The Iby and Aladar Fleischman
Faculty of Engineering
Tel Aviv University

Final Project – Introduction to ML

Professor: Mr. Dor Bank

Professor Assistance: Mr. Ilan Vasilevsky

Submitted by:

Itay Shnaider – 315507558

Gal Mishan – 315114363

Date of submission: 30.6.2023

❖ **Introduction:**

The following report presents our final project in Introduction to ML Course. This report contains the process we made during the last few weeks developing an AI model that helps us determine if a file is malicious or not.

❖ **Part A: Exploration**

The first part aims to explore the data, learn about feature distributions, their characteristics and more. We randomly split the data into train and validation datasets in order to evaluate our models. From this point all conclusions, visualizations and data processing will be on the train dataset. To begin, we aimed to understand the size and composition of our dataset. We found that there are 48,000 examples with 24 features. In the validation set there are 12,000 examples. The train data splits quite evenly between the two labels.

- ✓ According to our examination using the unique identifier ("sha256") of each example, none of the examples are repeated.
- ✓ We counted the number of Nulls in each feature.
- ✓ Using the unique and dtypes methods, we observed the number of distinct values in each feature. For example, the binary variables contain 2 values (0, 1), and there is no exceptional value present. The categorical variable C has 7 different values. Later on we will explore the impact of these categorical values on the data and investigate any potential relationships or patterns (to be discussed later in the report).
- ✓ We split our features into 3 groups: categorial, numeric and binary:
  - 'binary_vars' contain the next features: 'has_debug', 'has_relocations', 'has_resources', 'has_signature', 'has_tls'.
  - 'numeric_vars' contain the next features: 'size', 'vsize', 'imports', 'exports', 'symbols', 'numstrings', 'paths', 'urls', 'registry', 'MZ', 'printables', 'avlength', 'file_type_prob_trid', 'A', 'B'.
  - 'cat_vars' contain the next features: 'sha256', 'file_type_trid', 'C'.
- ✓ Using the Describe method, we obtained summary statistics for each feature, such as median, mean, percentiles, standard deviation and more (Figure 1).
- ✓ Conclusions from the histogram analysis (Figure 2):
  - Only feature A follows a normal distribution (significant in determining the approach for handling Null values).
  - Some of the binary features has one major value.
- ✓ We have decided to plot the histogram of both labels in each feature to see if there are differences between the distribution in each feature. By examining the histograms separately for different label values, we can observe any variations, trends, or outliers in the feature distributions that might be indicative of their predictive value. This analysis helps in identifying features that exhibit distinct distributions for different labels, which can be valuable for understanding the relationship between features and labels and making informed decisions during the model development and feature selection processes.
  - Most of the features split the same.
  - The few features that do not split the same are: 'avlength', 'vsize', 'registry' and 'exports' (Figure 3, 4).
- ✓ Conclusions from the correlation matrix (Figure 5):
  - There is a high correlation between the following features: 'MZ', 'size', 'numstrings'. High correlation allows us to reduce dimensions without significantly impacting the data since the behavior of these features is similar.
  - There is also a high correlation between 'avlength' and 'printables'.
  - There is no significant correlation (positive or negative) observed among the other features.

- ✓ We wanted to take a better look at the correlation between the labels and the features (Figure 6), It is noticeable that there is no significant correlation between any of the features and the label. Therefore, no specific feature carries more weight in determining whether the label is 1 or 0.
- ✓ We wanted to see how the binary features are distributed with the label (Figure 7), we can see that files that have debug and signature have better chances to be non-malicious. In the other features, it doesn't look like there is a significant difference.
- ✓ Based on our analysis, it's clear that the variables 'symbols', 'exports', and 'registry' are predominantly composed of 0 values.
  - When 'registry' and 'symbols' have a significant number of zeros for both labels, 'exports' shows a slight connection between labels and the value of the feature.
  - Since in 'registry' and 'symbols' the percentage of zeros is high in both labels we decided to drop them.
- ✓ In the binary vars we wanted to take a better look at how the features are distribute between the values. In some features there is a clear majority for specific value while in some there is a slight difference between the values (figure 8). Based on this information, we made the decision to impute the missing values in the binary features with the most frequent value.

❖ **Part B: Handling the data**

- ✓ Categorical Features:

  As shown before, there are 3 categorial features: sha256 ,file_type_trid and C.
  - sha256 - It is the unique identifier (hash) of each file. Since it does not have any impact on the training results, it can be safely removed as a feature.
  - C - The feature has 7 unique values. We performed a segmentation of each value in comparison to the label (Figure 9).
    It is evident that each value is evenly distributed with an equal 50-50 split. Therefore, a specific value does not have any influence on determining whether a file is malicious or not.
    Consequently, we have decided to exclude this feature from further analysis.
  - file_type_trid - It contains 88 different values. To simplify the analysis and modeling process, file types are categorized into families based on the first three letters of each file type. This classification allows us to identify common patterns and group files belonging to the same family together (Figure 10). We created additional binary variables based on the file family to which the file belongs and removed the original feature. However, after several experiments, we saw that it led to overfitting. Therefore, we have ultimately decided to completely remove the original feature and not include the new additional features.

- ✓ Feature selection:

  At the beginning we had 24 features (23 + label). We tried to reduce dimensions as much as possible while keeping high scores in our models. Small dimension problem can improve the chance of not getting overfitted models and can reduce the run time of the models significantly (we have a time constraint of one hour for all code execution).
  To reduce features there are several methods like PCA, backward selection or even manually.
  We made the decision not to utilize PCA because it results in a loss of domain knowledge and hampers our ability to track the importance of each feature in the model.
  - Manually: after analyzing the train dataset we have noticed some features that can be removed due to high correlation, uniqueness, significantly common value or evenly split:
    - 'C', 'file_type_trid' and 'sha256' - all categorial features.
    - 'file_type_prob_trid' - since it's probably meaningless without 'file_type_trid'.
    - 'numstrings' and 'MZ' - high correlation and more null values.
    - 'symbols' and 'registry' - these variables have an overwhelming majority of zeros.

- ‘printables’ - strong correlation with 'avlength' but higher number of null values.
- ‘A’ - it does not have input in the RF and XGboost models according to feature importance method in the models.

- Backward selection: we developed a function that does backward selection using Mallows Cp score as the evaluation metric. Since we saw that Random Forest has the highest AUC score in the models, we decided to use it as regressor. It then iteratively removes one feature at a time and evaluates the model's performance using Mallows Cp score. Finally, it plots the Mallows Cp score against the number of features and returns the indexes of the selected features with the lowest score and returns a list of all important features. We decided not to use it because it has a very long running time, and we have high scores for our model without it (Figure 11).

✓ Outliers:

Removing outliers from a dataset is important for improving the quality of the data, and enhancing model performance. Outliers are often caused by errors or anomalies and can distort the analysis or the training process. By eliminating outliers, we can ensure a more accurate representation of the data and reduce their influence on model parameters, resulting in more reliable and robust predictions. Even though it is important to lose outliers, there is a fine line between losing too much data to outliers and losing less outliers. Both examples can hurt the model and it’s important to find the balance. We have plotted boxplot for each numerical feature and looked for the iqr, upper bound and lower bound. We have chosen to use the 2.5th and 97.5th percentiles as boundary rules when loading the data. This means that we remove values below the 2.5th percentile and above the 97.5th percentile. After few tests we saw that those percentiles can give us a high AUC score but still we lose a significant number of outliers examples. In the labels histogram we noticed that ‘exports’ have more examples in the high values so in this specific feature we decided not to remove outliers at all.

✓ Null handling:

- Binary features: previously we saw how the binary feature distributes between their values. First, we decided to fill the nulls with the most common value of each feature. We wanted to improve our models, therefore we tried to use KNNImputer instead and it improved our models.
  - KNNImputer: It calculates the distances between data points to find the K-nearest neighbors for each missing value. Then, it imputes the missing value by taking the average of the corresponding feature values from those neighbors. The KNNImputer utilizes the similarity between data points to estimate missing values and preserve the overall patterns in the data.
  - We determined that the optimal value for the hyper-parameter was 51 after conducting several trials. This particular value consistently yielded the highest AUC score among our models.

✓ Normalize the data:

We decided to normalize the data since we chose to use SVM and KNN models, which are both distance-based models. By bringing features to a similar scale, it helps avoid dominance of certain features and improves the overall stability and performance of the model.

- Min-Max scaling: First, we tried Min-Max scaling. It works by subtracting the minimum value and dividing it by the difference between the maximum and minimum values. This scaling method is beneficial as it preserves the original distribution, maintains the relative relationships between features, and ensures that all features are on a comparable scale. However, it may not be suitable for datasets with outliers as it compresses most of the data, potentially diminishing the distinctiveness of outliers and affecting the overall data representation.

- We employed a small outlier threshold and utilized mean-std scaling to improve our models successfully.
- Mean-Std scaling: mean-std normalization can mitigate the impact of outliers and extreme values in the data. Since the transformation is based on the mean and standard deviation, it is less influenced by outliers compared to some other scaling techniques.

❖ **Part C: Modeling:**

We developed a grid search function that will help us to choose the best hyper parameter for each model. For more complex models (such as Random Forest and XGBoost), we implemented a different function using random search instead of grid search. This was primarily done to address the long runtime associated with these models. Furthermore, we developed a model evaluation function that uses K-Fold (K=5) and plot the ROC-AUC for each fold. Each model was evaluating by this function along with the best parameters obtained from the grid/random search.

Based on the available models, we applied the first part to the KNN and Logistic Regression models. The second part was utilized for the Decision Tree and Random Forest models. After comparing the performance of the Adaboost model and the Random Forest model, we found that the Random Forest model achieved better results. Therefore, we decided to proceed with the Random Forest model for further analysis. Furthermore, as part of our project, we incorporated an additional tool that was not covered in our course. This tool was specifically utilized when implementing the XGBoost model, and we will provide further details on its usage later in the report.

To review detailed explanation about the models, their hyper-parameters and feature importance please refer to Appendix #2: Models.

These are the Mean AUC scores that we got for each of our models by our model evaluation function:

Model	KNN	Logistic regression	Decision Tree	Random Forest	XGboost
Mean Train AUC-ROC	0.953	0.8	0.923	0.96	0.96

❖ **Part D: Model evaluation:**

✓ **Validation + Overfit**:

After conducting a train-validation split and training the models on the training data, we evaluated the models on the untouched validation dataset. We evaluate our models by AUC score, which evaluates the model's overall ability to distinguish between positive and negative instances. While the model may have a higher vulnerability to false negatives, optimizing the AUC score aims to strike a balance between sensitivity and specificity, considering both types of errors. The models were assessed using the validation data, and the following grades were assigned to each model based on their performance:

- **KNN**: The KNN model achieved a validation accuracy of 95%.
- **Logistic Regression**: The Logistic Regression model achieved a validation accuracy of 80%.
- **Decision Tree:** The Decision Tree model achieved a validation accuracy of 93%.
- **Random Forest**: The Random Forest model achieved a validation accuracy of 97%.
- **XGBoost**: The XGBoost model achieved a validation accuracy of 97%.

To determine if your model is overfitting, it's helpful to compare the training and validation scores. If the training score is significantly higher than the validation score, it could indicate overfitting.
By comparing the AUC scores obtained on the training and validation sets, we observed only slight differences, indicating that there is no significant overfitting in our models. These results indicate that the models have demonstrated good generalization to the validation data, suggesting that they have not overfit the training data. We decided to keep on with the **Random Forest model** due to is high performance on the validation set, while also there was a Negligible difference between the AUC score of the train set:

Model Train AUC score: 0.9972
Model Validation AUC score: 0.9738
AUC Score Difference: -0.0234

- ✓ **Feature Importance Analysis:**
  To gain insights into the predictive power of the features in our model, we performed a feature importance analysis. Using the Random Forest model, we obtained the feature importance scores and visualized them in a plot. This plot depicts the relative importance of each feature in contributing to the model's overall performance. The higher the importance score, the more influential the feature is in determining the target variable. From the feature importance plot (Figure 13), We can assume and try to reach conclusions about the importance of the features. In the appendices there is an analysis of how each feature may affect the probability of a file being malicious

- ✓ **Confusion Matrix**:
  - According to the Confusion Matrix (Figure 12) It is observed that the model has a higher number of false negatives (FN) compared to false positives (FP). The FN value represents the cases where the model incorrectly predicts negative labels for positive instances.
    TP = 5690, FP = 453, FN = 565, TN = 5292
    - **Accuracy** => *(TP+TN)/(TP+FP+FN+TN) = 0.915*
      That means we are correct in 92% precents of the cases when we need to determine whether a file is malicious or not.
    - **Precision** => *TP / (TP+FP) = 0.926*
      That means that in 93% of the cases we determined a file is malicious he indeed was.
    - **Sensitivity** => *TP / (TP+FN) = 0.91*
      That means that from the files that where actually malicious we predicted 91% correctly.
    - **Specificity** => *TN / (TN+FP) = 0.921*
      That means that from the files that wasn't malicious we predicted 92% correctly.

- ✓ The best parameters for the Random Forest model were obtained using RandomizedSearchCV, which performed a randomized search over the parameter space. Due to the random nature of the search process, the final results may exhibit slight variations or deviations when rerun. It is important to consider that the reported performance metrics and conclusions are based on the best parameters found during the search, but minor differences may arise if the search is repeated.

Appendix #1: Work distribution

In general, most of our progress was made when we got together and worked side by side. We consulted a lot with each other, shared our thoughts, and helped find solutions to each other's problems.

Shared responsibility:

- Data exploration
- Visualizations
- Filling NA's
- Modelling
- Final Report

Gal:

- Outliers
- Confusion Matrix
- Pipeline

Itay:

- Feature analysis
- Feature selection
- Model evaluation

❖ **Appendix #2: Models**

✓ **KNN**: (Figure 14)

- The chosen parameters for our KNN model are:
  - ***metric***: 'manhattan'.
    The metric parameter specifies the distance metric used to calculate the distances between instances in the dataset. Manhattan distance calculates the sum of absolute differences between the coordinates of two points.
  - ***n_neighbors***: 11.
    The n_neighbors parameter determines the number of neighbors considered when making predictions. The choice the number of neighbors depends on the complexity of the problem and the characteristics of the dataset. With more neighbors, the model may capture more nuanced patterns, but it could also introduce noise or make the decision boundary less smooth.
  - ***weights***: 'distance'.
    The weights parameter determines how the weights are assigned to the neighbors based on their distance. In this case, 'distance' has chosen, which means that closer neighbors have a higher influence on the prediction than distant neighbors.
- The mean result of our model evaluation function was 0.955.

✓ **Logistic Regression**: (Figure 15)

- The chosen parameters for our Logistic Regression model are:
  - ***C:*** 0.01.
    The C parameter is the inverse of regularization strength. A smaller value of C increases the regularization strength, which helps prevent overfitting by shrinking the coefficients towards zero. This result suggests that a lower regularization strength is preferred for our model.
  - ***penalty:*** 'L1'.
    The penalty parameter determines the type of regularization used in logistic regression. The chosen value of penalty is 'L1', which refers to Lasso regularization. This regularization adds a penalty term to the loss function, encouraging the model to use fewer features by driving some of the feature coefficients to zero.
  - ***solver:*** 'liblinear'.
    The solver parameter defines the optimization algorithm used for model training. The chosen value of solver is 'liblinear'. This result indicates that the 'liblinear' solver is the preferred choice for optimizing the logistic regression model.
- The mean result of our model evaluation function was 0.799.

✓ **Decision Tree**: (Figure 16)

- The chosen parameters for our Decision Tree model are:
  - ***criterion:*** 'entropy'.
    This parameter determines the quality of a split. In this case, 'entropy' is chosen as the criterion. Entropy measures the impurity or disorder of the data at a given node. By using entropy as the criterion, the model will aim to create splits that maximize the information gain and result in more homogeneous subsets of data at each node.
  - ***max_depth:*** 15.
    The max_depth parameter sets the maximum depth of the decision tree. A depth of 15 means that the tree can have a maximum of 15 levels, with each level

representing a split based on a feature. It limits the complexity of the tree and can prevent overfitting.

- ***min_samples_split:*** 9.
  This parameter determines the minimum number of samples required to split an internal node. In our case, the value of 9 means that a node will only be split if it contains at least 9 samples
- ***min_samples_leaf:*** 3.
  This parameter sets the minimum number of samples required to be at a leaf node. With a value of 3, leaf nodes will only be created if they have at least 3 samples. It helps to control the depth of the tree and prevents overfitting.

- The mean result of our model evaluation function was 0.925.

✓ **Random Forest**: (Figure 17)

- The chosen parameters for our Random Forest model are:
  - ***n_estimators:*** 250.
    This parameter specifies the number of decision trees in the random forest ensemble. In our case, the value of 250 means that the ensemble consists of 250 decision trees.
  - ***criterion:*** 'gini'.
    This parameter defines the function used to measure the quality of a split. In our case 'gini' is the best criterion. Gini impurity is a measure of node purity, where lower values indicate more homogeneous classes within the node. It is commonly used as the default criterion for Decision Trees.
  - ***max_depth:*** 60
    This parameter sets the maximum depth of each decision tree in the random forest. With a value of 60, each tree can have a maximum of 60 levels, limiting the complexity of the individual trees.
  - ***min_samples_split:*** 7.
    This parameter determines the minimum number of samples required to split an internal node during tree construction. With a value of 7, a node will only be split if it contains at least 7 samples.
  - ***min_samples_leaf:*** 3.
    This parameter sets the minimum number of samples required to be at a leaf node. A value of 3 means that a leaf node can have 3 samples. It helps to control the tree's complexity and can prevent overfitting.
  - ***max_features:*** 'log2'
    This parameter determines the maximum number of features to consider when looking for the best split at each node. The value 'log2' suggests that the number of features to consider is the base-2 logarithm of the total number of features.
- The mean result of our model evaluation function was 0.97.

✓ **XGboost:** (Figure 18)

- Brief explanation: XGBoost is a machine learning algorithm that's part of the ensemble learning family, just like the AdaBoost algorithm we learned about. But they work differently and have their own advantages. AdaBoost focuses on training weak learners, like decision trees, in a sequence. It gives more importance to the samples that were misclassified in each iteration. By adjusting the weights of the training examples, it focuses on the difficult cases and combines multiple weak learners to create a strong final model. In contrast**, XGBoost uses gradient boosting and gradient descent techniques** to build a sequence of decision

trees. It constructs each tree by minimizing a specific objective function that includes a loss function and a regularization term. This approach allows XGBoost to handle more complex relationships and optimize the model's performance.

- The chosen parameters for our XGboost model are:
    - ***n_estimators:*** 171.
      With 171 estimators, the model will consist of a larger ensemble of trees. Increasing the number of estimators can improve performance up to a certain point, after which the returns diminish. However, it's important to consider the computational cost associated with a larger number of estimators.
    - ***max_depth:*** 5.
      This parameter specifies the maximum depth of each decision tree in the ensemble. In this case, a value of 5 limits the complexity of the trees and strikes a balance between capturing intricate patterns and preventing overfitting.
    - ***learning_rate:*** 0.3.
      This parameter determines the step size or shrinkage applied to each tree's contribution during boosting. With a learning rate of 0.3, the contribution of each tree in the ensemble is scaled down, making the model more robust. A lower learning rate helps to reduce the impact of each individual tree, which may require a higher number of trees (n_estimators).
    - ***gamma:*** 1.87.
      The gamma parameter controls the minimum loss reduction required to make a split in a decision tree-based model. In this case, the value of 1.87 indicates that the model will be more selective when deciding to make splits, resulting in a more conservative and simpler model.
    - ***reg_alpha:*** 3.
      This parameter is a regularization term that penalizes the magnitudes of the model's weights. A value of 3 suggests a moderate regularization strength, which can be effective in reducing overfitting if the model is complex.
    - ***reg_lambda:*** 0.78.
      This parameter controls the L2 regularization term on the weights. With a value of 0.78, the regularization strength is moderately applied, contributing to the reduction of overfitting.

- The mean result of our model evaluation function was 0.96.

❖ **Appendix #3: Feature Importance analysis**

1. **B**: Without additional context, it's difficult to provide specific insights for this feature.
2. **imports**: The "imports" feature indicates the number of external functions imported by the file. Malicious files may have a different set or a larger number of imports compared to benign files, as they may require additional functionalities for their malicious activities.
3. **avlength**: This feature refers to the average length of string within the file. Malicious files may exhibit distinctive patterns or deviations in their component lengths, potentially indicating the presence of obfuscated or malicious code.
4. **size**: The "size" feature represents the overall size of the file. Malicious files can sometimes be larger due to the inclusion of additional malicious payloads or code, while benign files tend to have smaller sizes.
5. **vsize**: This feature refers to the virtual size of the file, which relates to the amount of memory required to load the file. Similar to the "size" feature, malicious files may have larger virtual sizes compared to benign files, reflecting the presence of potentially harmful or complex code.
6. **urls**: The presence or frequency of URLs within a file could be indicative of network activity or communication with malicious servers. Malicious files may exhibit a higher frequency of suspicious URLs.
7. **has_debug**: The presence of debug information in a file could indicate that it was compiled for debugging purposes. Malicious files might be less likely to have debug information, as it could make them easier to analyze and detect.
8. **has_signature**: This feature likely indicates whether the file has a valid digital signature. Malicious files may be less likely to have signatures since they often attempt to avoid detection and tampering.

The feature importance analysis revealed that the "has_debug" and "has_signature" features have the highest importance in classifying files out of the binary vars. This aligns with our earlier exploration, where we observed that files with debugging information or valid signatures were more likely to be labeled as non-malicious. These findings highlight the significance of these features in determining the probability of a file being malicious.

❖ Appendix #4: Plots & Graph

Figure 1:

	count	mean	std	min	25%	50%	75%	max	median
size	48000.0	-0.00	1.00	-0.33	-0.30	-0.22	-0.06	7.153000e+01	-0.22
vsize	48000.0	1983077.27	21864285.55	544.00	126976.00	466944.00	1835008.00	4.278288e+09	466944.00
imports	48000.0	106.07	206.08	0.00	1.00	39.00	146.00	1.504700e+04	39.00
exports	48000.0	30.14	498.33	0.00	0.00	0.00	0.00	4.884000e+04	0.00
has_debug	48000.0	0.40	0.48	0.00	0.00	0.00	1.00	1.000000e+00	0.00
has_relocations	48000.0	0.54	0.48	0.00	0.00	1.00	1.00	1.000000e+00	1.00
has_resources	48000.0	0.86	0.34	0.00	1.00	1.00	1.00	1.000000e+00	1.00
has_signature	48000.0	0.25	0.43	0.00	0.00	0.00	0.25	1.000000e+00	0.00
has_tls	48000.0	0.23	0.41	0.00	0.00	0.00	0.23	1.000000e+00	0.00
paths	48000.0	1.64	34.81	0.00	0.00	0.00	1.00	5.324000e+03	0.00
urls	48000.0	12.45	74.19	0.00	0.00	0.00	12.45	9.387000e+03	0.00
avlength	48000.0	31.53	1039.46	5.00	7.45	12.48	18.66	2.079909e+05	12.48
B	48000.0	5.81	0.62	0.00	5.51	5.83	6.30	6.580000e+00	5.83

Figure 2:

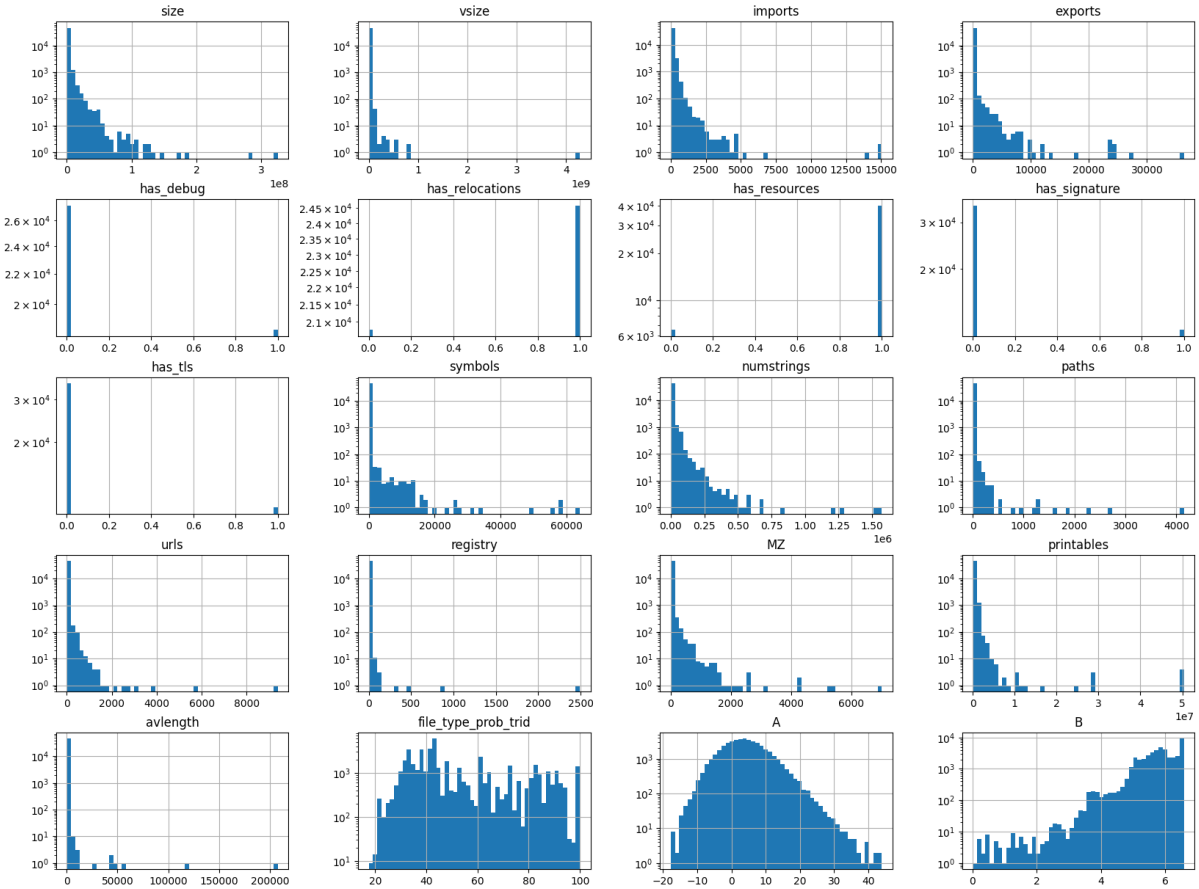

Figure 3,4:

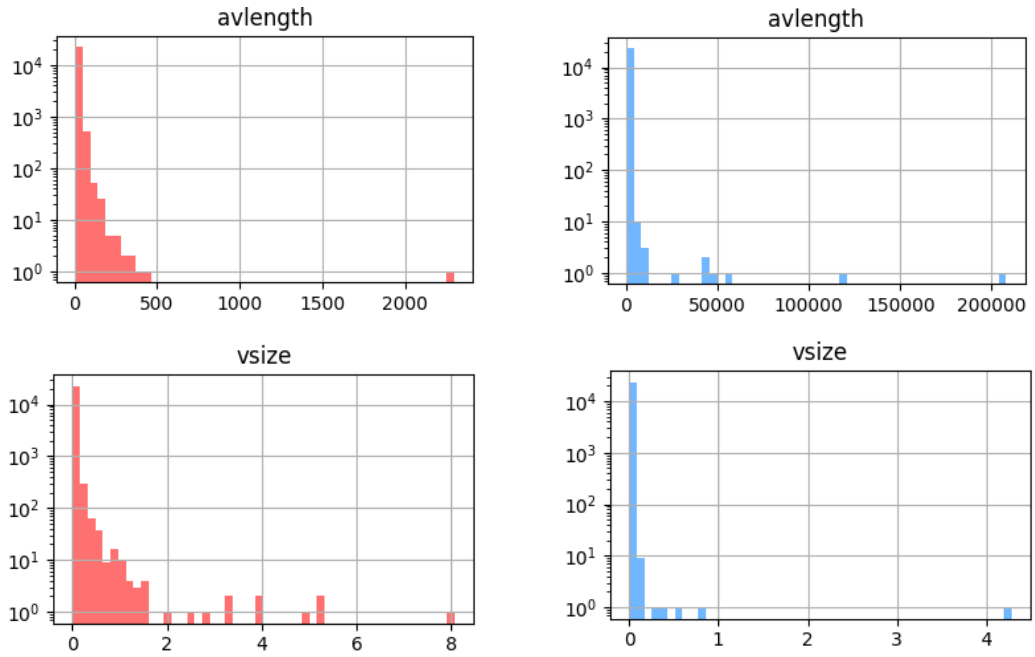

Figure 5:

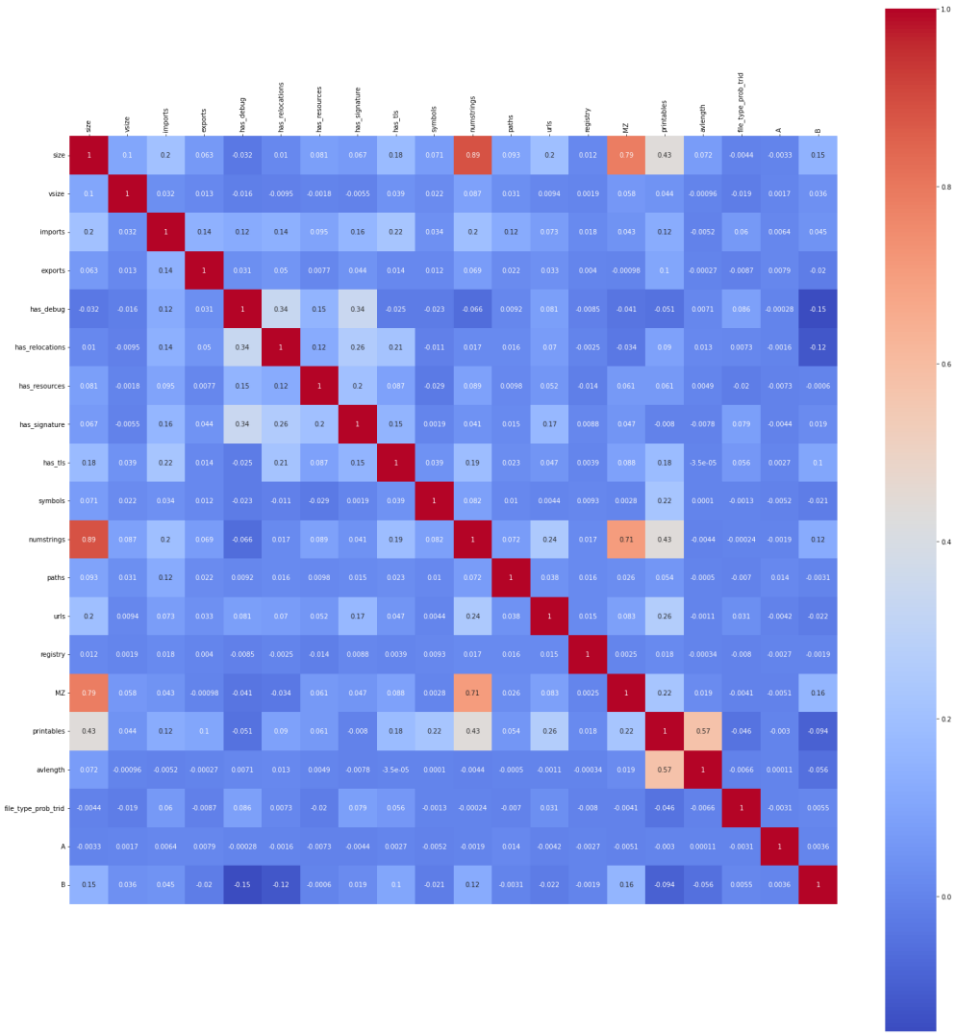

Figure 6:

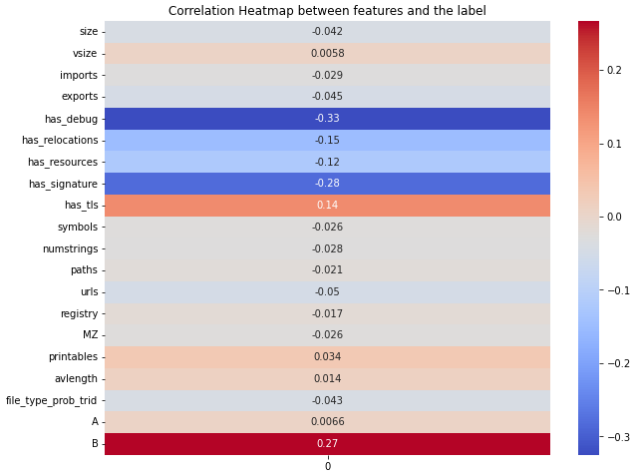

Figure 7:

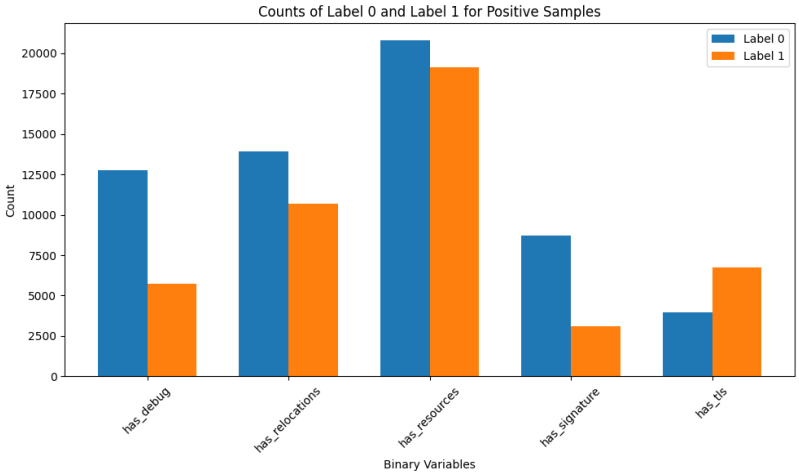

Figure 8:

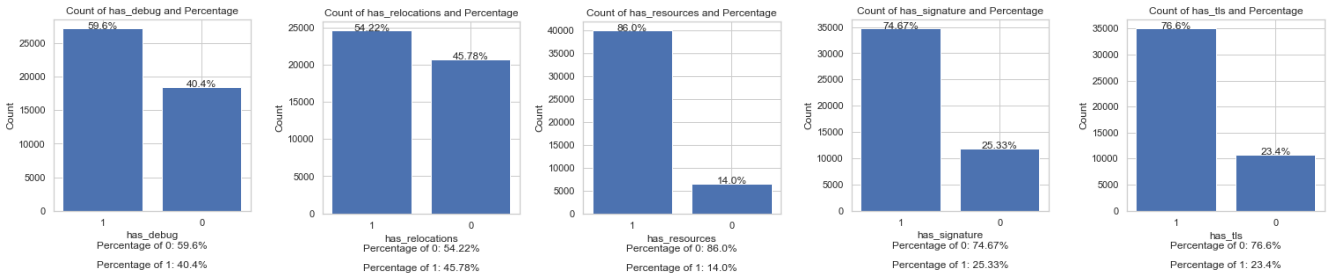

Figure 9:

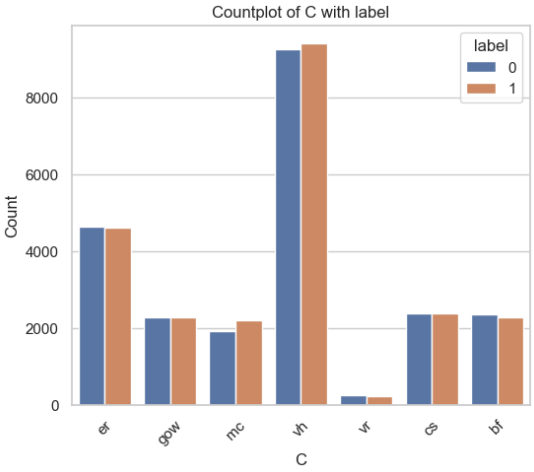

Figure 10:

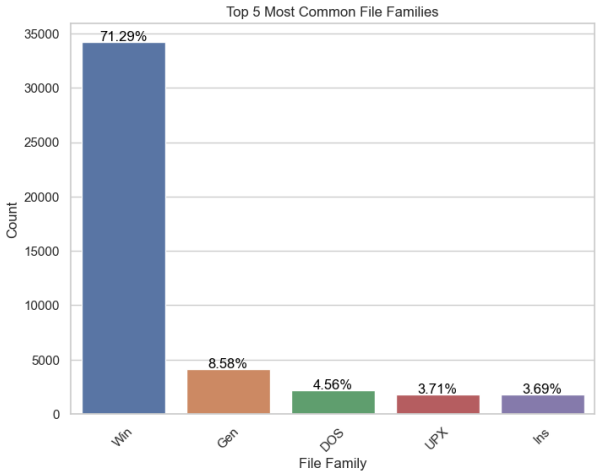

Figure 11:

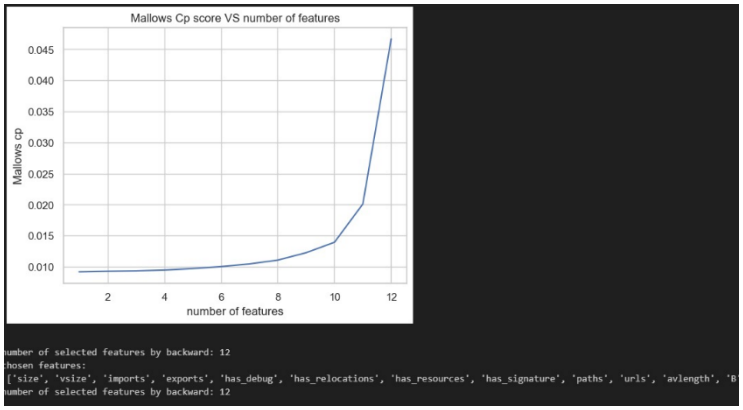

Figure 12:

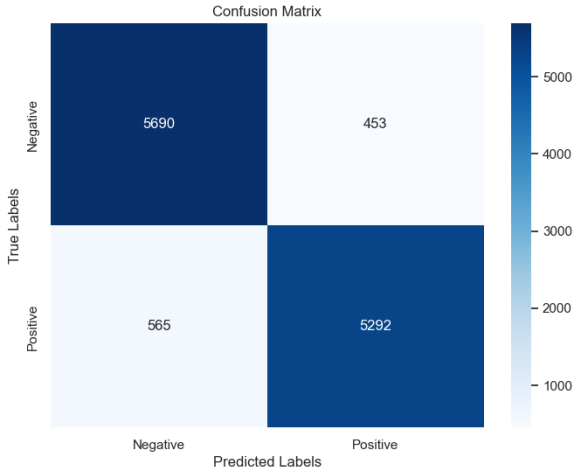

Figure 13:

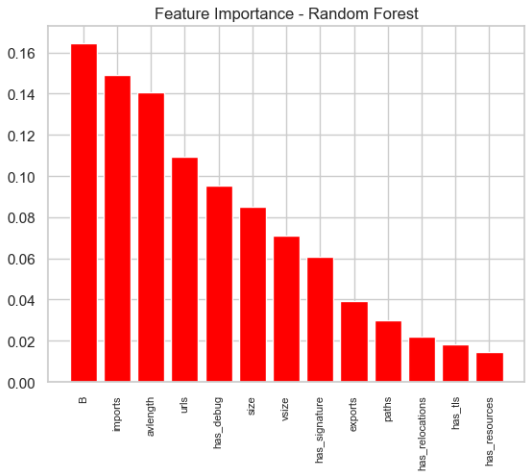

Figure 14(KNN):

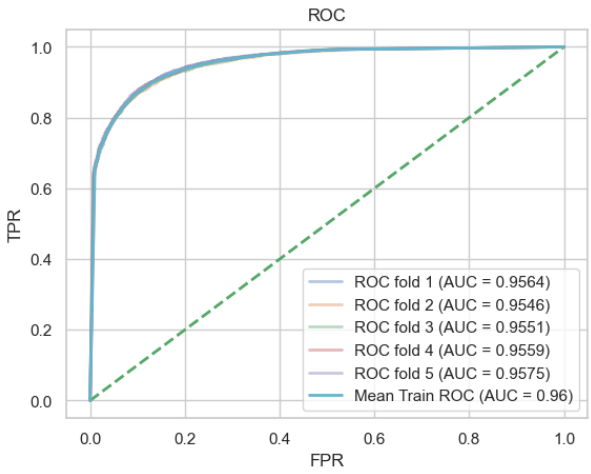

Figure 15(Logistic Regression):

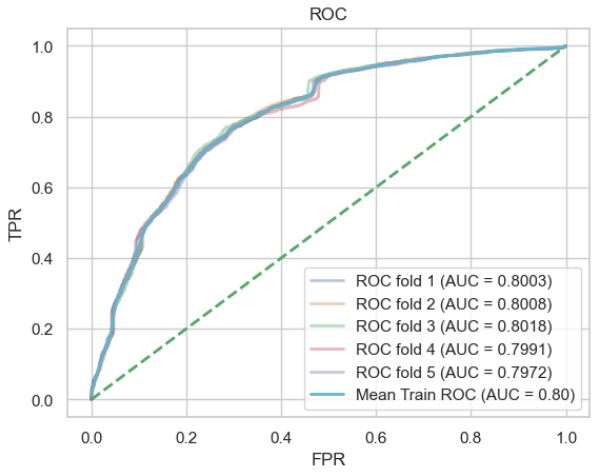

Figure 16(Decision Tree):

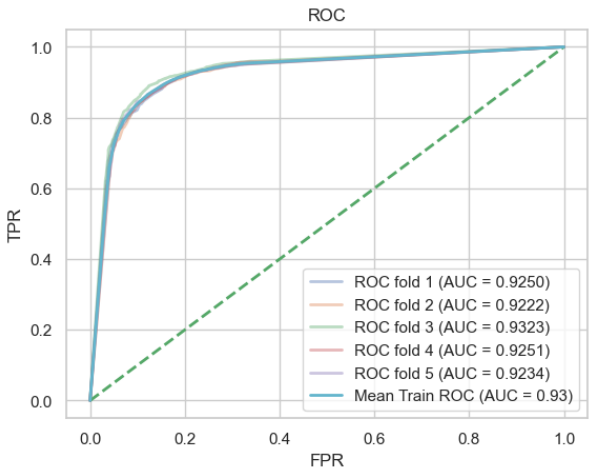

Figure 17 (Random Forest):

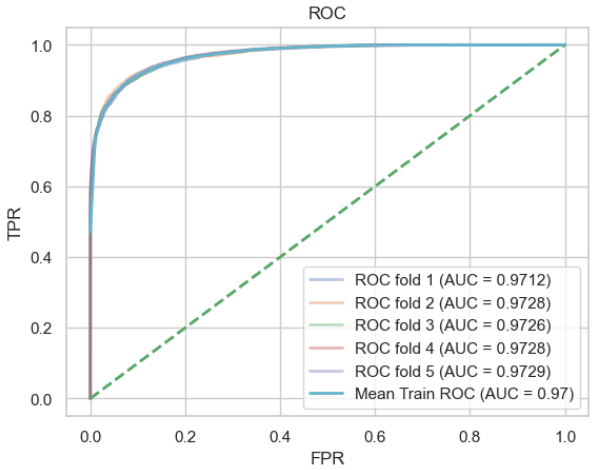

Figure 18 (XGboost):