Final Project Report: Malware Files Detection

Authors: Sophie Margolis and Jeremy Sprung.

In our project, we focused on analyzing Portable Executable (PE) files, which are executable file formats used in various operating systems. The name "Portable Executable" refers to the fact that the file format is not architecture-specific, making it versatile across different systems. Our primary goal was to develop a binary classifier capable of accurately identifying whether a given file is malicious or benign. To achieve this, we employed advanced techniques to examine the relationships between the file's features and identify patterns indicative of malicious behavior.

Part 1: Exploratory Data Analysis

After having imported the necessary libraries to begin our code, we loaded the **'train.csv**' and '**test.csv'** files respectively into pandas dataframes called **train_data** and **test_data**, and used the `head`,` shape` and `info` functions to have a look into the train data. From these functions we could see that we have a **large data set** (60000, 24) with different types of features: **continuous** and **binary** with float64 or int64 types values, and **categorical** features containing objects. For better exploration we **created 3 lists** containing the names of the three types of columns, called **binary_features**, **categorical_features** and **continuous_features**. We examined the **statistics** table to gain insights into various aspects such as the presence of negative values, the average values and the spread (std) of the features.

Features distributions- We chose to plot all our **continuous features** as histograms using **seaborn** (Appendix 1). We applied **np.log** to all continuous features except **'file_type_prob_trid'**, **'A'**, and **'B'** features to enhance visualization and interpretation, as these features showed extreme skewness. For the histograms, we used bins=20 to improve visualization and bin integration (we also experimented with 10 and 25 bins). Additionally, we set kde=True and kde_kws={'bw_method': 0.15} to include a smoothed representation of the underlying distribution using a detailed bandwidth. Our main observation was that several of our **continuous features displayed skewness or non-bell-shaped distributions, except for the 'A' feature, which followed a normal distribution.**

Then we looked at the number of **unique values** in our 3 **categorical features**, in which way we could see that the feature **`sha256`**, which contains the files names, wouldn't be useful in our dataset as it has 60000 different values which means that each file has its unique name and there would be no point in showing its proportion. The **`file_type_trid`** feature contains 87 **unique values**. We plotted it as stacked bar plots (Appendix 2) to visualize the proportion of different file types and the distribution of malicious files within each file type. Our goal was to identify the most common file types and detect any discernible patterns. We observed that after a few highly popular values, the number of files dramatically decreased. Notably, we noticed **a significant proportion of malicious files in the following file types:** InstallShield setup, UPX compressed Win32 Executable, Windows Control Panel Item (generic), Win32 EXE PECompact compressed (generic), and NSIS - Nullsoft Scriptable Install System. Additionally, **100% of malicious files were found in the DOS Executable Generic file type.** As for the **'C'** feature, which

consists of 7 unique values, we plotted its proportions (Appendix 3) in a similar manner to **`file_type_trid`**. We observed that the most prevalent value was **'vh'**, while the least prevalent was **'vr'**. .All of these values exhibited approximately 50% malicious files

Our next step was to examine the **binary features** and assess the proportion of malicious files (Appendix 4). Firstly, we wanted to determine the distribution of labels, considering that certain models assume balanced data and may require specific techniques to handle imbalanced datasets. Upon inspection, we found that **the labels are evenly balanced**. Regarding the other binary features, we thought of several explanations. We noticed that files with **no debug information** had a malicious presence exceeding 50%. This is often the case because malicious actors frequently attempt to remove or hide debug information in their files, making detection more challenging. Additionally, **files with thread local storage (tls) and without resources, relocations or digital signature** serve as additional indicators of potential .maliciousness. The **full conclusions** written in the notebook near the #BINARY FEATURES cell

Features correlation- We plotted a heatmap (Appendix 5) to evaluate the Pearson correlation coefficients between all of the features. We observed **two strong positive linear correlations** that were quite sensible: **size** and **number of strings** and **MZ** and **size**. However, we decided not to remove these correlations as there might be non-linear relationships that we will explore later. We didn't want to take the risk of removing these features prematurely, as they could potentially contribute to our models.

Part 2: Pre-processing

We separated our training data into two parts: **X**, which contains **all the feature columns**, and **y**, which contains only the **label column**. Then, we used the **train_test_split** function to split the data again into **training** and **validation sets**. After that, we combined **X_train** with **y_train** to create **X_y_train** and **X_val** with **y_val** to create **X_y_val**. This allows us to preprocess the entire training data and then preprocess the validation and test data together. Firstly, all the changes we make will be on the **X_y_train** dataset.

Investigating missing value- We implemented the **get_missing_values** function to determine the percentages of missing values for each feature. The results showed that out of the 24 features, 19 of them have missing values. However, the proportions of missing values in all features are below 6.3%, indicating that the missing values are relatively low. Therefore, we made the decision not to remove any feature based on this criterion. Additionally, we found that the rows with missing values account for 58.10% of the data, and we don't want to lose a significant amount of information. The **missing values in the continuous features** were filled with the medians of their respective columns, as it is known to be better for handling skewness. However, for the 'A' column, which follows a normal distribution, we filled the missing values with the mean. As for the categorical features, we observed that only 3.41% of the rows had missing values. Therefore, we made the decision to delete these rows using the `dropna` method. The **binary features** had 20% of **rows with missing values.** To handle these missing values, we decided to fill each one with the most frequent value of its respective column. After double-checking, we saw that we treated **all** of our missing values.

Normalization- We used the **PowerTransformer** class from scikit-learn. Unlike other methods we encountered in homework (StandardScaler or MinMaxScaler) this transformation aligns our skewed or non-normal feature distributions to be more Gaussian-like. Since we have negative values in the `A` feature, we applied the **Yeo-Johnson** method, which handles both positive and negative values. This transformation also applies zero-mean, unit-variance normalization by default. This choice may benefit our KNN model due to its distance-based nature (normalization helps reduce the influence of features with different scales, impact of outliers and overall representation of data), while non-parametric algorithms like Decision Tree and Random Forest remain unaffected.

Outlier detection- We used the **IQR** (Interquartile Range) method to **identify and detect outliers** in our data. This method measures the spread of the data by considering the range between the first quartile and the third quartile. We implemented the `outlier_count` function to plot the number of outliers within each feature. In this function, we defined the lower bound as (Q1 - 1.5 * IQR), which represents the smallest value in the dataset, and the upper bound as (Q3 + 1.5 * IQR), representing the largest value. Any data point that falls outside this range is considered an outlier. We can observe from the plot (Appendix 6) that the feature `exports` have the highest number of outliers. In total, we have 12,309 outliers which accounts for approximately 25% of our training samples. To prevent information loss, we applied the **Winsorization** technique, which **helps reduce the impact of outliers by replacing their values with upper and lower bounds.** In our case, we used the 5th and 95th percentiles as the bounds.

Categorical features- We took another look at the number of unique values in the categorical features, which had slightly changed after handling missing values. At this stage, we decided to remove the `sha256` feature (file names), as previously mentioned. For the feature `C`, which had only 7 unique values, we applied **One-Hot Encoding.** This encoding represents categorical variables as binary vectors, with each unique value assigned a new column. However, when considering the `file_type_trid` feature, which had 83 unique values, we decided not to use One-Hot Encoding to avoid the curse of dimensionality. Applying One-Hot Encoding in this case would result in expanding the dimensions to over 100, which could pose challenges. Instead, we chose to apply **Frequency Encoding**, where categorical variables are represented by their frequencies, ranging from 0 to 1. We believe that this feature contains informative data, and by encoding it based on frequencies, we aim to capture the relationship between different file types and their association with being malicious or benign. Finally, we dropped the 'C' feature and kept only its values column, and 'file_type_trid' became a continuous variable.

Feature engineering- After encoding the `file_type_trid` feature with numerical values based on the belongingness and frequency of each file, **we created a weighted feature that considers both the file type and its probability.** This was achieved by using the `file_type_prob_trid` feature, which provides the probabilities of different file types. Multiplying the frequency and probability values resulted in a new feature that captures the combined information. Then, we made the decision to drop both the `file_type_trid` and `file_type_prob_trid` features from our dataset since the information they provided was now captured in the newly created weighted feature called ‘weighted_file_type`. This decision had a positive effect on the performance of our models.

Dimensionality reduction- At this stage, our dataset contains 28 features after completing all the preprocessing steps. However, we need to be cautious about the **curse of dimensionality**, which can arise in high-dimensional spaces. In such cases, the data points tend to spread out, making accurate predictions challenging. Additionally, overfitting becomes a concern as the models become more complex. This means that the models may perform well on the training data but struggle to generalize to new, unseen data. We explored techniques like **PCA** and **backward selection** to reduce dimensionality. However, implementing these methods had a negative impact on the AUC scores of our models. Thus, we made the decision not to proceed with dimensionality reduction techniques in this case. Later on, we implemented decision tree and random forest models. It is important to note that applying PCA for dimensionality reduction may not be necessary or beneficial when using non-parametric algorithms as they are not affected in the same way by the curse of dimensionality. They have the capability to handle high-dimensional data effectively without the need for dimensionality reduction techniques. Therefore, .our models might still perform well

Test data treatment- We made a copy of the test dataset (as we will need to save the files names for the prediction) and developed a function called `preprocess_data` to apply all the previous preprocessing stages to both the test and validation datasets. The **only difference** was that instead of removing categorical feature rows with missing values, we filled them with the most frequent value from their respective column. This was necessary as we cannot remove samples from the test data. Before starting the modeling stage, we checked the validation and test datasets using the `head` function. We noticed that the columns were in a different order compared to the training dataset. To fix this, we rearranged the columns using the X_y_train.columns() method.

Part 3: Models

We started by splitting back the **X_y_train** and **X_y_val** datasets into **X_train**, **y_train**, **X_val** and **y_val**. For the **decision tree** model, we chose to optimize **the max_depth, min_samples_split** and **min_samples_leaf** hyperparameters. Tuning **max_depth** can help control the complexity of the decision tree. A tree with a large maximum depth may overfit the training data, while a tree with a small maximum depth may underfit the data. By adjusting this parameter, we can find the optimal balance between model complexity and generalization. **min_samples_split** and **min_samples_leaf** can also help control the complexity of the tree by setting minimum requirements for splitting internal nodes and creating leaf nodes, respectively. Increasing these values can help prevent overfitting by reducing the number of splits and making the tree more general. For the **random forest** model, we chose to optimize the **n_estimators** and **max_depth** hyperparameters. We don't want to repeat ourselves as the **detailed explanation of each model's hyper parameters is written in the notebook** near the #MODELING cell (easy search with ctrl+f). We saved our best models, trained them and printed their AUC scores on the validation test.
We also attempted to use the **Gaussian Naive Bayes classifier, considering that some techniques had made our data more Gaussian-like. However, the AUC score on the validation test was 79%, which was lower than that of Logistic Regression. Therefore, we decided not to implement it. It appears that our data may not be sufficiently Gaussian-like, despite our efforts to normalize it.

Part 4: Evaluation

Confusion matrix- We plotted our trained Random Forest model (rf_best) **confusion matrix** (Appendix 7). We calculated its values using the true labels **y_val** and the predicted labels **y_pred**, and visualized it using a ConfusionMatrixDisplay. At the notebook we implemented a function that calculates the **performance metrics**: **accuracy**, **precision**, **recall**, and **F1-score**. **Accuracy** measures the overall correctness of the model, **precision** measures how often the model is correct when it predicts a positive class, **recall** measures how well the model identifies positive instances, and **F1-score** is a measure of balance between precision and recall. Further explanation at the #CONFUSION MATRIX cell. Finally, we printed out the values of these metrics to evaluate the performance of our model.

ROC curves for each of the 5 cross-validation folds (Appendix 8)- The ROC curve is a visual representation of the trade-off between a model's True Positive Rate (TPR) and False Positive Rate (FPR) at different classification thresholds. A good classifier will have a ROC curve that is located at the top-left corner of the plot, which correctly identifies positive instances while minimizing false positives across various thresholds while the linear line represents the simplest classifier. In our case, we observed that the **Random Forest classifier showed such a curve, with an average AUC score of 96.5%**. This indicates its strong performance. Surprisingly, the **KNN classifier achieved an average AUC score of 91.4%**, performing well in capturing distances and scales as we had emphasized during the preprocessing stage. The Decision Tree classifier achieved an average AUC score of 90.4%, while the Logistic Regression classifier obtained a lower average AUC score of 81.6%.

Train and validation performance- We printed the AUC scores for both the training and validation datasets. The models showed better AUC scores on the validation dataset, indicating their ability to generalize well to new data. To prevent overfitting, we focused on hyperparameter tuning using techniques like GridSearchCV. For instance, we used penalty arguments in logistic regression to apply regularization to the model's complexity and minimize overfitting. The chosen penalty by grid search was l2 which as we explained at the notebook (#MODELING cell) indicated that all features contribute to the model, we guess that's why the backward selection didn't work for us. These efforts aimed to find a balance between model performance on the training data and its ability to generalize to unseen data.

We also examined the **feature importance** (Appendix 9) to see the contribution of `weighted_file_type` to the model. We observed that it ranked as the 7th most important feature, closely following the feature `size`.

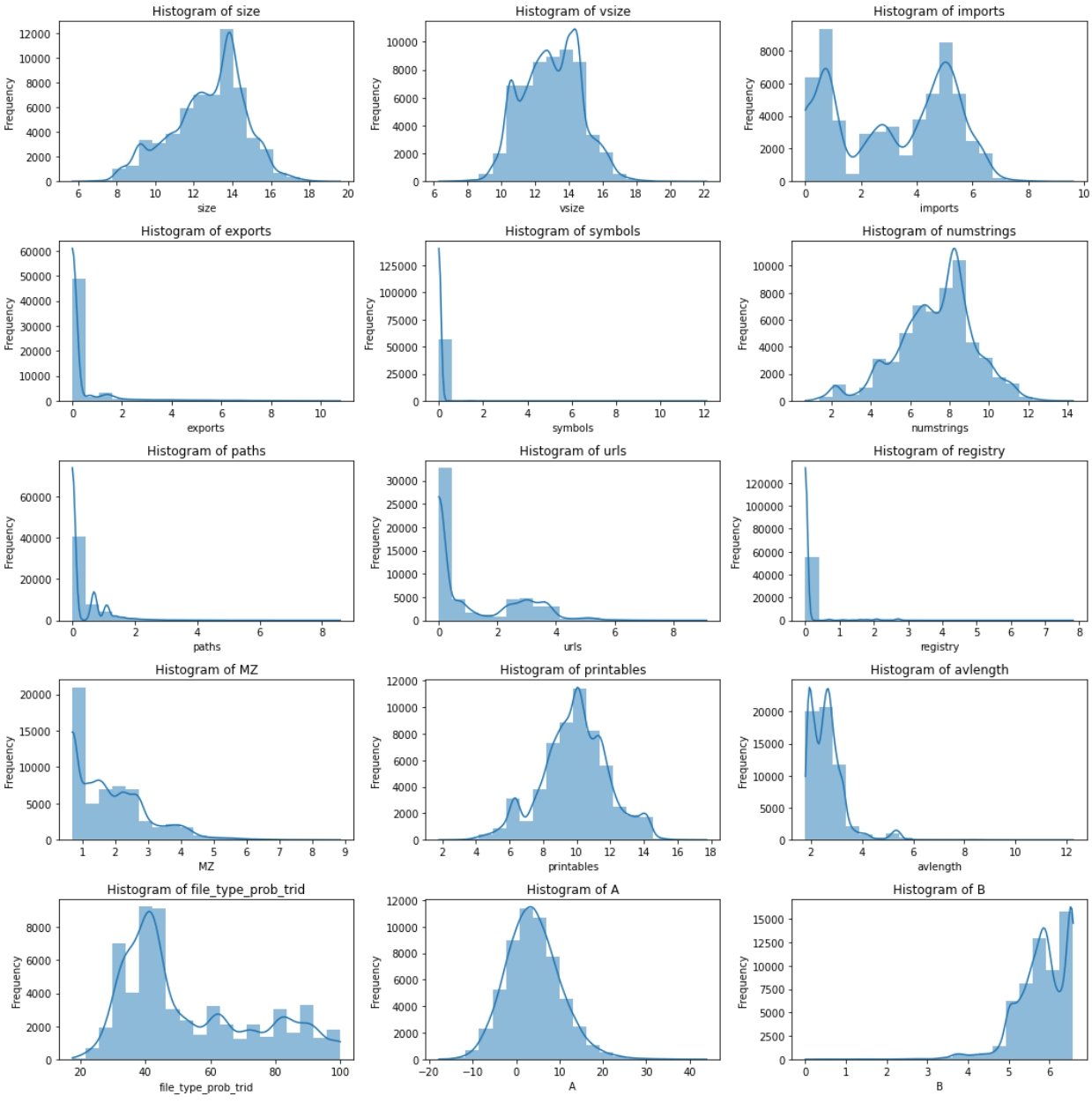
Part 5: Prediction

After reuniting **X_y_train** and **X_y_val** to form the entire **train_data**, we splitted it again into **X** and **y** where **X** contains the data of all features and **y** is the target data and trained the model on whole data. Finally, we created a dataframe with the predictions we found and the name of the files off the original test dataset, and saved the results into a CSV file named **results.csv**

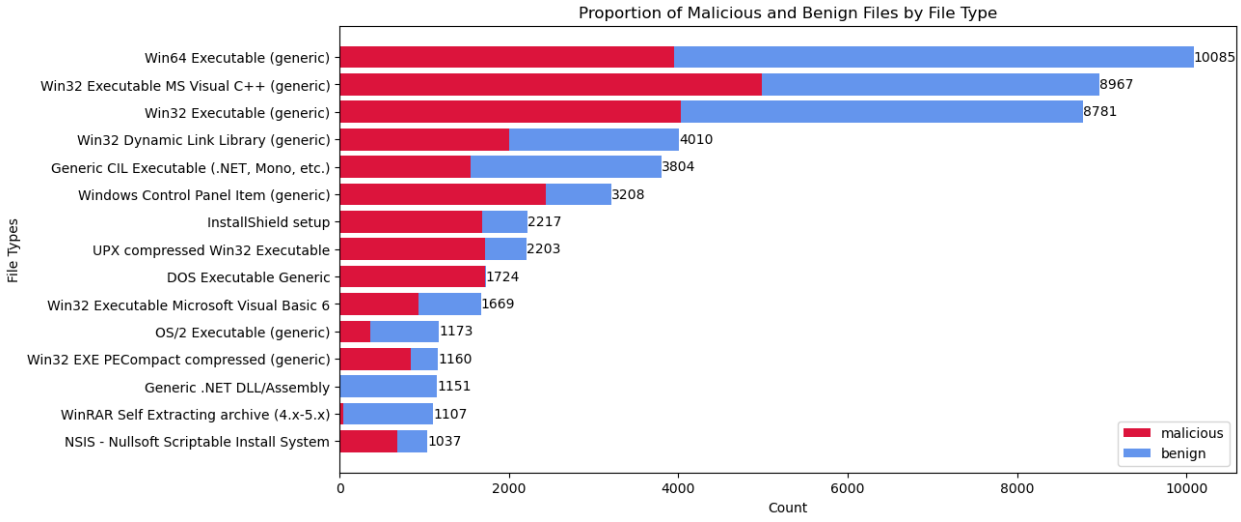
Part 6: Usage of tools not learned in the course- notebook

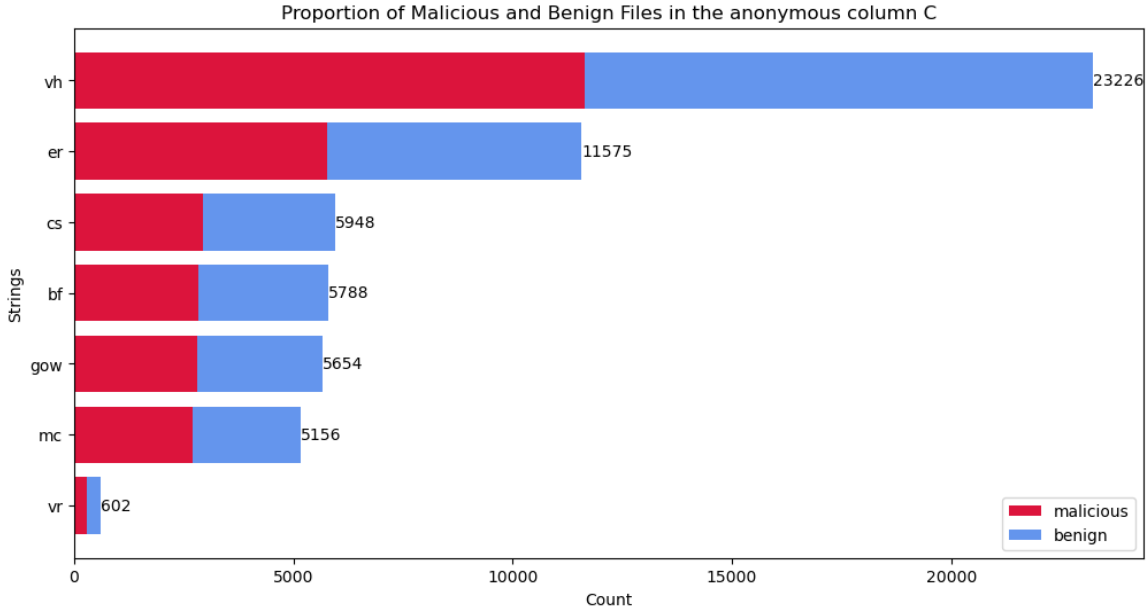
Appendices

Appendix 1- Histograms of the continuous features distributions

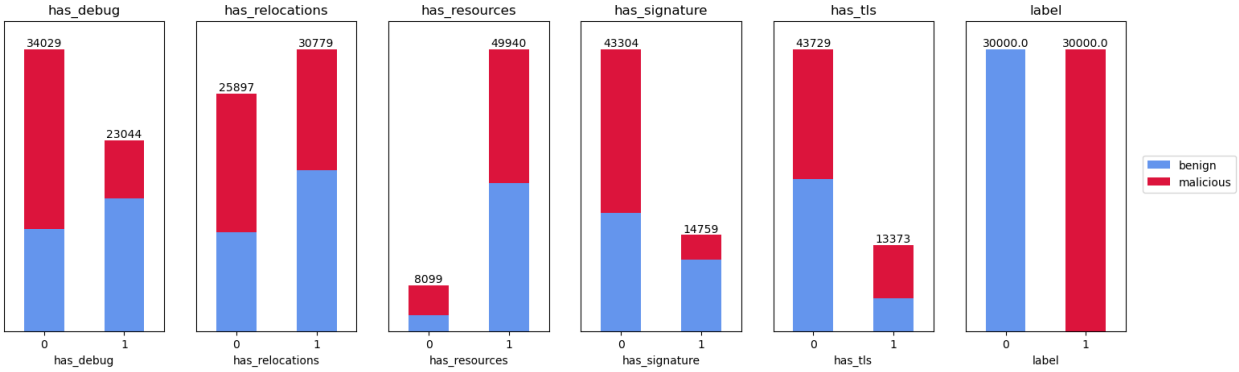


Appendix 2 and 3- Stacked horizontal bar plots of categorical features and malicious file proportions

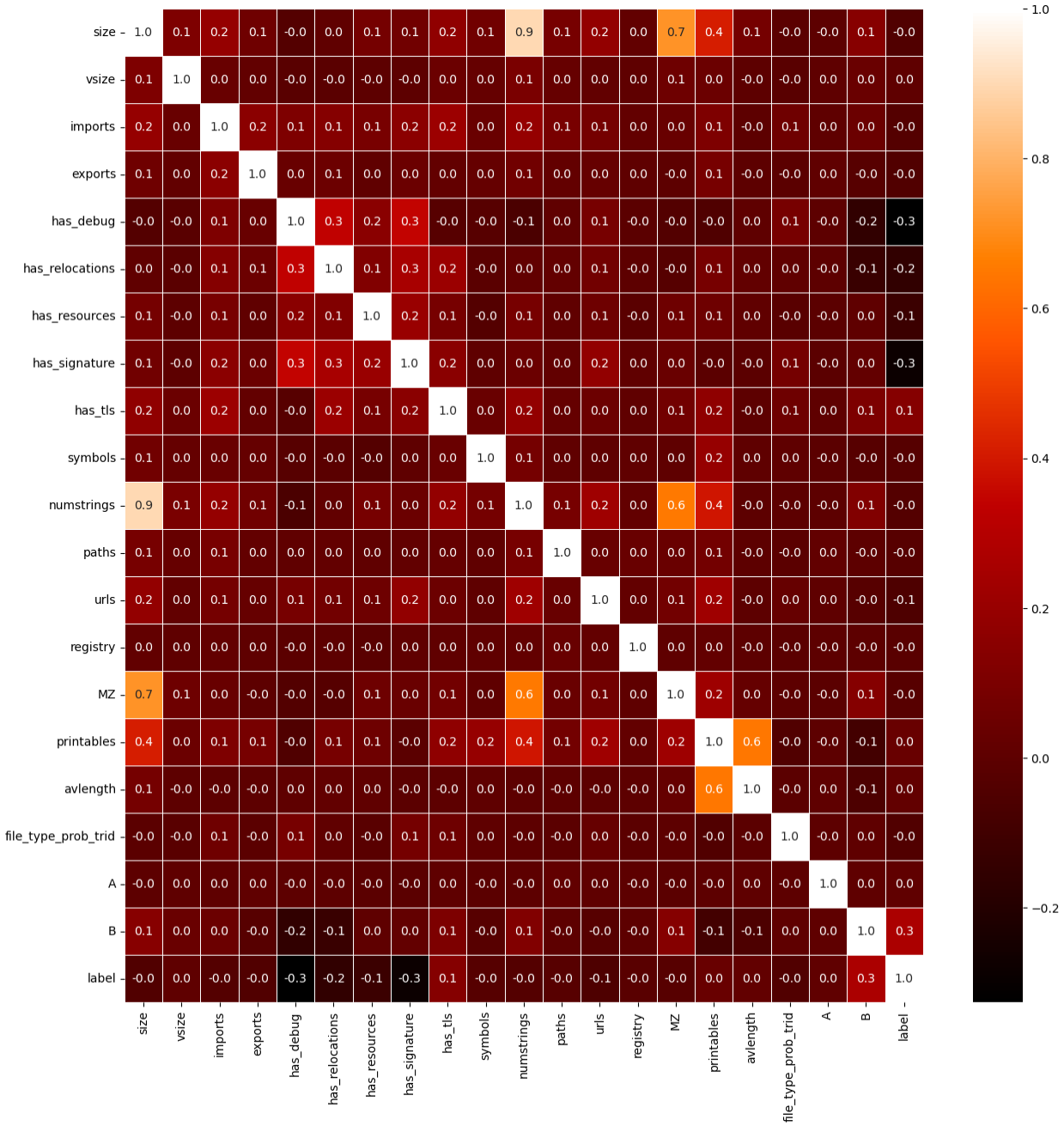




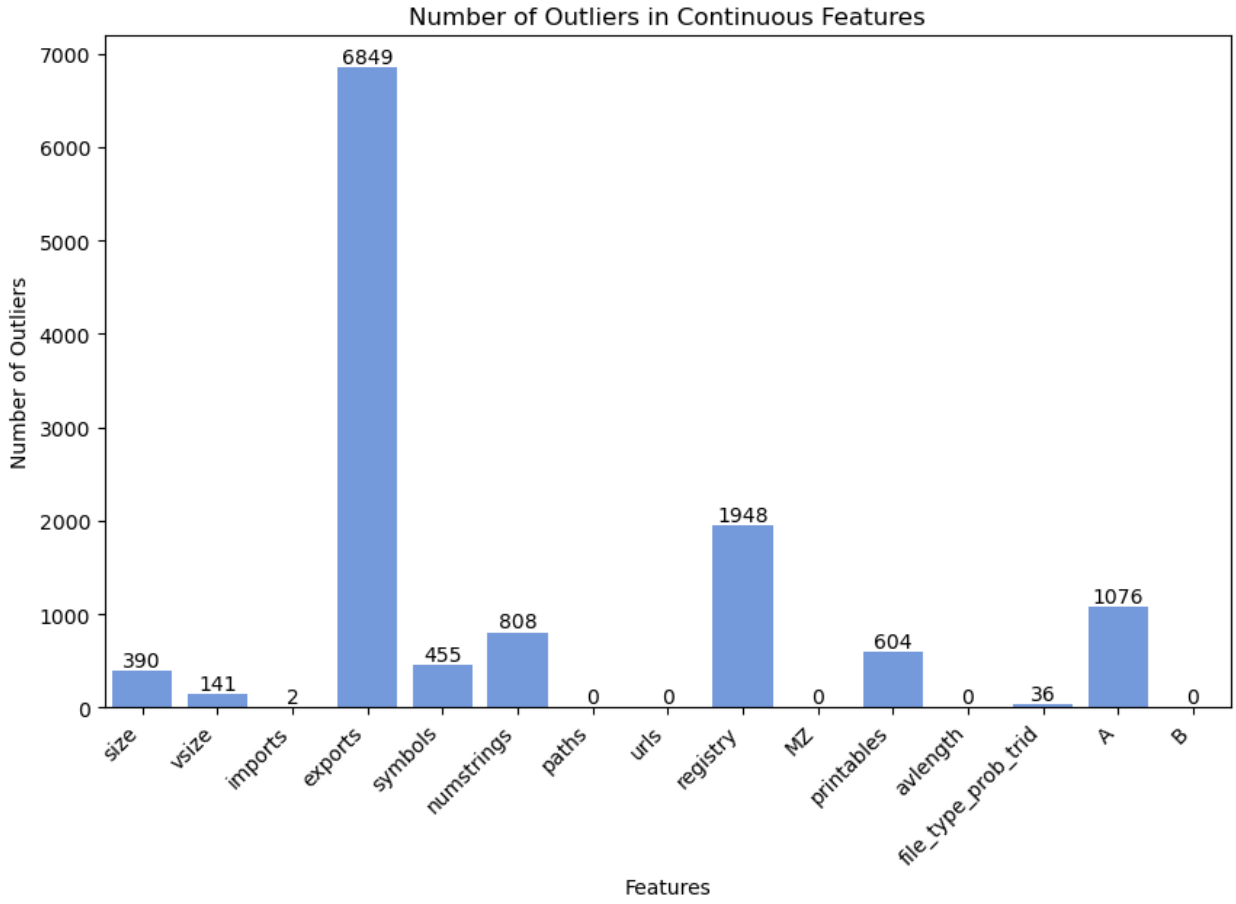
Appendix 4- Bar plots of binary features and malicious file proportions



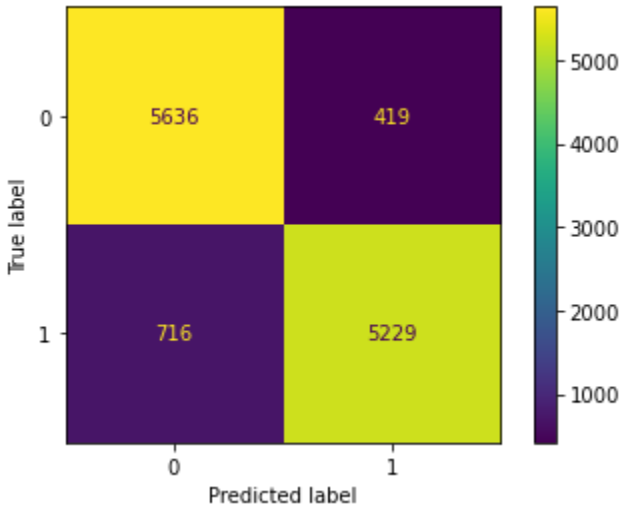
Appendix 5- Linear correlation heatmap



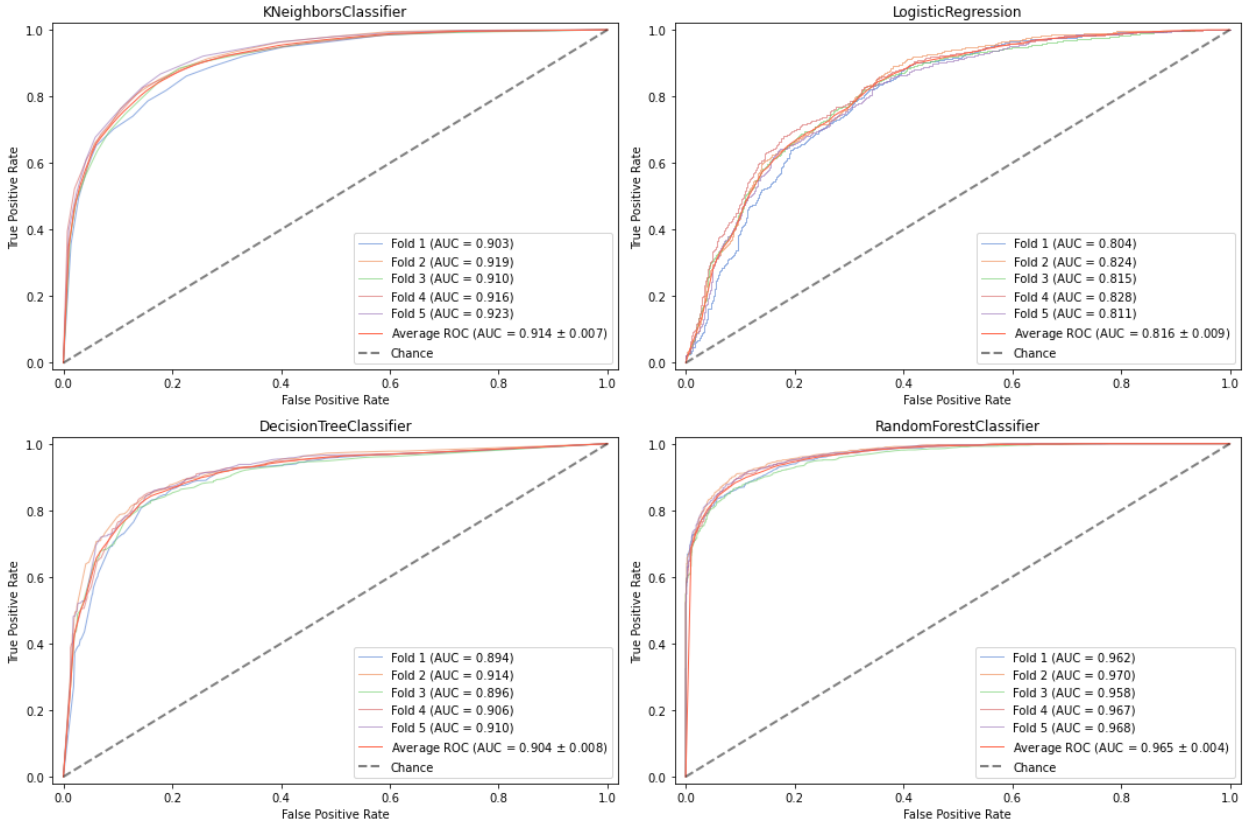
Appendix 6- Bar plot of the number of outliers within each feature



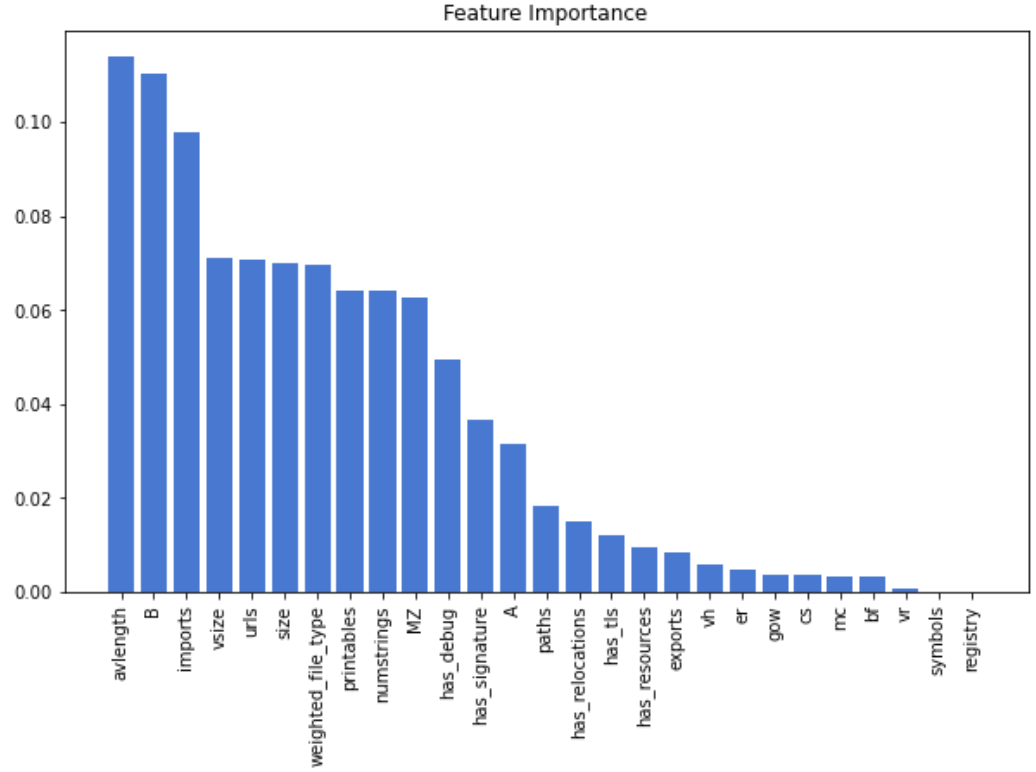
Appendix 7- Confusion Matrix of our best Random Forest model



Appendix 8- ROC curves for each of the 5 cross-validation folds of each model



Appendix 9- Feature importance



Appendix 10- each partner contribution to the project

We worked together on the **Exploratory Data Analysis** part to gain a better understanding of the data and extract meaningful insights. We discussed and decided on the visualizations that would help us achieve our goals. We created distribution plots for the continuous and binary features together. Sophie took charge of the categorical features plots and overall designs (of plots and notebook), while Jeremy did the heatmap. Each of us wrote our conclusions, and we combined them into a single summary before moving on to the next steps.

In the **Preprocessing** part, Jeremy was responsible for handling missing values and treating categorical features, while Sophie focused on detecting outliers, performing normalization, dimensionality reduction techniques and feature engineering. After we joined them together Jeremy programmed the preprocess_data function, which applies all the necessary changes to preprocess the test and validation data.

At the **Modeling** stage, Jeremy focused on modeling the Decision Tree and Random Forest models, while Sophie worked on KNN and Logistic Regression. After running separate tests with each model, we decided to combine them into a single code using a dictionary of models and various hyperparameter values. To optimize the hyperparameters, Sophie implemented GridSearchCV.

In the **Evaluation** part, Jeremy constructed the confusion matrix, explaining the interpretation of each value within the matrix, and created a corresponding plot. Sophie focused on plotting the ROC curves and conducting the analysis. Additionally, she generated the feature importance plot to see the contribution of the new feature to our model. We examined the AUC scores for both the training and validation sets to evaluate model performance. Finally, Jeremy worked on the **Prediction** part and created the final pipeline. At part **6** each of us wrote on a different technique, Jeremy wrote about the PowerTransformation and Sophie wrote about the Winsorization.

We had multiple meetings throughout the process to collaborate on merging our code, making necessary modifications, and discussing the next steps. During these meetings, we explained our work to each other, provided assistance when needed, and offered feedback and suggested improvements for each other's parts.