Saf Flatters

network intrusion prediction USING CLASSIFICATION

Machine Learning Model Evaluation Project – 2024

**Contents**

1. Introduction
2. Main Findings
3. Methodology
4. Data Preparation
   1. Data Types
   2. Missing Entries
   3. Incorrect Entries
   4. Duplicates
   5. Feature Engineering
   6. Irrelevant Attributes
5. Training and Test Data Sets
   1. Test Data Preparation
   2. Exported to ARFF
6. Feature Selection and Transformation
   1. Feature Selection
   2. Scaling / Standardisation
   3. Data Balance
7. Classification
   1. Stratification
   2. Hyperparameter tuning and Cross Validation
   3. Prediction Evaluation
      1. Accuracy
      2. Precision and Precision for Normal
      3. Recall
      4. F1 Score
   4. K-Nearest Neighbours Classifier
   5. Naïve Bayes Classifier
   6. Decision Trees Classifier
   7. Random Forest Classifier
   8. Histogram Gradient Boosting Classifier
8. Best Predictions
9. References
10. **Introduction**

Data mining classification techniques play a crucial role in addressing network intrusion challenges due to the ability to learn by example. By leveraging supervised machine learning, data specialists can classify various types of attacks using network traffic data with access to existing data, enabling them to protect systems autonomously. In this project, I performed predictive analytics using data preparation tasks such as feature extraction, data cleaning and data transformation on historical network data and then build (with Python) and evaluate supervised machine learning classifiers such as ‘k-Nearest Neighbours’, ‘Naïve Bayes’, ‘Decision Trees’ and two meta-classifiers, ‘Random Forest’ and ‘Gradient Boost’ to determine whether new network activity is normal or one of the eight malicious attack types (and what attack type it is).

This report will describe my methodology in which to classify attack categories. I also extend further, assuming that a priority is to avoid all attacks regardless of their type and I built a custom metric to evaluate these classifier models on their Precision score for “Normal”. This report will also describe how I pre-processed the provided training data, prepared the test data, selected and transformed appropriate features, tuned the hyperparameters of various classifiers, cross-validated and then made attack category predictions on the test data and finally how I evaluated the accuracy and precision across all models. This submission also includes a Python Notebook with my data exploration, two Python files with all the code required to replicate my project (a run file and a functions file), the given training set and two test sets, my processed training set and two test sets in arff format and two csv files called ‘predict1.csv’ and ‘predict2.csv’ with my two test data set predictions.

1. **Main Findings**

The Random Forest Classifier equalled with the Decision Tree Classifier with an accuracy of 70% for Test Data 1 and 82% for Test Data 2. However Random Forest Classifier outperformed all other classifiers in terms of reducing the misclassification of malicious attacks as ‘Normal’ with only misclassifying 7 attacks in Test Data 1 and 6 in Test Data 2 (out of 59809 instances).

1. **Methodology**

In this study, the first step in tackling network intrusion classification involved cleaning and preprocessing the given dataset to build a robust training set for the classifiers. Data exploration was conducted in a Jupyter Notebook (“Data\_Preprocessing\_Notebook.ipynb”) using Python, and the subsequent data cleaning and preprocessing steps were performed in “run.py,” with functions modularised in “Runpy\_Functions.py” to promote reusability and readability. The preprocessing included tasks such as identifying and converting data types, handling missing or incorrect entries, managing row and attribute duplication and deriving new features from existing attributes.

Once the dataset was preprocessed, the cleaned training data was exported to ARFF format ('processed\_training\_data.arff'). Similarly, the test data (two sets) underwent feature engineering, where new features were derived from existing ones, and the processed test sets were exported into ARFF format ('processed\_test\_set1.arff' and 'processed\_test\_set2.arff'). Feature selection and transformation was performed on the data sets to identify the most relevant features, and three final dataframes (one training set, two test sets) were created based on the selected features. This step was iterative to optimise the feature combinations and improve classifier performance. Once features were selected, numerical variables within this subset (except for binary attributes) were standardized by fitting a StandardScaler which was then used to transform the test data (using the same standard deviations). The data was also transformed by balancing the class distribution of the target variable (with SMOTE).

The classification phase of this study was also executed within 'run.py.' The classifiers employed were k-Nearest Neighbours, Naïve Bayes, Decision Trees, and two meta-classifiers, Random Forest, and Gradient Boost. The training data was split into training and validation sets using stratified splitting to maintain the proportional representation of all categories of the target variable (‘attack\_cat’). For each classifier, a Grid Search with Stratified k-Folds cross-validation was used to identify the best hyperparameters. Due to computational limitations, hyperparameter options were limited. Once the optimal hyperparameters were selected, the classifiers were trained (fitted) on the processed training set.

After training, the models were used to make predictions on both the training set and the two test sets. The performance of each model was evaluated using the metrics “Accuracy”, “F1 Score”, “Weighted Recall”, “Weighted Precision” and “Precision for Normal”, the latter a metric tracking the number of instances were predicted ‘Normal’ but were actually a malicious attack. These final predictions were visualised with confusion matrices and feature importance bar charts to compare the performance of the models. This evaluation process was iterative, and adjustments to the feature set and further feature engineering were made when necessary to improve model accuracy.

Finally, code was written in 'run.py' to automatically compare the results of all classifiers and select the two best-performing models. The predictions from these top two classifiers on the test sets were outputted to ‘predict\_1.csv’ and ‘predict\_2’ for final evaluation.

1. **Data Preparation**

The given training data set was a nondependent-orientated multidimensional data set in an excel spreadsheet. It contained over 145,000 instances with 42 attributes and 2 target attributes, in which the target attribute of interest is ‘attack\_cat’ which describes which attack type each instance is. As per good statistical practice, I relocated the target variable to the first column. The data preprocessing phase is highly application specific and a significant limitation of this study includes a lack of domain specific personnel on the task. This led to decisions being made on intuition and evidence only found within the dataset but may not reflect real world practices. Data Cleaning led to 145,853 instances, 117 variables (75 bool, 11 floats, 30 integers and 1 target variable).

* 1. **Data Types**

My preliminary exploration findings found the dataset contained three categorical variables (not including the targets) the rest were numerical. Of those that were numerical, two variables presented as binary (‘swin’ & ‘is\_sm\_ips\_ports’). There were a further four variables that appeared to be inconsistently binary and may require altering later on (‘dwin’, ‘is\_ftp\_login’, ‘ct\_ftp\_cmd’, ‘is\_sm\_ips\_ports’). As for the other numerical columns, most appeared to be continuous, however there were four variables that appeared to be discrete (‘sttl’, ‘dttl’, ‘trans\_depth’ and ‘ct\_state\_ttl’) and two numerical variables that may appear to be sequenced strings (‘stcpb’ and ‘dtcpb’).

See 4.7 Feature Engineering section for conversions into different data types (data portability).

* 1. **Missing Entries**

There are no ‘NULL” entries in this dataset.

* 1. **Incorrect Entries**

Incorrect instances were found by exploring the number of unique items in the categorical and binary-appearing variables. This included:

|  |  |
| --- | --- |
| **Incorrect Entry** | **Rectification** |
| ‘state’: “no” (1 entry and the rest of the categories are capitalised) | Deleted Instance (due to size of dataset, no big loss to sample size) |
| ‘dwin’: 244, 70 (2 entries that appeared non-consistent with the binary-appearing variable) | Deleted Instance (due to size of dataset, no big loss to sample size) |
| ‘is\_ftp\_login’: 4, 2 (24 entries that appeared non-consistent with the binary-appearing variable and not related to minority attack category) | Imputed entries with mode of all entries for variable (0) |
| ‘ct\_ftp\_cmd’: 4, 2 (24 entries that appeared non-consistent with the binary-appearing variable and not related to minority attack category) | Imputed entries with mode of all entries for variable (0) |

* 1. **Duplicates**

**Column Duplications:** When exploring the data, I found that two columns were identical.

|  |  |
| --- | --- |
| **Column Duplicate** | **Rectification** |
| ‘is\_ftp\_login’ with ‘ct\_ftp\_cmd’ – 100% duplication | Dropped ‘ct\_ftp\_cmd’ |

**Row Duplications:** During analysis, I found 38% of instances were duplicates with other instances. Due to the nature of the data being Network Intrusion data, with a lack of domain knowledge I have been hesitant to remove these duplicates. My intuitive reasoning is that some attacks may be replicating the profile many times over to try and gain access via brute force. I analysed the duplications and found that the percentages of duplicates were not even for the attack type. For example, Generic attacks were 89.55% duplications where Normal (non-attacks) were only 4.16% duplications. I simulated removing the duplicates and plotting the number of attacks as a percentage of all the data and found that removing the duplicates made the attack types even more disproportionate. This aligned with my intuition, and I ultimately kept the duplicated instances.

A graph of different colored bars

Description automatically generated

* 1. **Feature Engineering**

**Binarization of discrete features:** There were multiple attributes that required binarization where either to attribute was already binary but need to be in actual binary format or where the attribute could be turned into a binary format by binning the discrete entries into a 1 or a 0 (using data exploration graphs to confirm). Also had to change ‘is\_sm\_ips\_ports’ from an ‘int64’ datatype to a ‘bool’.

|  |  |
| --- | --- |
| **Attribute to Binarize** | **Feature Engineered** |
| ‘swin’ | ‘swin\_bin’ |
| ‘dwin’ | ‘dwin\_bin’ |
| ‘trans\_depth’ | ‘trans\_depth\_bin’ |
| ‘is\_ftp\_login’ | ‘is\_ftp\_login\_bin’ |
| ‘sttl’ | ‘sttl\_bin’ |

**One-hot-encoding:** For the categorical attributes, I one-hot-encoded ‘state’, ‘proto’ and ‘service’. This created binary attributes for every category within these attributes. Choosing which attributes to keep involved one-hot-encoding the attack categories temporarily and creating a correlation heat map to determine which ones were more associated with attack categories (after balancing the data). See example of correlation heatmap with ‘service’ below.

A colorful squares with black text

Description automatically generated with medium confidence

A black background with white text

Description automatically generated**Edit Distance:** ‘stcpb’ and ‘dtcpb’ are sequenced strings of numbers. I performed analysis on the edit distance between these two columns and determined that 26% of “Normal” (in ‘attack\_cat’) would have an edit distance less than 7 if all 0s were removed. In other attack categories, an edit distance under 7 was only 6%! The percentage of 0’s in ‘stcpb’ also indicated differences between attack categories as “Generic” was made up of 98% of 0’s (compared to “Normal” that was 30% and “Worms” that was only 11%). I derived a column called ‘edit\_distance’ where it was 0 to 10 (where 0 meant no sequence at all as there were no duplications between ‘stcpb’ and ‘dtcpb’). Edit Distance is calculated by:

Equation : (Aggarwal, 2015)

|  |  |
| --- | --- |
| **Attributes to calculate Edit Distance** | **Feature Engineered** |
| ‘stcpb’ and ‘dtcpb’ | ‘edit\_distance’ |

A graph with blue squares and black dots

Description automatically generatedA graph of a bar chart

Description automatically generated with medium confidence

**Taking the mean of multiple columns combined:** Using another correlation heat map I investigated starting with ‘ct’ and found that all except for 'ct\_state\_ttl' and 'ct\_flw\_http\_mthd' were highly correlated. I created a function that summed them all together and divided by 9 (the average of them all).

|  |  |
| --- | --- |
| **Attributes to combine and take the mean of** | **Feature Engineered** |
| 'ct\_dst\_ltm', 'ct\_src\_dport\_ltm', 'ct\_dst\_sport\_ltm', 'ct\_dst\_src\_ltm', 'ct\_src\_ltm', 'ct\_srv\_dst', | ‘ct\_mean’ |

A colorful squares with black text

Description automatically generated

**Rate:** I investigated all attributes separately during the data exploration phase and this one flagged for me. It is a strangely built where many of the instances had unusual numbers that had lots of common digits (generally those numbers were over 10,000) and many numbers of different amounts which visually looked some-what correlated with attack categories. Only 1% of ‘Normal’ was over 1000 in comparison to 98% of Generic was. I engineered two features (one binary, one discrete) that may assist in classification.

|  |  |
| --- | --- |
| **Attributes to engineer** | **Feature Engineered** |
| ‘rate’ | ‘rate\_over\_10000’ |
| ‘rate’ | ‘rate\_digit\_count’ |

**4.7 Irrelevant Attributes**

Irrelevant attributes will typically reduce classification accuracy and increase inefficiency. (Aggarwal, 2015). Although feature selection will be conducted post-preprocessing – there are attributes that may be removed from that selection process due to either being closely correlated with other attributes or have been used for feature engineering and therefore are now redundant (some, not all). Due to the set up of this project, I will not be dropping these columns off the training and test set, I will be just removing these from the Feature Selection options. There are 36 valid feature selections.

**Closely correlated columns removed:** Due to highly correlated with other variables, removed: dwin, dbytes, dloss, sbytes, sloss, swin, synack, ackdat, sinpkt

**Columns that have been used to derive better columns in feature engineering to be removed:** 'proto', 'state', 'sttl', 'state\_encoded', 'service\_encoded', 'service', 'stcpb', 'dtcpb', 'dttl', 'ct\_dst\_ltm', 'ct\_src\_dport\_ltm', 'ct\_dst\_sport\_ltm', 'ct\_dst\_src\_ltm', 'ct\_src\_ltm', 'ct\_srv\_dst'.

1. **Training and Test Data Sets**
   1. **Test Data Preparation**

Preparing the test sets is crucial to ensure that features engineered with the training data were also engineered in the test data to ensure that the model can fit the test set properly; if the features are not consistent between the training and test sets, the model will be unable to make accurate predictions, leading to skewed evaluation metrics and potentially misleading conclusions about its performance.

* 1. **Exported to ARFF**

Once the training set has been cleaned and the test sets were prepared, they were exported to ARFF format.

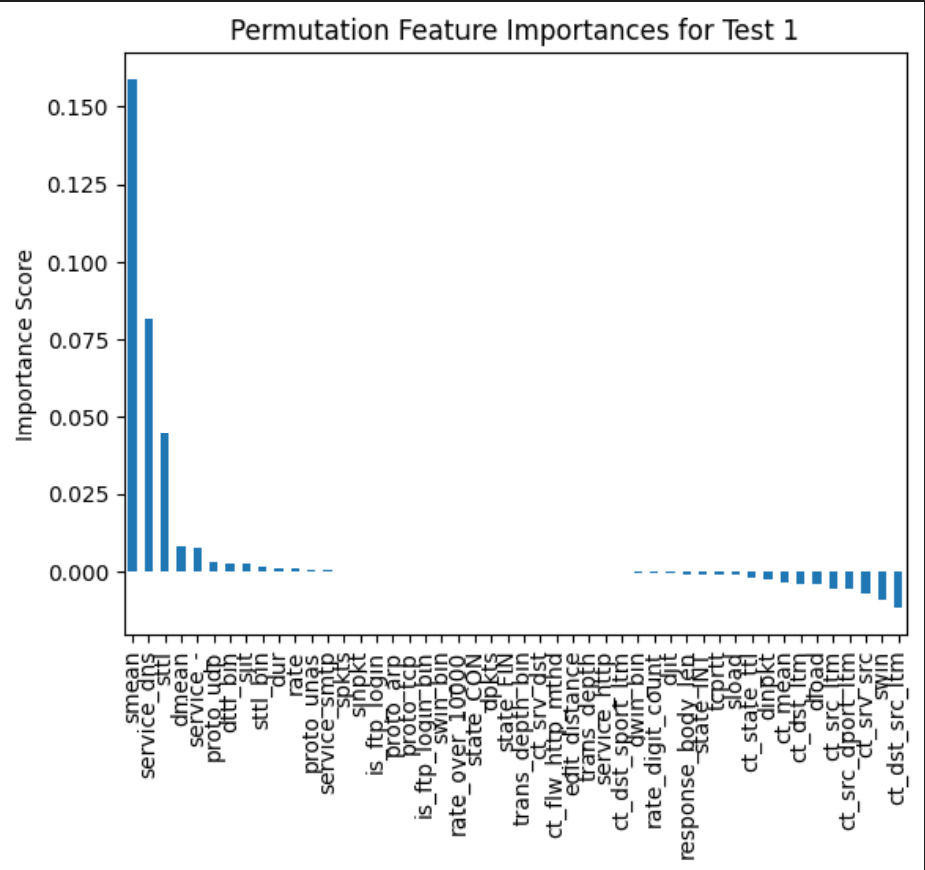
1. **Feature Selection and Transformation**

Feature selection and transformation should not be considered a part of data preprocessing as the features selected can be tightly integrated with the specific algorithm or embedded model. It is to be done before the Classification part of Supervised Learning (Aggarwal, 2015). Through my research I found that specific classifier algorithms tend to adapt better to particular data types.

* 1. **Feature Selection**

To select features, there are a few options to consider. There is Filter, Wrapper, Embedded and manual (Aggarwal, 2015). I went with the manual feature selection option and therefore I had the ability to interpret the process more as this has been my first machine learning exercise.

Once removing the features outlined in Section 4.7 Irrelevant Attributes, there were 36 features to make a subset for each classifier. I researched about what each classifier prefers as features and then once my classifiers were built, used my best classifier as a base for a permutation\_importance tool (Galli, 2024). I plotted all valid features using Gradient Boost and then removed all the features that scored negatively. Then through a manual process of refining the subselection for each classifier and its own permutation\_importance or feature\_importances\_ (for tree based classifiers to see splits) (Scikit Learn, 2024) to get the best result.

A graph with blue lines

Description automatically generated

I then interrogated each confusion matrix to determine the biggest weaknesses. For all classifiers except for Bernoulli Naïve Bayes (which used a very different subset due to binary data types only), I found the biggest concern was ‘Normal’ was being predicted as ‘Fuzzer’ attacks only in the Test sets (especially Test Set 1). During this study, I could not find a way to overcome this Data Drift, which is where the historical data may be outdated which I found when comparing Fuzzers in the Training Data to the Test Data (Ali, 2023). If this was a real life scenario, I would advise on more data collection or employ a domain expert to advise before further feature engineering

* 1. **Scaling / Standardisation**

I standardised all numerical columns (except binary). This was to avoid larger scaled attributes dominating classifier choices in certain scenarios (eg. Using Euclidean distance in kNN) (Aggarwal, 2015). Using Sklearn StandardScaler (Standardise rather than MinMaxScale), I fit the scaler to the training data so I could use the same scaler to the test data sets later (using the training data’s standard deviation). Standardisation is more robust than scaling as it compensates better with outliers.

* 1. **Data Balance**

To ensure good representation of all attack categories in the training data, I balanced the dataset through a combination of oversampling and undersampling, targeting 60,000 overall – this number was derived from the number in the test sets (I wanted it to be minimal as possible for speed but didn’t want to fall under the test set amount). I applied SMOTE (Synthetic Oversampling) to the minority classes, generating new data points between existing instances and their nearest neighbours, which is more effective than basic oversampling (Aggarwal, 2015). After reaching over 6,500 for all minority classes, I used undersampling on classes exceeding this count, resulting in a balanced dataset (Swastik, 2024). Ultimately, I found better accuracy without employing SMOTE.

1. **Classification**
   1. **Stratification**

I used sklearn’s train\_test\_split to divide the training and testing subsets for validation with 80% of the training set to be used to train the data and 20% to be used for testing. I included the ‘stratify=y’ parameter to ensure both training and test sets maintain the similar proportion of attack\_cat classes. This also helps maintain class representation across all classes especially if the classes are imbalanced (Aggarwal, 2015).

* 1. **Hyperparameter tuning and Cross Validation**

After selecting the appropriate classifier, I initialized it within my hypertune\_w\_CV function, which also implements GridSearch for hyperparameter tuning. GridSearch systematically evaluates all possible combinations of hyperparameters to identify the settings that maximize the classifier's accuracy (Scikit Learn, 2024). For a detailed overview of each classifier and its parameters, refer to their respective sections. Additionally, I incorporated the ‘cv=skf’ parameter within GridSearchCV, utilising StratifiedKFold with 10 folds. This cross-validation strategy divides the data into 10 segments while preserving the class distribution. By setting shuffle=True, the data is randomised before the split. The function then prints the best hyperparameters found during this process. These optimal parameters are subsequently employed to fit the classifier to 80% of the training data, allowing it to make predictions on the remaining 20%. I then assessed the generalisation of the model with cross\_validation\_score. This allowed evaluation on the performance of the tuned model and outputted “Mean Accuracy” and “Standard Deviation of Accuracy”. I also custom built a scorer as a second cross validation metric. Please see information about custom scorer in Section 7.3.2. Precision and Precision for Normal.

*Note for Assessor: In the submitted code, I have put an if statement in each classifier section where if ‘hypertune’ == “show” it will complete GridSearch and tune the hyperparameters. If ‘hypertune’ == “skip” it will skip this process and use the hyperparameters I found to have been the best during testing (and stated in the output and in this report). This is to increase the speed of the code.*

* 1. **Prediction Evaluation**

To evaluate how well a Classifier performs in a multi-class problem, I used a combination of evaluation scores from the confusion matrices. A limitation of this study was not having information of what would be deemed a priority however I made assumptions that can be generalised to real-world network intrusion problems. The score types are made from “True Positives” and “True Negatives” where the classifier has correctly predicted an instance to be the class it actually is, “False Positives” where the classifier incorrectly predicts an instance to be in the class when it actually is a different class and “False Negatives” where the classifier incorrectly predicts an instance to not be of a class but it should be. “Accuracy”, “Precision for Normal” and “F1 Score”, “Weighted Average Precision” and “Weighted Average Recall” are printed to terminal and the first three are published on each Confusion Matrix.

* + 1. **Accuracy**

The score “Accuracy” refers to and is a general measure how well the classifier performs. However, in cases of imbalanced datasets, accuracy can be misleading, as it may be inflated or deflated depending on the frequency of certain classes (Aggarwal, 2015). For instance, if an attack class is rare but the cost of missing an attack is high, accuracy alone is not a reliable metric to assess how well the classifier detects that specific attack. This will still be a metric I will use as actual accuracy.

* + 1. **Precision and Precision for Normal**

The score “Precision” calculates the proportion of instances that were correctly identified by which measures how accurate a positive prediction is (Evidently AI, 2024). A higher precision would result in fewer false alarms in a two class scenario. However, in this multi-class scenario I wanted to use a metric that evaluated how many instances would be predicted as “Normal” but would actually be an attack. In 2022, the Australian government introduced harsher penalties for serious data breaches (Dreyfus, 2022). As businesses are under immense data security scrutiny, face large fines and shareholder pressure to protect their network from malicious attacks – I have made increasing this score a priority. I have coded ‘Precision for Normal’ figures as a metric printed to the terminal and on the Confusion Matrix. This calculation used sklearn.metrics precision\_score plug in. See below for an example image of a confusion matrix with showing how ‘Precision for Normal’ is calculated.

This metric calculates:

*Where: ‘Predicted ‘Normal’ but Attack’ is the sum of ‘Predicted Normal’ but True to another class (all other classes are attacks).*

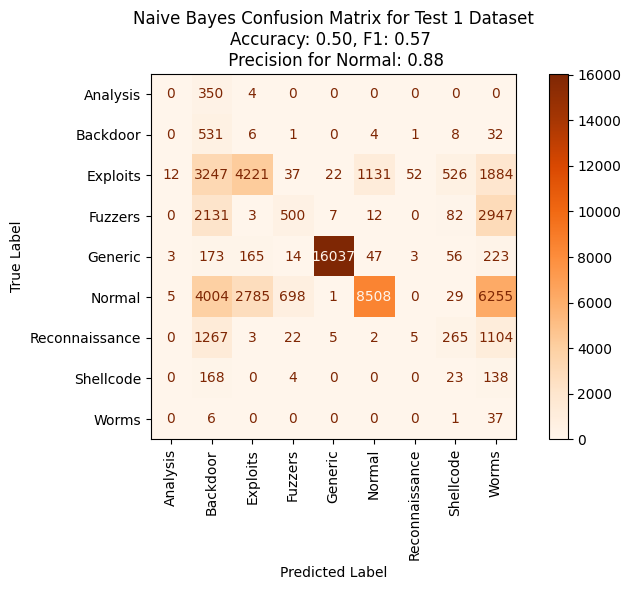


Figure : An example of a 'Precision for Normal' calculation using Confusion Matrix

This custom metric is also employed as a secondary measure during Cross-Validation to assess the robustness of the model's performance across different subsets of the data.

* + 1. **Recall**

For two class problems, “Recall” would be a better metric to track when failing to detect an attack is priority as is: (Evidently AI, 2024). However in this multi-class scenario, I could not use this as the sklearn.metric recall\_score would only calculate when the True class in Normal but the Predicted classes were attacks. Recall is however used as part of “F1 Score”

* + 1. **F1 Score**

These scores “Recall” and “Precision” result in a lot of metric scores so it’s common for them to be combined together across the classes. Both can be measured with ‘weighted’ which takes into account the frequency of each class or ‘macro’ which calculates each class separately and then takes the average (not taking into account class imbalances).

The score ‘F1” is calculated by and is a good measure when there is an uneven class distribution and a balance between Precision and Recall (Kundu, 2022).

* 1. **K-Nearest Neighbours Classifier**

**What:** K-Nearest Neighbours (KNN) is a type of instance-based learning algorithm. Instead of learning a model during training, KNN simply stores the entire training dataset. When making predictions, it calculates the distance between a new data point and all the points in the training set (Aggarwal, 2015). Based on a specified number, k, KNN identifies the k nearest neighbours and assigns the class of the majority among those neighbours to the new data point.

**The Good:** KNN is easy to implement, works well with smaller datasets, and performs better with lower dimensionality. However, I noticed that increasing the number of features improved my test results. It handles all data types, but it's important to standardise numerical data for optimal performance.

**The Bad:** KNN is slow because it calculates distances between each test point and the entire training set, making it inefficient for large datasets. It's also sensitive to noise and outliers, which can negatively impact its performance.

**Feature Selection:** To improve KNN (within computational limitations), I used SMOTE to under sampled all classes to 12500 each (rather than 125000). I reduced features to between 6 – 15 for lower dimensionality.

These features were:  'sttl', 'smean', 'dmean', 'service\_-‘, 'service\_smtp', 'service\_dns', 'proto\_udp', 'tcprtt', 'ct\_state\_ttl', 'ct\_mean',  'sjit',  'dur',  'sload'

**Hyperparameters:** Initially, I used Grid Search with Cross Validation to find the optimal number of neighbours, weight type (distance or uniform), and distance metric (Euclidean, Manhattan, or Minkowski) (Scikit Learn, 2024). However, due to computational constraints, I narrowed the selection to only odd values of n neighbours between 3 and 17, selected weights as ‘distance’ meaning closer neighbours have more influence (decided through trial and error) and used the ‘manhattan’ distance metric which is more robust with outliers and high dimensional data (Chugani, 2024).

**Optimal Hyperparameters:** The optimal hyperparameters chosen during the Grid Search with Cross Validation were: 'metric': 'manhattan', 'n\_jobs': -1, 'n\_neighbors': 11, 'p': 1, 'weights': 'distance'

**Cross Validation:** 10 fold cross validation with mean accuracy of 0.9071, standard deviation of accuracy of 0.0017, mean Precision score for ‘Normal’ of 0.9968 and standard deviation of Precision score for ‘Normal’ of 0.0011. The low standard deviation of cross validation proves the model is consistent across subsets and therefore robust. This suggests similar scores on unseen data if the test data follows similar trends.

**Best Results:**

KNN scoring on Training set:

Accuracy: 0.91, F1 score: 0.9, Precision score for 'Normal': 0.9966

Weighted Averaged Precision: 0.91, Weighted Average Recall: 0.91

KNN scoring on Test 1 set:

Accuracy: 0.69, F1 score: 0.7, Precision score for 'Normal': 0.9911

Weighted Averaged Precision: 0.82, Weighted Average Recall: 0.69

KNN scoring on Test 2 set:

Accuracy: 0.8, F1 score: 0.82, Precision score for 'Normal': 0.9937

Weighted Averaged Precision: 0.8588, Weighted Average Recall: 0.8

A graph of data analysis

Description automatically generated with medium confidenceA graph of different numbers

Description automatically generated with medium confidenceA chart with numbers and a number of different colored squares

Description automatically generated with medium confidence

* 1. **Naïve Bayes Classifier**

**What:** One example of a probabilistic classifier is Naïve Bayes, which utilises conditional probabilities for class classification. There are three main types of Naïve Bayes classifiers: Gaussian, Multinomial, and Bernoulli. Gaussian is suitable for continuous data that follows a normal distribution, while Bernoulli is designed for binary features (Scikit Learn, 2024). The type of Naïve Bayes classifier chosen will ultimately influence the feature selection approach. I tested both of these with different subsets of features in an attempt to improve accuracy scores. I ultimately used Bernoulli because it gave me a chance to explore some features that the other classifiers didn’t look at due to the subset having to be binary only. The Bernoulli Naïve Nayes formula is: .

**The Good:** It can handle large datasets. It is fast

**The Bad:** Data Types are important to determine which Naïve Bayes to use and will affect feature selection. An increase in dimensionality can result in a decrease in estimation accuracy. If the training data has a zero probability for a particular class, the model will fail to recognize that class in the test data if it appears (Jayaswal, 2020). However, Naïve Bayes employs "Laplace Smoothing" by default, which adds 1 to any probabilities that are 0%.

A graph with blue and white bars

Description automatically generated**Feature Selection:** I found better results with Bernoulli Naïve Bayes and therefore my feature selection was ['attack\_cat', 'dttl\_bin', 'sttl\_bin', 'proto\_udp', 'state\_CON', 'service\_dns', 'proto\_arp', 'is\_sm\_ips\_ports', 'service\_-', 'proto\_ipv6', 'service\_pop3', 'service\_snmp', 'proto\_rsvp', 'proto\_cbt', 'proto\_st2', 'proto\_xnet', 'proto\_hmp', 'proto\_mux', 'proto\_emcon', 'proto\_sat-mon', 'proto\_iatp', 'proto\_iso-ip', 'proto\_tlsp', 'proto\_sep', 'proto\_tcf', 'proto\_rvd', 'proto\_pup', 'proto\_sccopmce', 'proto\_swipe', 'proto\_iso-tp4', 'service\_dhcp', 'proto\_ippc', 'proto\_pim', 'proto\_netblt', 'proto\_sun-nd', 'proto\_aes-sp3-d', 'proto\_micp', 'proto\_mtp', 'proto\_encap', 'proto\_larp', 'proto\_mobile', 'proto\_irtp', 'proto\_ax.25', 'proto\_egp', 'proto\_igmp', 'proto\_ipip', 'proto\_nvp', 'proto\_trunk-1', 'proto\_argus', 'proto\_mfe-nsp', 'service\_ssl', 'proto\_gre', 'proto\_etherip', 'proto\_igp', 'proto\_visa', 'proto\_sps', 'proto\_srp', 'proto\_leaf-2', 'service\_radius', 'proto\_bbn-rcc', 'proto\_narp', 'proto\_ipcv', 'proto\_vmtp', 'proto\_mhrp']

This was found by inputting the entire data set that were binary data type. Then using Permutation Importance iteratively to remove features scoring negatively. See figure for an example of Test 1 negative perumutation importance.

**Hyperparameters:** I removed var\_smoothing from the parameters to test as the change in results was negligible.

**Cross Validation:** 10 fold cross validation with mean accuracy of 0.7744, standard deviation of accuracy of 0.0026, mean Precision score for ‘Normal’ of 0.9958 and standard deviation of Precision score for ‘Normal’ of 0.0013. The low standard deviation of cross validation proves the model is consistent across subsets and therefore robust. This suggests similar scores on unseen data if the test data follows similar trends.

**Best Results:**

Naive Bayes scoring on Training set:

Accuracy: 0.77, F1 score: 0.75, Precision score for 'Normal': 0.9963

Weighted Averaged Precision: 0.8, Weighted Average Recall: 0.77

Naive Bayes scoring on Test 1 set:

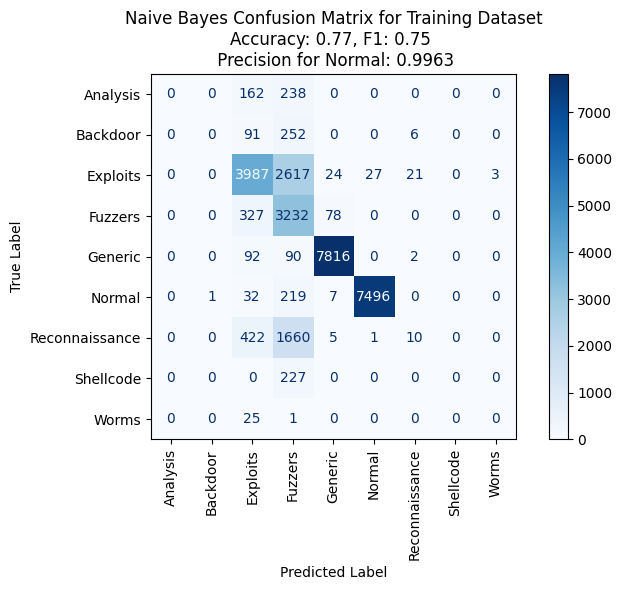
Accuracy: 0.65, F1 score: 0.67, Precision score for 'Normal': 0.9925

Weighted Averaged Precision: 0.81, Weighted Average Recall: 0.65

Naive Bayes scoring on Test 2 set:

Accuracy: 0.74, F1 score: 0.75, Precision score for 'Normal': 0.9939

Weighted Averaged Precision: 0.83, Weighted Average Recall: 0.74

A chart with numbers and a number of different colored squares

Description automatically generated with medium confidenceA chart with numbers and a number of different colored squares

Description automatically generated with medium confidence

* 1. **Decision Trees Classifier**

**What:** Decision Trees are a classification process using a set of hierarchical decisions. The split criterion is the condition on one feature that divides the subset of the data into two or more parts – essentially creating a fork in the tree. It will continue to do this until the leaf nodes only represent one class of the target variable or a stopping criterion has been set. Decision Trees can be pre-pruned or post-pruned. I chose to pre-prune with hyperparameters due to time limitations during this study.

**The Good:** Decision Trees typically work best with lower dimensionality, all data types used in this study, decreased number of outliers and large sample size. It does not require the data to be normally distributed, standardised or linear (it is non-parametric). (Anshul, 2024)

**The Bad:** Decision Trees however can be sensitive to noisy data, small variations and imbalanced data sets. Can be prone to overfitting.

**Feature Selection:**  To improve the Decision Tree model, I reduced the features to between 6-15 features to lower dimensionality, I ensured balanced classes. Through trial and error, my feature selection was also impacted by the results of feature importances metric. This showed me where the tree split on a variable and how important that variable is. I also used a pydotplus gragh = however it was too big to be interrogate.

These features were: ['attack\_cat', 'dur',  'spkts', 'rate', 'sttl', 'sjit', 'smean', 'dmean', 'service\_’, 'service\_dns', 'proto\_udp', 'proto\_swipe', 'proto\_sun-nd', 'proto\_mobile' ]

**Hyperparameters:** The parameters of the DecisionTreeClassifier() includes a choice of techniques on how to decide on the best split called “Criterion” using either quantifying the impurity of a split (Gini Index) or measuring information gain of a split (Entropy). The lower the Gini impurity, the better quality of the split and is calculated by (Karabiber, 2024). Another is “max\_depth” which sets a stopping criterion and without it can lead to unpruned trees (increasing computational complexity and overfitting). Splitter is another parameter that chooses either best or random for split strategy. Other finer tuning parameters were not used in this study. (Scikit Learn, 2024)

**Optimal Hyperparameters:** The optimal hyperparameters chosen during the Grid Search and Cross Validation were:

'criterion': 'gini', 'max\_depth': 30, 'splitter': 'best'

**Cross Validation:** 10 fold cross validation with mean accuracy of 0.91, standard deviation of accuracy of 0.0024, mean Precision score for ‘Normal’ of 0.9897 and standard deviation of Precision score for ‘Normal’ of 0.0019. The low standard deviation of cross validation proves the model is consistent across subsets and therefore robust. This suggests similar scores on unseen data if the test data follows similar trends.

**Best Results:**

Decision Tree scoring on Training set:

Accuracy: 0.91, F1 score: 0.91, Precision score for 'Normal': 0.9911

Weighted Averaged Precision: 0.91, Weighted Average Recall: 0.91

Decision Tree scoring on Test 1 set:

Accuracy: 0.7, F1 score: 0.71, Precision score for 'Normal': 0.9788

Weighted Averaged Precision: 0.83, Weighted Average Recall: 0.7

Decision Tree scoring on Test 2 set:

Accuracy: 0.82, F1 score: 0.83, Precision score for 'Normal': 0.9841

Weighted Averaged Precision: 0.87, Weighted Average Recall: 0.82

A graph of a tree confusion matrix

Description automatically generatedA chart with numbers and a number of different colored squares

Description automatically generated with medium confidenceA graph of a tree

Description automatically generated with medium confidence

* 1. **Random Forest Classifier**

**What:** Random Forest is an ensemble method that uses a subset of variables to build decision trees to improve stability, prediction accuracy and avoid overfitting. It also works on the “Bagging” principle which involves choosing a random subset, then row sampling with replacement, training a model with this Bootstrapped sample then combining the results of all models through majority voting and then generating an output (known as “Aggregation”) (Sruthi, 2024).

**The Good:** It can handle noisy data, it is robust, it doesn’t need standardised data to work

**The Bad:** It is slower than Decision Trees

**Feature Selection:**  This classifier has feature importance tool that I also used on Decision Trees. This helped select the best features.

These features were: [ 'dur', 'spkts', 'rate', 'sttl', 'sjit', 'smean', 'dmean', 'sttl\_bin', 'dttl\_bin', 'service\_-', 'service\_dns',

 'service\_smtp', 'proto\_udp', 'proto\_unas',]

**Hyperparameters:** The parameters I tuned were n\_estimators which is the number of trees to build before it takes the average, max\_depth which sets a stopping criterion and without it can lead to unpruned trees (increasing computational complexity and overfitting), min\_sample\_leaf determines the number of leaves required to split an internal node, and criterion which determines how to split the node in each tree (Scikit Learn, 2024).

**Optimal Hyperparameters:** The optimal hyperparameters chosen during the Grid Search and Cross Validation were:

'criterion': 'gini', 'max\_depth': 30, 'min\_samples\_leaf': 10, 'n\_estimators': 200,

**Cross Validation:** 10 fold cross validation with mean accuracy of 0.91, standard deviation of accuracy of 0.0019, mean Precision score for ‘Normal’ of 0.9992 and standard deviation of Precision score for ‘Normal’ of 0.0004. The low standard deviation of cross validation proves the model is consistent across subsets and therefore robust. This suggests similar scores on unseen data if the test data follows similar trends.

**Best Results:**

Random Forest scoring on Training set:

Accuracy: 0.91, F1 score: 0.9, Precision score for 'Normal': 0.9995

Weighted Averaged Precision: 0.917, Weighted Average Recall: 0.91

Random Forest scoring on Test 1 set:

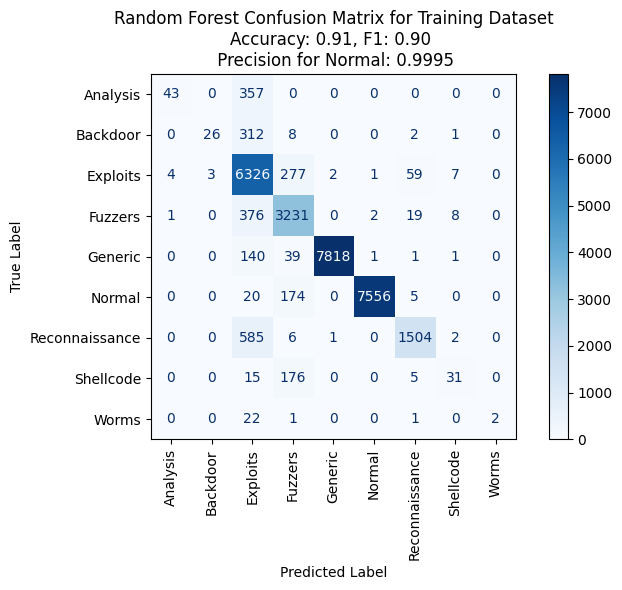
Accuracy: 0.7, F1 score: 0.7, Precision score for 'Normal': 0.9992

Weighted Averaged Precision: 0.84, Weighted Average Recall: 0.7

Random Forest scoring on Test 2 set:

Accuracy: 0.82, F1 score: 0.82, Precision score for 'Normal': 0.9995

Weighted Averaged Precision: 0.87, Weighted Average Recall: 0.82

A graph of different numbers and a number of objects

Description automatically generated with medium confidenceA graph of different numbers and a number of different numbers

Description automatically generated with medium confidence

* 1. **Histogram Gradient Boosting Classifier**

**What:** Gradient Boosting Classifier is also an ensemble classifier. It builds models one at a time and each new model corrects the mistakes of the previous model. It combines the weak learners (shallow decision trees in this case) to build a more powerful model (Aarshay, 2024).

Due to the computational power limitations, I altered this Gradient Boosting Classifier to use HistGradientBoostingClassifier(), this involves binning the continuous data type features into discrete intervals before determining tree splits. This significantly increased the speed of the algorithm but reduced the precision (Scikit Learn, 2024).

**The Good:** Handles diverse data types, robust against overfitting, handles non-linear and non-normal data better.

**The Bad:** Training time is slow, hence why I am using a faster but less precise version and it is less interpretable although I am still relying on feature importance.

**Feature Selection:**  I’m using a Permutation feature importance tool to assess which features bring the most value.

These features were: ['spkts', 'sttl', 'smean', 'dmean','dttl\_bin', 'service\_-', 'service\_dns', 'proto\_udp']

**Hyperparameters:** The parameters I tuned were max\_iter which the number of iterations of the boosting process multiplied by the number of classes (9), learning\_rate which controls how much change in the models response to estimated error and max\_depth which sets the stopping criterion. Class Weight determines if to should take into account the imbalance of the classes. (Scikit Learn, 2024).

**Optimal Hyperparameters:** The optimal hyperparameters chosen during the Grid Search and Cross Validation were:

'class\_weight': None, 'learning\_rate': 0.01, 'max\_depth': 8, 'max\_iter': 200

**Cross Validation:** 10 fold cross validation with mean accuracy of 0.91, standard deviation of accuracy of 0.0024, mean Precision score for ‘Normal’ of 0.9992 and standard deviation of Precision score for ‘Normal’ of 0.0001. The low standard deviation of cross validation proves the model is consistent across subsets and therefore robust. This suggests similar scores on unseen data if the test data follows similar trends.

**Best Results:**

Gradient Boost scoring on Training set:

Accuracy: 0.91, F1 score: 0.9, Precision score for 'Normal': 0.9992

Weighted Averaged Precision: 0.91, Weighted Average Recall: 0.91

Gradient Boost scoring on Test 1 set:

Accuracy: 0.7, F1 score: 0.7, Precision score for 'Normal': 0.9973

Weighted Averaged Precision: 0.84, Weighted Average Recall: 0.7

Gradient Boost scoring on Test 2 set:

Accuracy: 0.81, F1 score: 0.82, Precision score for 'Normal': 0.9979

Weighted Averaged Precision: 0.87, Weighted Average Recall: 0.81

**A graph of a number of data

Description automatically generated with medium confidenceA graph of a number of data

Description automatically generated with medium confidenceA graph of a number of data

Description automatically generated with medium confidence**

1. **Best Predictions**

The top two predictions by accuracy for each test data set are printed to the terminal, displaying the accuracy score and the precision score for 'Normal'. Additionally, a CSV file is exported for each test data – predict\_1.csv and predict\_2.csv. These have got attack\_cat encoded to numbers. If there is a requirement for them to not be encoded, there are unencoded versions in the Saved\_Outputs folder.

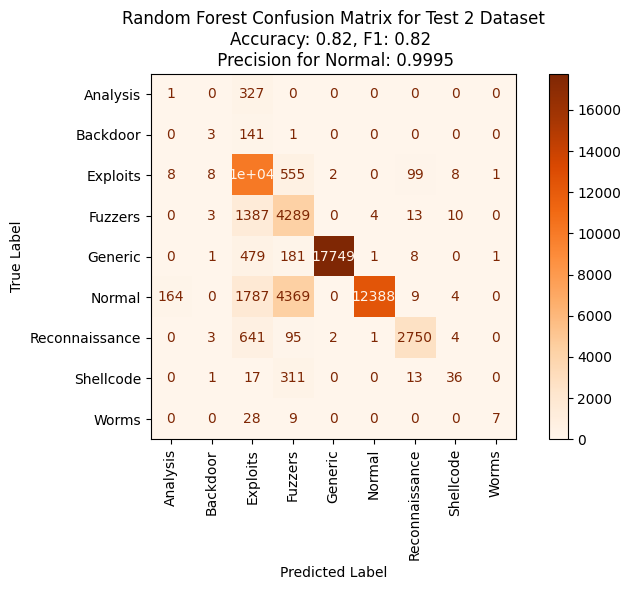
\*The encoded dictionary is:

"Analysis": 1, "Backdoor": 2, "Exploits": 3, "Fuzzers": 4, "Generic": 5, "Normal": 6, "Reconnaissance": 7,

"Shellcode": 8, "Worms": 9

The first column contains an index number starting from 1, the second column lists the top predictions, and the third column contains the second top predictions in each file. The top predictions are:

Test Data Set 1 - Top Two Accuracy Predictions:



**Model: DT\_t1, Accuracy: 0.7, Precision Score for 'Normal': 0.9788**

**Model: RF\_t1, Accuracy: 0.7, Precision Score for 'Normal': 0.9992**

Test Data Set 2 - Top Two Accuracy Predictions:

**Model: DT\_t2, Accuracy: 0.82, Precision Score for 'Normal': 0.9841**

**Model: RF\_t2, Accuracy: 0.82, Precision Score for 'Normal': 0.9995**

Random Forest was the best classifier in this study for accuracy but also for each test set misclassified only 7 (for test set 1) and 6 (for test set 2) attacks as normal (as shown in the Confusion Matrix). This is out of 59809 instances. This was significantly better than the other classifiers and due to the increasing pressure for organisations to protect their data and focus on their cyber security, I focused on the Precision for ‘Normal’ metric I made that calculates how many instances were predicted Normal (not a malicious attack) with how many were actually ‘Normal’. As a future project, I would want to continue this effort focusing on getting this number to 0.

With more time, I would like to try different ensemble classifiers and conduct some more feature engineering to improve accuracy of the classifiers.

**References**

Aarshay (2024, Oct 15). Complete Machine Learning Guide to Parameter Tuning in Gradient Boosting (GBM) in Python. *Analytics Vidhya.* https://www.analyticsvidhya.com/blog/2016/02/complete-guide-parameter-tuning-gradient-boosting-gbm-python/

Aggarwal, C., C. (2015). *Data Mining: The Textbook.* Springer. <https://doi.org/DOI10.1007/978-3-319-14142-8>

Anshul (2024, Sep 10). What is Decision Tree? *Analytics Vidhya.* <https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/>

Ali, M. (2023, Jan 11). Understanding Data Drift and Model Drift: Drift Detection in Python. *Data Camp.* https://www.datacamp.com/tutorial/understanding-data-drift-model-drift

Chugani, V. (2024, Jul 17). What is Manhattan Distance. *Data Camp.* https://www.datacamp.com/tutorial/manhattan-distance

Dreyfus, M. (2022, Oct 22). *Tougher Penalties for Serious Data Breaches* [Press Release]. Attorney-General’s Portfolio. <https://ministers.ag.gov.au/media-centre/tougher-penalties-serious-data-breaches-22-10-2022>

Evidently AI. (2024). *Classification Metrics Guide: Accuracy, Precision, and Recall in Multi-class Classification.* Evidently AI. <https://www.evidentlyai.com/classification-metrics/multi-class-metrics#:~:text=To%20calculate%20the%20recall%2C%20we,predictions%20(true%20and%20false)>

Galli, S. (2024, Jan, 11). Understanding Permutation Feature Importance for Model Interpretation. *Train In Data.* <https://www.blog.trainindata.com/permutation-feature-importance/#:~:text=Permutation%20feature%20importance%20is%20a%20metric%20obtained%20by%20randomly%20shuffling,different%20values%20for%20feature%20importance>.

Jayaswal V. (2020, Nov 22). Laplace Smoothing in Naïve Bayes Algorithm. *Towards Data Science.* <https://towardsdatascience.com/laplace-smoothing-in-na%C3%AFve-bayes-algorithm-9c237a8bdece>

Kundu, R. (2022, Dec 16). F1 Score in Machine Learning: Intro & Calculation. *V7 Labs.* https://www.v7labs.com/blog/f1-score-guide

Karabiber, F. (2024). *Gini Impurity.* Learn Data Science. <https://www.learndatasci.com/glossary/gini-impurity/>

Scikit Learn. (2024). *KNeighborsClassifier*. Scikit Learn. <https://scikit-learn.org/dev/modules/generated/sklearn.neighbors.KNeighborsClassifier.html>

Scikit Learn. (2024). *DecisionTreeClassifier*. Scikit Learn. https://scikit-learn.org/dev/modules/generated/sklearn.tree.DecisionTreeClassifier.html

Scikit Learn. (2024). *RandomForestClassifier*. Scikit Learn. https://scikit-learn.org/dev/modules/generated/sklearn.ensemble.RandomForestClassifier.html

Scikit Learn. (2024). *HistGradientBoostingClassifier*. Scikit Learn. https://scikit-learn.org/dev/modules/generated/sklearn.ensemble.HistGradientBoostingClassifier.html

Scikit Learn. (2024). *1.9.4 Bernoulli Naïve Bayes*. Scikit Learn. <https://scikit-learn.org/1.5/modules/naive_bayes.html>

Scikit Learn. (2024). *Permutation Importance vs Random Forest Feature Importance (MDI)*. Scikit Learn https://scikit-learn.org/stable/auto\_examples/inspection/plot\_permutation\_importance.html#sphx-glr-auto-examples-inspection-plot-permutation-importance-py

Scikit Learn. (2024). *GridSearchCV.* Scikit Learn. https://scikit-learn.org/dev/modules/generated/sklearn.model\_selection.GridSearchCV.html

Sruthi (2024, Oct 8). Understanding Random Forest Algorithm with Examples. *Analytics Vidhya.* <https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/>

Swastik. (2024, Oct 16). SMOTE for Imbalanced Classification with Python. *Analytics Vidhya.* https://www.analyticsvidhya.com/blog/2020/10/overcoming-class-imbalance-using-smote-techniques/