Tel Aviv University

Sciences for High-Tech

Introduction to Machine Learning

Final Project Report

Detecting PE Malicious Files

Submitted by:

Asaf Biniamini - 316137017

Guy Maoz – 316084821

Group #18

30.06.2023

Introduction

The following report presents a machine learning project aimed at training a model to accurately classify Portable Executable (PE) files as either malicious or benign. In this report, we will discuss our approach, data exploration, conclusions, data transformations, assumptions, model development and training, and model evaluation to achieve optimal results.

General methods

Before commencing the project, we have established a set of general methods to ensure an organized and efficient workflow. Firstly, we compiled a list of features based on our conclusions, which we intend to remove from the dataset. This allows us to work with a refined dataset in each stage, building upon the progress made thus far. Additionally, we have implemented a custom function naming convention, starting with 'pre_processing', to facilitate the testing of various data changes. This approach offers two primary advantages:

a. It grants us flexibility in exploring different methods and data processing techniques to determine the optimal approach.

b. It simplifies the implementation of the data processing pipeline on the test dataset.

To ensure a reliable validation process, we divided the data into separate train and validation sets **before** conducting any exploration or preprocessing. This allowed us to test our work on unseen validation data, assess the performance of our model, and determine if overfitting occurred.

1. Part 1 – Data Exploration

Upon loading the dataset, our initial step involved gathering insightful information about the data, including feature details, distributions, and other relevant characteristics to guide our data processing approach. Utilizing the info() method, we gain some basic information about the data before and after the train validation split and about the features. (Basic insights can be seen in the notebook and in Appendix C)

Further analysis, referencing the feature information PDF, revealed that the 'Sha256' attribute serves as a unique identifier for each sample, so it'll this identifier has no potential for improving our model's performance, therefore we decided to remove this feature.

To gain insights into the statistical characteristics of the features, we employed the describe() method, yielding the following key findings:

- The "has_###" features are binary variables and will be explored further considering their binary nature.
- The mean and median values of the labels are 0.5, indicating a near-equal probability distribution between classes. This can be advantageous when constructing a Naive Bayes model by leveraging priors derived from the training data.
- A notable observation is the significant disparity between the 75th percentile and maximum values in most continuous features, suggesting potential low variance or sparsity.

1.1 Sparse feature exploration was conducted by counting '0' values in each continuous numeric feature. We focused on features with '0' values in over half of the samples. The following conclusions were drawn: (visualization in Appendix D + E, the full process in the notebook)

- The 'exports' feature has a large number of '0' values (~37K out of 45K samples) but still exhibits some variation between '0' and non-zero values, so it may still offer some utility to the model. After testing our model with and without this feature, we found it to have no significant contribution to the results. Therefore, we decided to remove it to reduce dimensionality.
- The 'symbols' and 'registry' features predominantly consist of '0' values (~42K out of 45K samples) and show minimal variation in label distribution. These features will be removed.
- The 'urls' and 'paths' features have a lower number of '0' values and distinct results between '0' and non-zero values, warranting further analysis and retention.

1.2 feature correlation - to identify and eliminate highly correlated features we build a correlation matrix (Appendix F). We set a correlation threshold (85%), and features that passed the threshold were appended to the 'features_to_remove' list ('numstrings' only).

1.3 Binary features - We explored the binary features to determine if any combination of them could effectively predict malicious files. By creating contingency tables for each feature with the 'label' variable, we assessed their predictive power (Appendix G). Using the 'combinations' function from the 'itertools' library, we checked all possible feature combinations and the probability of them to indicate malicious files. Although we found no strong connections for predicting malicious files, we discovered a few combinations that indicated non-malicious files. From these combinations, we selected the one with the lowest probability (Appendix H) and created a new feature (when both features are 1 it'll be 1, otherwise 0). When this new feature is set to 1, it suggests that the file is likely benign.

1.4 Categorial features – We had two remaining categorical features that required attention. (Plot in Appendix I). Upon analyzing the 'label' distribution for each category in feature 'C' (Appendix J), we observed an equal split. As a result, we concluded that this feature would not significantly contribute to the prediction and added it to the 'features_to_remove' list. For the 'file_type_trid' feature, which had numerous uncommon categories, we developed a function to retain only the most common categories, replacing the uncommon ones with 'others' (Appendix K).

1.5 Other features exploration and transformation– Through exploration and analysis of additional features (see Appendix L), we identified six features that displayed a log-normal distribution. To address this, we applied a logarithmic transformation (outlined in Appendix M + N), resulting in three of these features closely resembling a normal distribution (Although not exact, we assumed that they are). Consequently, we developed a function to scale the dataset's features accordingly. (Full process in the notebook).

1.6 Size And V-Size Exploration - The features 'size' and 'vsize' describe different size aspects of the files, so we aimed to investigate if a notable disparity between these two variables could serve as an indicator of the label. To assess this, we introduced a new feature representing the absolute difference between 'size' and 'vsize'. However, after evaluation (see Appendix O), we found that this new feature had minimal impact on the results. Therefore, we made the decision to exclude it from our analysis. (Full process in the notebook).

2. Part 2 – Preprocessing

2.1 Outliers

Our feature set consists of three types: continuous, binary, and categorical. Outliers pose no issue for binary features, given their limited values of 0 or 1. For categorical features, rare categories are grouped as 'others' rather than treated as outliers. As a result, our primary focus centers on the continuous features, which we further divided into two distinct cases.

1. features that we assumed were normally distributed - we applied the IQR (Interquartile Range) method to identify outliers. This approach involves calculating the range between the 25th and 75th percentiles and considering data points outside a specified threshold as outliers.
2. For features with undiagnosed distributions, we employed the Isolation Forest algorithm from the 'pyod' library (We will discuss this in detail later).

We chose not to remove the outliers for two primary reasons: Firstly, we aimed to devise a solution that can be applied to our test data as well, Secondly, it cause us for overfitting. Instead, for normally distributed features, we replaced the outliers with the median value (**of the train data**). We observed that dealing with outliers using the Isolation Forest for the unknown distribution features harmed the results, so we decided only to address outliers in the normal features.

2.2 Dealing with missing data:

To impute the missing data, we utilized the SimpleImputer. For this purpose, we developed a function capable of implementing the imputation process based on the specified strategy and features. This approach enabled us to handle different feature types and test various strategies accordingly.

After researching, we found that the appropriate imputation strategy depends on the assumption of missing values and data distribution. If missing values are random and the data follows a normal distribution, using the 'mean' strategy is suitable. For data with outliers or skewness, the 'median' strategy is more robust. The 'most_frequent' strategy is effective for categorical or discrete features. Considering these factors and assuming random missing values in normal distribution features, we implemented our methodology to handle missing data.

Although the only categorical feature, 'file_type_trid,' did not have any NULL values in the train data, we realized the possibility of encountering them in the validation or test data.

We explored the K-nearest neighbors (KNN) imputer method but found it to be time-consuming without providing better results. Hence, we chose not to use it. Additionally, our experiments confirmed that the most effective approach aligned with our research findings.

2.3 Dealing with categorical features

To accommodate our models, which do not support categorical features, we transformed our categorical feature into a numeric representation. We opted to use the OneHotEncoder from the ‘sklearn’ library instead of alternatives like get_dummies from 'pandas'. This choice ensures compatibility with future test data, enabling us to handle new examples that were not seen during training.

2.4 Data normalization

The data isn’t normalized. normalization of the data is crucial for several reasons, like preventing dominance by larger magnitudes, many algorithms operate optimally when dealing with features of similar scales for example KNN that we used in our model.
To achieve normalization, we employed the MinMaxScaler method from ‘sklearn’ library.

2.5 Dimensions analysis

Large dimensionality occurs when the number of features is significantly higher than the number of samples. In our dataset with around 20 to 30 features and 60,000 examples, this is unlikely to be a problem. However, high dimensionality can cause issues such as increased computational complexity, longer training times, and potential overfitting. Checking for high feature correlation and addressing multicollinearity can help identify if dimensions are too large. (We already removed the features with high correlation).

2.6 Dimensionality reduction's impact on the model

We opted to employ Sequential Forward Selection (SFS), a widely used feature selection method. Comparing our model performance with and without dimensionality reduction, we observed significantly higher train and validation scores when utilizing SFS.

2.7 Implementing the process on the test data

Upon conducting research and gaining insights into the topic, we made a significant realization. When it comes to imputing values in the test data, relying solely on the test data's statistics (such as mean, median, or mode) is not viable. This limitation stems from the test data potentially having a small sample size, making it unreliable for learning purposes. Therefore, we devised an alternative approach to address this challenge.

To tackle this issue, we adopted a two-fold implementation process: one for training data and another for test data. During the training implementation, we stored the desired statistics for each function in a dictionary. These calculated statistics serve as valuable references for imputing missing data in the test (and validation) implementation. For further details, see the "Implementing the preprocessing on the test data" section in the notebook.

3. Part 3 – Building the models

After initially testing all seven available models, we identified the ones that yielded the best results. While some other models produced similar outcomes, they required significant computational time without offering superior performance compared to our top models: KNN, Naïve Bayes Classifier, Random Forest, and Decision Tree. We selected these four

models as they provided optimal results while maintaining reasonable computational efficiency. The chosen parameters for each model as well as an explanation of the model hyperparameters and their impact on the bias and variance tradeoff can be found in **Appendix S**.

4. Part 4 - Model evaluation

For model evaluation, we created a dictionary with our final models and parameters, and a summarizing function that evaluates them. The evaluation includes:

1. Train scoring using mean K-Fold Cross Validation 'roc auc' scoring, with a graph showing scores for each fold.
2. Validation scoring using roc-auc scoring.
3. Printing train and validation scores for each model.
4. Printing and saving the winning model with its validation score.
5. Plotting two graphs: one comparing model scores and another one that compares differences between train and test data, to test overfitting.
6. Plotting the confusion matrix of the winning model and printing its data.

Scores Summary:

	KNN	***Naïve Bayes***	***Decision Tree***	***Random Forest***
Val Score	93.381978%	81.235409%	83.520368%	**95.542460%**
Train Score	93.320724%	81.451625%	89.124101%	98.030044%

4.1 Confusion matrix analysis

You can find the confusion matrix of the winning model in Appendix P and in the notebook.

Interpreting the matrix cells in our model:

- **True Positive (TP):** The model correctly predicted 5699 samples as malicious.
- **False Positive (FP):** The model incorrectly predicted 350 samples as malicious when they were benign.
- **False Negative (FN):** The model incorrectly predicted 1814 samples as benign when they were malicious.
- **True Negative (TN):** The model correctly predicted 7137 samples as benign.

Analyzing the model's performance:

FPR (False Positive Rate): 5.79%. This means 5.79% of benign files were incorrectly classified as malicious.

FNR (False Negative Rate): 20.27%. This indicates 20.27% of malicious files were incorrectly classified as benign.

Conclusions:

The false positive rate of 5.79% suggests that the model has a relatively low rate of misclassifying benign files as malicious. This is important to minimize false alarms and prevent legitimate files from being incorrectly flagged as malicious.

The false negative rate of 20.27% indicates that the model has a higher rate of misclassifying malicious files as benign. This is a concern as it means the model may miss some potentially harmful files, leading to false negatives.

Overall, while the model demonstrates high accuracy, there is room for improvement in terms of reducing false negatives to enhance its ability to correctly identify malicious files. For our work it's great, but for a real-life scenario, it must be improved significantly.

An interesting analysis of why our results aren't enough in real life is provided in Appendix T. (recommended)

4.2 Train / test scoring difference (why?)

To assess the disparity between the training and test scores, we compared and displayed the scores side by side and visualized the differences through a graph. The validation data, which remained unaffected by preprocessing steps and

served as clean test data, was used for this purpose. To ensure a more accurate and less overfitted evaluation of the training data, we employed K-Fold cross-validation.

Initially, when testing the models at the project's inception, we observed a substantial disparity between the training and validation scores (as evidenced in Appendix Q). To address the issue of overfitting, we employed smarter data processing techniques. The following key steps significantly aided in mitigating overfitting:

1. Divide the missing values and outliers implementation to train and test procedures, and use train statistics to impute those values in the validation data.
2. For the Random Forest model, which exhibited the highest degree of overfitting, we fine-tuned the hyperparameters. By employing the Random Search method, which enables us to explore a wider range of parameter values compared to Grid Search, we achieved significant improvements in model performance.
3. For the Decision Tree model, Incorporating the 'max_features' parameter into the tested parameter grid effectively reduced dimensionality and significantly mitigated overfitting in the model.
4. In the KNN and Naïve Bayes models, we used SFS to reduce dimensionality and minimized the difference between train and validation data to almost zero.

By implementing these steps, we successfully reduced the overfitting in our models, enabling them to generalize better and perform more accurately on unseen data.

To further reduce overfitting in the Random Forest and Decision Tree models, additional dimensionality reduction methods and hyperparameter tuning can be explored for regularization. For Decision Trees, controlling the maximum depth or limiting the number of leaf nodes helps prevent complexity and noise capture. In Random Forests, adjusting parameters such as the number of trees, maximum depth, and minimum samples per leaf can effectively control model complexity and combat overfitting.

4.3 Feature Importance Analysis

After identifying the Random Forest Model as our best performer, we aimed to analyze the feature importance within this model to gain insights into the most significant features. To achieve this, we developed a function that calculates the feature importance and generates a corresponding graph, highlighting the top 15 contributing features.

Interestingly, we observed that the categorical features we created had the least impact on the model's performance. Our newly constructed binary feature ranked 11th in terms of contribution, with a score of 0.0344, and the unknown feature B emerged as the most influential, despite its ambiguous meaning.

In future endeavors, an approach to enhance our models could involve evaluating the cumulative contribution of features and selecting those that collectively account for 0.99 of the model's performance. This strategy would offer benefits such as dimensionality reduction, mitigating overfitting, and improving runtime efficiency. (Plots of this chapter are in Appendix R)

5. Part 5 – Prediction

We developed a function that takes the selected model, files names, and loads the test data. This function performs the necessary preprocessing steps on the data, generates predictions using the model, and exports the results to a CSV file.

6. Part 6 – Unlearned tools - SFS

We opted to use **Sequential Forward Selection (SFS),** a feature selection technique that gradually builds the best feature subset. SFS stands out for its robustness to overfitting (evaluating each subset's performance using a cross-validation), better generalization to unseen data, support for floating search, and flexible scoring metrics (we used roc_auc). That is why we chose to use SFS over single-step forward or backward techniques. However, applying SFS to the Random Forest and Decision Tree models incurred high runtime costs without improving performance. Research revealed that these models are already adept at handling high-dimensional data and offer the 'max_features' hyperparameter for dimension reduction. Therefore, we decided to not use SFS on these models and rely on feature removal based on our findings and the models' built-in capabilities.

7. Appendix

6.1 Appendix A – Partners' Responsibilities and work plan:

Our collaborative work process was successful, characterized by strong teamwork, continuous learning, and mutual improvement. Although we initially divided responsibilities and tasks, we worked closely together throughout the entire project. Constant communication, assessment, support, and shared decision-making were integral to our collaborative approach. Therefore, it is challenging to attribute specific contributions to each individual, as we consistently contributed to all aspects of the project together.

However, we did allocate specific "dry" responsibilities as follows:

Asaf:

- Handling outliers
- Data normalization
- Dealing with categorical features
- Dimension reduction
- Building the new binary feature
- Exploring size v-size relationship
- Features log transformation

Guy:

- Exploration of sparse features
- Exploration of features correlation
- Dealing with missing data
- Implementing preprocessing and conclusions on the train and test data
- Building and tuning the models' hyperparameters
- Model evaluation
- Test data prediction
- Pipeline implementation

6.2 Appendix B – Preprocess instruction Answers references

As we answered part of these questions in the exploration part and the other in the Preprocess part, we add this appendix to help you find each answer.

- *האם קיימים נתונים חריגים (Outliers) בדאטה? אם כן, עליכם להסירם או לפחות לתת עליהם את הדעת* - **Section 2.1 – Outliers**.
- *האם הנתונים מנורמלים? אם לא- האם צריך לנרמל אותם? מה החשיבות של נרמול הנתונים בבעיה?* – **Section 2.4 – Data Normalization**
- *האם ישנם נתונים חסרים? כיצד בחרתם לטפל בהם ומדוע באופן זה?* – **Section 2.2 – Dealing With Missing Data**
- *התמודדות עם משתנים קטגוריאליים* - **Section 2.3 – Dealing with categorical features**
- *האם המימדיות של הבעיה גדולה מדי? למה מימדיות גדולה עלולה ליצור בעיה? איך נזהה כי מימדיות הבעיה גדולה מדי?* – **Section 2.5 – Dimensions analysis**
- *הקטנת המימדיות על ידי טכניקה אחת שנלמדה בכיתה , PCA – ו/או על ידי בחירת תת קבוצה של פיצ'רים קיימים כיצד הקטנת המימדיות השפיעה על המודל?* – **Section 2.6 - Dimensionality reduction's impact on the model**
- *בניית פיצ'רים חדשים ו/או מניפולציה מתמטית על פיצ'רים קיימים* – **Sections 1.2 – Binary features and 1.5 - other features exploration and transformation**
- *החלת העיבוד המקדים על סט הTest* - **Section 2.7 - Implementing the process on the test data**
- *ניתן לבצע ניסיונות נוספים אשר לא נלמדו בקורס על מנת לעבד את הפיצ'רים הניתנים לכם* - ***Size Vsize Exploraion, Sparse features exploration, SFS, Isolation Forest Outliers detaction***
- *יש לתת הסבר על משמעות ההיפר פרמטרים שבחרתם לשנות וכיצד הם משפיעים על המודל מבחינת שונות והטייה* – Appendix S

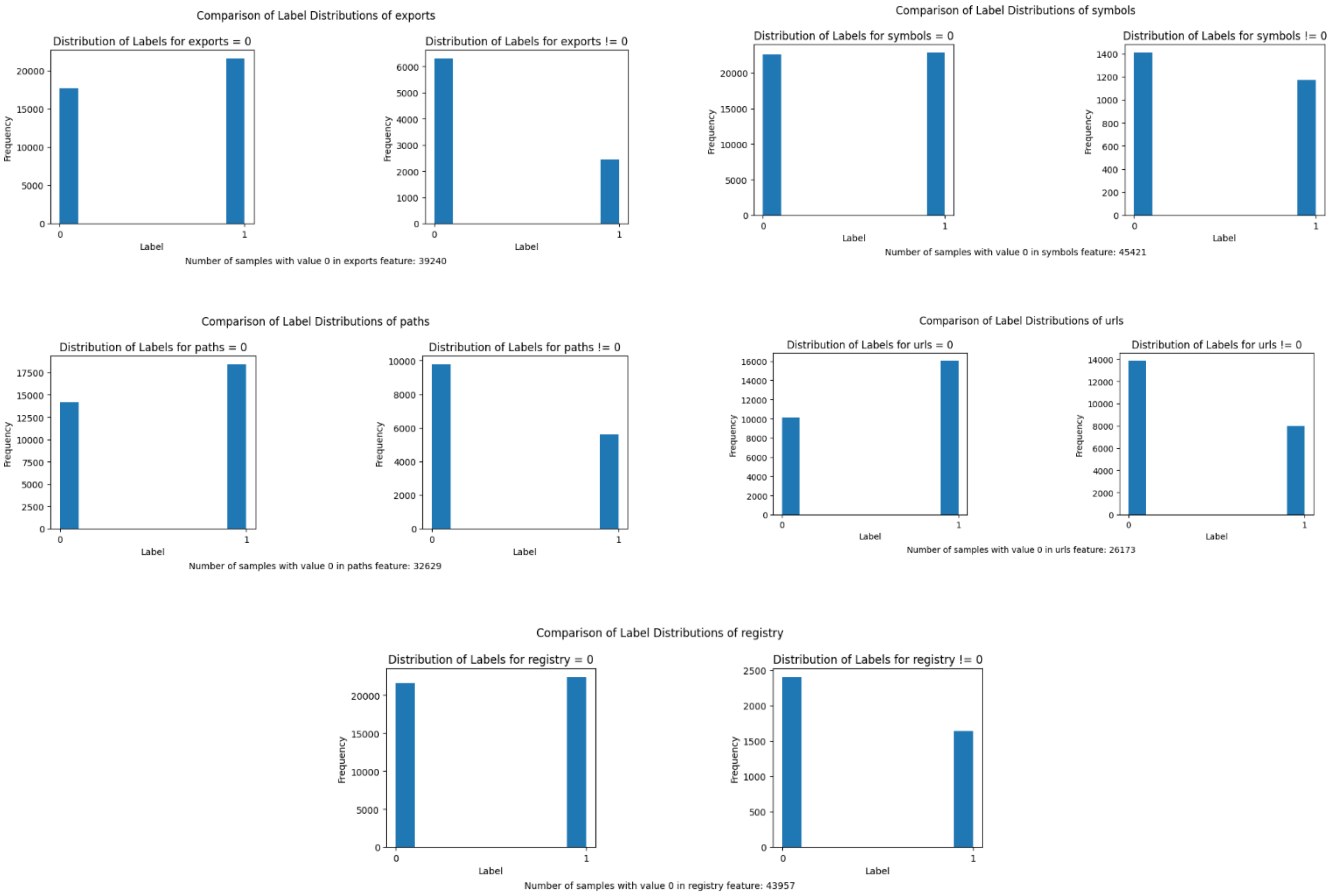
6.3 Appendix C – Insights from the info tables:

- The dataset comprises 60,000 samples.
- The train validation split splits our data into 45K samples in the train and 15K samples in the validation
- There are 23 features, predominantly numeric (float64 or int type), with 3 categorical (object type) features requiring special attention during processing.
- Notably, most features exhibit approximately 2000-4000 null values.

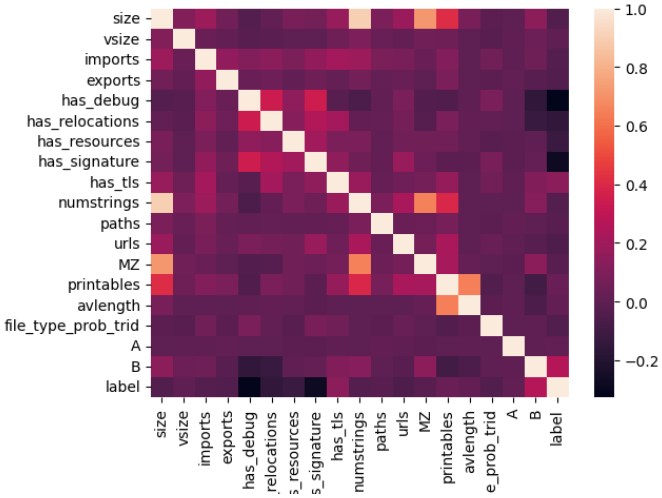
6.4 Appendix D – Zero value counting for continuous features.

```
counting zeros for each feature:
Number of samples with value 0 in size feature: 0
Number of samples with value 0 in vsize feature: 0
Number of samples with value 0 in imports feature: 5077
Number of samples with value 0 in exports feature: 39240
Number of samples with value 0 in symbols feature: 45421
Number of samples with value 0 in numstrings feature: 0
Number of samples with value 0 in paths feature: 32629
Number of samples with value 0 in urls feature: 26173
Number of samples with value 0 in registry feature: 43957
Number of samples with value 0 in MZ feature: 0
Number of samples with value 0 in printables feature: 0
Number of samples with value 0 in avlength feature: 0
Number of samples with value 0 in file_type_prob_trid feature: 0
Number of samples with value 0 in A feature: 0
Number of samples with value 0 in B feature: 0
```

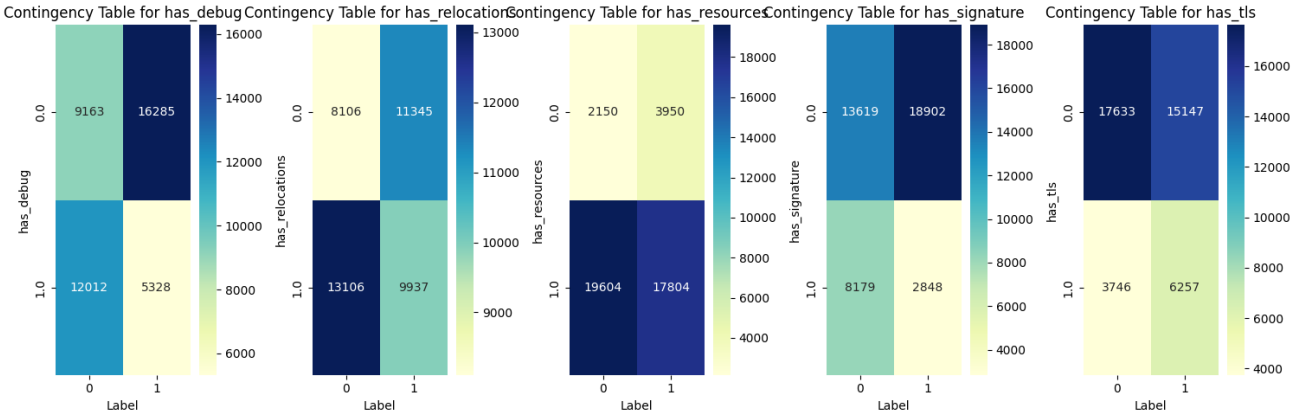
6.5 Appendix E – High zero value features exploration:



6.6 Appendix F – features correlation matrix:



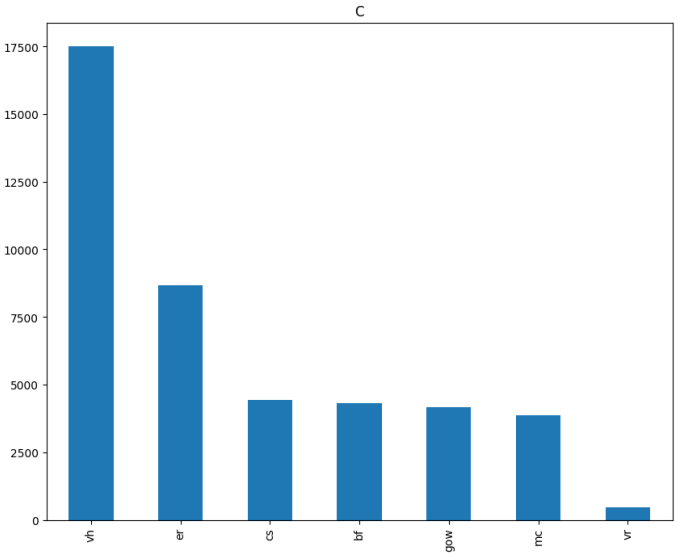
6.7 Appendix G – Contingency Table of binary features to the label:

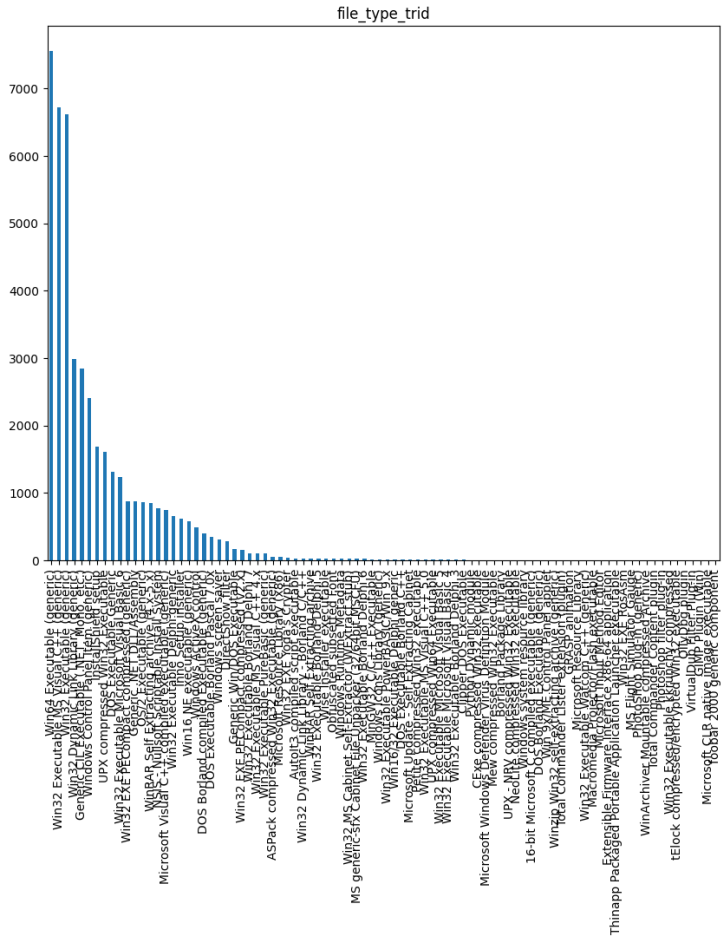


6.8 Appendix H – probability of the minimum combination:

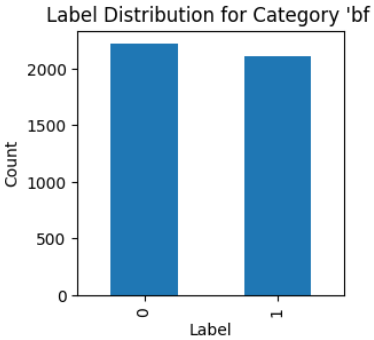
Features: ('has_relocations', 'has_signature')
Probability of Label 1: 0.2601 => Probability of Label 0: 0.7399

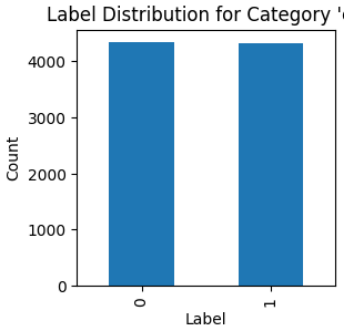
6.9 Appendix I – histogram of categorial features:

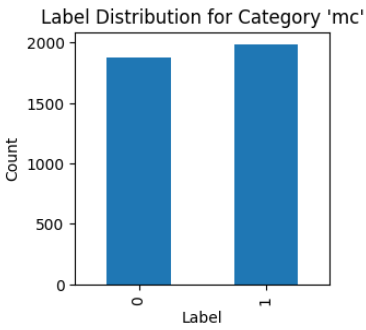


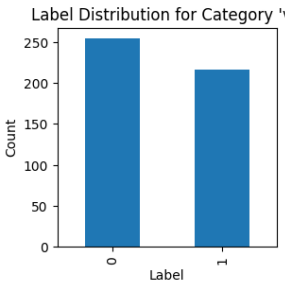


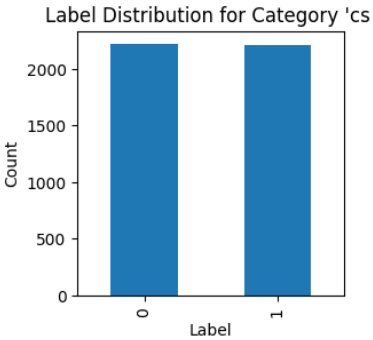
6.10 Appendix J – label distribution of each category of 'C':

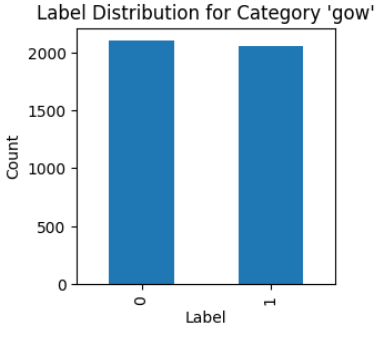


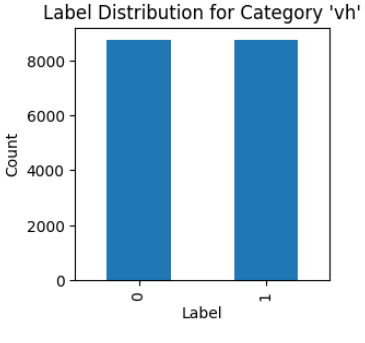




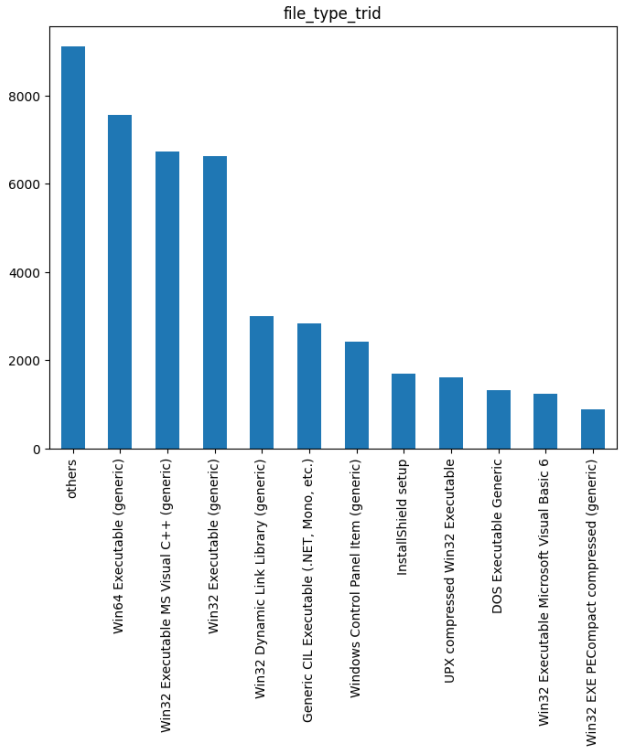




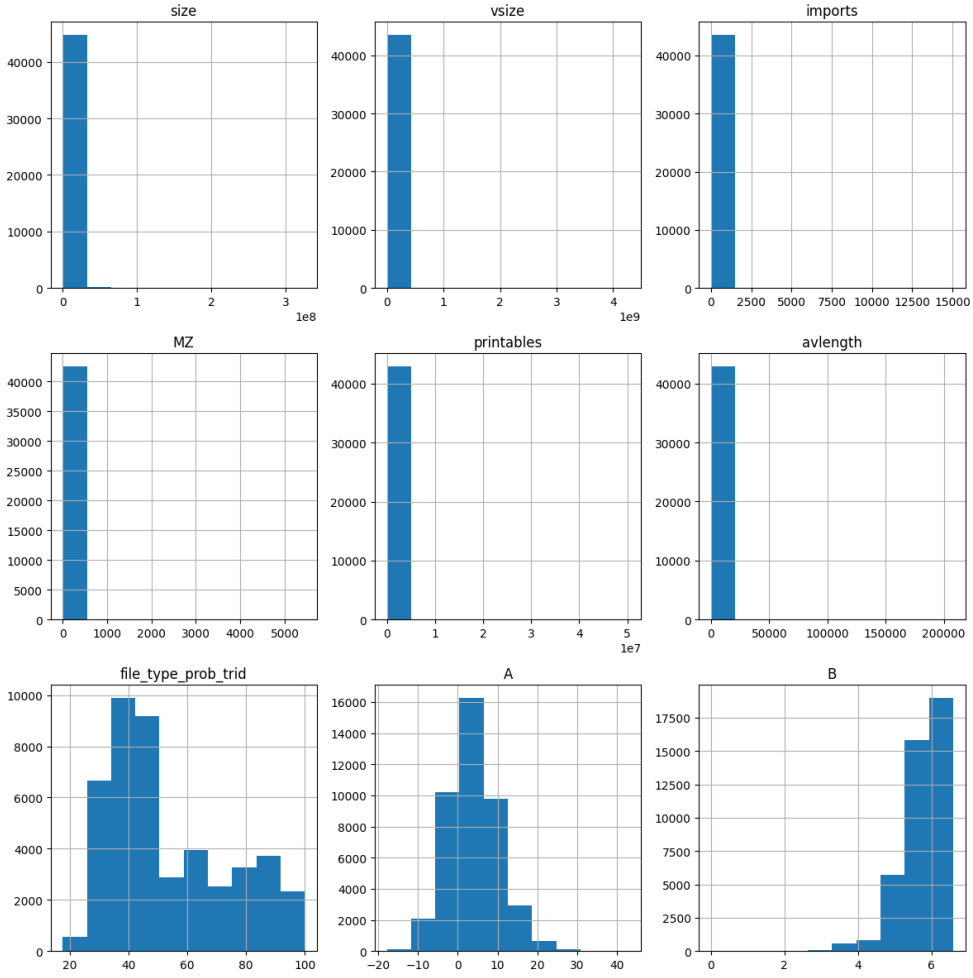




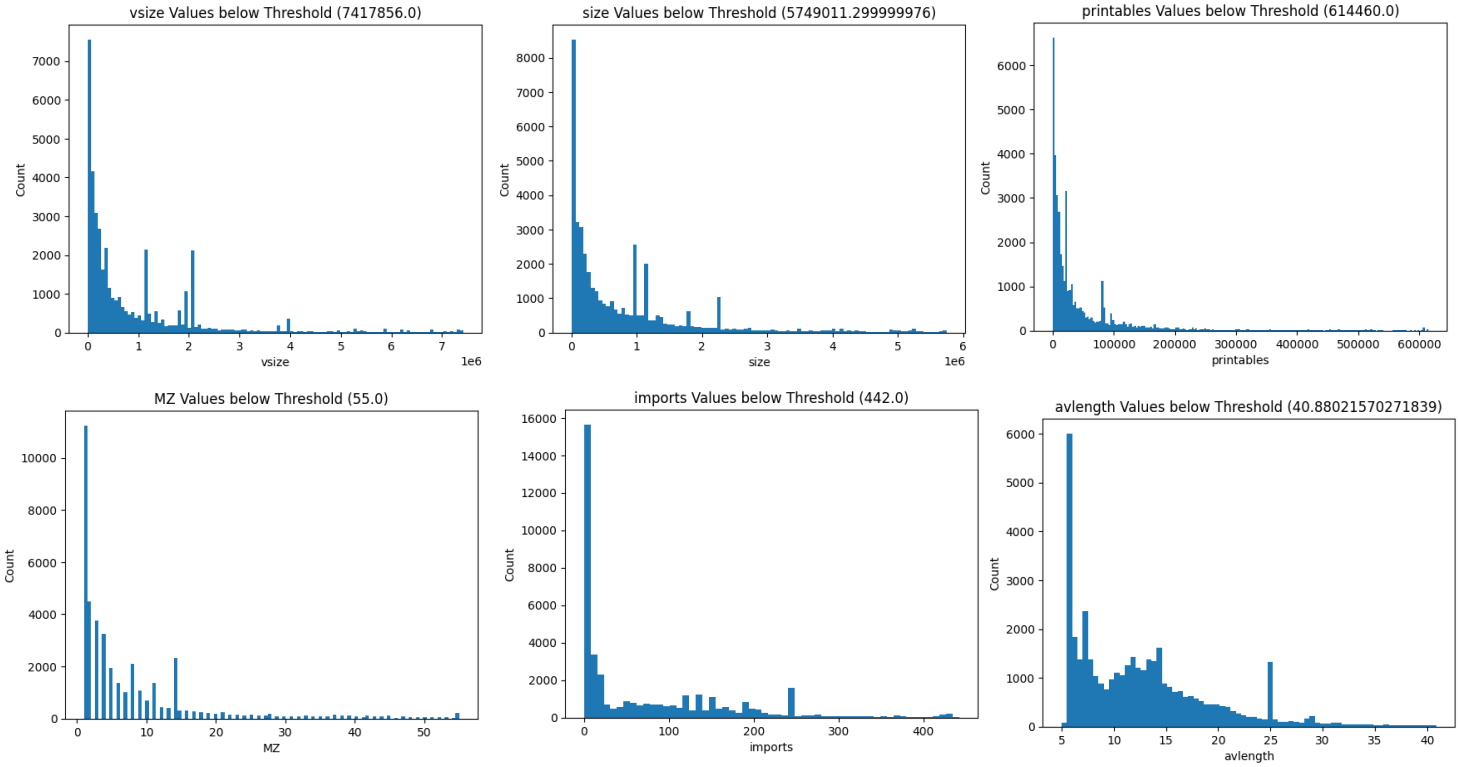
6.11 Appendix K - histogram of 'file_type_trid' after reducing categories:



6.12 Appendix L - histogram of other features:



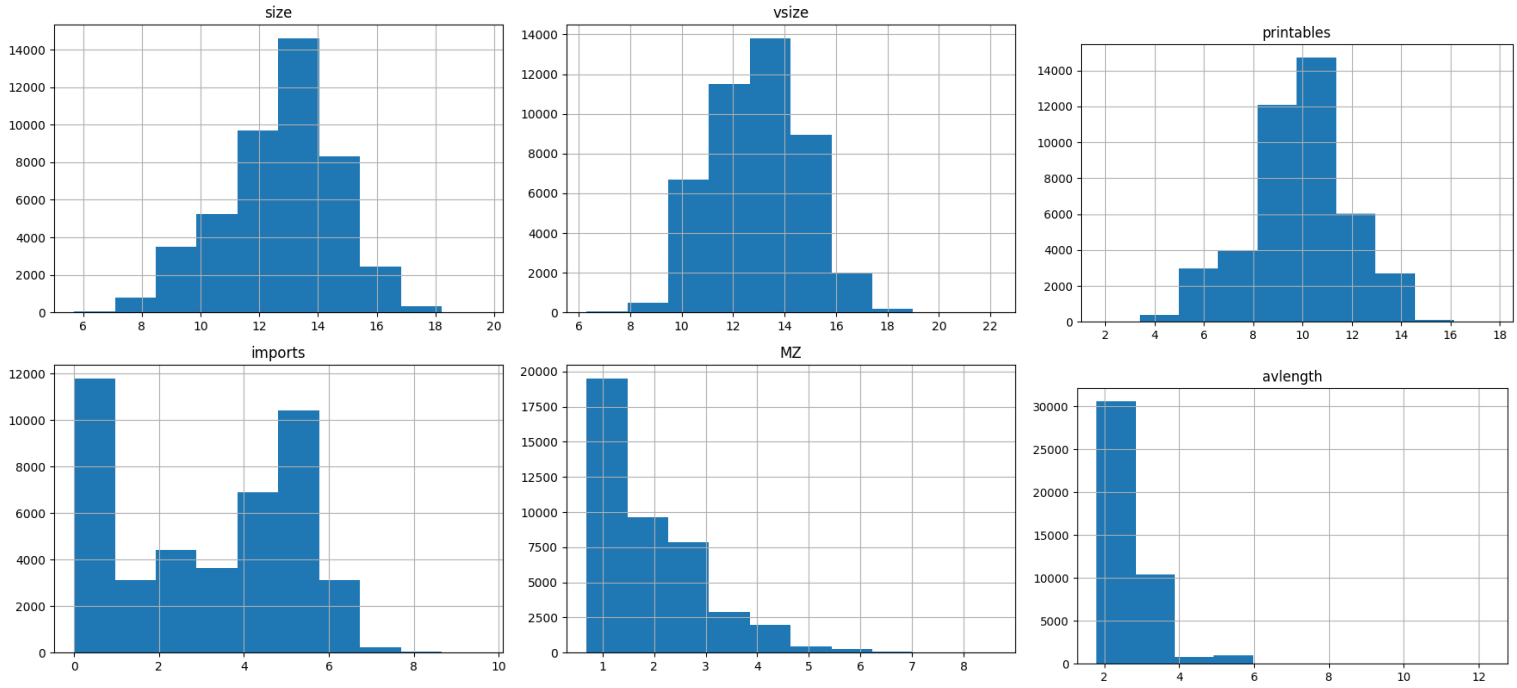
6.13 Appendix M - We wanted to try to set a threshold as the upper limit of the one big bar we saw in the histograms plots of these features, as we thought that most of the interesting data is there.



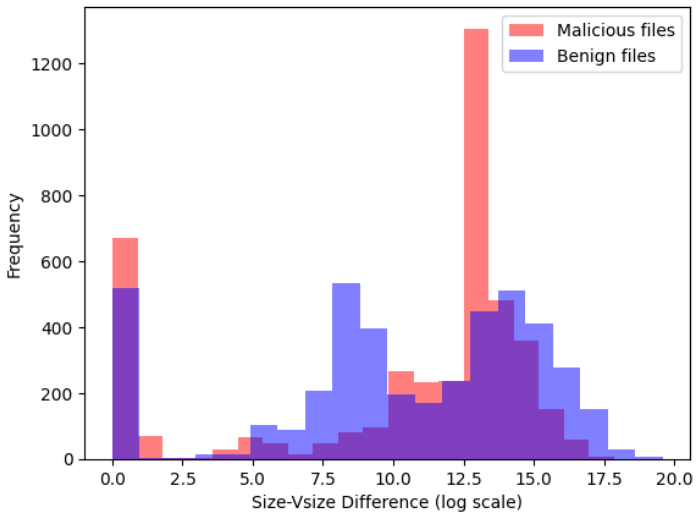
Log-Normal Distribution: The log normal distribution may seem similar to the normal distribution but it has a few difference:

- It's skewed to the right, meaning it has a positive fat-tail.
- It only contains positive values.

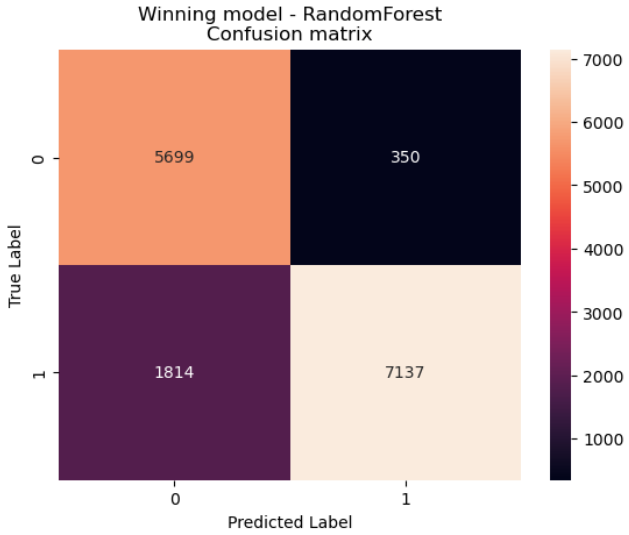
6.14 Appendix N – histogram of the features after scale changing:



6.15 Appendix O - Distribution of Size-Vsize Difference by Label:



6.16 Appendix P– Winning model confusion matrix:




```
Winning model - RandomForest
Confusion matrix :
 [[5699  350]
 [1814 7137]]
False positive rate: 5.79%
False negative rate: 20.27%
```

6.17 Appendix Q – Overfitted results during the process

Random Forest overfitted result before & after testing and tuning wider parameters with Random Search:

```
Above plot results:
  ID    Algoritm       train score[%]  val score[%]
   3.   RandomForest   97.970775%       87.307521%
```

```
 ID    Algoritm       train score[%]  val score[%]
  3.   RandomForest   98.095642%       93.013653%
```

KNN overfitted result before & After implementing the final outliers technique:

```
Above plot results:
  ID    Algoritm       train score[%]  val score[%]
   2.   KNN            92.353608%       88.558330%
```

```
Above plot results:
  ID    Algoritm       train score[%]  val score[%]
   2.   KNN            92.410444%       92.794230%
```

GNB overfitted result before & after reducing dimensionality with SFS:

```
Above plot results:
  ID    Algoritm       train score[%]  val score[%]
   1.   GNB            77.924223%       75.133537%
Next Model:
```

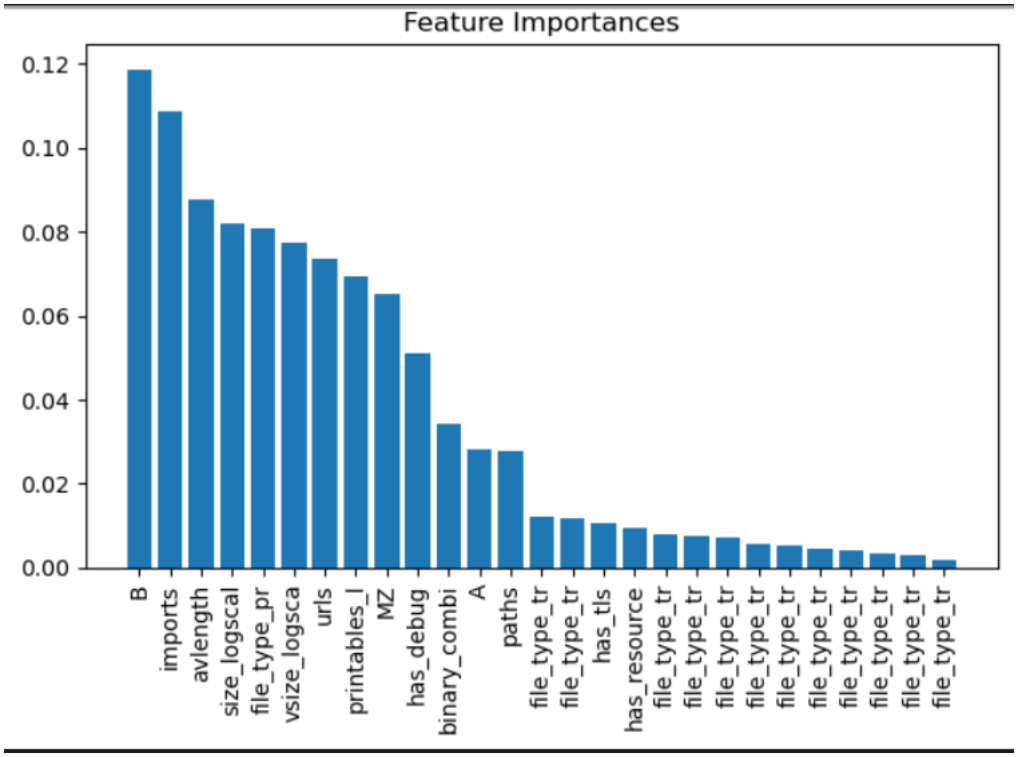
```
Above plot results:
  ID    Algoritm       train score[%]  val score[%]
   1.   GNB            81.494201%       81.210844%
```

Decision Tree overfitted result before & after reducing dimensionality with ‘max_features’ parameter:

```
Above plot results:
  ID    Algoritm       train score[%]  val score[%]
   4.   DecisionTree   90.947531%       82.469962%
Next Model:
```

```
Above plot results:
  ID    Algorithm      train score[%]  val score[%]
   4.   DecisionTree   89.124101%       83.520368%
```

6.18 Appendix R – Feature Importance Analysis




```
Top 15 Feature Contributions:
B: 0.1187
imports: 0.1087
avlength: 0.0879
size_logscal: 0.0822
file_type_pr: 0.0810
vsize_logsca: 0.0775
urls: 0.0736
printables_l: 0.0693
MZ: 0.0654
has_debug: 0.0511
binary_combi: 0.0344
A: 0.0282
paths: 0.0278
file_type_tr: 0.0123
file_type_tr: 0.0118
```

6.19 Appendix S – models hyper-parameters explanation:

6.1.1 2 basic models

6.1.1.1 KNN

Hyper - Parameters tuning:

In the KNN model, there are 2 hyper-parameters that can be configured - the metric and the number of the closest neighbors that the model calculates by.

- **K:** The number of nearest neighbors to consider. A smaller K leads to lower bias but higher variance, while a larger K results in higher bias but lower variance.
- **Distance metric:** The measure used to calculate distances between data points. Different metrics can impact the bias-variance trade-off.

By default, the KNeighborsClassifier uses the Euclidean distance metric and k=5 (5 nearest neighbors).

To choose the appropriate hyperparameters, we used the grid search technique, with this grid:

param_grid = {'n_neighbors': [3, 5, 7], 'metric': ['euclidean', 'manhattan']}

We wanted to check a few numbers of neighbors, above and under the default value, and both metrics to find the best combination for our model.

The parameters that were chosen by the grid search were - **{'metric': 'manhattan', 'n_neighbors': 3}**

The parameters might change a bit in every run of the project, as the differences between some combinations can be very small, and the exact train data change every time (because of the random train and validation split).

6.1.1.2 Naïve-Bayes Classifier

There are three types of Naive Bayes models: Gaussian, Multinomial, and Bernoulli.

1. **Gaussian Naive Bayes** – This is a variant of Naive Bayes that supports continuous values and has an assumption that each class is normally distributed.
2. **Multinomial Naive Bayes** – This is another variant that is an event-based model that has features as vectors where the sample(feature) represents frequencies with which certain events have occurred.
3. **Bernoulli** – This variant is also event-based where features are independent boolean which are in binary form.

Based on the given description, we excluded the Multinomial and Bernoulli types as they are more suitable for discrete (like word frequencies or presence/absence indicators) and binary features, respectively. Since our dataset primarily consists of continuous features, we determined that Gaussian Naïve Bayes is the most suitable choice among these types.

When utilizing this model, we made the following assumptions:

Independence of features: Gaussian Naive Bayes assumes that the features are independent of each other, given the class label. This assumption considers that the presence or absence of one feature does not affect the presence or absence of any other feature. Although we acknowledge that this assumption may not hold true in our case, even after removing highly correlated features, we still incorporated it into the model.

Gaussian distribution of features: Another assumption is that the numerical features in our data follow a Gaussian (normal) distribution. This enables the model to estimate the class-specific mean and variance parameters for each feature. Although we recognize that this assumption may not be valid for all features, we still accounted for it in the model.

Hyper - Parameters tuning:

In the Gaussian Naive Bayes Classifier model, there are typically no hyperparameters to tune, as the algorithm makes strong assumptions about the independence of features given the class label. However, in scikit-learn's implementation of Naive Bayes, there is one hyperparameter that we can specify: the prior probabilities.

By the train data, as we saw in the exploration part, the prior probabilities are very close to 0.5. We'll check the exact value of it and assume that it reflects the real priors probabilities.

The priors that were calculated and inserted into the model were - **[0.50028889 0.49971111]**

The priors might change a bit in every run of the project, as the exact train data change every time (because of the random train and validation split).

6.1.2 2 advanced models

6.1.2.1 Decision Tree

Hyper - Parameters tuning:

In a decision tree model, hyperparameters play a crucial role in controlling the model's complexity and, consequently, its bias-variance trade-off. The important hyperparameters in a decision tree are:

- **Maximum Depth (max_depth):** This hyperparameter determines the maximum depth or the maximum number of levels in the decision tree. A deeper tree can represent more complex relationships in the data, potentially leading to lower bias. However, deeper trees also tend to have higher variance as they may overfit the training data. Restricting the maximum depth can help control overfitting and reduce variance at the cost of increased bias.
- **Minimum Samples Split (min_samples_split) (default=2):** This hyperparameter sets the minimum number of samples required to split an internal node. It controls when to stop splitting nodes based on the number of samples. A smaller value allows for more splits and finer partitions of the data, potentially leading to lower bias. However, a smaller value can also increase the likelihood of overfitting and higher variance. Increasing the minimum samples split helps control overfitting, reducing variance while introducing a slightly higher bias.
- **Minimum Samples Leaf (min_samples_leaf) (default=1):** This hyperparameter sets the minimum number of samples required to be at a leaf node. Similar to min_samples_split, it controls when to stop growing the tree. Setting a smaller value allows the tree to create more specific leaves, potentially resulting in lower bias. However, smaller values can lead to overfitting and higher variance. Increasing the minimum samples leaf helps control overfitting, reducing variance while introducing a slightly higher bias.
- **Maximum Features (max_features) (default=None):** This hyperparameter determines the maximum number of features to consider when looking for the best split at each node. A smaller value reduces the number of features considered, potentially reducing variance. However, it may also increase bias if important features are not considered. A larger value allows for more features to be considered, potentially reducing bias but increasing variance.

To choose the optimal parameters for our decision tree model, we utilized the grid search technique.
We carefully selected a range of options to ensure that each parameter's value does not lead to overfitting. For instance, we deliberately omitted the 'None' option for the max_depth parameter to prevent the formation of deep overfitted trees.
By conducting the grid search with these specific parameter options, and a cross-validation technique we aimed to identify the best values for each parameter, striking a balance between model complexity and generalization.
The parameters that were chosen by the grid search were - **{'max_depth': 9, 'max_features': 'log2', 'min_samples_leaf': 2, 'min_samples_split': 10}**
The parameters might change a bit in every run of the project, as the differences between some combinations can be very small, and the exact train data change every time (because of the random train and validation split).

6.1.2.2 Random Forest

Hyper - Parameters tuning:

We began by examining the default parameters of the Random Forest model, which provided a lengthy list. To determine where to start, we consulted the Scikit-Learn documentation, which highlighted the key settings: n_estimators (number of trees) and max_features (number of features considered for splitting).
Instead of diving into research papers, we opted for an empirical approach, trying out various hyperparameter values to observe their impact. The hyperparameters we focused on are:

- **n_estimators:** It represents the number of trees in the random forest. Increasing the number of trees generally reduces the variance of the model, as the predictions from multiple trees are averaged. However, a very large number of trees may lead to increased computational complexity and potential overfitting if the trees start to memorize the training data.
- **max_features:** It determines the maximum number of features considered for splitting a node in each tree. A smaller value reduces the correlation between individual trees, leading to lower variance. However, setting it too low may result in high bias, as important features might be ignored.
- **max_depth:** It defines the maximum depth or the maximum number of levels in each decision tree. A deeper tree can capture complex relationships, potentially reducing bias. However, deeper trees are more prone to overfitting and can result in higher variance. Restricting the maximum depth helps control overfitting but may introduce a slightly higher bias.
- **min_samples_split:** It sets the minimum number of data points required in a node before it can be split. Increasing this value reduces the complexity of the model, resulting in higher bias but lower variance. It helps prevent overfitting by requiring a certain number of samples for a split to occur.
- **min_samples_leaf:** It determines the minimum number of data points allowed in a leaf node. Similar to min_samples_split, increasing this value reduces the complexity of the model, potentially increasing bias and reducing variance.
- **bootstrap:** It specifies the method for sampling data points for training each tree. When set to True, random sampling with replacement (bootstrap) is used. This introduces randomness and helps reduce variance. If set to False, the entire dataset is used for training each tree, which can lead to higher bias but lower variance.

By adjusting these hyperparameters, we aimed to strike a balance between bias and variance, enhancing the performance of our Random Forest model.
Given the large number of possible combinations of hyperparameter settings (5 * 4 * 3 * 3 * 3 * 2 = 1080), conducting a grid search would require extensive computation time. To overcome this challenge, we opted for a more efficient approach called random search.
Random search selects parameter values at random, allowing us to sample a diverse range of combinations without exhaustively testing every possibility. While it may not guarantee the absolute best parameter combination, it provides a good enough result by exploring a wide range of values. This approach significantly reduces the computational burden associated with grid search, making it a practical choice for our scenario.
The parameters that were chosen by the random search were -

{'n_estimators': 500, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_features': 'log2', 'max_depth': None, 'bootstrap': False}

The parameters might change a bit in every run of the project, as the differences between some combinations can be very small, and the exact train data change every time (because of the random train and validation split).
And for this model, they are also chosen with the random search that chooses randomly which combination to test each time.

6.20 Appendix T – real-life model evaluation based on our statistics.

So why 95.54% is not enough in real life?

Because you can't just consider the sensitivity/specificity of the algorithm, you must consider the malicious over legitimate traffic ratio to understand how many alerts will be generated. And this ratio is extremely low. Let's consider that you have 1 malicious event every 10 000 events (it's a high ratio) and 1 000 000 events per day, you will have:

- 100 malicious events, 95 identified by the tool, and 19 false negatives (20.27% FNR but let's consider 20% here)
- 999 900 legitimate events, around 57894 were identified as malicious (5.79% FPR)

So, in the end, the analyst would receive 57894 alerts per day with only 95 true positives in it (0.001%). Your IDS is useless here. (Out of curiosity, the same problem applies when detecting cancer at a large scale (in French))

If we take an example, a standard Windows 7 install has around 15000 binary files (like our validation data size), which means that on average 350 files would be detected incorrectly as malicious. And that's a bad antivirus.