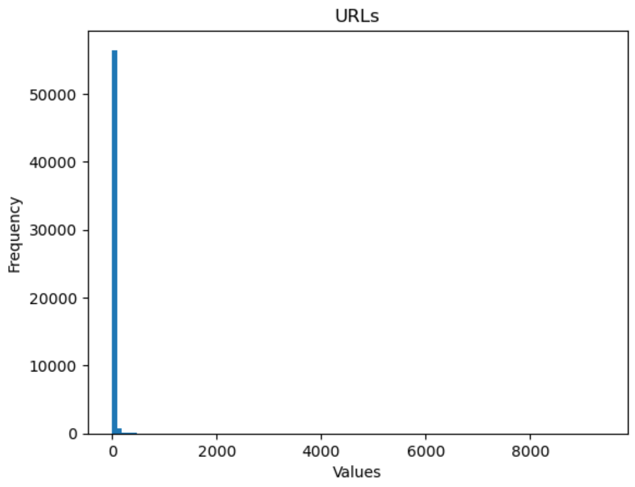

Has TLS
NULL
4.8%
(2898)
1.0
22.3%
(13372)
72.9%
(43729)
0.0

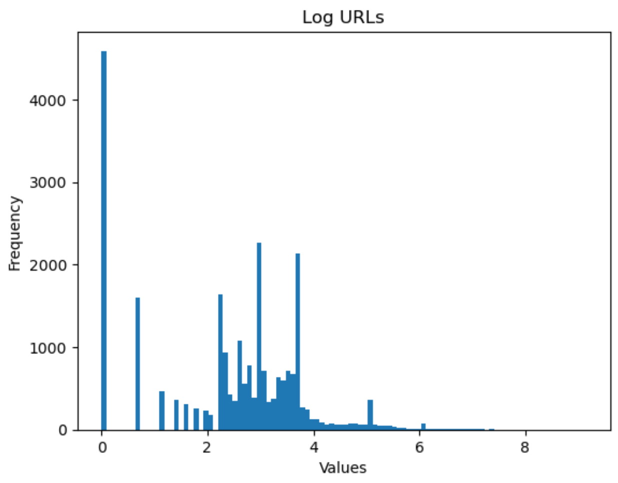

Symbols
Greater than 0
1.0%
99.0%
0

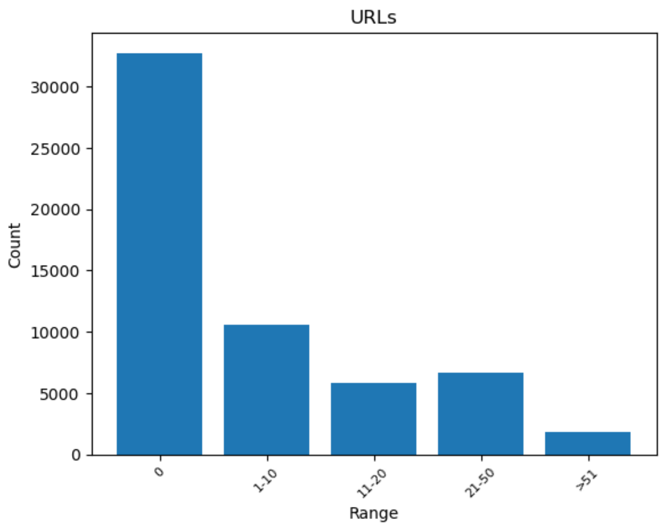

Symbols
Frequency
Values
60000
50000
40000
30000
20000
10000
0
0
25000
50000
75000
100000
125000
150000
175000

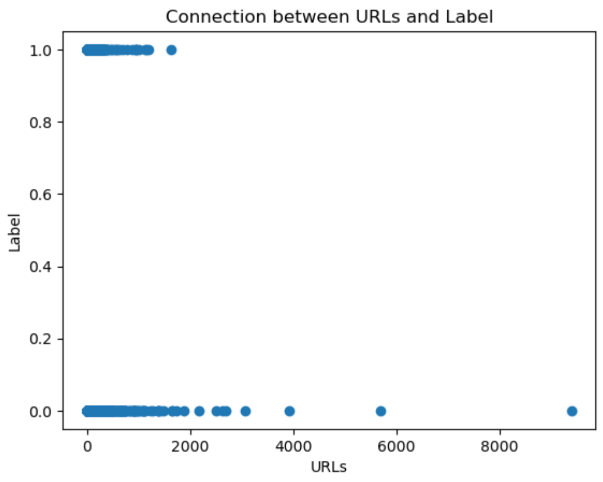

Log Symbols
Frequency
Values
60
50
40
30
20
10
0
0
2
4
6
8
10
12

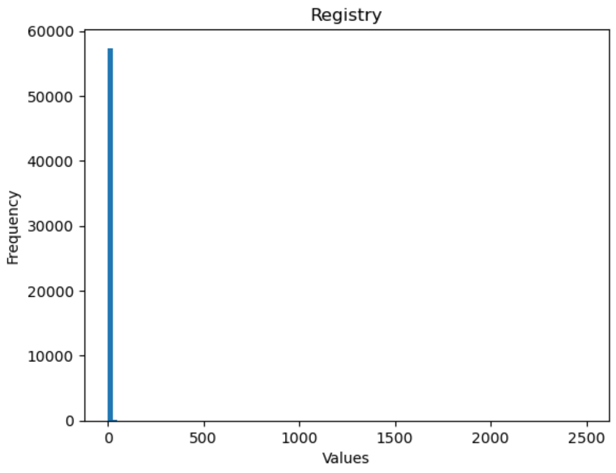

Connection between Symbols and Label
Label
Symbols
1.0
0.8
0.6
0.4
0.2
0.0
0
25000
50000
75000
100000
125000
150000
175000

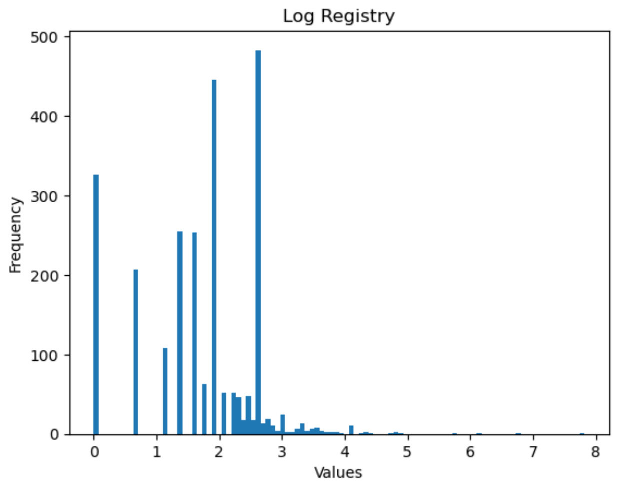

Numstrings
Frequency
Values
1e6

Log Numstrings
Frequency
Values

Connection between Numstrings and Label
Label
Numstrings
1e6

Paths
Frequency
Values

Log Paths
Frequency
Values

Paths
Count
0
1-50
51-6,000
Range

Connection between Paths and Label
Label
Paths

URLs
Frequency
Values

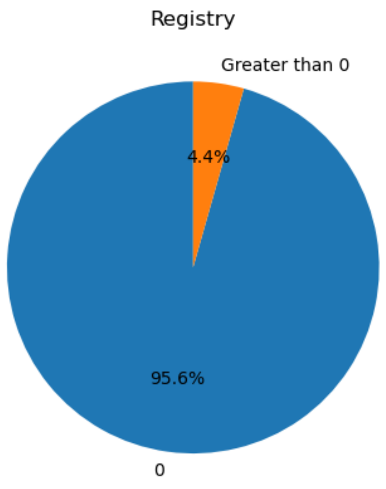

Log URLs
Frequency
Values
0
1000
2000
3000
4000
0
2
4
6
8

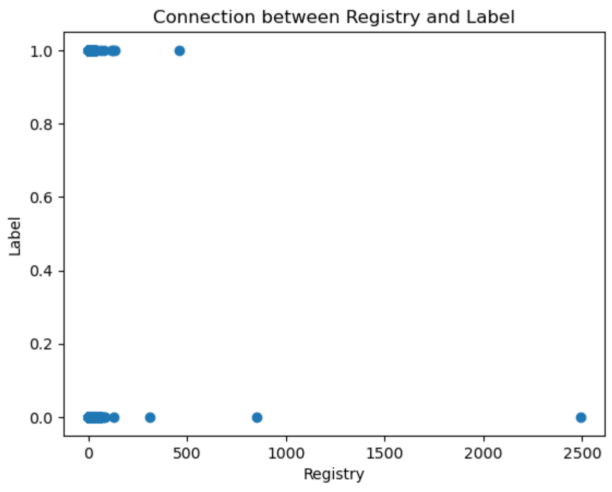

URLs
Count
Range
0
5000
10000
15000
20000
25000
30000
0
1-10
11-20
21-50
>51

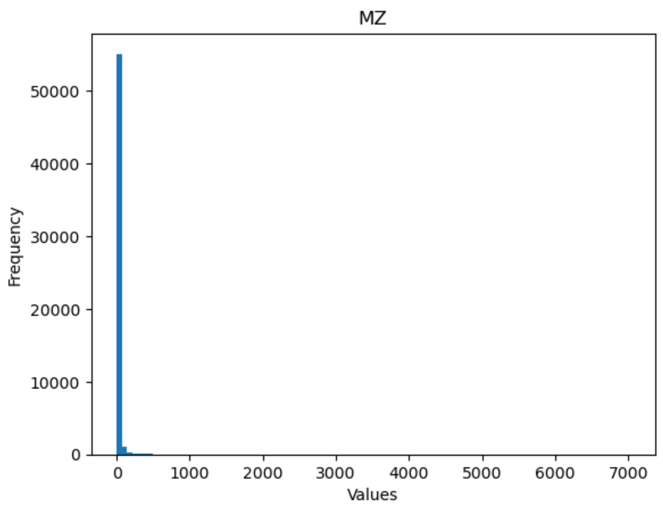

Connection between URLs and Label
Label
URLs
0.0
0.2
0.4
0.6
0.8
1.0
0
2000
4000
6000
8000

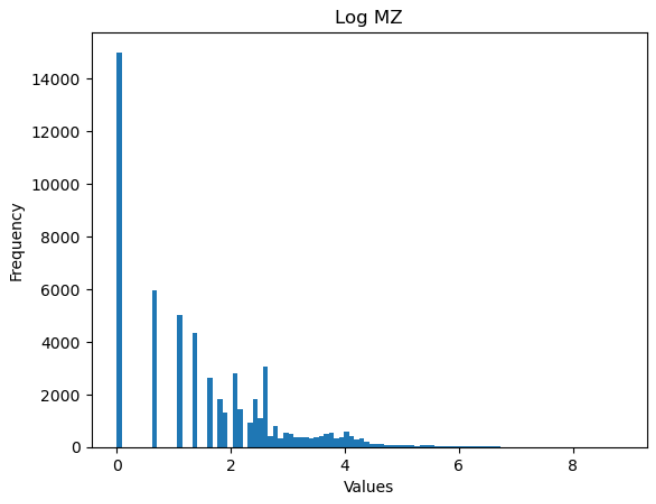

Registry
Frequency
Values
0
10000
20000
30000
40000
50000
60000
0
500
1000
1500
2000
2500

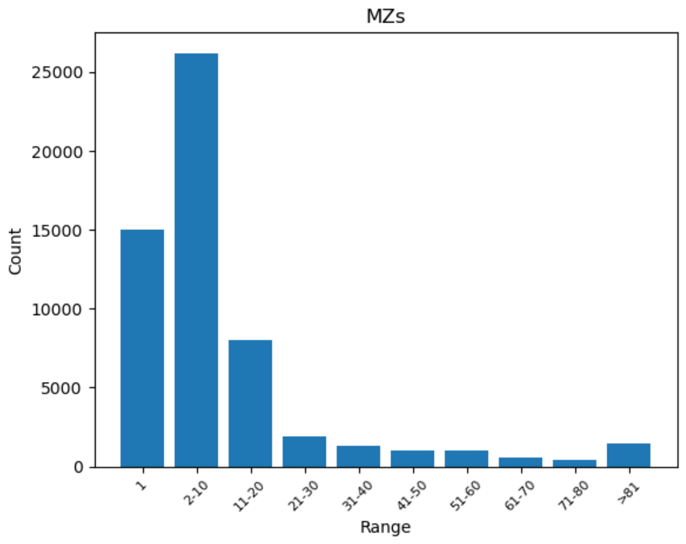

Log Registry
Frequency
Values
0
100
200
300
400
500
0
1
2
3
4
5
6
7
8

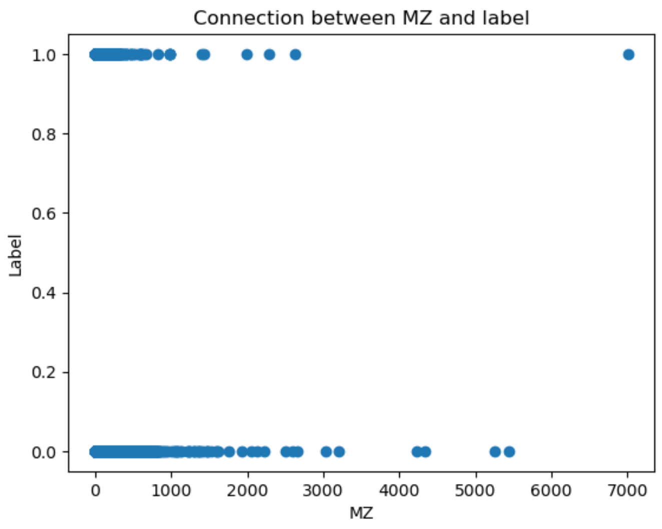

Registry
Greater than 0
4.4%
95.6%
0

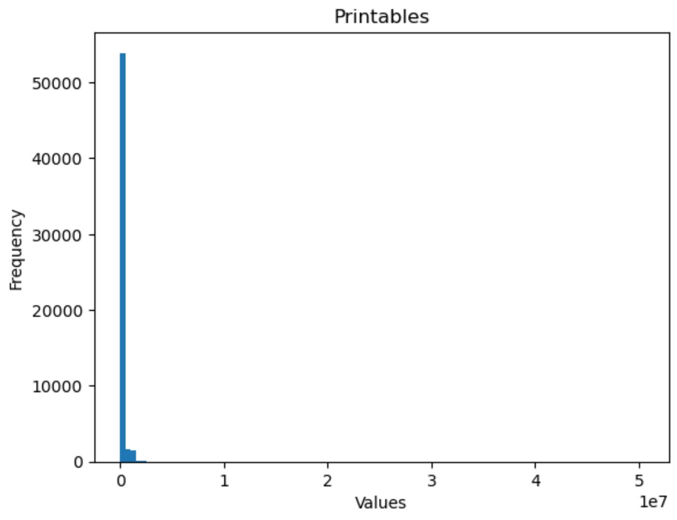

Connection between Registry and Label
Label
Registry
0.0
0.2
0.4
0.6
0.8
1.0
0
500
1000
1500
2000
2500

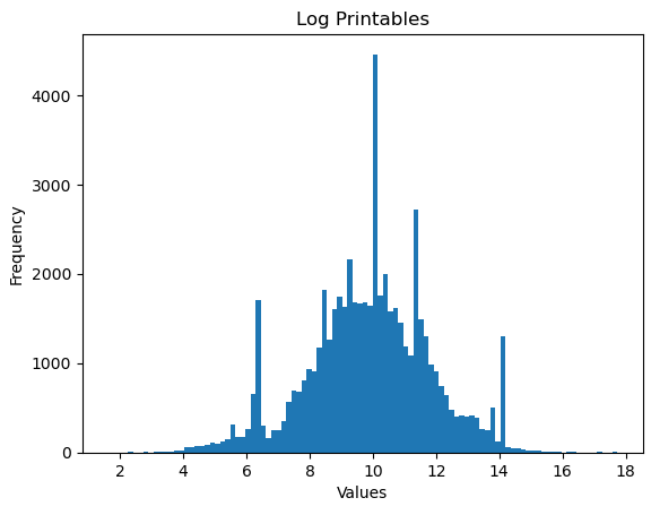

MZ
Frequency
Values
0
10000
20000
30000
40000
50000
0
1000
2000
3000
4000
5000
6000
7000

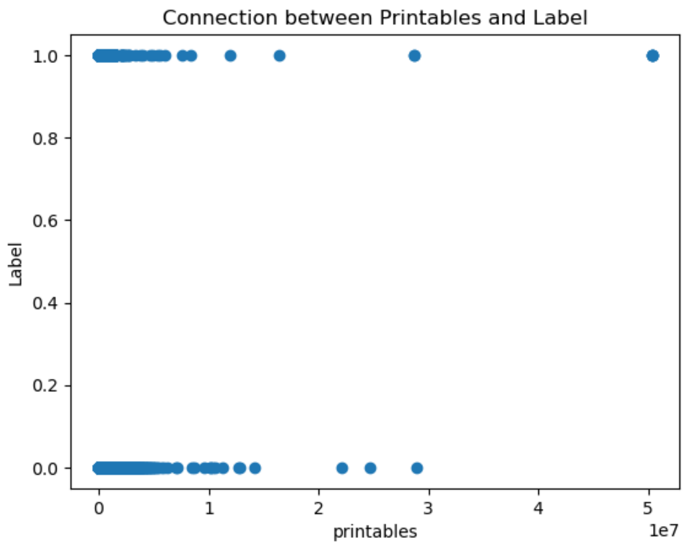

Log MZ
Frequency
Values

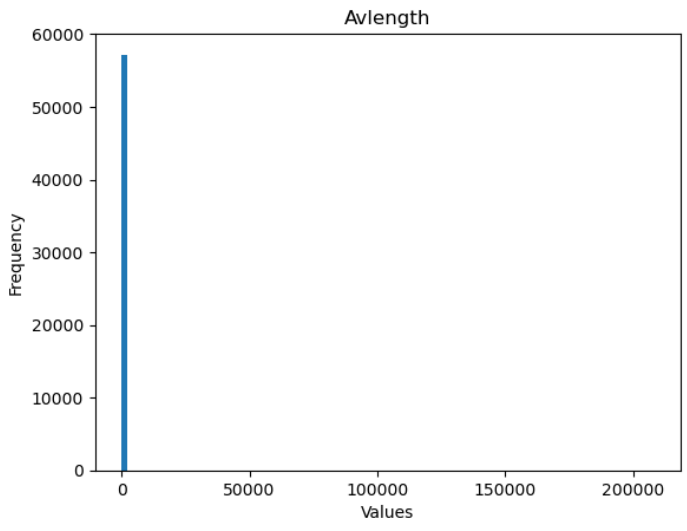

MZs
Count
Range
1
2-10
11-20
21-30
31-40
41-50
51-60
61-70
71-80
>81

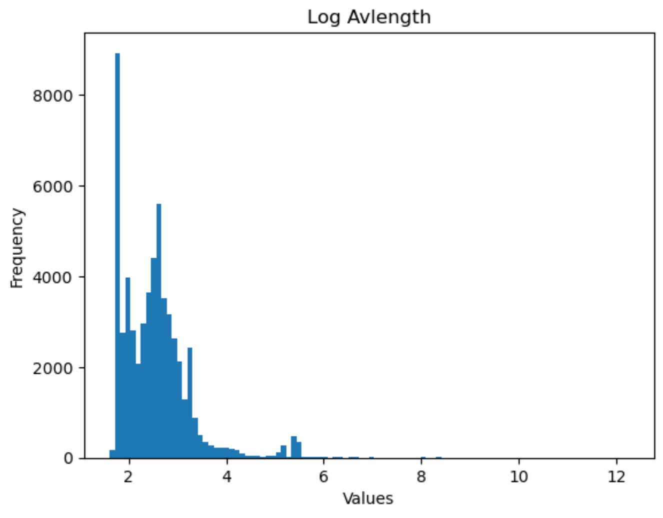

Connection between MZ and label
Label
MZ

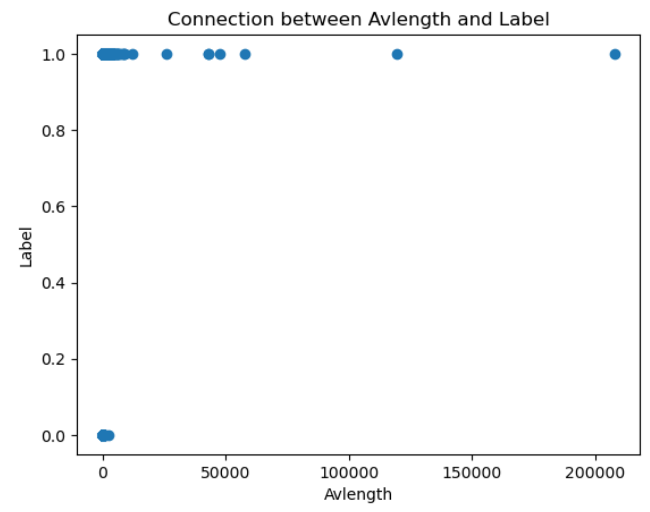

Printables
Frequency
Values
1e7

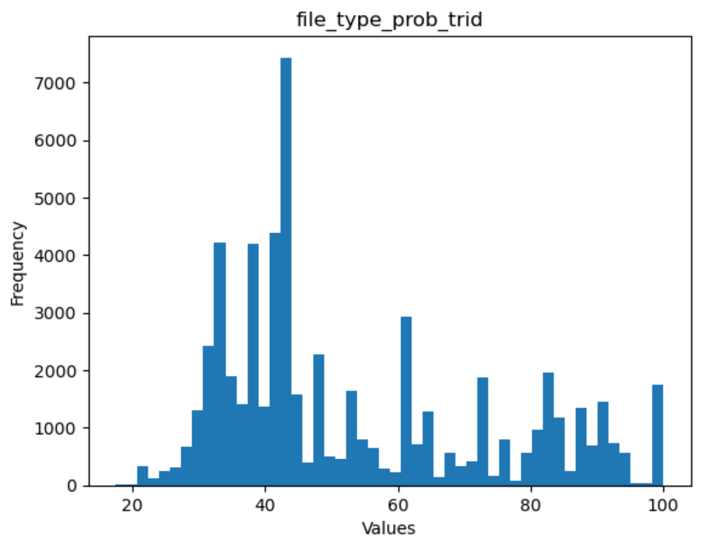

Log Printables
Frequency
Values

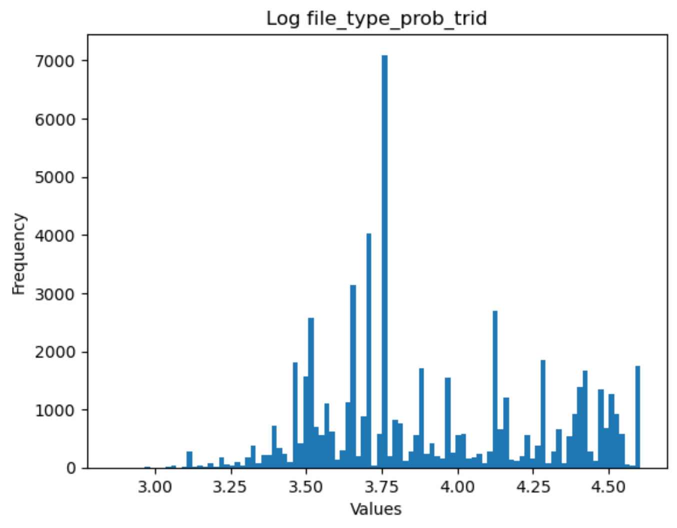

Connection between Printables and Label
Label
printables
1e7

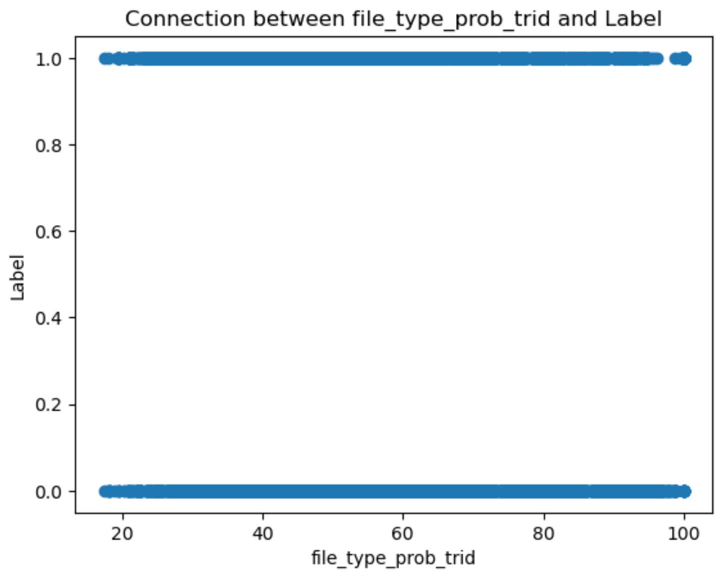

Avlength
Frequency
Values

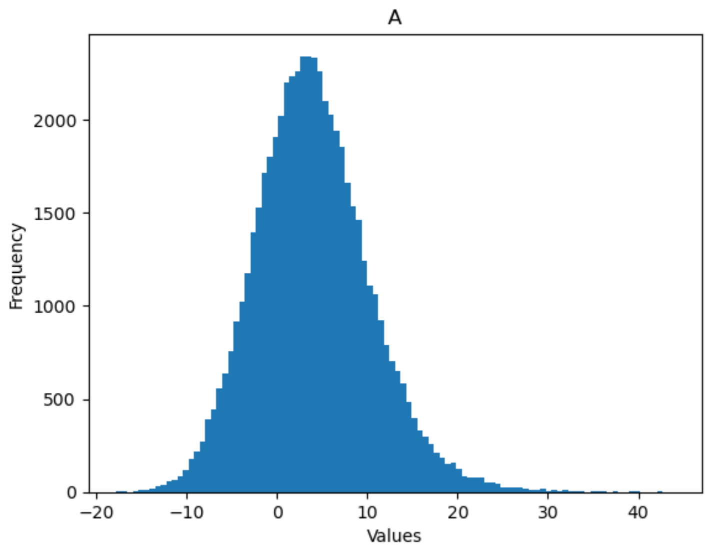

Log Avlength
Frequency
Values

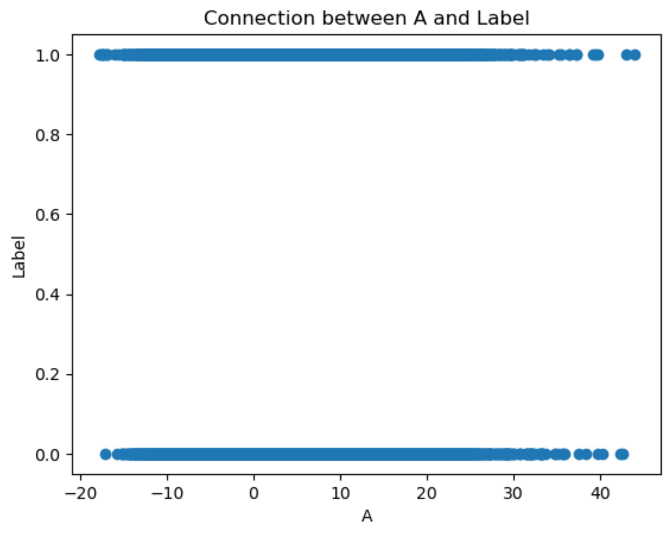

Connection between Avlength and Label
Label
Avlength

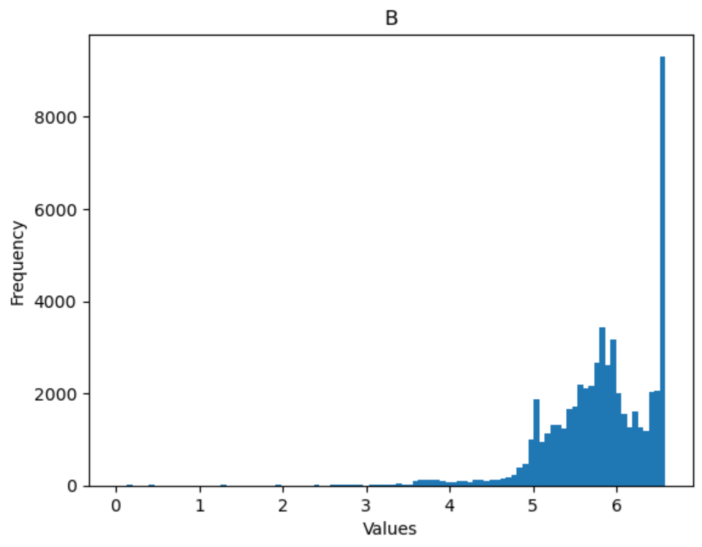

file_type_prob_trid
Frequency
Values

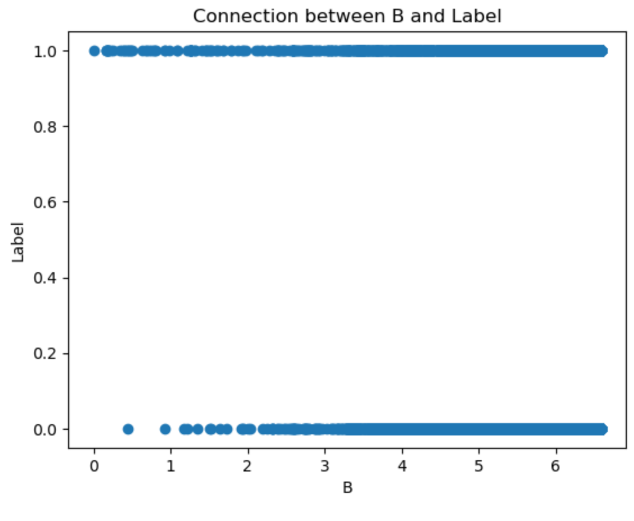

Log file_type_prob_trid
Frequency
Values

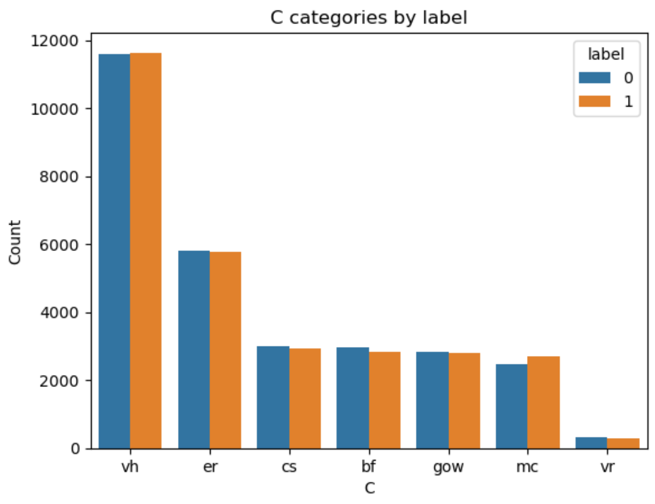

Connection between file_type_prob_trid and Label
Label
file_type_prob_trid

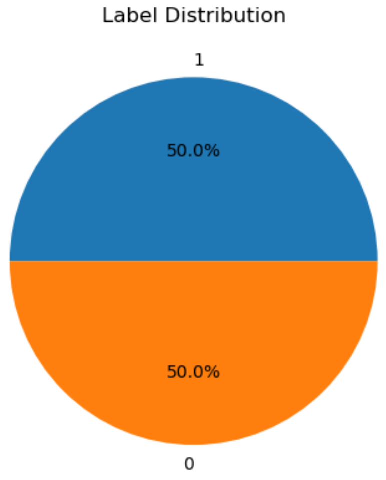

A
Frequency
Values


Connection between A and Label
Label
A

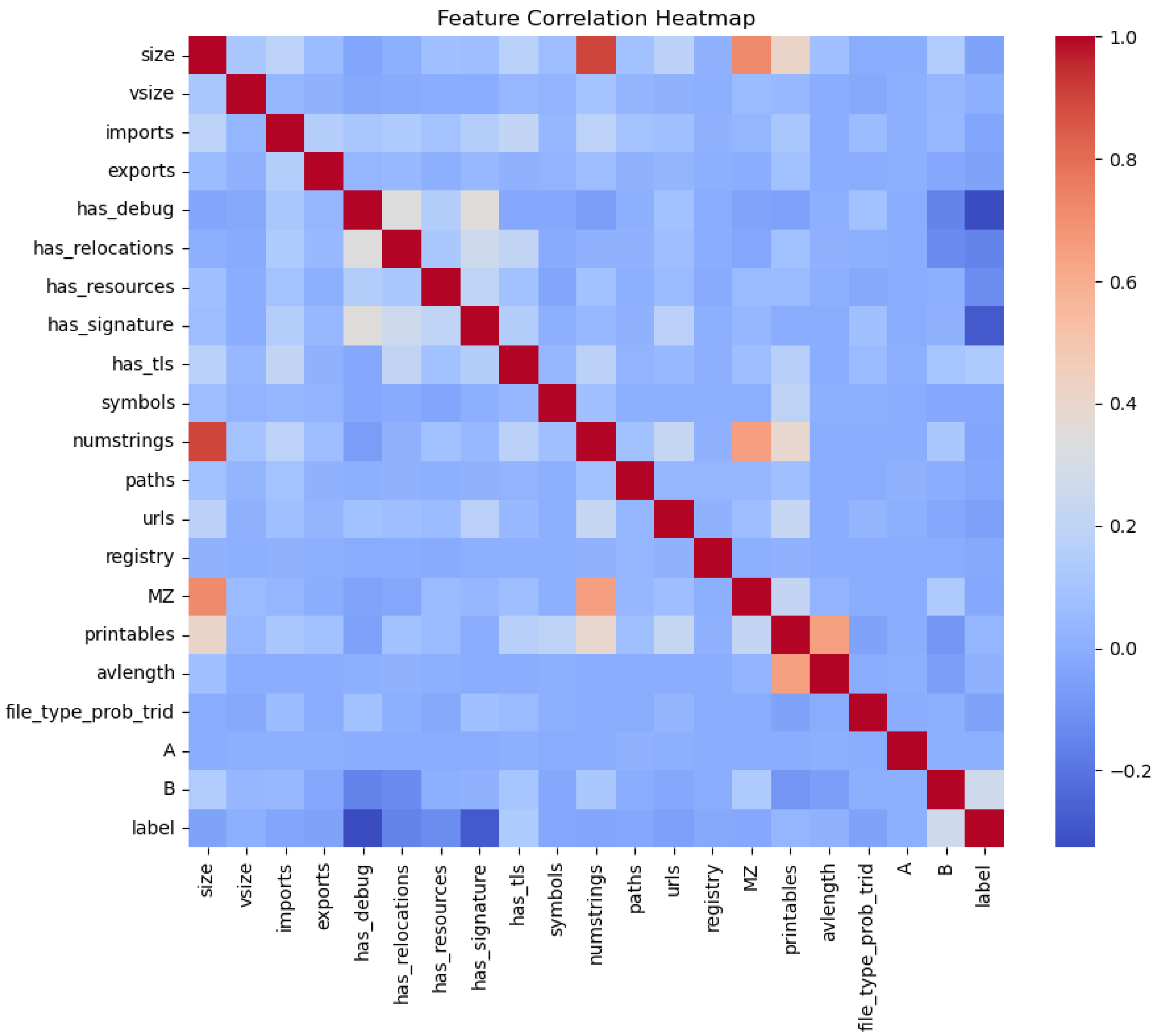

B
Frequency
Values

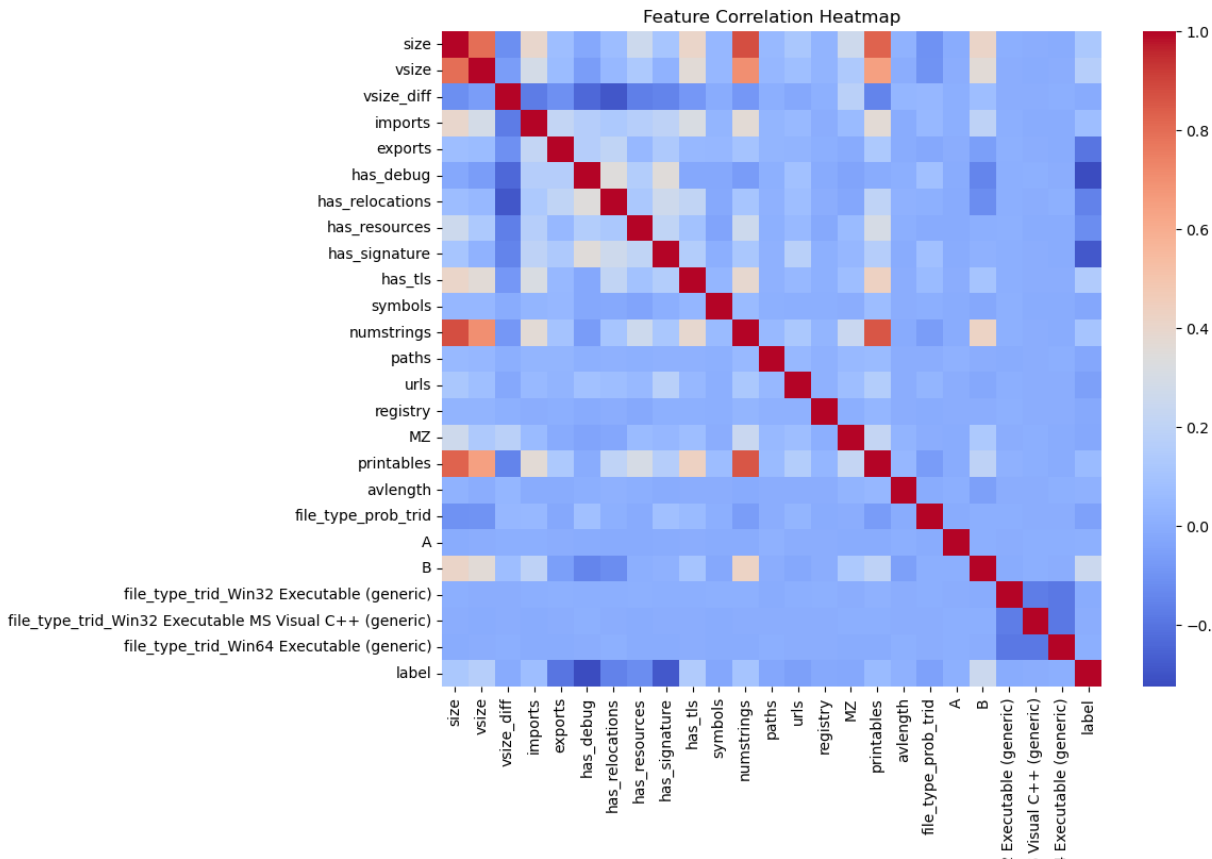

Connection between B and Label
Label
B

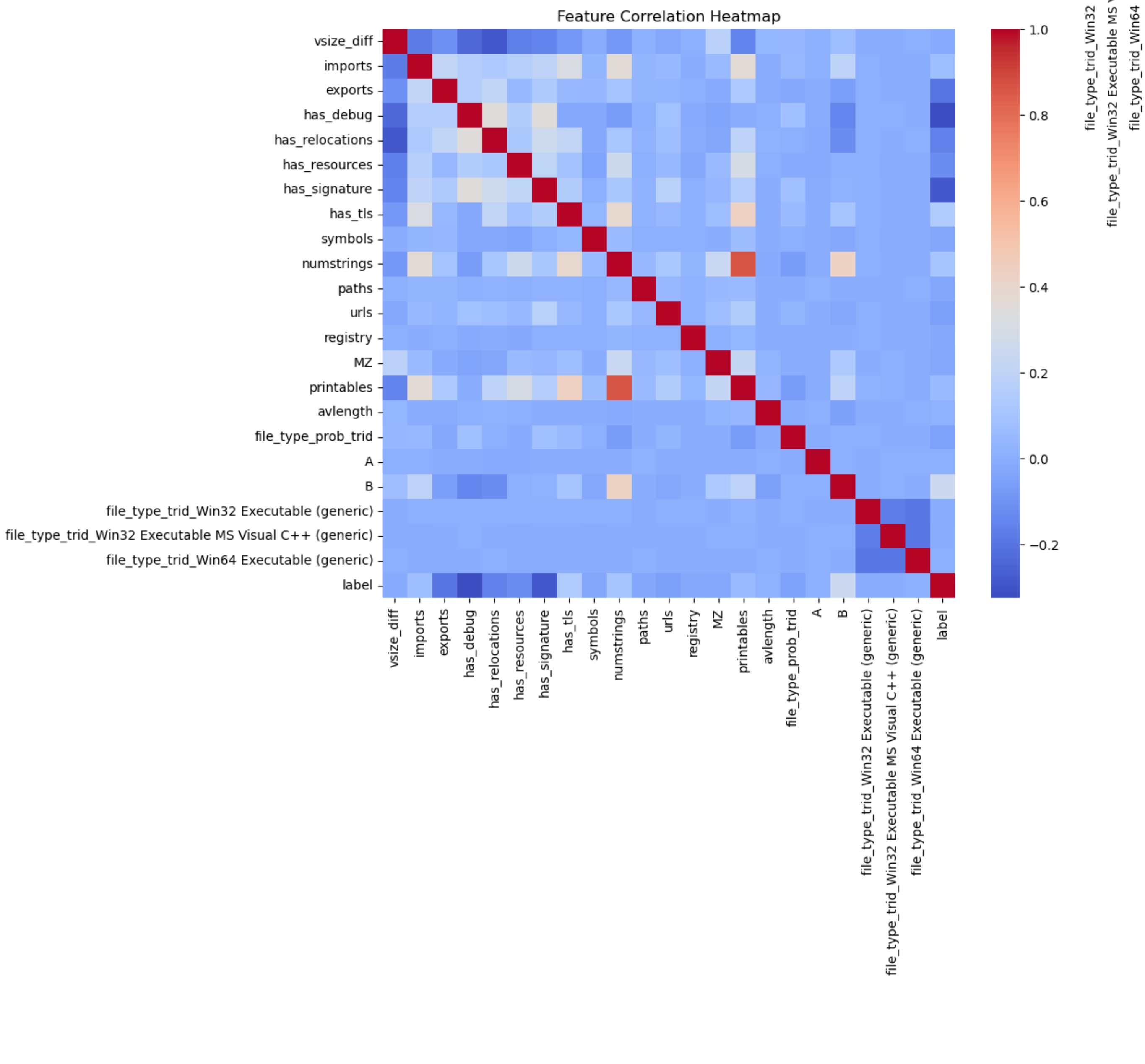




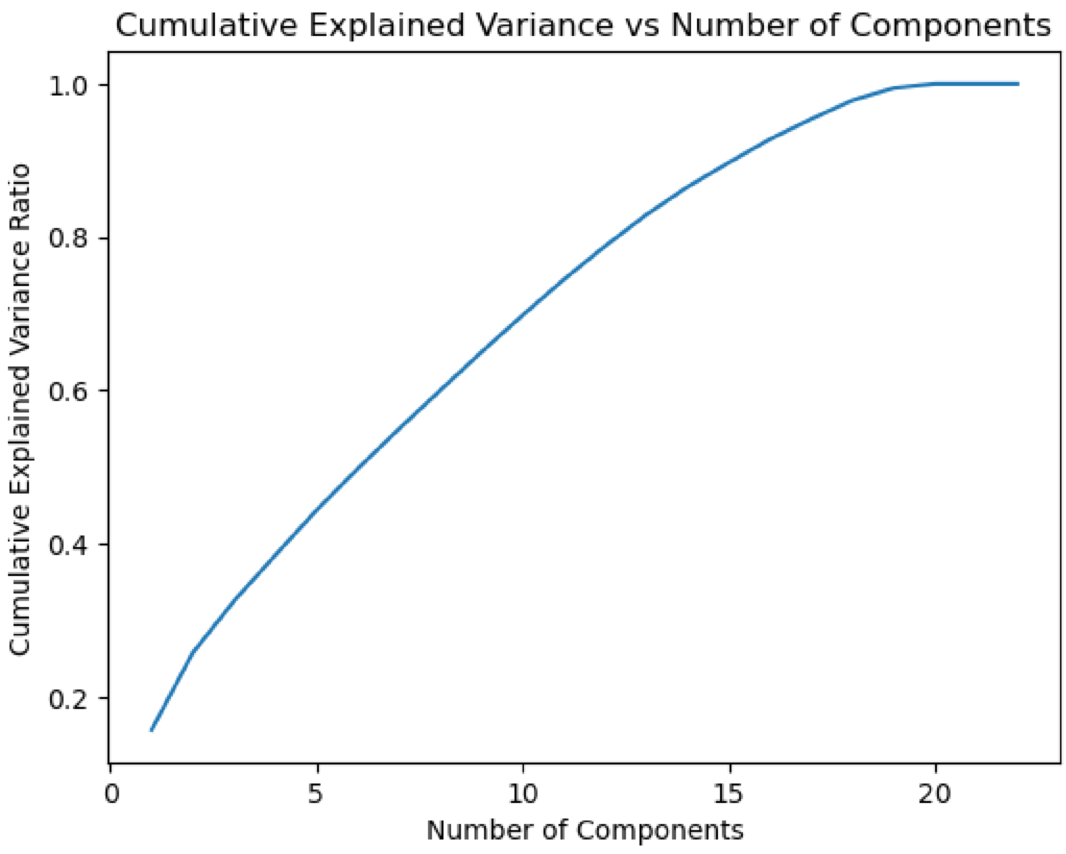
Next are the correlation heatmaps: they will be numbered Fig 2.01, Fig 2.02 and so on.

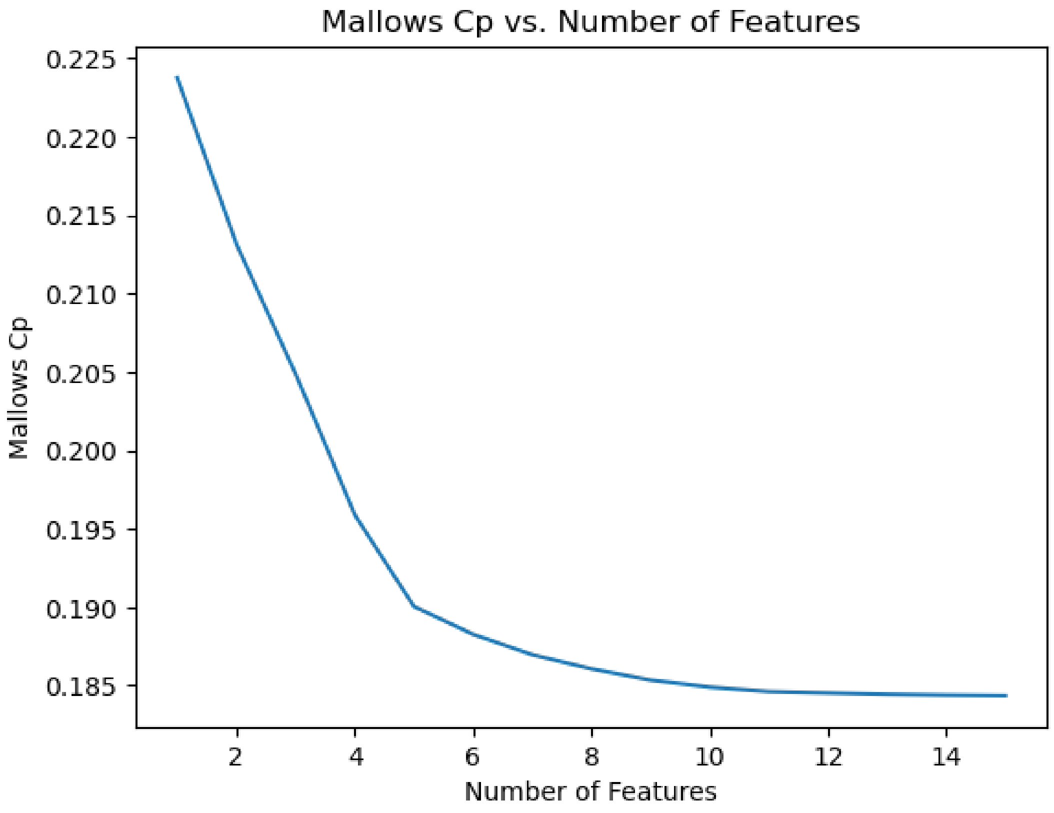



Feature Correlation Heatmap
size
vsize
vsize_diff
imports
exports
has_debug
has_relocations
has_resources
has_signature
has_tls
symbols
numstrings
paths
urls
registry
MZ
printables
avlength
file_type_prob_trid
A
B
file_type_trid_Win32 Executable (generic)
file_type_trid_Win32 Executable MS Visual C++ (generic)
file_type_trid_Win64 Executable (generic)
label
1.0
0.8
0.6
0.4
0.2
0.0
−0.2

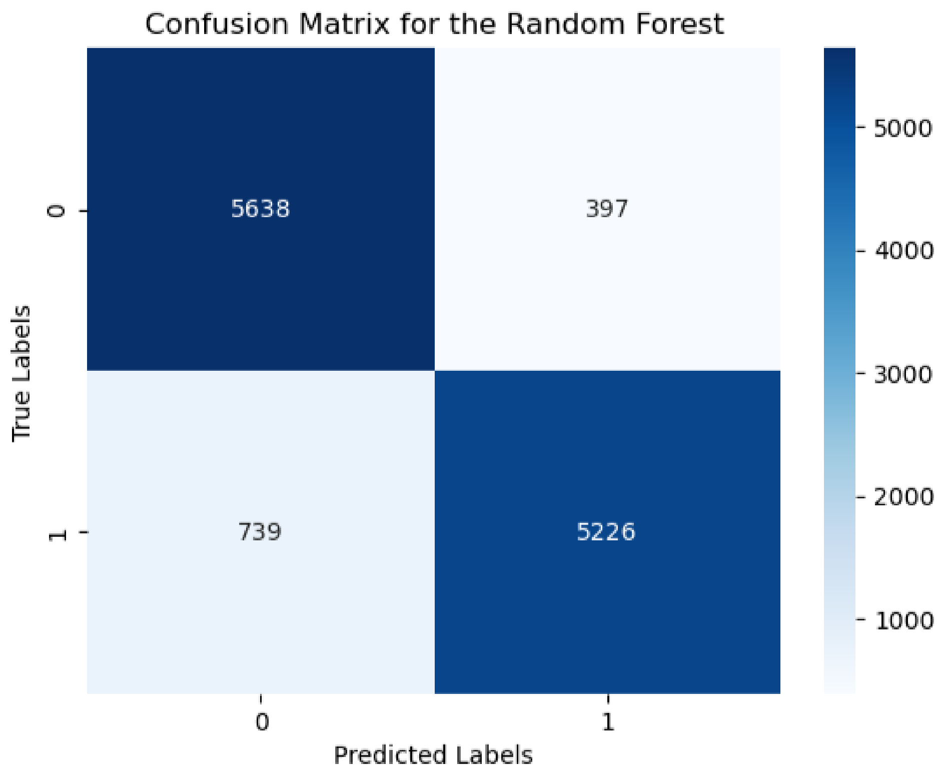

Feature Correlation Heatmap
vsize_diff
imports
exports
has_debug
has_relocations
has_resources
has_signature
has_tls
symbols
numstrings
paths
urls
registry
MZ
printables
avlength
file_type_prob_trid
A
B
file_type_trid_Win32 Executable (generic)
file_type_trid_Win32 Executable MS Visual C++ (generic)
file_type_trid_Win64 Executable (generic)
label
1.0
0.8
0.6
0.4
0.2
0.0
−0.2

PCA and Forward Selection plots: will numbered as Fig 3.01, Fig 3.02 and so on.

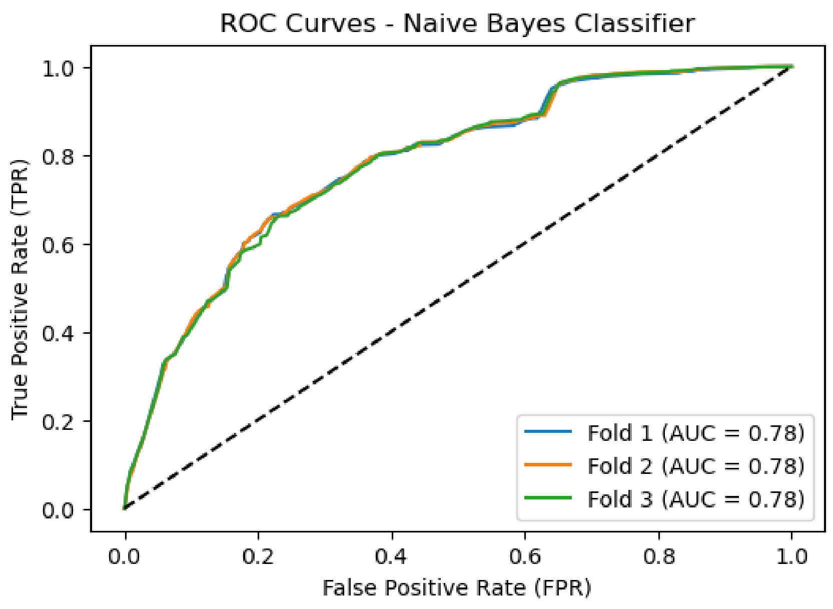


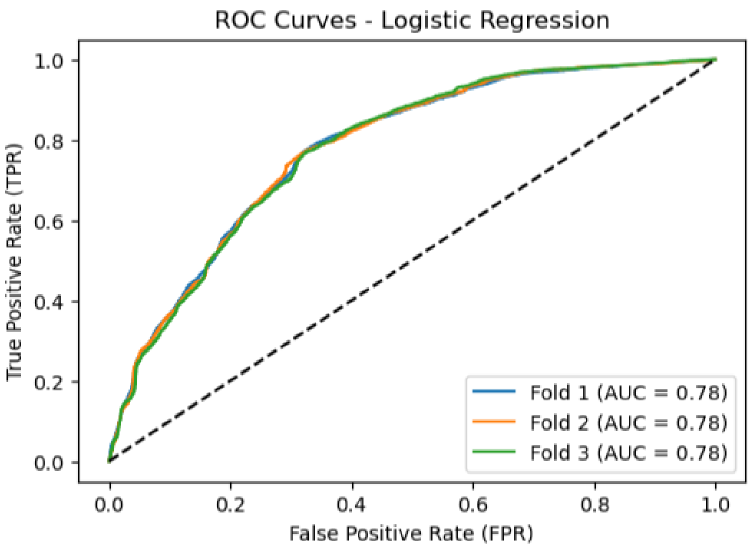


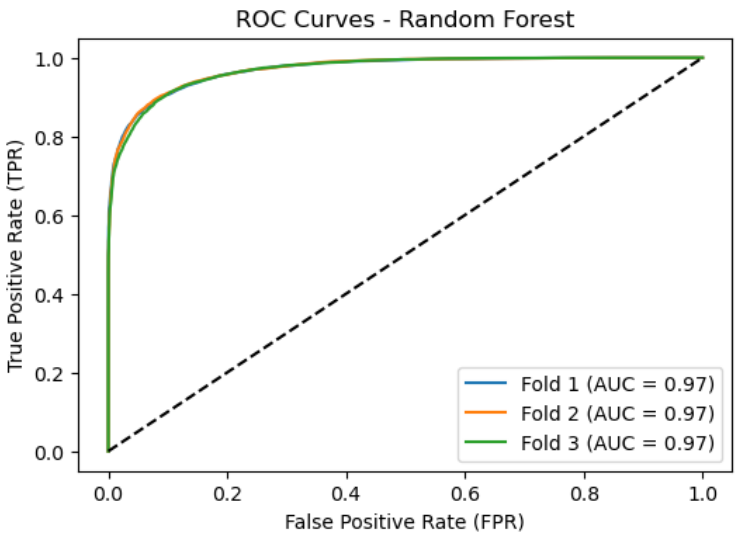
Confusion Matrix plot, will numbered as Fig 4.01:

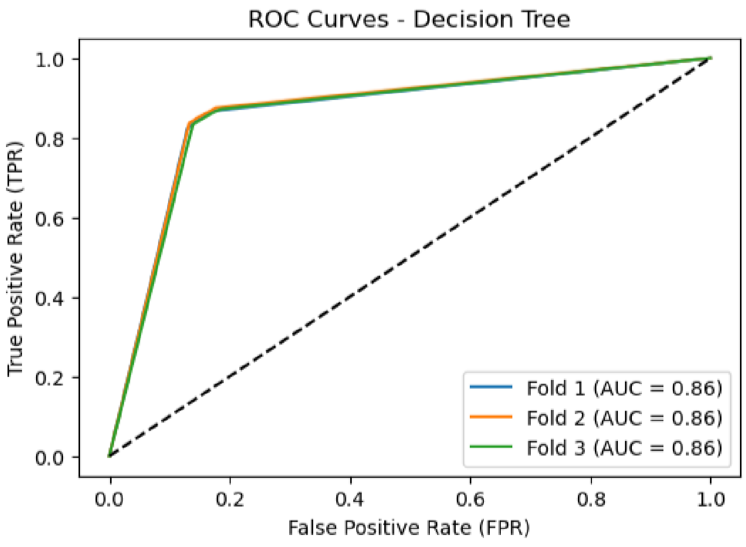


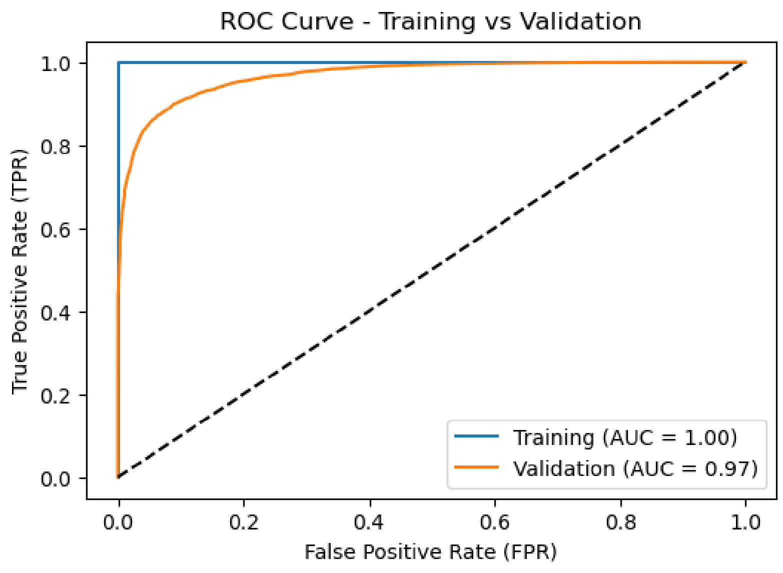
ROC Curves plots for each model, will be numbered as Fig 5.01 and so on.




ROC Curves - Logistic Regression
True Positive Rate (TPR)
False Positive Rate (FPR)
0.0
0.2
0.4
0.6
0.8
1.0
Fold 1 (AUC = 0.78)
Fold 2 (AUC = 0.78)
Fold 3 (AUC = 0.78)


ROC Curves - Random Forest
True Positive Rate (TPR)
False Positive Rate (FPR)
0.0
0.2
0.4
0.6
0.8
1.0
Fold 1 (AUC = 0.97)
Fold 2 (AUC = 0.97)
Fold 3 (AUC = 0.97)


ROC Curves - Decision Tree
True Positive Rate (TPR)
False Positive Rate (FPR)
0.0
0.2
0.4
0.6
0.8
1.0
Fold 1 (AUC = 0.86)
Fold 2 (AUC = 0.86)
Fold 3 (AUC = 0.86)



Additional screenshot from our code - Fig 6.01.

```
Win64 Executable (generic)                 10085
Win32 Executable MS Visual C++ (generic)    8967
Win32 Executable (generic)                  8781
Win32 Dynamic Link Library (generic)        4010
Generic CIL Executable (.NET, Mono, etc.)   3804
                                            ...
VirtualDub Filter Plug-in                      1
GIMP Plugin (Win)                              1
Photoshop filter plug-in                       1
Microsoft CLR native image executable          1
foobar 2000 generic component                  1
Name: file_type_trid, Length: 89, dtype: int64
```