Malicious Website Data Analysis

Milan Mitrovic | s4663796

Contents

Executive Summary

Malicious websites pose a significant threat to both individuals and organizations, leading to financial losses, security breaches, and compromised personal information if not addressed promptly. Early detection and prediction of malicious websites is crucial for mitigating the risks associated with cyberattacks and safeguarding users from potential harm. This project is centred on malicious website prediction and aims to identify the most effective supervised machine learning algorithm for this critical task. We conducted a comprehensive analysis, utilizing multiple supervised machine learning algorithms to predict the likelihood of a website being malicious based on both given and unseen data. This was accomplished by conducting an in-depth data analysis of malicious websites by leveraging advanced data analytical tools and data mining methods to uncover key hidden patterns, trends, insights, and correlations between malicious and benign website data. The primary aim was to determine the best algorithm for accurately detecting malicious websites, thereby enhancing cybersecurity defences and reducing vulnerability to cyberattacks which could have significant implications for cybersecurity and user protection. Specifically, the data mining method of classification was employed to build predictive models to classify websites as malicious or benign, aiming to enhance cybersecurity strategies and protect against online threats. Ultimately, after thorough testing, we found the Random Forest Model to be the most effective performance wise and highly recommend it for malicious website detection; it possesses the best evaluation metrics out of all models, however what sealed the deal was its specificity score – it was the highest and specificity is the most important metric as it represents the evaluation for correctly detecting malicious websites. This evaluation ensures that our model detects as many malicious websites as possible, minimizing the risk of missing threats, thereby safeguarding users from cyberattacks/malicious websites very effectively. The limitations are the results, the results are overall excellent, however it could be propelled further, especially for specificity, with more extensive experimenting such as oversampling the minority class or vigorous hyperparameter tuning.

Introduction

The proliferation of malicious websites represents a growing threat to individuals and organizations across the world, with the potential to cause severe financial losses, data breaches, compromised security, and significant disruptions to digital infrastructure if not identified and mitigated promptly. Identifying malicious websites early is paramount for protecting users and infrastructure. According to a report by the Center for Strategic and International Studies, cybercrime, including malicious websites, costs the global economy over $600 billion annually, and a survey by the Anti-Phishing Working Group found that over 60% of internet users have encountered some form of online scam, highlighting the urgent need for accurate detection methods (*Center for Strategic and International Studies*, 2024; *Anti-Phishing Working Group*, 2024). The solution: Supervised machine learning algorithms, specifically classification algorithms. Classification algorithms are particularly well-suited for identifying malicious websites as they can analyse vast amounts of website data; playing a crucial role in malicious website detection by leveraging malicious and benign website data to identify key patterns, trends, characteristics, and risk factors associated with malicious and non-malicious websites; leading to accurate predictions of malicious or benign websites (classification) which subsequently leads to early detection – crucial for mitigating the risks of cyberattacks and protecting users from potential threats. By conducting an in-depth data analysis, using advanced data analytics and classification techniques, this project seeks to uncover hidden patterns, trends, insights, and correlations between malicious and benign datasets, specifically in the provided malicious_and_benign_websites dataset – classification will be employed on this data. By doing so, we mainly aim to develop predictive models that can classify malicious and benign websites with a high degree of accuracy and determine which model is the most accurate. Predictive models can provide real time threat assessments, providing valuable tools for cybersecurity professionals to intervene early and implement protective measures. This proactive approach can significantly reduce the likelihood of cyberattacks and the associated risks to users and organizations. The outcomes of this project have the potential to significantly improve online safety, offering a proactive defense mechanism against the ever-evolving landscape of cyber threats.

We'll evaluate three distinct classification algorithms to determine the most effective one for diagnosing websites correctly – malicious or benign.

Classification Algorithms:

1. **Decision Tree**: A decision tree is a supervised machine learning algorithm, which is utilized for both classification and regression tasks. It constructs a tree-like structure to partition the feature space into regions, each corresponding to a specific class label. At each node of the tree, the algorithm selects the feature that best separates the data into the purest possible subsets based on some criterion, such as information gain. This process continues until a stopping criterion is met; reaching a maximum depth, minimum number of samples per leaf, or when further splits no longer improve the purity of the subsets. In the context of classifying websites as either malicious or benign, a decision tree evaluates a website based on its features, such as URL length, WHOIS information, and server details. During classification, the website's feature values are used to traverse the tree from the root node to a leaf node. At each internal node, the tree applies decision rules to determine the path taken. Once a leaf node is reached, the tree assigns the website a class label (malicious or benign) based on the majority class of the instances that fall into that leaf node. This approach enables the decision tree to categorize websites by analyzing their attributes and determining the most likely class label through the hierarchical decision process.

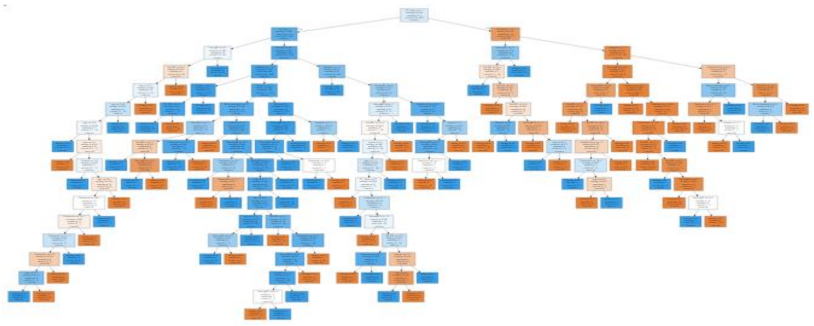


Figure 1. From A Guide to any Classification Problem Durgance Guar, 2022

2. **K-Nearest Neighbor (KNN)**: KNN is a supervised machine learning algorithm that relies on proximity to make classifications about the grouping of a particular data point. It achieves this by identifying the k-nearest neighbors among future examples. In classification tasks, KNN determines the category of a data point based on the category of its closest neighbor. For instance, if K equals 1, the data point would be assigned to the class of its closest neighbor. The value of K is determined by a majority vote among its neighbors. Typically, the optimal value of K is usually found to be the square root of the total number of samples (N). In essence, KNN operates on the assumption that similar entities are located close to each other -- calculating the distance between data points with distance metrics such as Euclidean, Manhattan, and Minkowski distances. In this case, KNN would classify the type of website (malicious/benign) by comparing it to its K nearest neighbors by distance in the training data and making a classification based on the majority label of these neighbors.

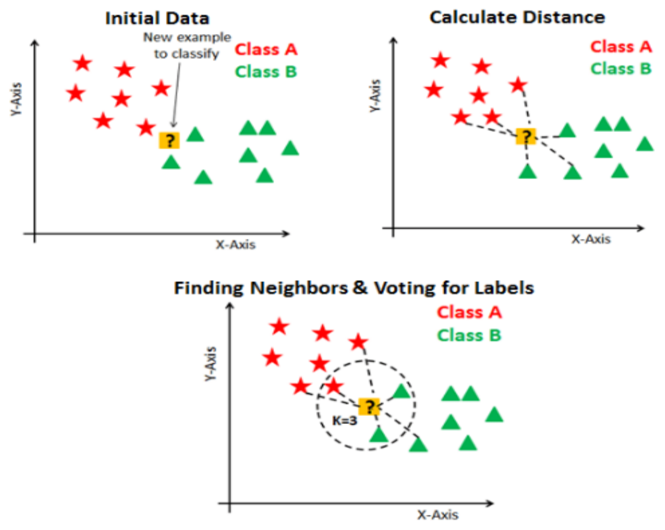


Figure 2. From A Guide to any Classification Problem Durgance Guar, 2022

3. **Random Forest Classifier** – A Random Forest Classifier is a supervised machine learning algorithm used for classification tasks. It improves upon individual decision trees by constructing multiple trees and making predictions based on the collective results of these trees. Each tree in the forest is built using a different bootstrap sample of the training data, and at each split, only a random subset of features is considered. This approach reduces overfitting and variance, enhancing the model's accuracy and robustness. In this situation, for classifying a website as either malicious or benign, the Random Forest Classifier evaluates the website based on its features, aggregates predictions from all decision trees in the forest, and assigns the most common class label. This method effectively handles complex datasets and can provide insights into feature importance, making it a powerful tool for classification tasks.

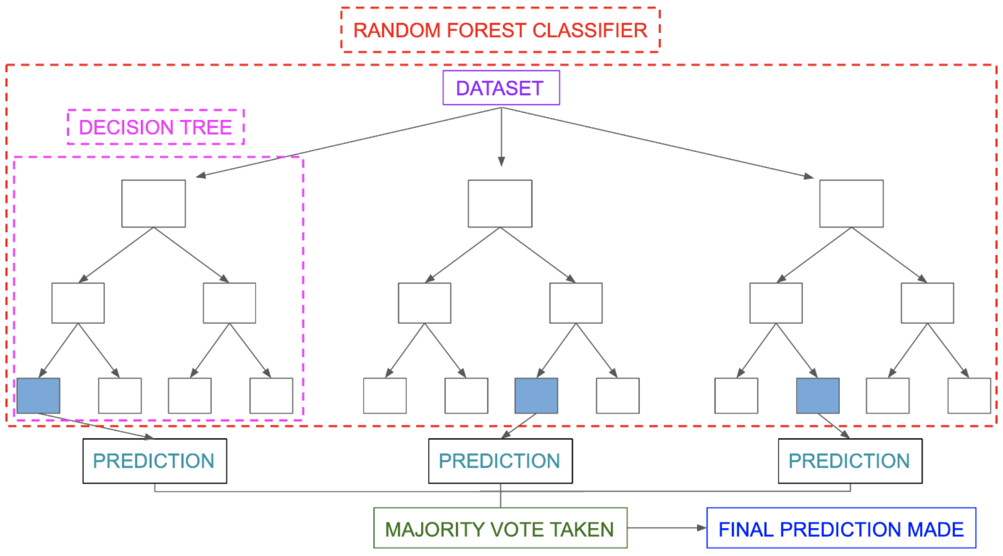


Figure 3. From A Guide to any Classification Problem Durgance Guar, 2022

We will measure the performance of our models by using these evaluation metrics:

- **Accuracy:** Accuracy is calculated as the proportion of accurate predictions made by the model relative to the total number of input samples, expressed as a percentage. This is calculated by the following formula:

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$$

Basically, the total correct answers are divided by the total number of entries. It's an efficient way to measure the predictive success of the model.

- **Precision/Positive Predictive Value:** Precision/Positive Predictive Value evaluates a model's effectiveness by indicating the proportion of positive predictions made by the model that are correct. It is computed as the ratio of true positive predictions to the sum of true positive and false positive predictions.

  $$Precision = \frac{TP}{TP+FP}$$

- **Recall/Sensitivity:** Recall/Sensitivity represents the proportion of positive predictions made by the model relative to all samples that should have been identified as positive. It measures the accuracy of identifying actual positive classes. This metric is computed by dividing the number of true positive predictions by the sum of true positive and false negative predictions. This metric is quite important as it represents the evaluation for correctly detecting benign websites. This evaluation ensures that benign websites are not incorrectly flagged as malicious, reducing false alarms.

  $$Recall = \frac{TP}{TP+FN}$$

- **Specificity:** Specificity represents the proportion of true negative predictions made by the model relative to all samples that should have been identified as negative. It measures the accuracy of identifying actual negative classes. This metric is computed by dividing the number of true negative predictions by the sum of true negative and false positive predictions. This metric is particularly crucial in our case as it represents the evaluation for correctly detecting malicious websites. This evaluation ensures that our model detects as many malicious websites as possible, minimizing the risk of missing threats.

  $$Specificity = \frac{TN}{TN+FP}$$

- **Balanced Accuracy**: Balanced Accuracy provides a balanced measure of the model's performance by averaging the accuracy of identifying both positive and negative classes. It is particularly useful when dealing with imbalanced datasets, ensuring that the model performs well across both classes. This metric is computed as the average of Recall (Sensitivity) and Specificity. It provides an overall measure that balances the detection of both classes, making it an ideal metric for measuring the detection in both malicious and benign websites correctly.

$$Balanced\ Accuracy = \frac{Recall + Specificity}{2}$$

- **Confusion Matrix** – A confusion matrix is an N X N matrix, where N represents the number of classes being predicted. It's a tool for summarizing the performance of a classification algorithm; it gives us a clear picture of the specific classification models performance and the types of errors produced by the model. The summary provides correct and incorrect predictions broken down by each category, represented by a matrix. Almost all performance metrics are based on the confusion matrix and the numbers inside it.

- **Confusion Matrix Elements:**

1. **True Positives (TP):** The number of observations that are actually positive and are correctly predicted as positive by the model. In other words, the model accurately predicted 'benign'.

2. **True Negatives (TN):** The number of observations that are actually negative and are correctly predicted as negative by the model. In this context, the model accurately predicted 'malicious'.

3. **False Positives (FP):** The number of observations that are actually negative but are incorrectly classified as positive by the model – also known as a Type I error. In this case, the model incorrectly predicted 'benign' when it was 'malicious'.

4. **False Negatives (FN):** The number of observations that are actually positive but are incorrectly classified as negative by the model – also known as a Type II error. The model incorrectly predicted 'malicious' when it was 'benign'.

TN and FP are crucial as they directly affect whether a malicious website is correctly/incorrectly detected. TP and FN are of lesser importance since they relate to cases where the website is benign – no risk of cyberattacks.

Literature Review

The rapid expansion of the internet has significantly increased the number of websites that users visit every day. However, this growth has also led to a surge in malicious websites aimed at exploiting or harming users. Detecting these websites is vital for cybersecurity. This literature review examines recent advancements in malicious website detection, emphasizing machine learning techniques and contrasting them with traditional methods. The review also argues that machine learning algorithms outperform human detection due to their scalability, speed, adaptability, pattern recognition, and consistency.

Machine learning approach:

Machine learning has become essential in identifying malicious websites because of its capacity to manage large datasets, adapt to new threats, and detect patterns that may not be obvious to human analysts. Several contemporary studies have demonstrated the effectiveness of various machine learning algorithms in this field. One widely used technique is Random Forests. This algorithm has proven effective in classifying websites by analyzing features like URL length, domain age, and specific keywords. For example, a study featured in the *IEEE Transactions on Information Forensics and Security* showed that Random Forests could achieve high accuracy in detecting phishing sites by examining both URL-based and content-based features extracted from web pages.

Other approaches:

Signature-Based Detection: This conventional method involves checking the characteristics of a website against a database of known malicious signatures. If a match is found, the website is marked as malicious. This technique is commonly employed in antivirus software and intrusion detection systems. However, it is less effective when the signature database is not updated or when dealing with new or altered threats such as zero-day attacks.

Heuristic Analysis: Heuristic analysis seeks to identify potentially malicious websites based on their behavior rather than specific signatures. This approach involves evaluating actions a website might perform, such as redirecting users to unfamiliar pages or attempting unauthorized downloads. While more adaptable than signature-based detection, heuristic methods can generate false positives and need constant updates to remain effective against new threats.

Justification for machine learning approach over human readers:

Although human expertise is valuable for analyzing and detecting malicious websites, machine learning algorithms offer several distinct advantages that make them more efficient and effective in today's digital landscape:

- **Scalability:** Human readers are constrained by time and cognitive limits, which hinders their ability to process large volumes of data. In contrast, machine learning algorithms can handle vast datasets simultaneously, making them ideal for analysing the extensive scale of web traffic and detecting potential threats in real-time.

- **Adaptability**: Machine learning models can be continuously updated with new data, allowing them to adapt to emerging threats. This flexibility ensures that the detection system remains effective against new types of attacks, such as zero-day exploits. Human readers, while capable of learning, cannot update their knowledge as quickly or efficiently as a machine learning model that is constantly being retrained.

- **Pattern Recognition**: Machine learning algorithms are highly skilled at identifying complex patterns in data that might be too subtle or intricate for human readers to detect consistently. For example, sophisticated phishing websites or malicious URLs might use obscure techniques that are difficult for humans to spot but can be identified by algorithms trained on large datasets. This ability to recognize nuanced patterns makes machine learning models more dependable in detecting threats that could be overlooked by human analysts.

- **Speed:** Machine learning models can analyse websites and identify malicious patterns much faster than human readers. These systems can classify and scan websites in milliseconds, offering immediate protection to users. Human analysis, even with the aid of automated tools, cannot match this speed, particularly when monitoring global web traffic on a large scale.

- **Consistency:** Human analysis is prone to variability due to factors like fatigue, cognitive biases, and differing levels of expertise. Machine learning algorithms, however, provide consistent and objective analysis, minimizing the risk of errors caused by human factors. This consistency is crucial for maintaining high security, as even a minor oversight by a human reader could result in a security breach.

Dataset Description

Feature	Sample	Description
URL	M0_109	Unique identifier of a website
URL_LENGTH	16	Number of characters in the URL
NUMBER_SPECIAL_CHARACTERS	7	Number of special characters in the URL
CHARSET	iso-8859-1	Character encoding used by the website
SERVER	nginx	The OS of the server retrieved from the packet response
CONTENT_LENGTH	263	Content size of the HTTP header
WHOIS_COUNTRY	US	Country of the server
WHOIS_STATEPRO	AK	State of the country of the server
WHOIS_REGDATE	10/10/2015 18:21	Server date & time
WHOIS_UPDATED_DATE	12/09/2013 0:45	Last update of the server
TCP_CONVERSATION_EXCHANGE	7	Number of TCP packets swapped between the server and the honeypot client
DIST_REMOTE_TCP_PORT	0	Number of ports noted and different to the TCP
REMOTE_IPS	2	Number of IP's connected to the honeypot
APP_BYTES	700	Number of bytes transferred
SOURCE_APP_PACKETS	9	Number of packets sent from honeypot to server
REMOTE_APP_PACKETS	10	Number of packets received from the server
SOURCE_APP_BYTES	1153	Number of bytes sent from honeypot to server
REMOTE_APP_BYTES	823	Number of bytes received from the server
APP_PACKETS	9	Number of IP packets generated during communication between honeypot and server
DNS_QUERY_TIMES	2	Number of DNS packets generated during

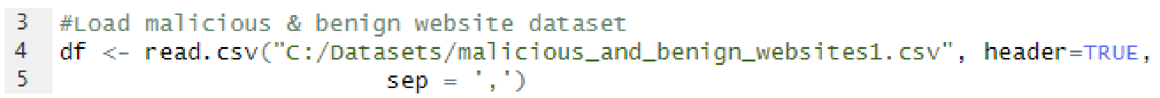
		communication between honeypot and server
Type	1	Indicates whether website is malicious (1) or benign (0)

This dataset: *malicious_and_benign_websites1.csv* which is used in this project for malicious website detection was provided for. The dataset encompasses comprehensive records of website interactions collected from honeypots and real-world online traffic, including detailed information on website characteristics, network traffic, server information, and domain registration details. This dataset contains every single possible feature that holds merit, when it comes to identifying and differentiating between malicious and benign websites. Currently, this dataset contains 21 features which can be analyzed and used to predict the nature of websites (malicious or benign) by utilizing supervised machine learning algorithms. The feature Type determines whether a particular website is malicious or benign; this is the categorical label we are going to predict – classification. Classification algorithms will be employed to fulfill this task by training the algorithms with this data, teaching them how to differentiate the values for the website type. Once trained, the algorithms can be applied to new data that doesn't include a Type feature, simulating real-world scenarios of predicting our target feature (dependent variable) Type in unseen website interactions, enabling the forecast of whether a website is malicious or benign in various online environments – subsequently providing early detection of malicious websites.

Data Cleaning

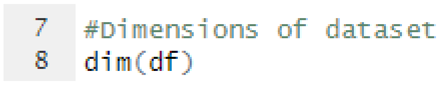
Data cleaning is a crucial step in the data analytics process and must be conducted thoroughly before proceeding to subsequent stages. This is because unclean data can result in misleading results and inaccurate conclusions.

First, I must import the dataset:

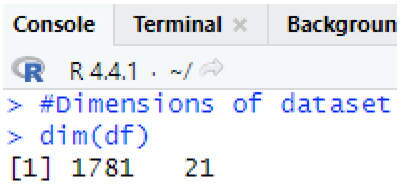


```r
#Load malicious & benign website dataset
df <- read.csv("C:/Datasets/malicious_and_benign_websites1.csv", header=TRUE,
                              sep = ',')
```

I load the dataset by utilizing the read.csv() function to read the dataset, specifying the parameters header=TRUE and sep = ','. The header=TRUE parameter tells R that the first row of the file contains the column names, while sep = ',' specifies that commas are used as the delimiter to separate values in the dataset – necessary as this is the layout of *malicious_and_benign_websites1.csv*. I assigned the dataset to the variable 'df'.



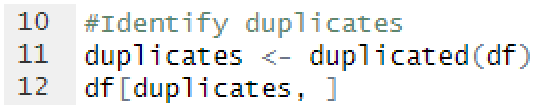
```
#Dimensions of dataset
dim(df)
```



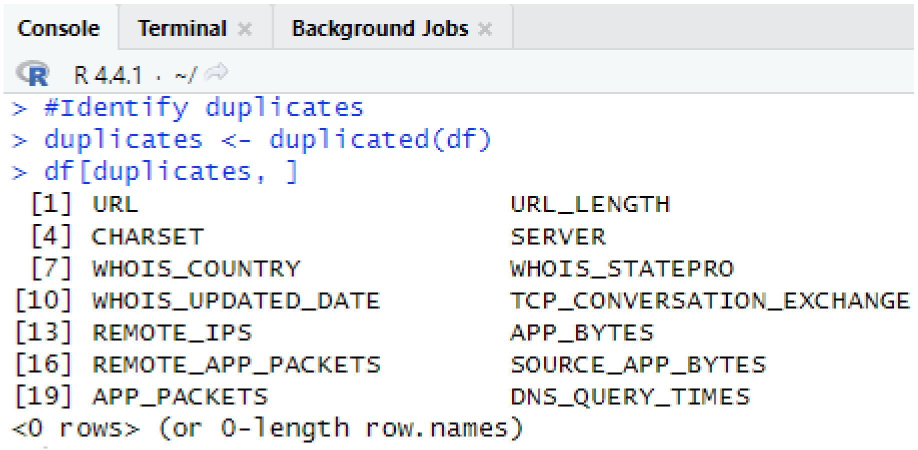
```
> #Dimensions of dataset
> dim(df)
[1] 1781   21
```

Currently, the dataset dimensions before conducting data cleaning: 1781 rows (observations) and 21 columns (variables). This is verified by the dim() function; which works by returning a vector with the number of observations and variables in a data frame concisely.

I will start with identifying duplicates and removing them if they are present:



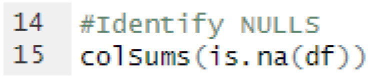
```
#Identify duplicates
duplicates <- duplicated(df)
df[duplicates, ]
```



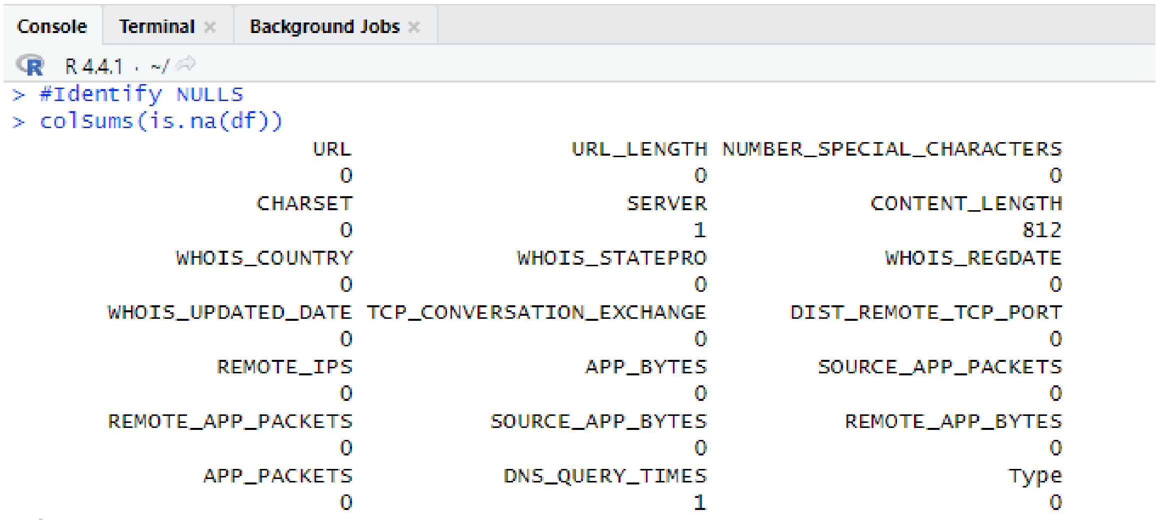
```
> #Identify duplicates
> duplicates <- duplicated(df)
> df[duplicates, ]
 [1] URL                         URL_LENGTH
 [4] CHARSET                     SERVER
 [7] WHOIS_COUNTRY               WHOIS_STATEPRO
[10] WHOIS_UPDATED_DATE          TCP_CONVERSATION_EXCHANGE
[13] REMOTE_IPS                  APP_BYTES
[16] REMOTE_APP_PACKETS          SOURCE_APP_BYTES
[19] APP_PACKETS                 DNS_QUERY_TIMES
<0 rows> (or 0-length row.names)
```

There are no duplicates present in the dataset. This is confirmed by utilizing the duplicated() function, which works by returning all duplicate rows within the dataset.

Now, I am going to analyse the NULLS/missing values within the dataset and subsequently deal with them:



```
#Identify NULLS
colSums(is.na(df))
```

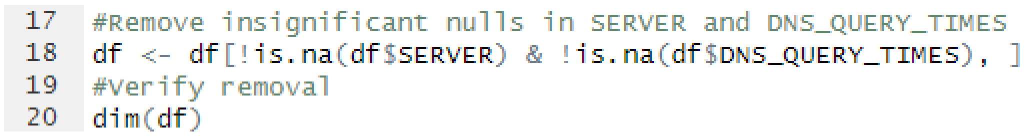


```
> #Identify NULLS
> colSums(is.na(df))
                      URL                URL_LENGTH NUMBER_SPECIAL_CHARACTERS
                        0                         0                         0
                  CHARSET                    SERVER            CONTENT_LENGTH
                        0                         1                       812
            WHOIS_COUNTRY             WHOIS_STATEPRO            WHOIS_REGDATE
                        0                         0                         0
       WHOIS_UPDATED_DATE TCP_CONVERSATION_EXCHANGE      DIST_REMOTE_TCP_PORT
                        0                         0                         0
               REMOTE_IPS                 APP_BYTES        SOURCE_APP_PACKETS
                        0                         0                         0
       REMOTE_APP_PACKETS          SOURCE_APP_BYTES          REMOTE_APP_BYTES
                        0                         0                         0
              APP_PACKETS           DNS_QUERY_TIMES                      Type
                        0                         1                         0
```

By utilizing the colSums and is.na() functions, we obtain the null values in every column within the dataset. Analyzing the output above, we can observe there are null values present in multiple features. Specifically in these 3 features: SERVER and DNS_QUERY_TIMES both contain 1 null value – insignificant. In contrast, CONTENT_LENGTH possesses 812 null values which is quite significant.

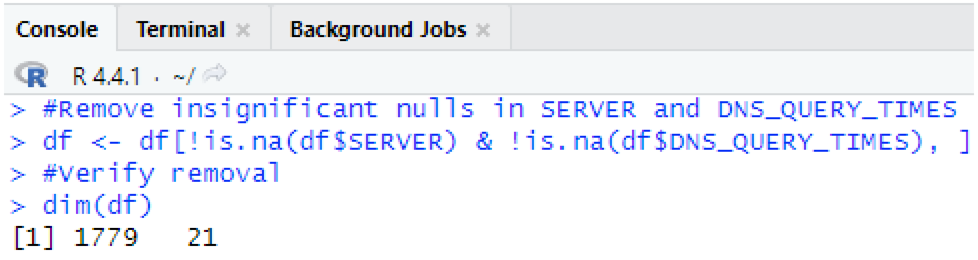
I deal with the insignificant and significant null values accordingly:

- **Insignificant null values**:



```
#Remove insignificant nulls in SERVER and DNS_QUERY_TIMES
df <- df[!is.na(df$SERVER) & !is.na(df$DNS_QUERY_TIMES), ]
#Verify removal
dim(df)
```

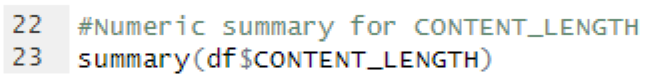
- I handle the insignificant nulls in SERVER and DNS_QUERY_TIMES by removing them. Exactly 2 null containing rows have been dropped; this approach is the most efficient as it's only 2 rows removed in total, which equates to less than 0.2% of the entire dataset: insignificant, maintaining data integrity and quality – removing nulls. The removal was achieved by using the is.na() function, specifically with the NOT operator (!); essentially working as a filter to provide a data frame free of nulls in these specific columns.



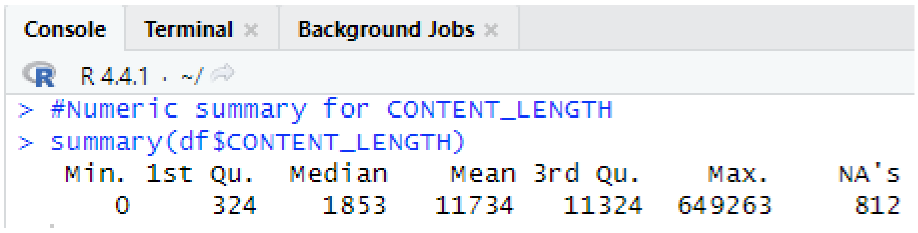
```
> #Remove insignificant nulls in SERVER and DNS_QUERY_TIMES
> df <- df[!is.na(df$SERVER) & !is.na(df$DNS_QUERY_TIMES), ]
> #Verify removal
> dim(df)
[1] 1779   21
```

- Clearly, the 2 null values have been removed from the dataset.

- **Significant null values:**

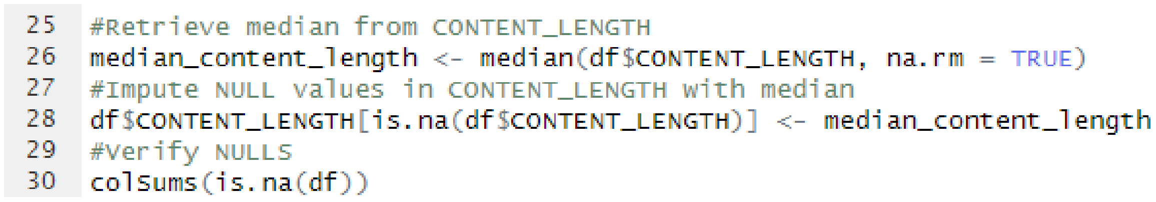


```
#Numeric summary for CONTENT_LENGTH
summary(df$CONTENT_LENGTH)
```



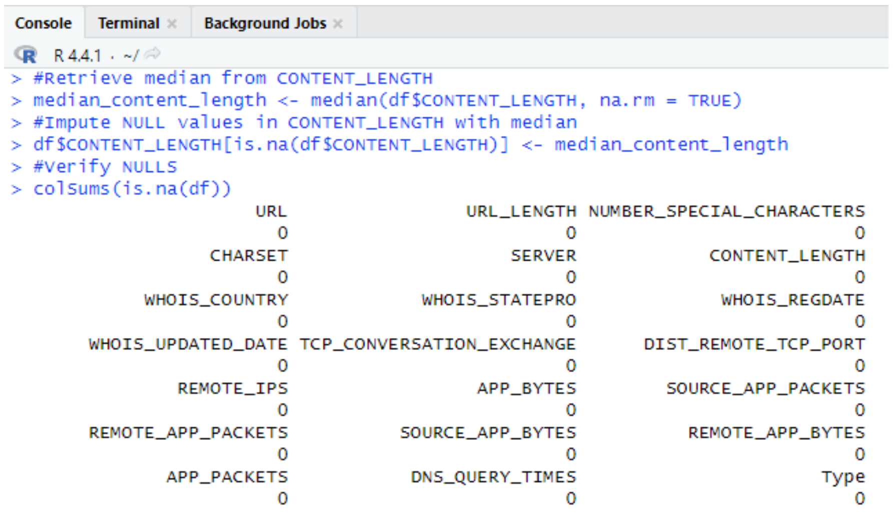
```
> #Numeric summary for CONTENT_LENGTH
> summary(df$CONTENT_LENGTH)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
      0     324    1853   11734   11324  649263     812 
```

- Since the null values in CONTENT_LENGTH represent a significant proportion of the dataset I need to impute them with a central tendency statistic. I use the summary() function to provide a concise summary of several key statistics in CONTENT_LENGTH to aid in my decision of imputation. By observing the results, imputation with median is the most optimal option here as the distribution is skewed – evident by the disparity between median and mean. Median is a robust measure against skewed distribution and outliers, which appears to be the problem here.



```
#Retrieve median from CONTENT_LENGTH
median_content_length <- median(df$CONTENT_LENGTH, na.rm = TRUE)
#Impute NULL values in CONTENT_LENGTH with median
df$CONTENT_LENGTH[is.na(df$CONTENT_LENGTH)] <- median_content_length
#Verify NULLS
colSums(is.na(df))
```

- I replace the 812 null values in CONTENT_LENGTH by imputing them with it's median, as justified previously, median is generally the best option when dealing with skewed data. It provides a measure of central tendency that is not overly influenced by the extreme values (outliers); ensuring the imputed values are not unduly affected by outliers, resulting in a more accurate and stable dataset for the overall analysis. Essentially, this is mainly done to preserve 812 rows, which equates to approximately 46% of the dataset – extremely significant. The imputation was done by calculating the median for CONTENT_LENGTH by using the median() function, the na.rm() argument explicitly tells R to ignore/remove null values whilst computing the median. I then store the median in the variable 'median_content_length'. The is.na() function identifies the null value rows and replaces them with our median variable. Finally, I verify if the imputation was successful.


Console Terminal Background Jobs

R 4.4.1 · ~/

```
> #Retrieve median from CONTENT_LENGTH
> median_content_length <- median(df$CONTENT_LENGTH, na.rm = TRUE)
> #Impute NULL values in CONTENT_LENGTH with median
> df$CONTENT_LENGTH[is.na(df$CONTENT_LENGTH)] <- median_content_length
> #Verify NULLS
> colSums(is.na(df))
                      URL                URL_LENGTH NUMBER_SPECIAL_CHARACTERS 
                        0                         0                         0 
                  CHARSET                    SERVER            CONTENT_LENGTH 
                        0                         0                         0 
            WHOIS_COUNTRY             WHOIS_STATEPRO             WHOIS_REGDATE 
                        0                         0                         0 
        WHOIS_UPDATED_DATE TCP_CONVERSATION_EXCHANGE       DIST_REMOTE_TCP_PORT 
                        0                         0                         0 
               REMOTE_IPS                 APP_BYTES        SOURCE_APP_PACKETS 
                        0                         0                         0 
        REMOTE_APP_PACKETS          SOURCE_APP_BYTES          REMOTE_APP_BYTES 
                        0                         0                         0 
              APP_PACKETS           DNS_QUERY_TIMES                      Type 
                        0                         0                         0 
```

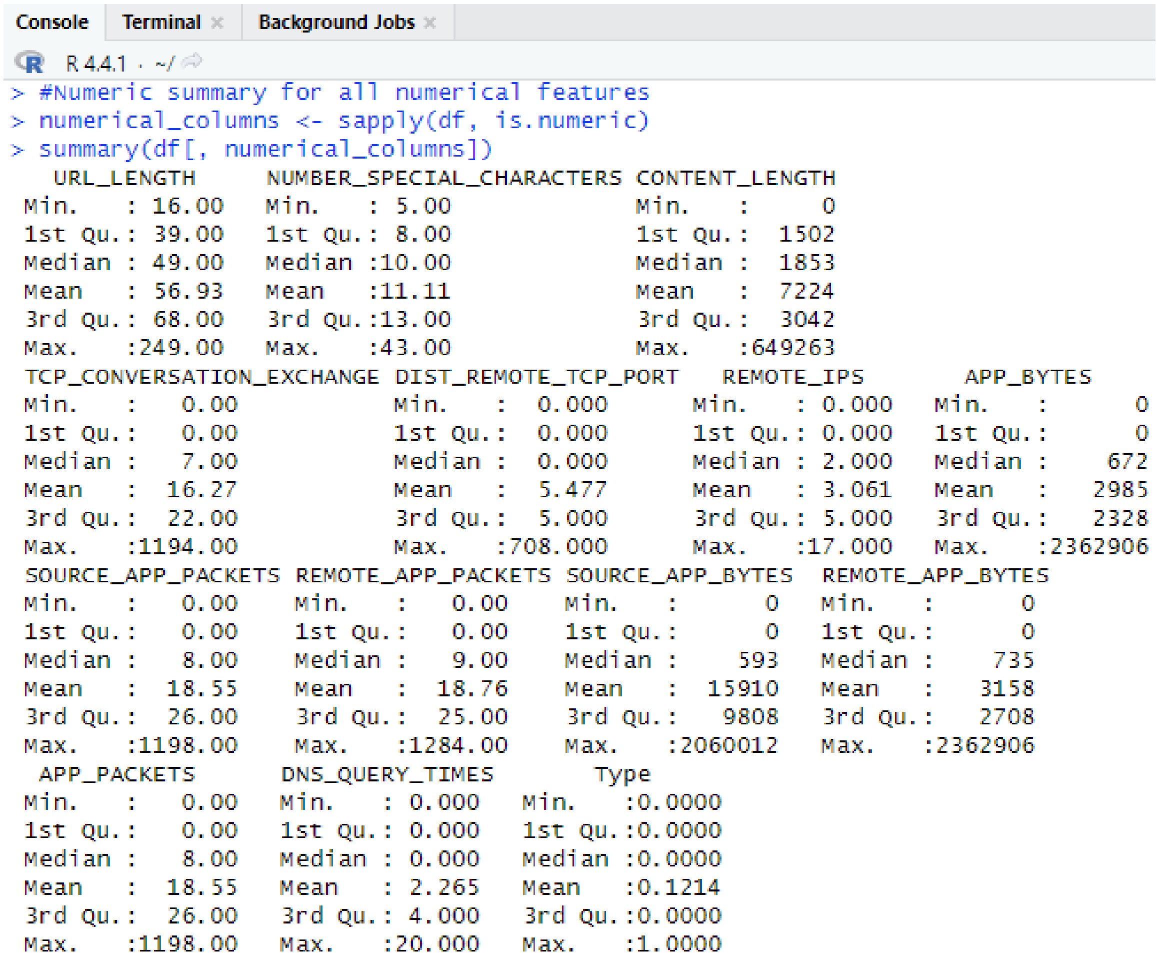
- Clearly, the removal of insignificant nulls and the imputation of significant nulls is successful, as there are no more nulls present in any feature within the dataset.

Overall, the approach I took eliminates null values, subsequently enhancing data integrity and quality whilst preserving the dataset. Addressing the null values was necessary because they would have otherwise distorted the data and introduced bias, negatively impacting the modelling performance.

Now that I've addressed the nulls in the dataset, it's time to investigate the outliers in all numerical features:

```
#Numeric summary for all numerical features
numerical_columns <- sapply(df, is.numeric)
summary(df[, numerical_columns])
```

- The summary() function is mainly deployed to provide a concise summary of several key statistics in all numerical features. It's a convenient and efficient way of determining outliers by analyzing the key statistical values.



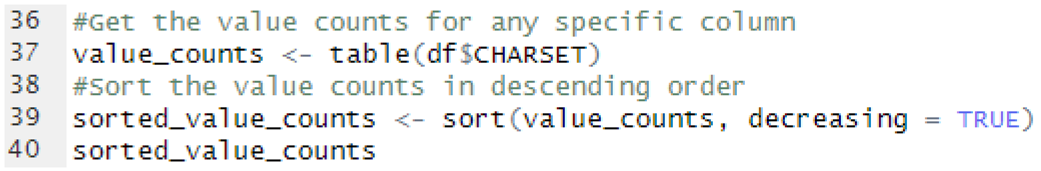
```
> #Numeric summary for all numerical features
> numerical_columns <- sapply(df, is.numeric)
> summary(df[, numerical_columns])
   URL_LENGTH     NUMBER_SPECIAL_CHARACTERS CONTENT_LENGTH
 Min.   : 16.00   Min.   : 5.00             Min.   :     0
 1st Qu.: 39.00   1st Qu.: 8.00             1st Qu.:  1502
 Median : 49.00   Median :10.00             Median :  1853
 Mean   : 56.93   Mean   :11.11             Mean   :  7224
 3rd Qu.: 68.00   3rd Qu.:13.00             3rd Qu.:  3042
 Max.   :249.00   Max.   :43.00             Max.   :649263
 TCP_CONVERSATION_EXCHANGE DIST_REMOTE_TCP_PORT   REMOTE_IPS       APP_BYTES
 Min.   :   0.00           Min.   :  0.000      Min.   : 0.000   Min.   :      0
 1st Qu.:   0.00           1st Qu.:  0.000      1st Qu.: 0.000   1st Qu.:      0
 Median :   7.00           Median :  0.000      Median : 2.000   Median :    672
 Mean   :  16.27           Mean   :  5.477      Mean   : 3.061   Mean   :   2985
 3rd Qu.:  22.00           3rd Qu.:  5.000      3rd Qu.: 5.000   3rd Qu.:   2328
 Max.   :1194.00           Max.   :708.000      Max.   :17.000   Max.   :2362906
 SOURCE_APP_PACKETS REMOTE_APP_PACKETS SOURCE_APP_BYTES   REMOTE_APP_BYTES
 Min.   :   0.00    Min.   :   0.00    Min.   :      0   Min.   :      0
 1st Qu.:   0.00    1st Qu.:   0.00    1st Qu.:      0   1st Qu.:      0
 Median :   8.00    Median :   9.00    Median :    593   Median :    735
 Mean   :  18.55    Mean   :  18.76    Mean   :  15910   Mean   :   3158
 3rd Qu.:  26.00    3rd Qu.:  25.00    3rd Qu.:   9808   3rd Qu.:   2708
 Max.   :1198.00    Max.   :1284.00    Max.   :2060012   Max.   :2362906
  APP_PACKETS      DNS_QUERY_TIMES        Type
 Min.   :   0.00   Min.   : 0.000   Min.   :0.0000
 1st Qu.:   0.00   1st Qu.: 0.000   1st Qu.:0.0000
 Median :   8.00   Median : 0.000   Median :0.0000
 Mean   :  18.55   Mean   : 2.265   Mean   :0.1214
 3rd Qu.:  26.00   3rd Qu.: 4.000   3rd Qu.:0.0000
 Max.   :1198.00   Max.   :20.000   Max.   :1.0000
```

The above table depicts a simple statistical summary of every numerical feature within the dataset: highlighting the min, Q1, median, mean, Q3, and max values of every numerical feature.

- **Min**: The smallest value in the column.
- **Q1 (25% Percentile)**: The value below which 25% of the data fall.
- **Median**: The middle value of the dataset.
- **Mean**: The average (mean) of each column.
- **Q3 (75% Percentile)**: The value below which 75% of the data fall.
- **Maximum (max)**: The largest value in the column.

I will only be focusing on the minimum and maximum values, as these values are the outliers present. Upon thorough observation, none of the minimum or maximum values immediately stand out as invalid outliers (errors). They all appear plausible within the context of web traffic and server communication. The outliers seem to be valid within the context of the features, as described previously. The high values, while outliers, could represent legitimate variations in the dataset, especially considering the diversity in web traffic and server responses. For example, CONTENT_LENGTH: While the maximum value is quite large, it's possible for web content to be this extensive, especially for rich media or large files. The minimum value of 0 is also plausible, as some HTTP responses might not have content. All outliers from features fall under this plausible notion, because of this I won't be removing/imputing any outliers as I believe they are valid. However, to be absolutely certain these are valid outlier values, generally, is to conduct extensive research and to consult with domain knowledge experts to provide more context, set boundaries, and validate these value ranges.

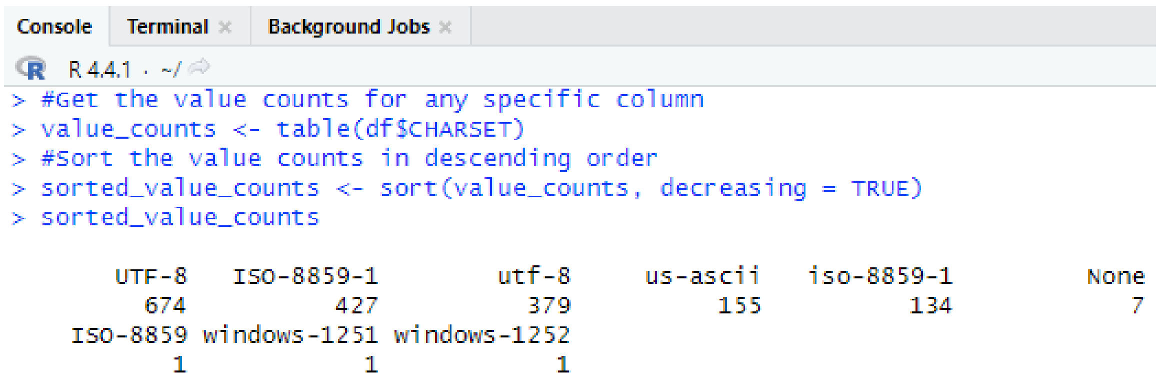
To finalize the data cleaning phase, I will be finishing off by thoroughly analyzing every categorical feature value and ascertain if they need to be modified. This is done to see if there are any values that shouldn't be there – data entry errors.



```r
#Get the value counts for any specific column
value_counts <- table(df$CHARSET)
#Sort the value counts in descending order
sorted_value_counts <- sort(value_counts, decreasing = TRUE)
sorted_value_counts
```

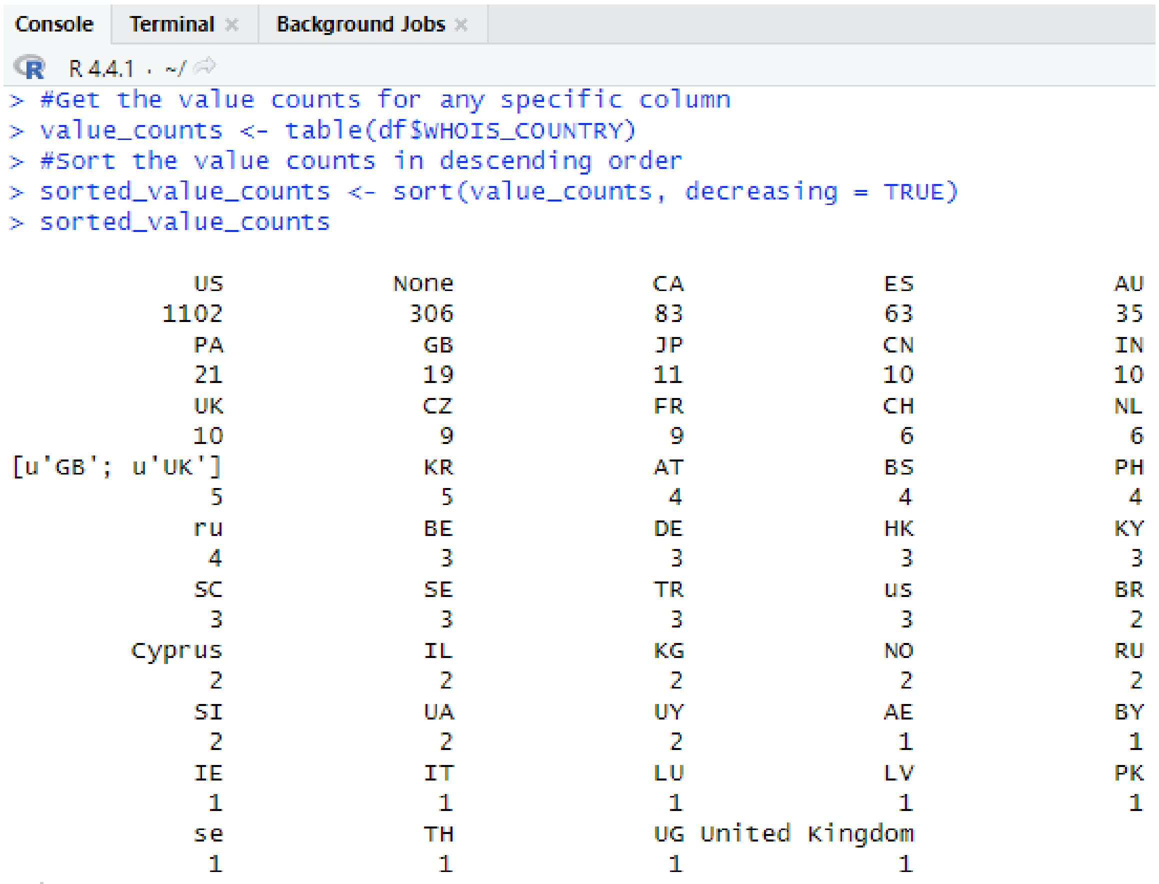
- Observing the values of categorical features by calculating the frequency of each unique value in the specified column (CHARSET) using the table() function, which creates a contingency table of counts. Then, it sorts these counts in descending order with the sort() function, specifying decreasing = TRUE to ensure the most frequent values appear first. The result: sorted_value_counts, it shows a summary of the distribution of a specified feature values in descending order.

- **CHARSET** values:



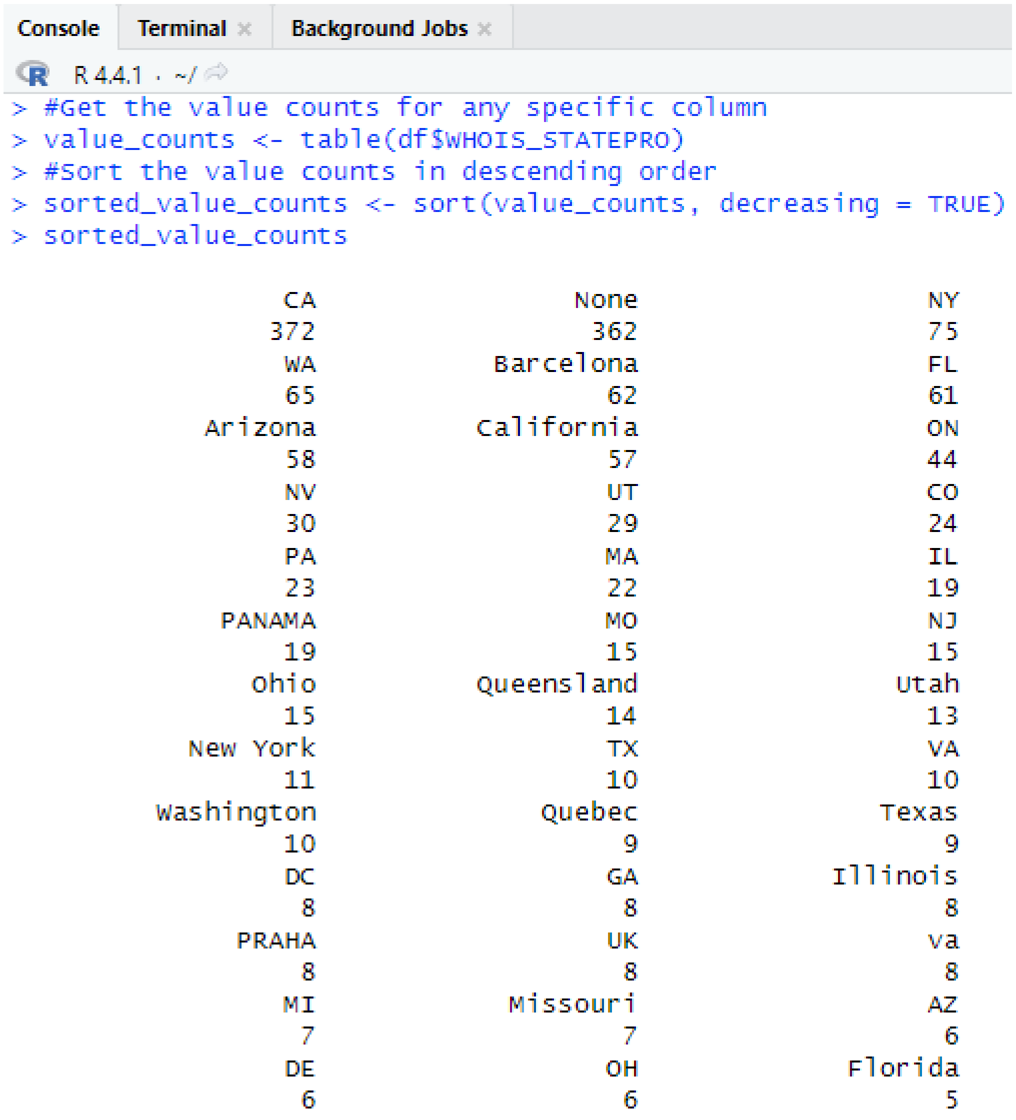
```
> #Get the value counts for any specific column
> value_counts <- table(df$CHARSET)
> #Sort the value counts in descending order
> sorted_value_counts <- sort(value_counts, decreasing = TRUE)
> sorted_value_counts

        UTF-8    ISO-8859-1         utf-8      us-ascii    iso-8859-1          None 
          674           427           379           155           134             7 
     ISO-8859  windows-1251  windows-1252 
            1             1             1 
```

- **WHOIS_COUNTRY** values:



```
> #Get the value counts for any specific column
> value_counts <- table(df$WHOIS_COUNTRY)
> #Sort the value counts in descending order
> sorted_value_counts <- sort(value_counts, decreasing = TRUE)
> sorted_value_counts

             US            None              CA              ES              AU 
           1102             306              83              63              35 
             PA              GB              JP              CN              IN 
             21              19              11              10              10 
             UK              CZ              FR              CH              NL 
             10               9               9               6               6 
[u'GB'; u'UK']              KR              AT              BS              PH 
              5               5               4               4               4 
             ru              BE              DE              HK              KY 
              4               3               3               3               3 
             SC              SE              TR              us              BR 
              3               3               3               3               2 
         Cyprus              IL              KG              NO              RU 
              2               2               2               2               2 
             SI              UA              UY              AE              BY 
              2               2               2               1               1 
             IE              IT              LU              LV              PK 
              1               1               1               1               1 
             se              TH              UG United Kingdom 
              1               1               1               1 
```

WHOIS_STATEPRO values:



```
> #Get the value counts for any specific column
> value_counts <- table(df$WHOIS_STATEPRO)
> #Sort the value counts in descending order
> sorted_value_counts <- sort(value_counts, decreasing = TRUE)
> sorted_value_counts

          CA         None           NY
         372          362           75
          WA    Barcelona           FL
          65           62           61
     Arizona   California           ON
          58           57           44
          NV           UT           CO
          30           29           24
          PA           MA           IL
          23           22           19
      PANAMA           MO           NJ
          19           15           15
        Ohio   Queensland         Utah
          15           14           13
    New York           TX           VA
          11           10           10
  Washington       Quebec        Texas
          10            9            9
          DC           GA     Illinois
           8            8            8
       PRAHA           UK           va
           8            8            8
          MI     Missouri           AZ
           7            7            6
          DE           OH      Florida
           6            6            5
```

In this dataset, several features exhibit inconsistencies in their values due to variations in data entry errors. These inconsistencies can lead to inaccuracies in analysis and modelling, as they represent the same entity or concept in different formats.

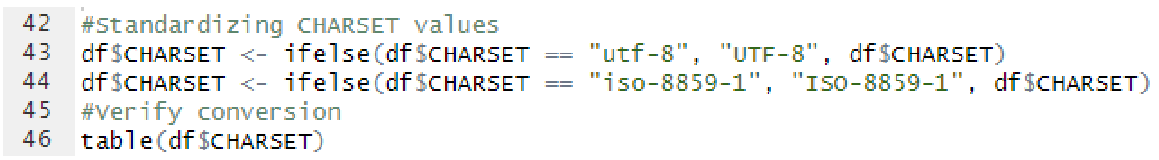
- **CHARSET Feature:**
  - The CHARSET feature includes values such as UTF-8, utf-8, ISO-8859-1, and iso-8859-1. Character encoding names are generally case-insensitive, meaning UTF-8 and utf-8 should be treated as identical. The presence of these variations suggests that data has been entered inconsistently, with different case conventions being applied arbitrarily. Without standardization, these

variations could be incorrectly treated as distinct categories, leading to misleading/poor results in the modelling phase.

- **WHOIS_COUNTRY Feature:**
    - The WHOIS_COUNTRY feature contains variations such as us, US, GB, UK, United Kingdom, [u 'GB'; u 'UK'], ru, RU, se, and SE. These values represent country codes and names, but they appear in different formats: us vs. US, gb vs. GB, etc., where the only difference is the case of the letters. For the United Kingdom, for example, the dataset includes multiple representations: GB (ISO country code), UK (alternative abbreviation), United Kingdom (full name), and even a more complex entry [u 'GB'; u 'UK'], which appears to be a mistakenly formatted list. These inconsistencies can cause the dataset to misrepresent the actual distribution of countries, as the same country might be counted under multiple different names, skewing the data, impacting the datasets integrity and quality greatly.
- **WHOIS_STATEPRO Feature:**
    - The WHOIS_STATEPRO feature includes variations like CA, California, NY, New York, etc. This feature should ideally standardize on a single format for states or provinces. CA vs. California, NY vs. New York. The abbreviation and the full name refer to the same state, but they are recorded differently. Like the other features, different cases (e.g., ca vs. CA) add to the inconsistency. These discrepancies can complicate any analysis that involves state-level data, such as identifying patterns in cyber-attacks by state. Without standardization, the dataset may incorrectly split data between what should be a single entity.

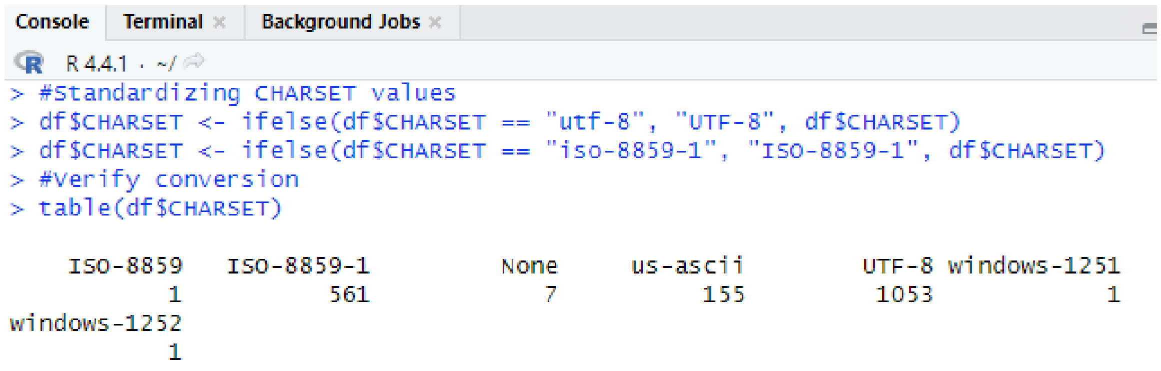
I deal with the described issues above, accordingly:

- **CHARSET:**



```
#Standardizing CHARSET values
df$CHARSET <- ifelse(df$CHARSET == "utf-8", "UTF-8", df$CHARSET)
df$CHARSET <- ifelse(df$CHARSET == "iso-8859-1", "ISO-8859-1", df$CHARSET)
#verify conversion
table(df$CHARSET)
```

- I have standardized the CHARSET values by merging variations using the ifelse() function. Specifically, values of "utf-8" are replaced with "UTF-8", and "iso-8859-1" values are merged into "ISO-8859-1". This operation ensures consistent labeling of character encodings, addressing data entry errors and enhancing the accuracy and reliability of the dataset for analysis.



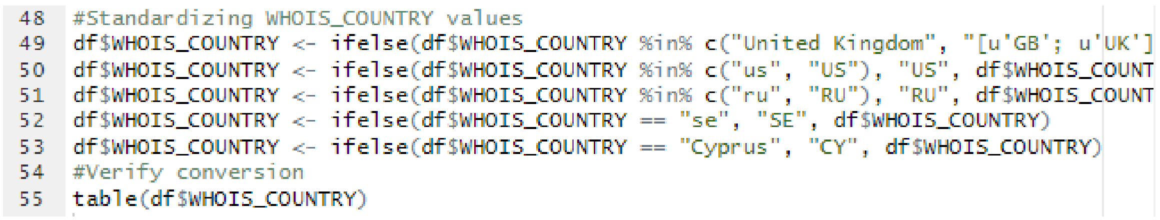
Console Terminal Background Jobs

R 4.4.1 · ~/

```
> #Standardizing CHARSET values
> df$CHARSET <- ifelse(df$CHARSET == "utf-8", "UTF-8", df$CHARSET)
> df$CHARSET <- ifelse(df$CHARSET == "iso-8859-1", "ISO-8859-1", df$CHARSET)
> #Verify conversion
> table(df$CHARSET)

    ISO-8859   ISO-8859-1         None     us-ascii        UTF-8 windows-1251 
           1          561            7          155         1053            1 
windows-1252 
           1 
```

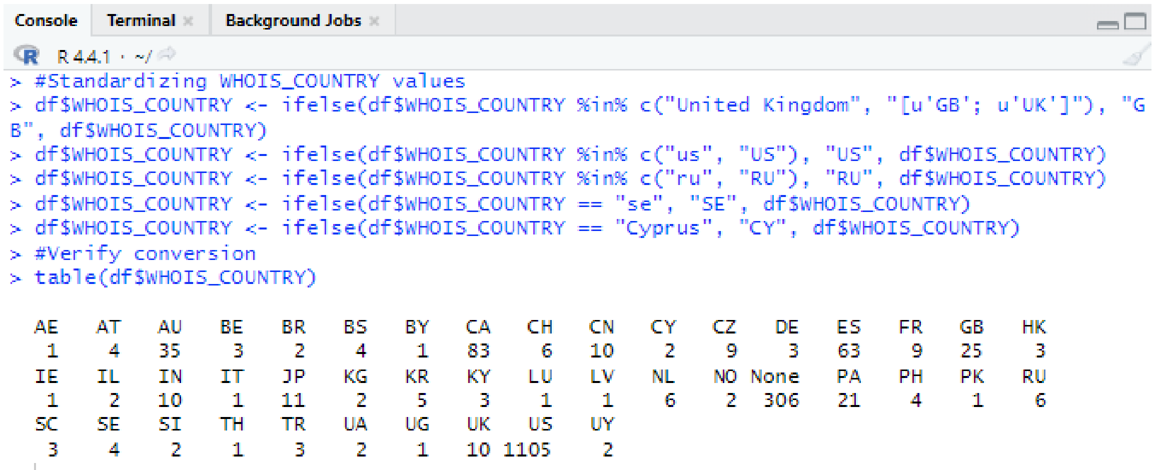
- By analysing the output above, we can identify the successful operation of merging utf-8: UTF-8 and iso-8859-1: ISO-8859-1.

- **WHOIS_COUNTRY:**



```
#Standardizing WHOIS_COUNTRY values
df$WHOIS_COUNTRY <- ifelse(df$WHOIS_COUNTRY %in% c("United Kingdom", "[u'GB'; u'UK']
df$WHOIS_COUNTRY <- ifelse(df$WHOIS_COUNTRY %in% c("us", "US"), "US", df$WHOIS_COUNT
df$WHOIS_COUNTRY <- ifelse(df$WHOIS_COUNTRY %in% c("ru", "RU"), "RU", df$WHOIS_COUNT
df$WHOIS_COUNTRY <- ifelse(df$WHOIS_COUNTRY == "se", "SE", df$WHOIS_COUNTRY)
df$WHOIS_COUNTRY <- ifelse(df$WHOIS_COUNTRY == "Cyprus", "CY", df$WHOIS_COUNTRY)
#Verify conversion
table(df$WHOIS_COUNTRY)
```

- I standardized the WHOIS_COUNTRY values by consolidating variations using the ifelse() function. For instance, entries like "United Kingdom" and the incorrect list "[u'GB'; u'UK']" were unified under the ISO code "GB." Similarly, different cases of "us" and "US" were standardized to "US," while "ru" and "RU" were unified as "RU." The code also converted "se" to "SE" and replaced "Cyprus" with its ISO code "CY." These adjustments ensure uniform country code labeling, correcting data entry errors and improving the dataset's quality and integrity.



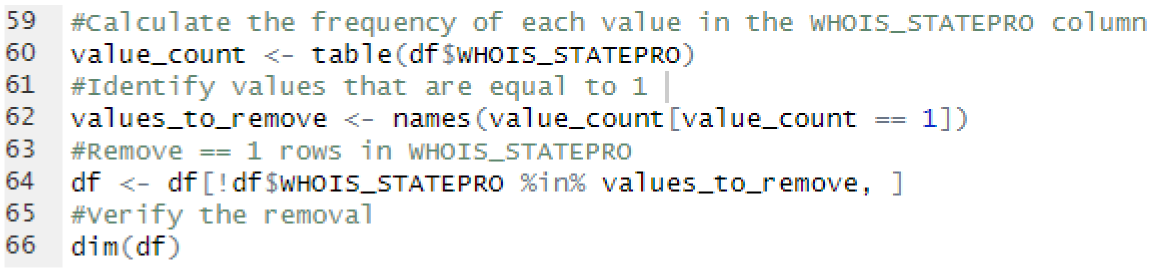
Console Terminal Background Jobs

R 4.4.1 · ~/

```
> #Standardizing WHOIS_COUNTRY values
> df$WHOIS_COUNTRY <- ifelse(df$WHOIS_COUNTRY %in% c("United Kingdom", "[u'GB'; u'UK']"), "G
B", df$WHOIS_COUNTRY)
> df$WHOIS_COUNTRY <- ifelse(df$WHOIS_COUNTRY %in% c("us", "US"), "US", df$WHOIS_COUNTRY)
> df$WHOIS_COUNTRY <- ifelse(df$WHOIS_COUNTRY %in% c("ru", "RU"), "RU", df$WHOIS_COUNTRY)
> df$WHOIS_COUNTRY <- ifelse(df$WHOIS_COUNTRY == "se", "SE", df$WHOIS_COUNTRY)
> df$WHOIS_COUNTRY <- ifelse(df$WHOIS_COUNTRY == "Cyprus", "CY", df$WHOIS_COUNTRY)
> #Verify conversion
> table(df$WHOIS_COUNTRY)

  AE   AT   AU   BE   BR   BS   BY   CA   CH   CN   CY   CZ   DE   ES   FR   GB   HK 
   1    4   35    3    2    4    1   83    6   10    2    9    3   63    9   25    3 
  IE   IL   IN   IT   JP   KG   KR   KY   LU   LV   NL   NO None   PA   PH   PK   RU 
   1    2   10    1   11    2    5    3    1    1    6    2  306   21    4    1    6 
  SC   SE   SI   TH   TR   UA   UG   UK   US   UY 
   3    4    2    1    3    2    1   10 1105    2 
```

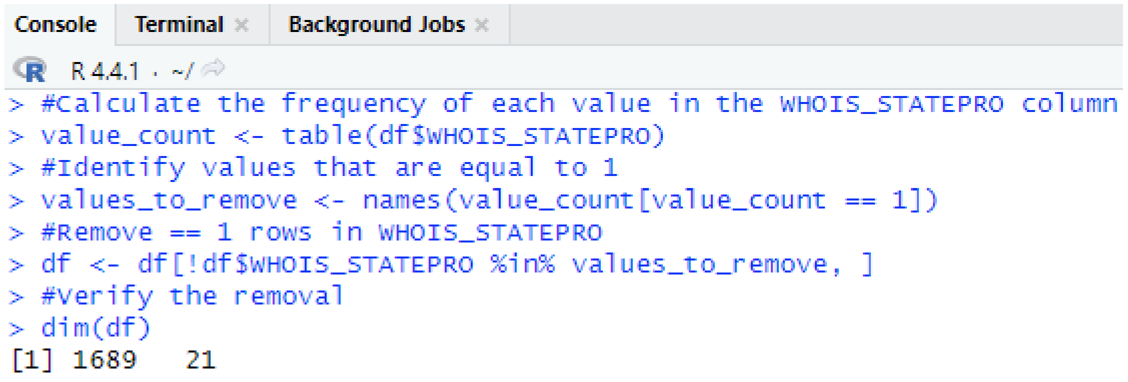
- Clearly, by the output, we can observe the issue of data entry errors are now resolved and all values are formatted correctly in WHOIS_COUNTRY.

- **WHOIS_STATEPRO:**



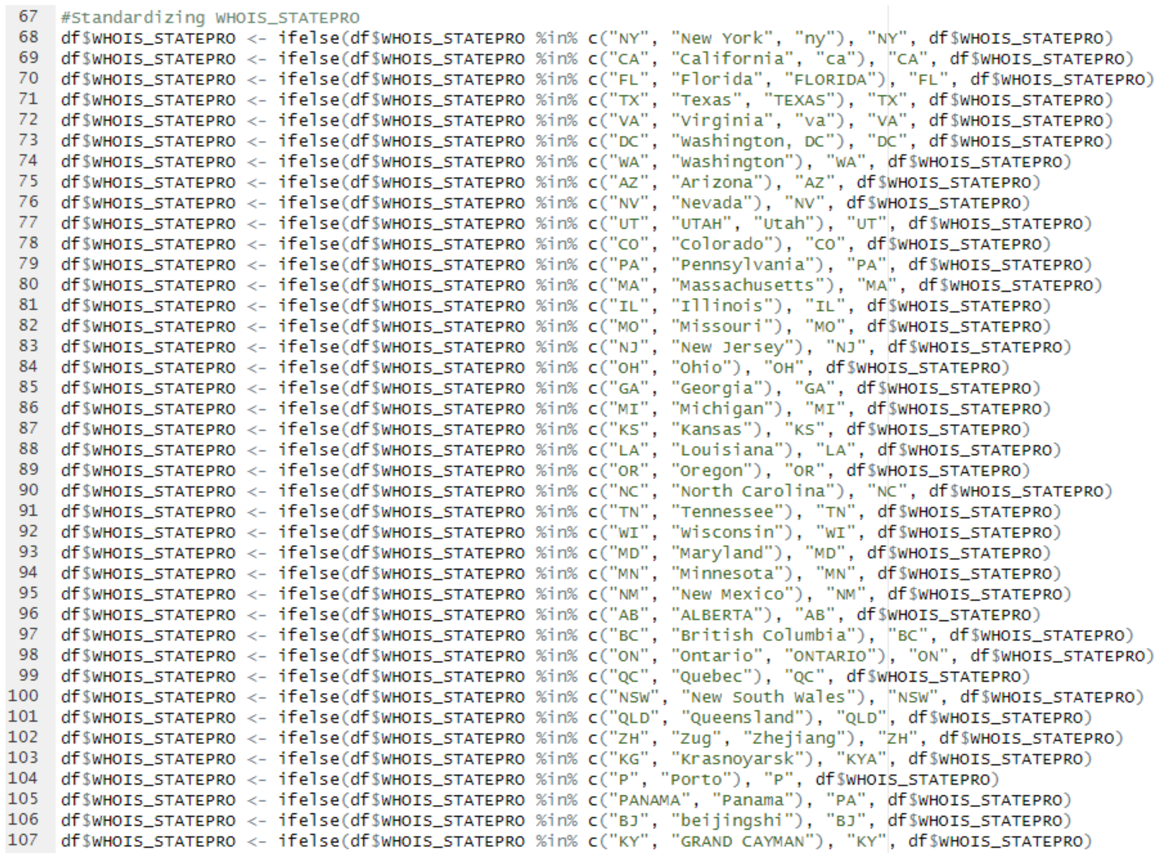
```
#Calculate the frequency of each value in the WHOIS_STATEPRO column
value_count <- table(df$WHOIS_STATEPRO)
#Identify values that are equal to 1
values_to_remove <- names(value_count[value_count == 1])
#Remove == 1 rows in WHOIS_STATEPRO
df <- df[!df$WHOIS_STATEPRO %in% values_to_remove, ]
#Verify the removal
dim(df)
```

- Before I standardize, I need to trim the samples in WHOIS_STATEPRO due to its extensive variety. Compared to the other two features we have standardized; this feature is magnitudes larger in terms of sample size: 181 unique values. I decide to remove all rows which are equal to 1 in this feature, as it's an efficient way of trimming down the extensive noisy values. This was achieved by using the table() function to calculate the frequency of each unique value in WHOIS_STATEPRO feature. It then identifies values that appear only once by comparing the frequency table to 1. The names() function is used to extract these values, which are then removed from the dataset using the ! operator and %in% function. This effectively filters out rows with these infrequent values. Finally, the dim() function is used to check the dimensions of the cleaned dataset, confirming the number of remaining rows after the removal process. This cleaning step reduces noise and streamlines the data, making it more manageable for further analysis and standardization.

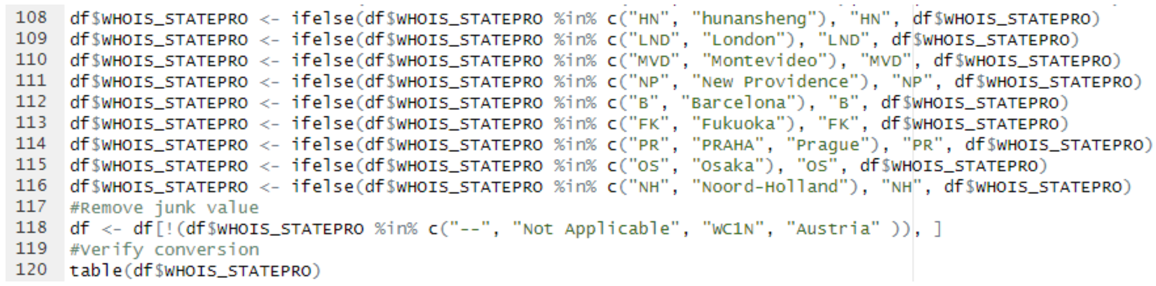


```
> #Calculate the frequency of each value in the WHOIS_STATEPRO column
> value_count <- table(df$WHOIS_STATEPRO)
> #Identify values that are equal to 1
> values_to_remove <- names(value_count[value_count == 1])
> #Remove == 1 rows in WHOIS_STATEPRO
> df <- df[!df$WHOIS_STATEPRO %in% values_to_remove, ]
> #Verify the removal
> dim(df)
[1] 1689   21
```

- 90 rows have been dropped in total from the trimming process; equating to approximately 5% of the dataset, which is insignificant overall.

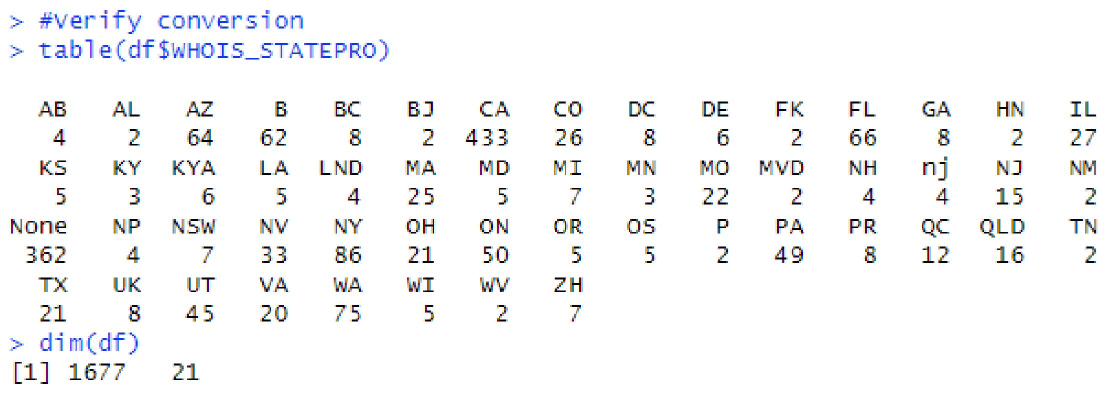


```r
#Standardizing WHOIS_STATEPRO
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("NY", "New York", "ny"), "NY", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("CA", "California", "ca"), "CA", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("FL", "Florida", "FLORIDA"), "FL", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("TX", "Texas", "TEXAS"), "TX", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("VA", "Virginia", "va"), "VA", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("DC", "Washington, DC"), "DC", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("WA", "Washington"), "WA", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("AZ", "Arizona"), "AZ", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("NV", "Nevada"), "NV", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("UT", "UTAH", "Utah"), "UT", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("CO", "Colorado"), "CO", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("PA", "Pennsylvania"), "PA", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("MA", "Massachusetts"), "MA", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("IL", "Illinois"), "IL", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("MO", "Missouri"), "MO", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("NJ", "New Jersey"), "NJ", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("OH", "Ohio"), "OH", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("GA", "Georgia"), "GA", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("MI", "Michigan"), "MI", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("KS", "Kansas"), "KS", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("LA", "Louisiana"), "LA", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("OR", "Oregon"), "OR", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("NC", "North Carolina"), "NC", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("TN", "Tennessee"), "TN", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("WI", "Wisconsin"), "WI", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("MD", "Maryland"), "MD", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("MN", "Minnesota"), "MN", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("NM", "New Mexico"), "NM", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("AB", "ALBERTA"), "AB", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("BC", "British Columbia"), "BC", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("ON", "Ontario", "ONTARIO"), "ON", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("QC", "Quebec"), "QC", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("NSW", "New South Wales"), "NSW", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("QLD", "Queensland"), "QLD", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("ZH", "Zug", "Zhejiang"), "ZH", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("KG", "Krasnoyarsk"), "KYA", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("P", "Porto"), "P", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("PANAMA", "Panama"), "PA", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("BJ", "beijingshi"), "BJ", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("KY", "GRAND CAYMAN"), "KY", df$WHOIS_STATEPRO)
```



```r
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("HN", "hunansheng"), "HN", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("LND", "London"), "LND", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("MVD", "Montevideo"), "MVD", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("NP", "New Providence"), "NP", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("B", "Barcelona"), "B", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("FK", "Fukuoka"), "FK", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("PR", "PRAHA", "Prague"), "PR", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("OS", "Osaka"), "OS", df$WHOIS_STATEPRO)
df$WHOIS_STATEPRO <- ifelse(df$WHOIS_STATEPRO %in% c("NH", "Noord-Holland"), "NH", df$WHOIS_STATEPRO)
#Remove junk value
df <- df[!(df$WHOIS_STATEPRO %in% c("--", "Not Applicable", "WC1N", "Austria" )), ]
#Verify conversion
table(df$WHOIS_STATEPRO)
```

- Standardizing the WHOIS_STATEPRO feature is crucial because it ensures consistency in the data. Without standardization, the dataset might have multiple representations of the same state or province (e.g., "California," "CA," "ca"), which could lead to inaccurate analysis or reporting. By converting all entries to a standardized format, the data becomes uniform, making it easier to perform accurate and meaningful analysis. This process also helps in reducing redundancies and improving the overall quality of the dataset. The primary function used in the code is ifelse(), which checks if the value in WHOIS_STATEPRO matches any of the specified conditions. If it does, the value is replaced with a standardized abbreviation. For example, if the value in WHOIS_STATEPRO is "New York," "NY," or "ny," it will be replaced with the standardized abbreviation "NY." This same logic is applied to various other states, provinces, and regions such as "CA" for California, "TX" for Texas, and so on. After standardizing the values, the code removes any irrelevant or junk values from the dataset. The df <- df[!(df$WHOIS_STATEPRO %in% c("--", "Not Applicable", "WC1N", "Austria")),] line ensures that any entries with these values are excluded from the dataset. Finally, the code uses table(df$WHOIS_STATEPRO) to display the frequency distribution of the standardized WHOIS_STATEPRO values. This helps to verify that the conversion has been successfully applied and allows for an easy check of the standardized data.



```
> #verify conversion
> table(df$WHOIS_STATEPRO)

  AB   AL   AZ    B   BC   BJ   CA   CO   DC   DE   FK   FL   GA   HN   IL 
   4    2   64   62    8    2  433   26    8    6    2   66    8    2   27 
  KS   KY  KYA   LA  LND   MA   MD   MI   MN   MO  MVD   NH   nj   NJ   NM 
   5    3    6    5    4   25    5    7    3   22    2    4    4   15    2 
None   NP  NSW   NV   NY   OH   ON   OR   OS    P   PA   PR   QC  QLD   TN 
 362    4    7   33   86   21   50    5    5    2   49    8   12   16    2 
  TX   UK   UT   VA   WA   WI   WV   ZH 
  21    8   45   20   75    5    2    7 
> dim(df)
[1] 1677   21
```

- By analyzing the output, we can conclude that the values for WHOIS_STATEPRO have been rectified, and all values are now correctly formatted. 12 rows have been dropped during this process, which amounts to less than 1% of the dataset: insignificant.

The data cleaning phase is now complete, ultimately, we have significantly enhanced the quality and integrity of our data, setting a solid foundation for extracting key insights in the subsequent steps of our data analysis. With the data cleaning phase complete, we are now ready to proceed to the next stage: exploratory data analysis (EDA).

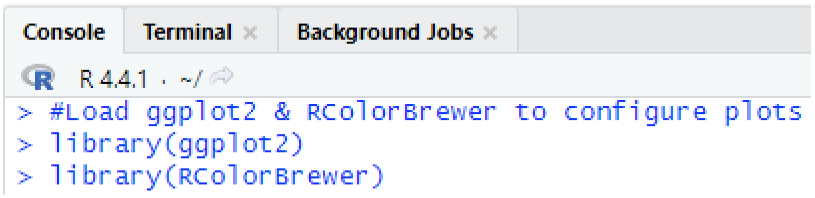
Exploratory Data Analysis

Exploratory Data Analysis (EDA): the process of analyzation and visualization to summarize the features and their corresponding relationships within the data, providing valuable insights into the factors contributing to malicious websites.

Before I begin EDA, I will be importing the ggplot2 & RColorBrewer package, as I will be using these two packages to conduct and configure several plots.

```r
#Load ggplot2 & RColorBrewer to configure plots
library(ggplot2)
library(RColorBrewer)
```

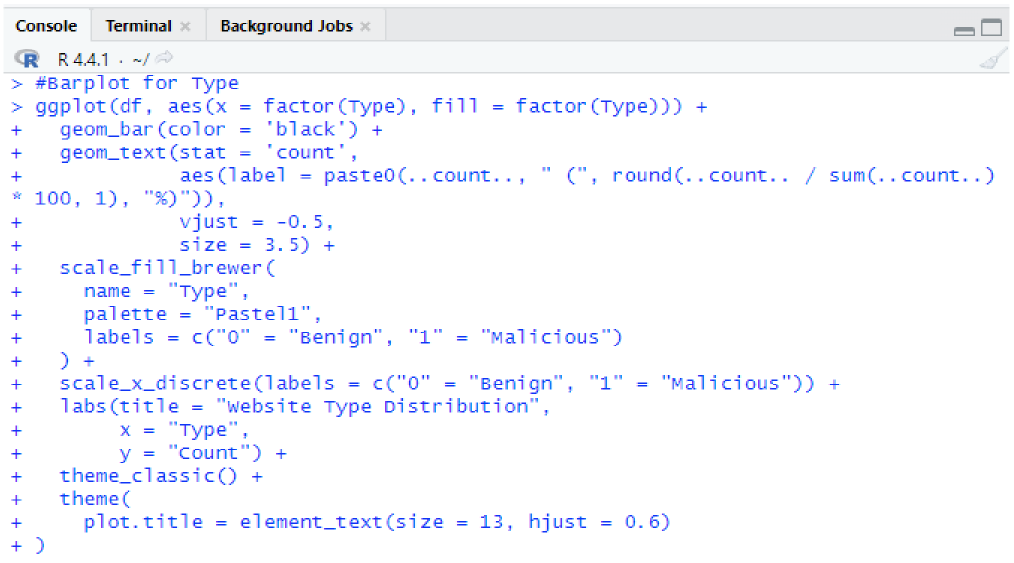
- Ggplot2 is a popular R package used for creating a wide range of static graphics and data visualizations. It's highly versatile and allows for the creation of complex plots with layered elements.
- RColorBrewer is an R package that provides a set of colour palettes designed to be useful for data visualization. It offers palettes that are colourblind-friendly and suitable for different types of data, such as categorical, sequential, and diverging data.



```
> #Load ggplot2 & RColorBrewer to configure plots
> library(ggplot2)
> library(RColorBrewer)
```

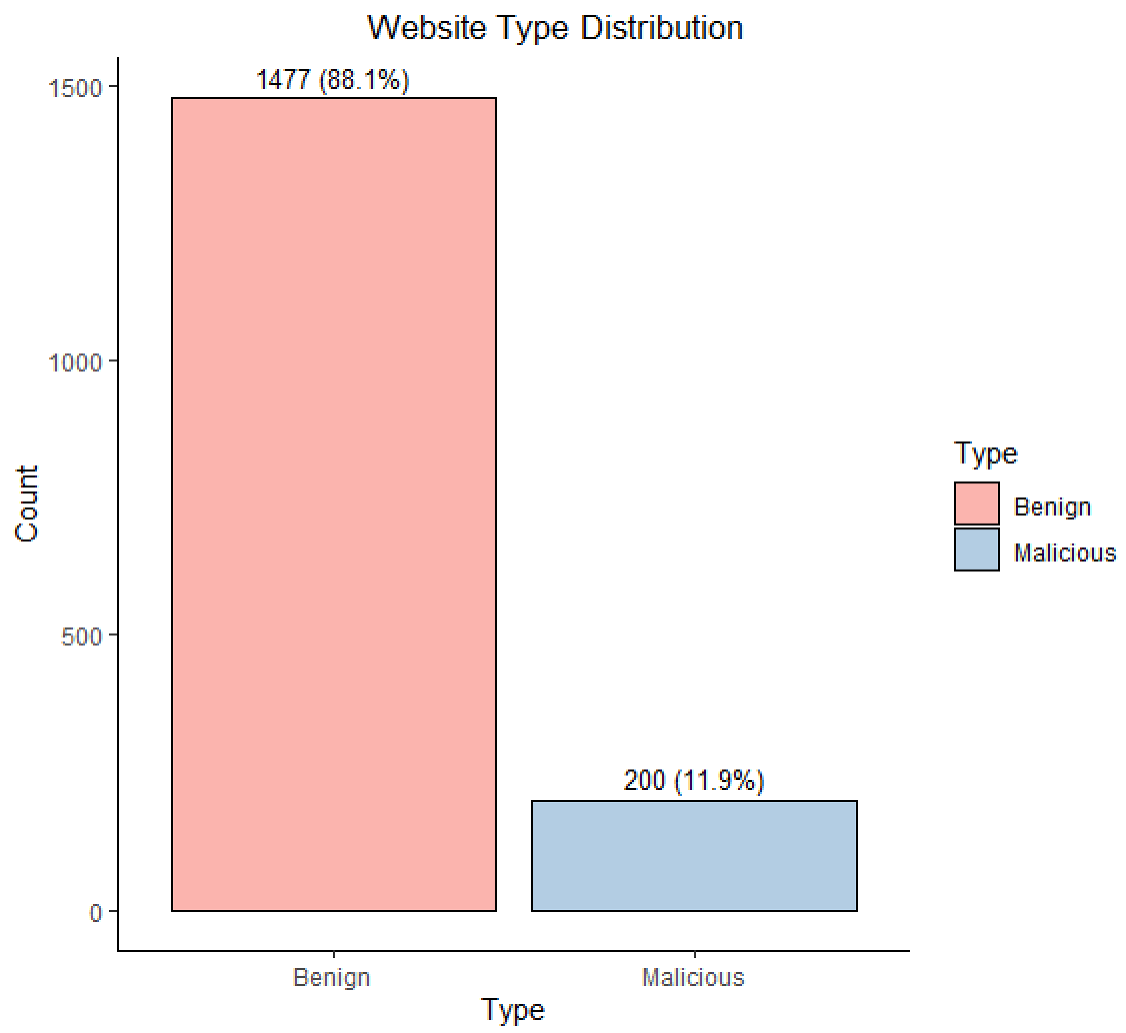
Bar plot layout:

```r
#Barplot for Type
ggplot(df, aes(x = factor(Type), fill = factor(Type))) +
  geom_bar(color = 'black') +
  geom_text(stat = 'count',
            aes(label = paste0(..count.., " (", round(..count.. / sum(..cou
            vjust = -0.5,
            size = 3.5) +
  scale_fill_brewer(
    name = "Type",
    palette = "Pastel1",
    labels = c("0" = "Benign", "1" = "Malicious")
  ) +
  scale_x_discrete(labels = c("0" = "Benign", "1" = "Malicious")) +
  labs(title = "Website Type Distribution",
       x = "Type",
       y = "Count") +
  theme_classic() +
  theme(
    plot.title = element_text(size = 13, hjust = 0.6)
)
```

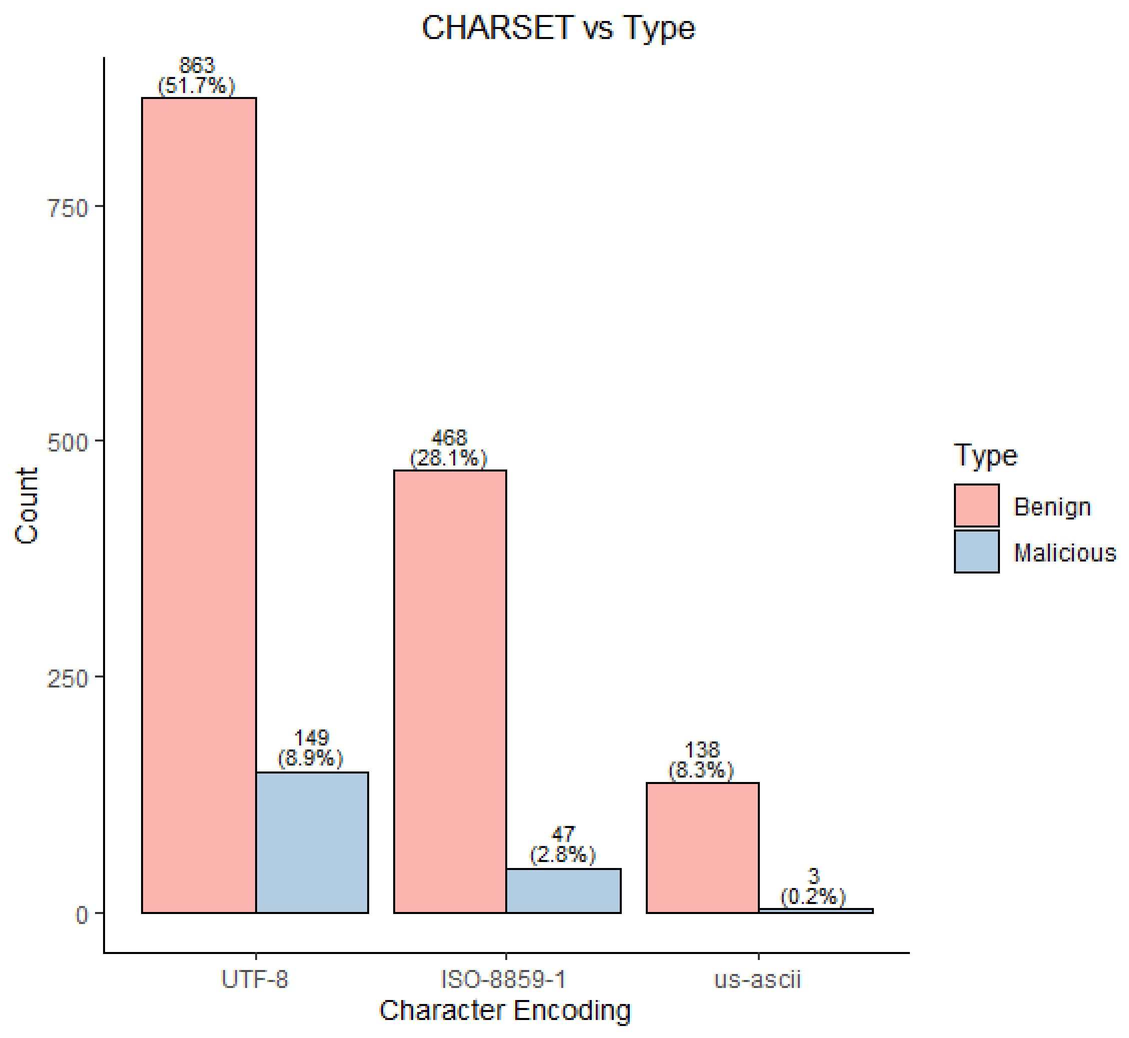


```
> #Barplot for Type
> ggplot(df, aes(x = factor(Type), fill = factor(Type))) +
+   geom_bar(color = 'black') +
+   geom_text(stat = 'count',
+             aes(label = paste0(..count.., " (", round(..count.. / sum(..count..)
* 100, 1), "%)")),
+             vjust = -0.5,
+             size = 3.5) +
+   scale_fill_brewer(
+     name = "Type",
+     palette = "Pastel1",
+     labels = c("0" = "Benign", "1" = "Malicious")
+   ) +
+   scale_x_discrete(labels = c("0" = "Benign", "1" = "Malicious")) +
+   labs(title = "Website Type Distribution",
+        x = "Type",
+        y = "Count") +
+   theme_classic() +
+   theme(
+     plot.title = element_text(size = 13, hjust = 0.6)
+ )
```

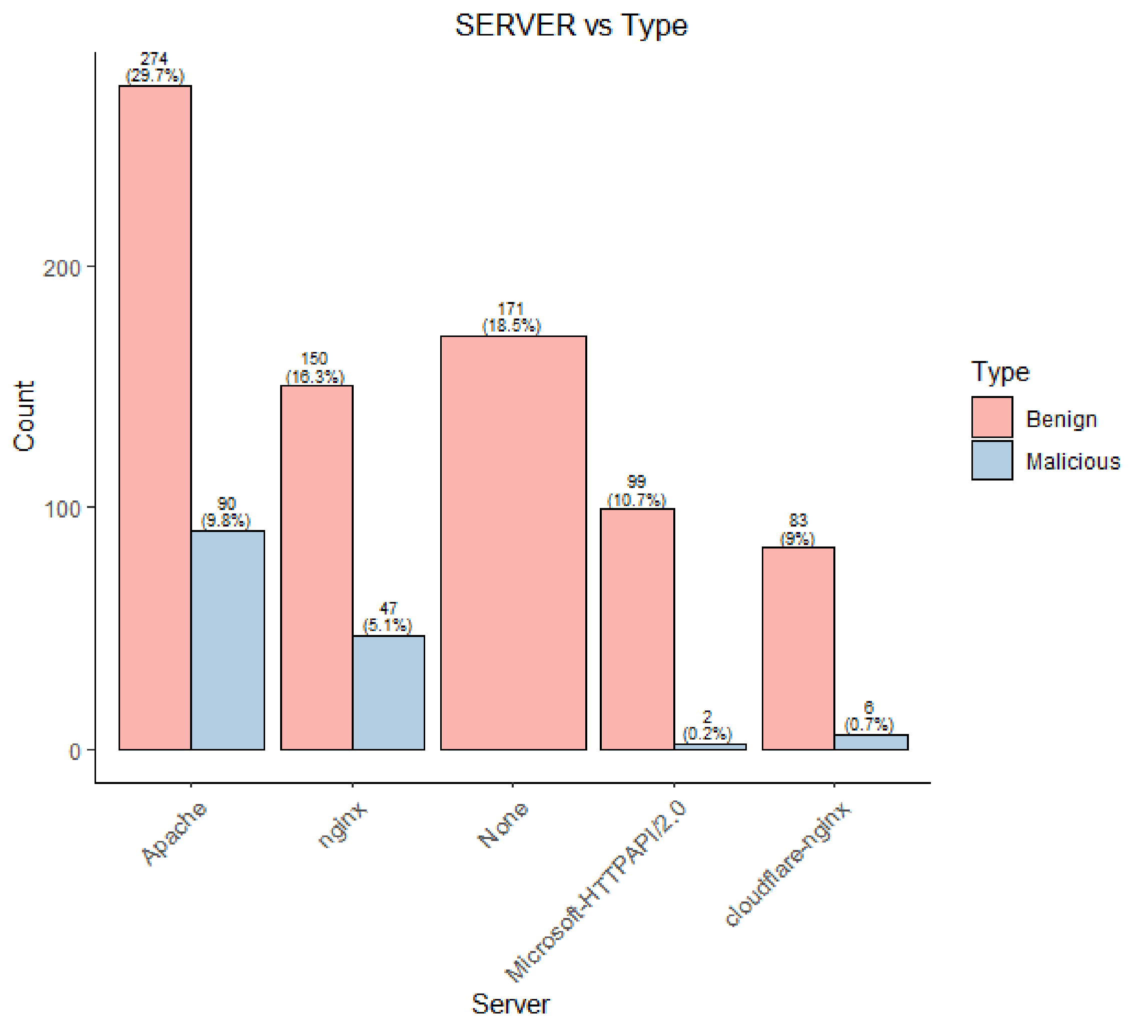
This code creates a bar plot using ggplot2 to display the distribution of "Benign" and "Malicious" values within the Type variable from the dataset. It initializes the plot with Type on the x-axis and uses it for bar colors. The geom_bar() function generates the bars with black outlines, while geom_text() adds the count and percentage labels on top of each bar, slightly positioned above. The scale_fill_brewer() function applies the "Pastel1" color palette from ColorBrewer, with customized legend labels for "Benign" and "Malicious." The x-axis labels are also set to these terms. The labs() function adds a title and axis labels, and theme_classic() provides a clean, minimal theme, with theme() adjusting the title's size and positioning it in the middle. This results in a clear and informative plot with a visually appealing design. Essentially this is the code I will be slightly reconfiguring differently for every categorical feature in the dataset. Because of the slight reconfigurations needed for every other feature, I wont be supplying the code for them as it would be redundant (mostly minimal changes required) and would take away from a neat and tidy EDA – this stage is also not a necessary requirement.



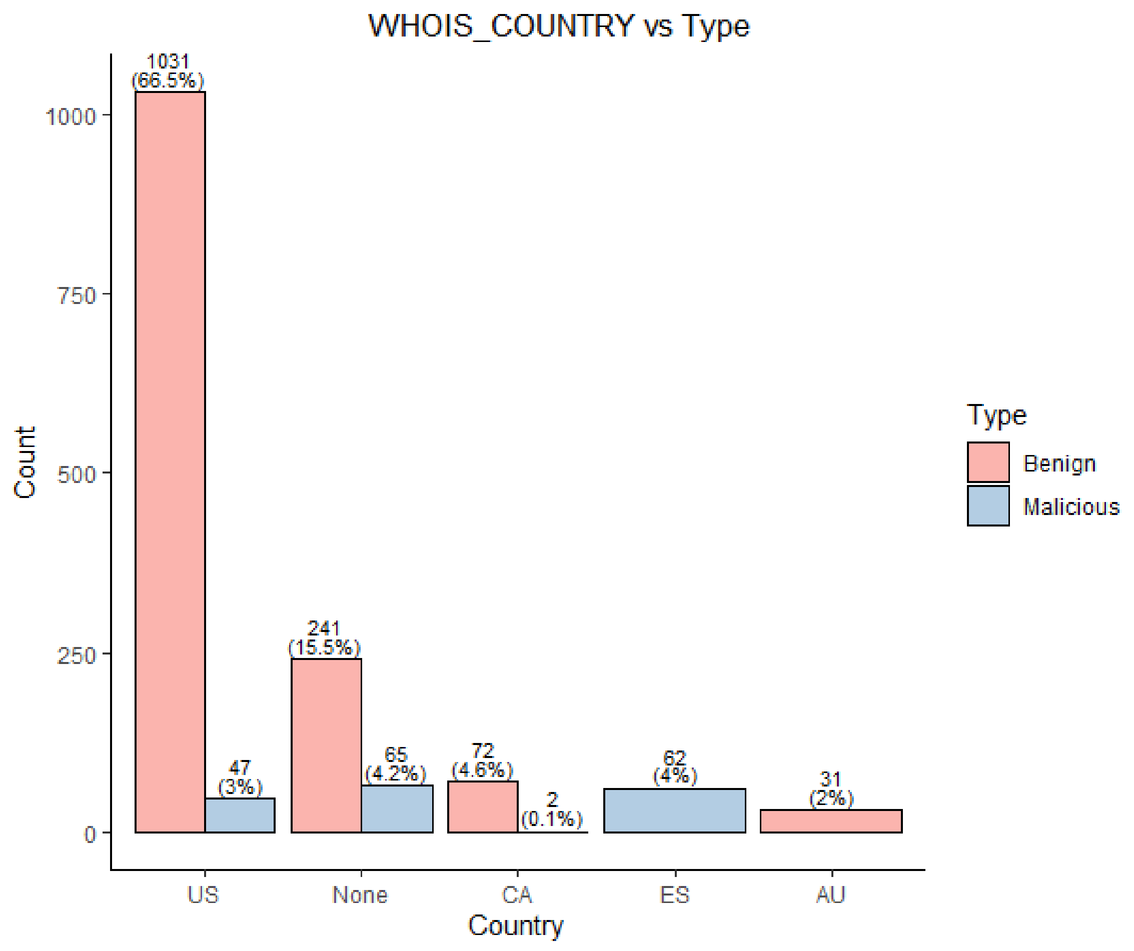
This bar plot displays the count of the Type feature (target variable). Clearly there is a vast difference between the values: 1477 (or 88.1%) of websites are benign, whereas only 200 (or 11.9%) of websites are malicious in nature within this dataset. The dataset is quite unbalanced and possibly needs to be amended before the modelling phase, otherwise it could lead to biased and skewed data — impacting the models evaluation on predicting whether websites are malicious catastrophically.



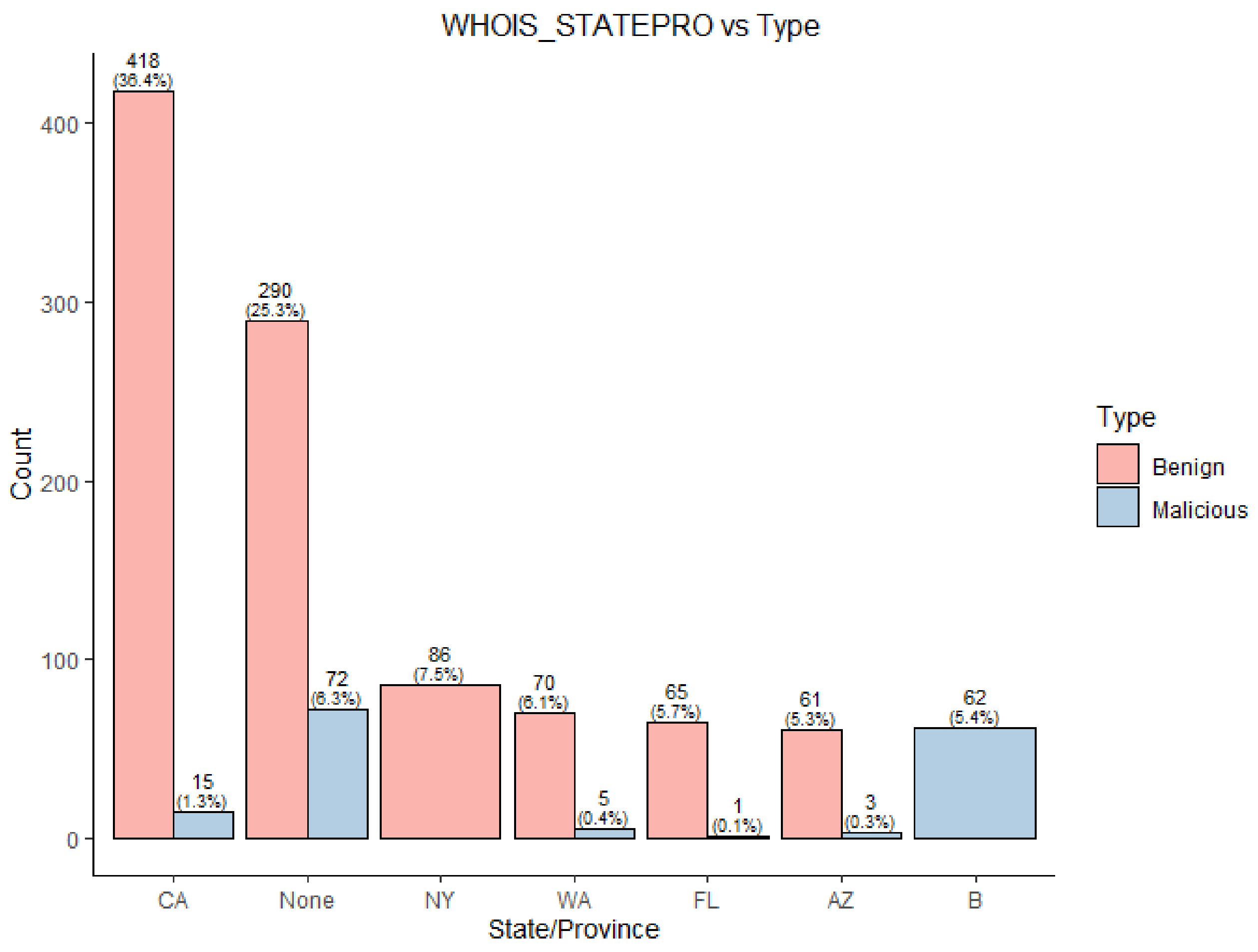
This bar plot illustrates the relationship between CHARSET and Type, showcasing the top 3-character encodings for websites in descending order of frequency within the dataset. The analysis reveals a notable disparity: the UTF-8-character encoding is predominant, with 863 benign websites (51.7%) and 149 malicious websites (8.9%). In contrast, the us-ascii charset has the smallest count of website types in total, with 138 benign and 3 malicious websites. Proportionally, UTF-8 has the highest rate of malicious websites, with approximately 15% of UTF-8 websites being labelled as malicious, indicating the highest rate of malicious activity among the character encodings in this dataset.



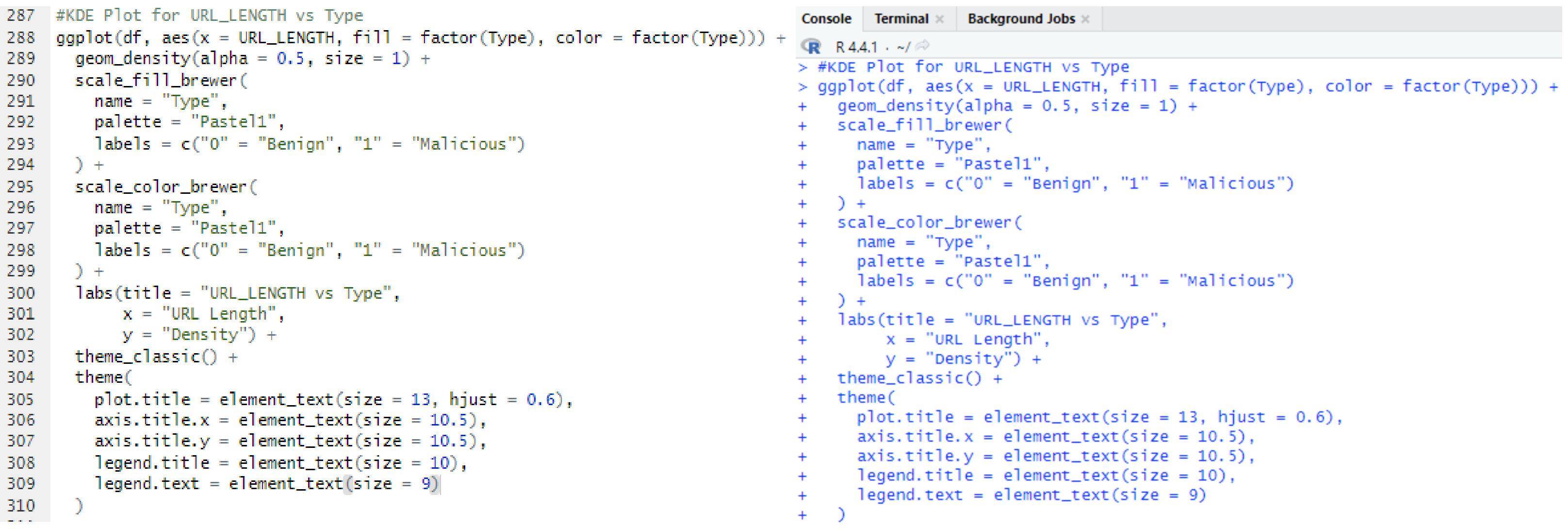
This bar plot demonstrates the correlation between SERVER and Type, highlighting the top 5 servers based on their frequency in descending order within the dataset. The analysis indicates a significant trend: Apache is the most prevalent server, hosting 274 benign websites (29.7%) and 90 malicious websites (9.8%). On the other hand, the cloudflare-nginx server shows the lowest total count, with only 83 benign and 6 malicious websites. Notably, Apache also exhibits the highest proportion of malicious websites, with about 25% of Apache servers being associated with malicious activity, marking it as the server with the greatest occurrence of malicious websites in this dataset.



This bar plot depicts the relationship between WHOIS_COUNTRY and Type, showcasing the top 5 countries with the highest frequency in the dataset. The findings reveal that the United States is the most prevalent country, with 1031 benign websites (66.5%) and 47 malicious websites (3%). On the contrary, Spain, despite being among the lowest in total website count, hosts only malicious sites, making it the country with the highest proportion of malicious websites. Spain represents 30% of the malicious sites in the dataset, marking it as the country most closely linked with malicious activity.



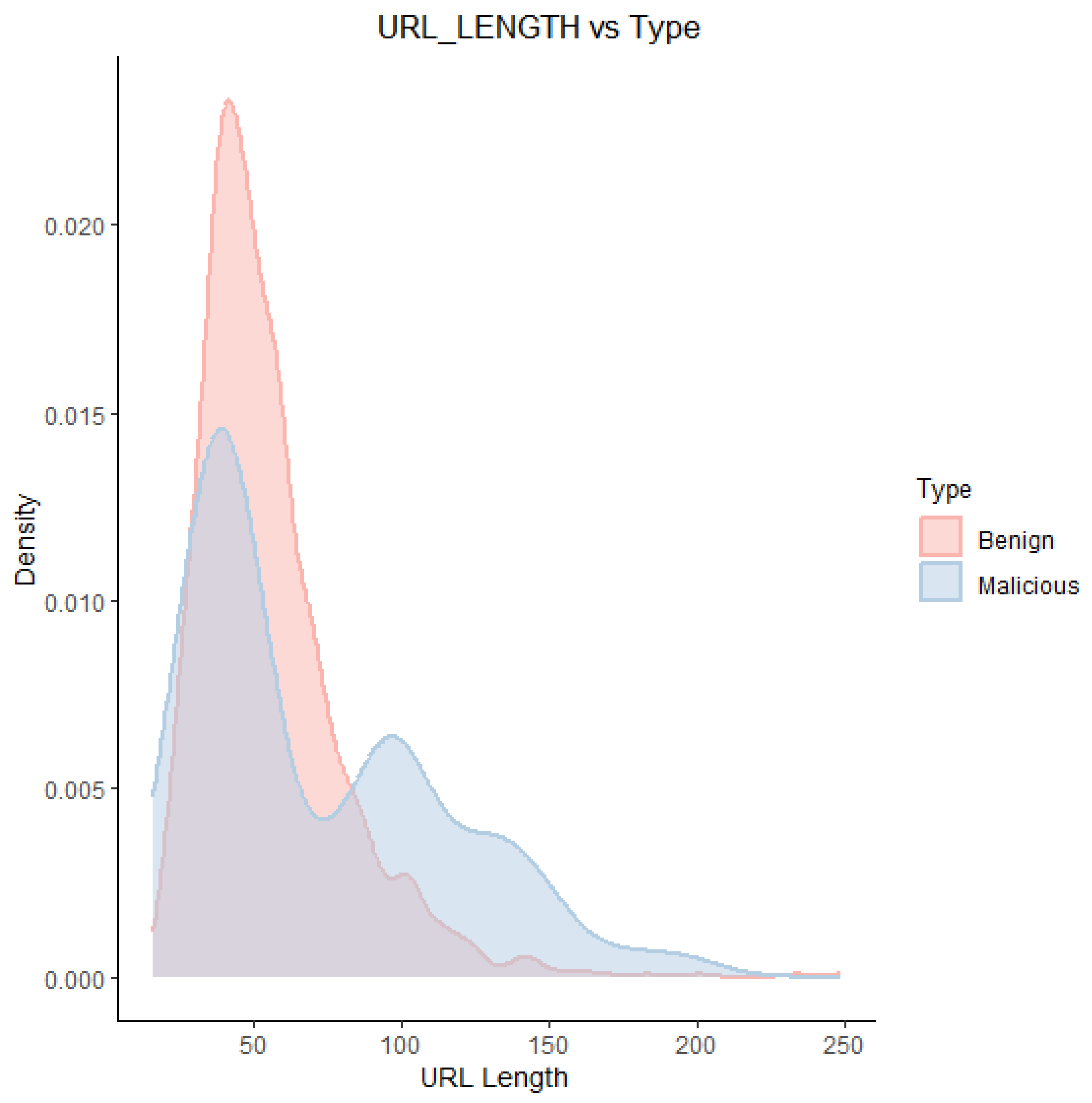
This bar plot illustrates the relationship between WHOIS_STATEPRO and Type, focusing on the top 7 states or provinces with the highest frequency of website types in the dataset. The results show that California is the most common state, with 418 benign websites (36.4%) and 15 malicious websites (1.3%). The second most frequent entry is "None" (indicating unknown), which has 290 benign websites but a significantly higher count of malicious sites, totaling 72. Conversely, Barcelona, the Spanish province with the fewest total websites, stands out with the highest proportion of malicious websites—62 in total and none that are benign. This makes Barcelona the region with the highest rate of malicious activity, accounting for 30% of all malicious sites in the dataset.



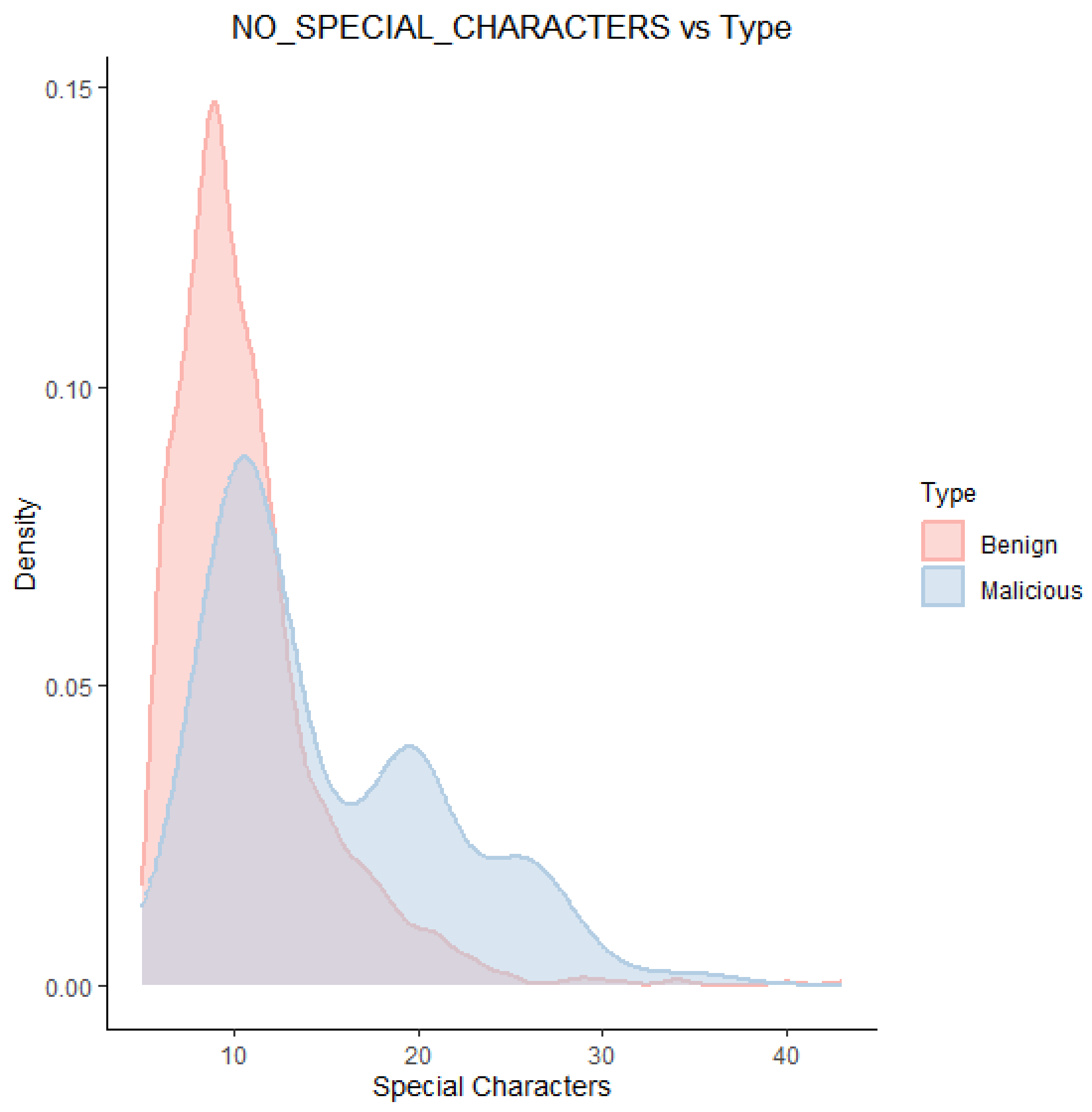
```
#KDE Plot for URL_LENGTH vs Type
ggplot(df, aes(x = URL_LENGTH, fill = factor(Type), color = factor(Type))) +
  geom_density(alpha = 0.5, size = 1) +
  scale_fill_brewer(
    name = "Type",
    palette = "Pastel1",
    labels = c("0" = "Benign", "1" = "Malicious")
  ) +
  scale_color_brewer(
    name = "Type",
    palette = "Pastel1",
    labels = c("0" = "Benign", "1" = "Malicious")
  ) +
  labs(title = "URL_LENGTH vs Type",
       x = "URL Length",
       y = "Density") +
  theme_classic() +
  theme(
    plot.title = element_text(size = 13, hjust = 0.6),
    axis.title.x = element_text(size = 10.5),
    axis.title.y = element_text(size = 10.5),
    legend.title = element_text(size = 10),
    legend.text = element_text(size = 9)
  )
```

```
> #KDE Plot for URL_LENGTH vs Type
> ggplot(df, aes(x = URL_LENGTH, fill = factor(Type), color = factor(Type))) +
+   geom_density(alpha = 0.5, size = 1) +
+   scale_fill_brewer(
+     name = "Type",
+     palette = "Pastel1",
+     labels = c("0" = "Benign", "1" = "Malicious")
+   ) +
+   scale_color_brewer(
+     name = "Type",
+     palette = "Pastel1",
+     labels = c("0" = "Benign", "1" = "Malicious")
+   ) +
+   labs(title = "URL_LENGTH vs Type",
+        x = "URL Length",
+        y = "Density") +
+   theme_classic() +
+   theme(
+     plot.title = element_text(size = 13, hjust = 0.6),
+     axis.title.x = element_text(size = 10.5),
+     axis.title.y = element_text(size = 10.5),
+     legend.title = element_text(size = 10),
+     legend.text = element_text(size = 9)
+   )
```

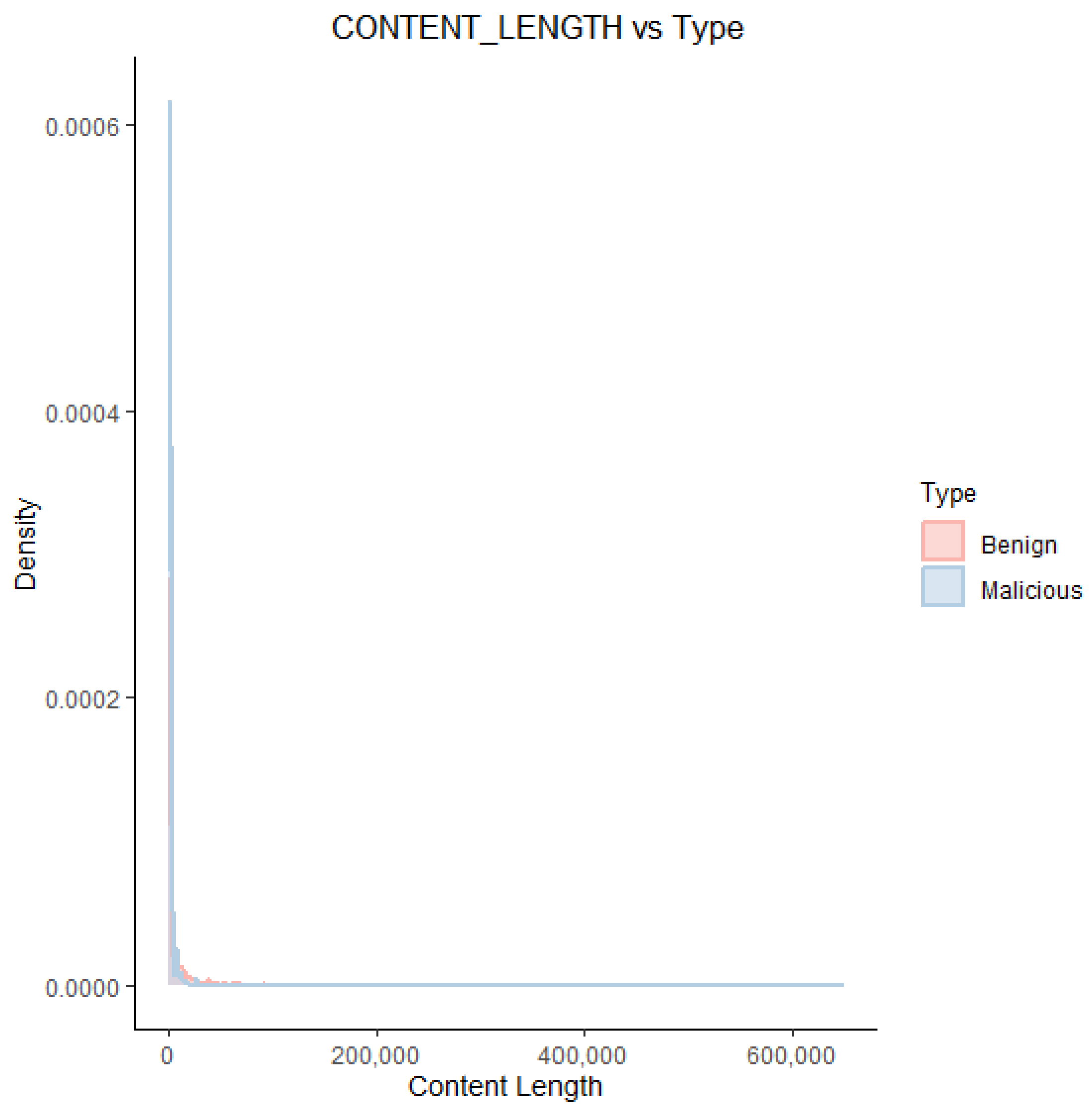
This code creates a KDE (Kernel Density Estimate) plot with ggplot2 to show the distribution of URL lengths for "Benign" and "Malicious" categories in the Type variable. The plot features URL_LENGTH on the x-axis, with density curves differentiated by color and fill according to the Type variable. The geom_density() function is used to generate the density curves with a semi-transparent fill (alpha = 0.5), which helps in visualizing areas where the categories overlap. The scale_fill_brewer() and scale_color_brewer() functions apply the "Pastel1" color palette from ColorBrewer, and set the legend labels to "Benign" and "Malicious" for clear category distinction. The labs() function adds a title and axis labels to improve the plot's clarity. Using theme_classic() ensures a clean and minimal design, while theme() adjusts the title size and positions, along with the axis and legend text sizes, for a refined look. This plot effectively compares the density distributions of URL lengths between the two categories, revealing differences in their distribution patterns. As previously noted for the bar plots of categorical features, this code will be similarly adapted for each numerical feature in the dataset. Given the minimal adjustments required for each numerical feature and to maintain a streamlined EDA process, I will not provide the code for every numerical feature visualization individually. This approach avoids redundancy and preserves the clarity of our exploratory data analysis.



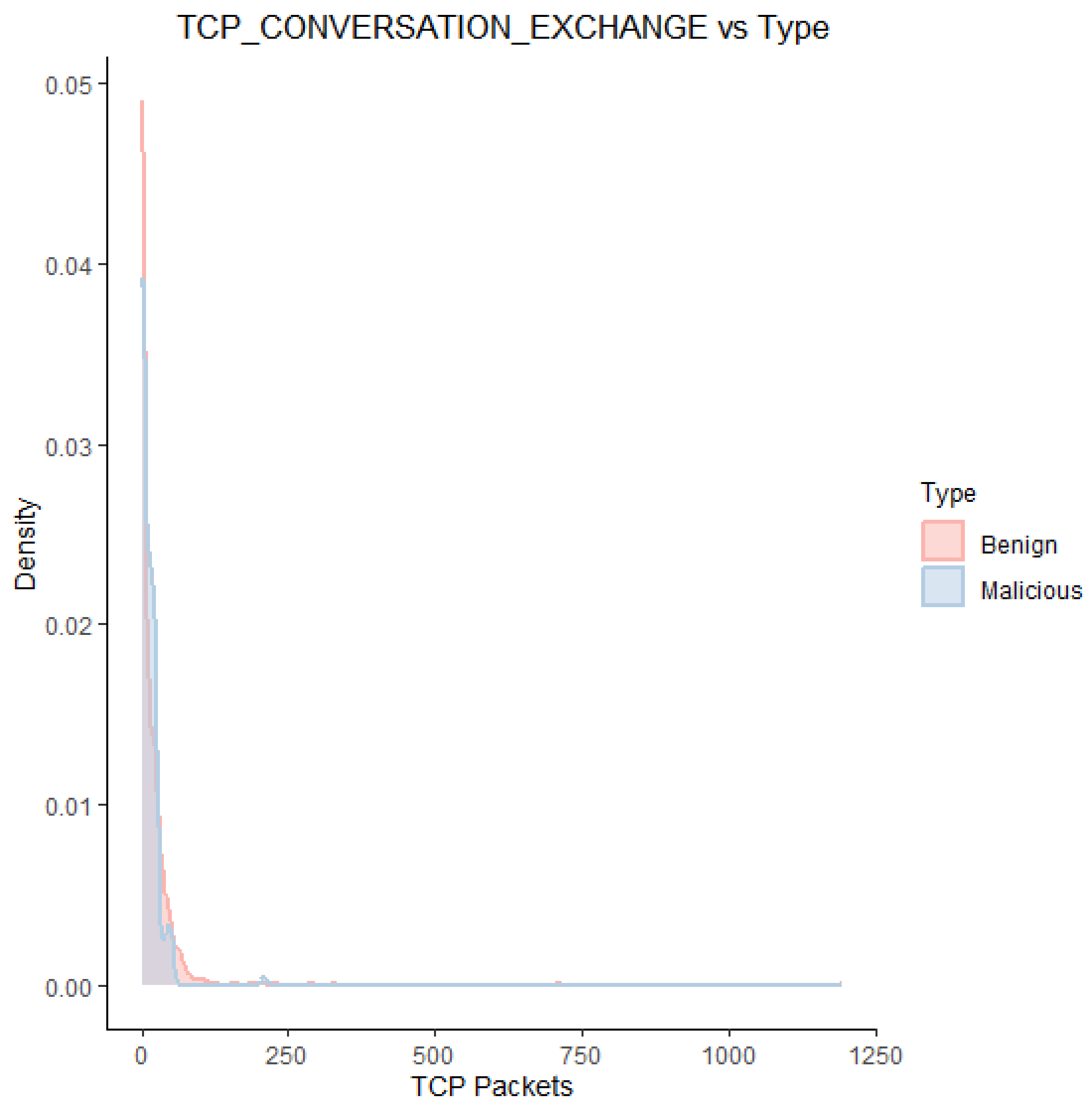
This Kernel Density Estimation (KDE) plot illustrates how the length of a URLs characters correlates with a websites type (benign or malicious). The URL length ranges from 16-249 characters; notably, the curve begins to ascend around the range of 18 characters for both websites and reaches its peak at approximately 48 characters for malicious websites, and around 50 characters for benign websites – indicates that a significant portion of malicious websites also have similar URL lengths as benign websites. However, the malicious curve is more spread out and shows a heavier tail extending toward longer URLs. This suggests that malicious websites are more likely to have a wider variety of URL lengths, including some that are significantly longer than benign websites.



The Kernel Density Estimation (KDE) plot depicts the relationship between the number of special characters in a URL and the likelihood of a website being benign or malicious. The number of special characters in the URLs varies from 5 to 43. The plot shows that malicious websites tend to have a broader distribution of special characters, with a noticeable peak at around 12 characters, while benign websites have a sharp peak near 10 characters. The broader and heavier tail of the malicious curve indicates that these sites often contain a higher and more varied number of special characters in their URLs. This suggests that URLs with a larger count of special characters are more commonly associated with malicious websites, potentially serving as a red flag for identifying them. Additionally, the spread of the malicious curve across a wider range of special character counts highlights the unpredictability and complexity of malicious URL structures compared to their benign counterparts.

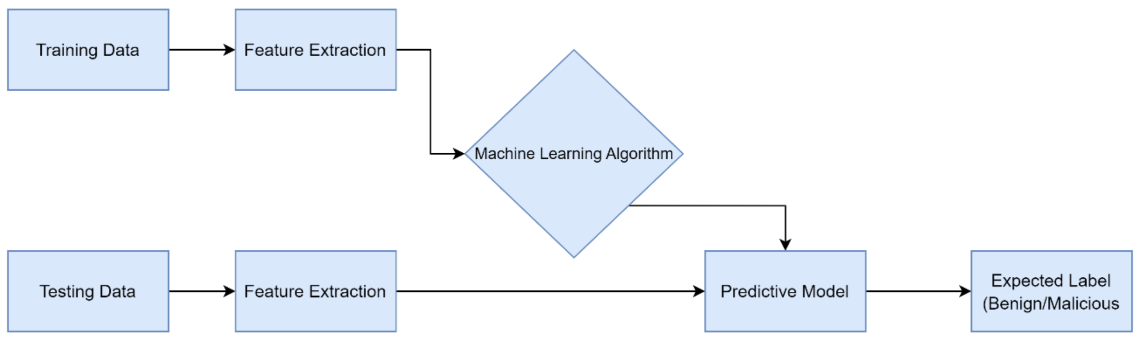


The Kernel Density Estimation (KDE) plot illustrates the relationship between the content length of the HTTP header and the probability of a website being classified as either benign or malicious. The content length values span from 0 to 649,263 bytes. The plot reveals a pronounced positive correlation with malicious websites, as indicated by the sharp peak and the dominance of the blue curve representing the malicious category. In contrast, there is minimal presence of the red curve, which corresponds to benign websites, suggesting that benign sites rarely exhibit significant content length in their HTTP headers. This observation implies that malicious websites typically have a substantially larger and more varied content size in their HTTP headers, which may serve as a distinguishing factor. The broader tail of the malicious curve further emphasizes that these sites not only have larger content lengths but also display a wide range of values, indicating a more complex and extensive use of HTTP header content compared to benign websites.



The Kernel Density Estimation (KDE) plot displays the relationship between the number of TCP packets exchanged during communication between a server and a honeypot client, and the likelihood of a website being labelled as benign or malicious. The range of TCP packets spans from 0 to 1,194. Both benign and malicious websites peak at the same TCP packet count, but the benign curve is notably sharper, indicating a higher concentration of TCP packets around the peak value. In contrast, the curve for malicious websites, while peaking at the same point, features a much longer tail, extending across the entire range of TCP packet values. This suggests that malicious websites are more likely to exhibit a wider variety of TCP packet exchanges, with some reaching significantly higher counts than those typically seen in benign websites.

Methodology



1. **Training Data**: Aggregate a dataset containing labelled examples of both benign and malicious websites to conduct training.
2. **Feature Extraction**: Extract relevant features from website data, such as URL length, WHOIS information, and website content-based attributes, to create feature vectors.
3. **Machine Learning Algorithm:** Use the labelled feature vectors from the training data to train a machine learning algorithm. During training the algorithm learns to distinguish between benign and malicious websites based on the extracted features (predictor variables).
4. **Testing Data**: Use a separate dataset (testing dataset) containing unlabelled website data. Simulating real world scenarios in predicting the type of website (Benign/Malicious).
5. **Feature Extraction**: Similar to the training phase, extract relevant features from the testing dataset.
6. **Predictive Model:** Apply the trained machine learning model to the feature vectors from the testing dataset to generate predictive labels, classifying each website as either benign or malicious.
7. **Expected Label:** The model outputs the classification of each website, allowing for the detection of potentially malicious websites.

Data Pre-Processing/Technical Demonstration

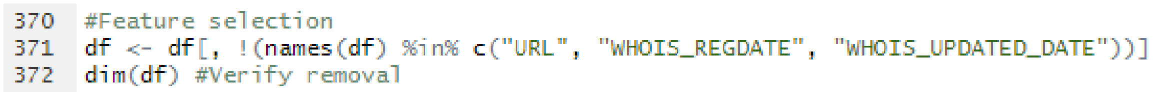
For feature selection we are going to remove irrelevant features which would hinder the models performance:

URL:

- The URL feature was excluded from the dataset because it acts as a unique identifier for each website. Unlike other features, URLs do not inherently provide information about the general characteristics of maliciousness or benignity.
- Since URLs are inherently unique and specific to each website, they do not contribute to identifying generalizable patterns that can be used for classification; irrelevant feature as it does not provide any valuable information for analysis nor is it beneficial for the modelling process.
- Including URL as a feature could introduce unnecessary complexity into the model without adding meaningful insights, potentially leading to overfitting and reduced model performance.

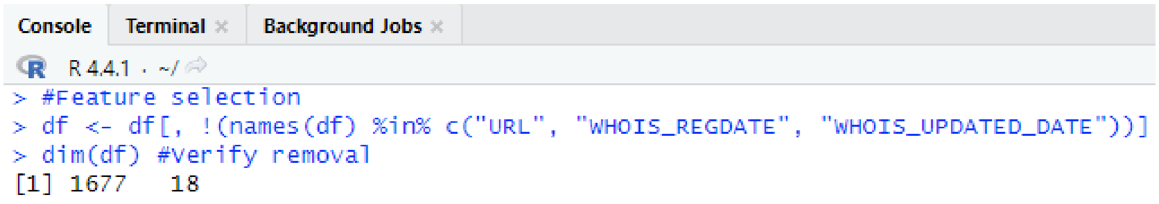
WHOIS_REGDATE & WHOIS_UPDATED_DATE:

- The WHOIS_REGDATE and WHOIS_UPDATED_DATE features were removed due to their temporal nature. These dates reflect the registration and last update times of domain names, which, while potentially useful in some contexts, do not directly correlate with malicious activity in a predictive manner.
- Temporal features such as these often do not capture patterns that are indicative of malicious behaviour and can be challenging to convert into numerical representations that are useful for model training.
- By excluding these date-related features, the model focuses on more relevant and actionable attributes, enhancing its interpretability and performance while reducing the risk of overfitting and computational overhead.



```r
#Feature selection
df <- df[, !(names(df) %in% c("URL", "WHOIS_REGDATE", "WHOIS_UPDATED_DATE"))]
dim(df) #Verify removal
```

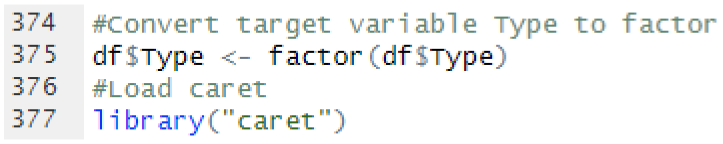
- Removing URL, WHOIS_REGDATE, and WHOIS_UPDATED_DATE. This was achieved by retrieving column names by using the names() function, identifying which columns to exclude, negating this information to retain the desired columns, and then updating the data frame to reflect these changes. This method ensures that only the relevant columns are kept, streamlining the data for further analysis or modeling.



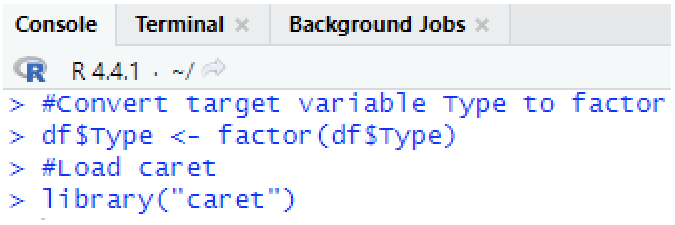
```
> #Feature selection
> df <- df[, !(names(df) %in% c("URL", "WHOIS_REGDATE", "WHOIS_UPDATED_DATE"))]
> dim(df) #verify removal
[1] 1677   18
```

- By examining the output, we can gauge the removal of these particular features was successful.

Overall, this is mainly done to improve the models performance. By refining our dataset in this way, we are ensuring that only the most relevant and useful information is used to build our classification models - maximizing our models evaluation metrics.

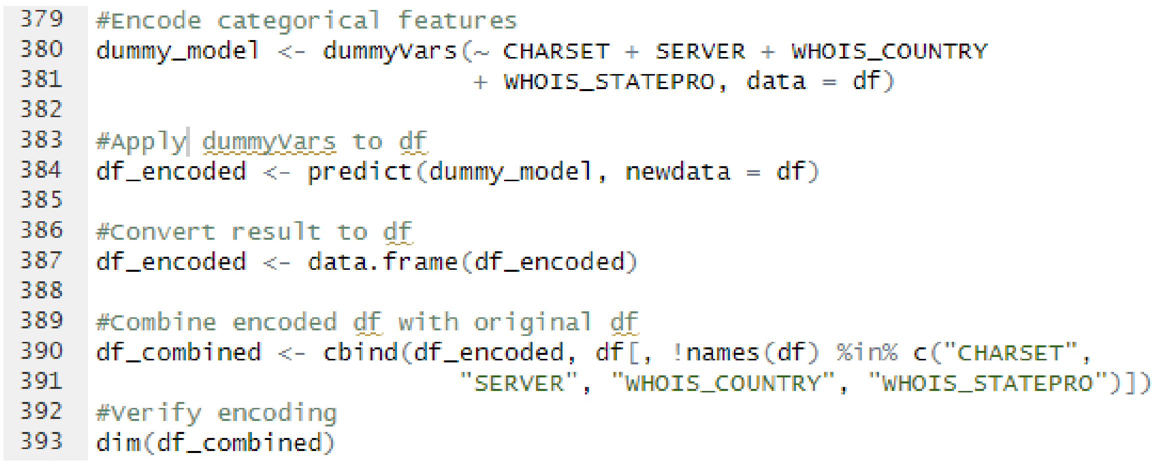


```
#Convert target variable Type to factor
df$Type <- factor(df$Type)
#Load caret
library("caret")
```



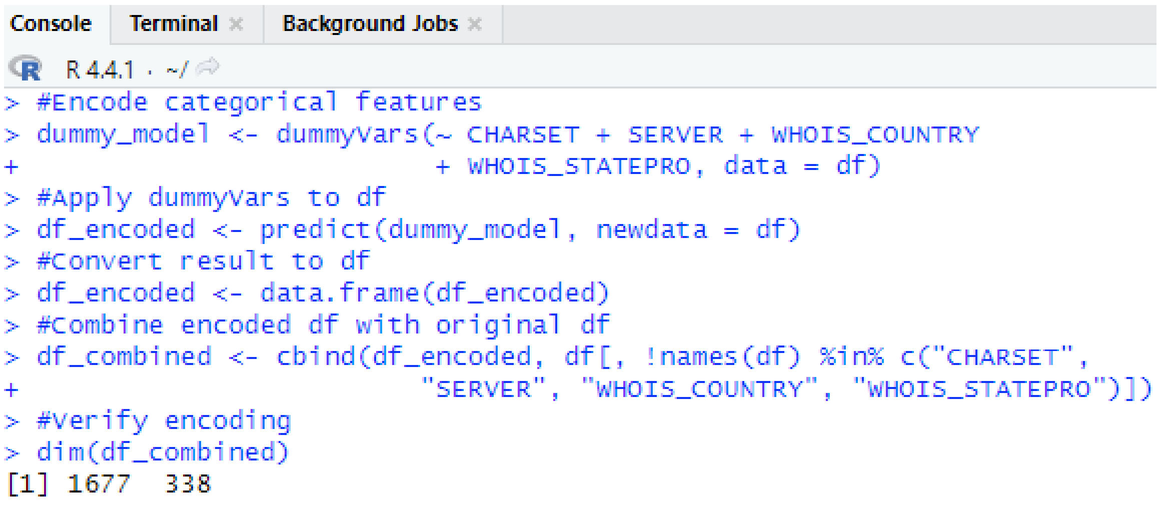
```
> #Convert target variable Type to factor
> df$Type <- factor(df$Type)
> #Load caret
> library("caret")
```

- I convert the target feature label to a factor to guarantee consistency as it is a categorical variable; for the classification we will be doing later, this ensures that our models interpret and process the data correctly. It prevents misinterpretation, ensures correct computation of performance metrics, and avoids errors or warnings from the algorithm. The df$Type <- factor (df$Type) line converts the Type column of the entire dataset into a factor by utilizing the factor() function – this is necessary as machine learning algorithms cannot process words, only numbers.
- I also load the caret library, which provides functions to streamline the modelling process to create predictive models for classification. It is very convenient and efficient when it comes to conducting models and the overall functionality it provides makes it the preferred choice.



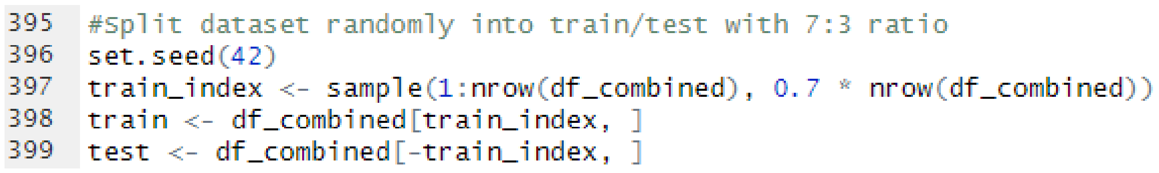
```
#Encode categorical features
dummy_model <- dummyVars(~ CHARSET + SERVER + WHOIS_COUNTRY
                         + WHOIS_STATEPRO, data = df)

#Apply dummyVars to df
df_encoded <- predict(dummy_model, newdata = df)

#Convert result to df
df_encoded <- data.frame(df_encoded)

#Combine encoded df with original df
df_combined <- cbind(df_encoded, df[, !names(df) %in% c("CHARSET",
                        "SERVER", "WHOIS_COUNTRY", "WHOIS_STATEPRO")])
#Verify encoding
dim(df_combined)
```

Now we have to convert categorical data to numerical data. This is necessary because the features CHARSET, SERVER, WHOIS_COUNTRY, and WHOIS_STATEPRO contain textual values, and machine learning algorithms require numerical input for processing. I'll be utilizing the one-hot encoding dummyVars() function from the caret package to handle the conversion process. This function transforms categorical data into dummy variables, creating binary (0/1) variables for each unique value in the categorical features. This is achieved by first defining a model with dummyVars(), which specifies the categorical features to be encoded. Next, the predict() function applies this model to the data, generating a matrix of binary variables. The result is then converted into a data frame and combined with the rest of the original dataset, excluding the original categorical columns. The final output is a fully numerical dataset that is ready for machine learning algorithms.



```
> #Encode categorical features
> dummy_model <- dummyVars(~ CHARSET + SERVER + WHOIS_COUNTRY
+                          + WHOIS_STATEPRO, data = df)
> #Apply dummyVars to df
> df_encoded <- predict(dummy_model, newdata = df)
> #Convert result to df
> df_encoded <- data.frame(df_encoded)
> #Combine encoded df with original df
> df_combined <- cbind(df_encoded, df[, !names(df) %in% c("CHARSET",
+                        "SERVER", "WHOIS_COUNTRY", "WHOIS_STATEPRO")])
> #Verify encoding
> dim(df_combined)
[1] 1677  338
```

By observing the output of the dim() function, we can see there are 338 variables – one-hot encoding is successful.

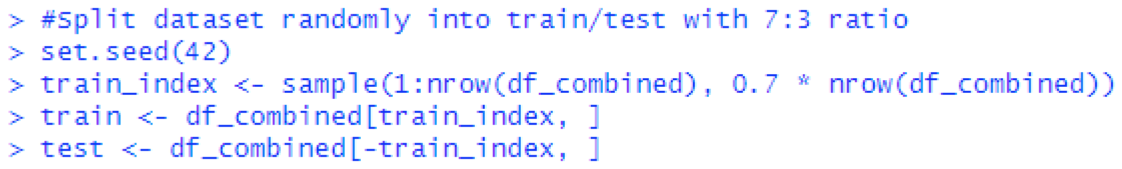


```
#Split dataset randomly into train/test with 7:3 ratio
set.seed(42)
train_index <- sample(1:nrow(df_combined), 0.7 * nrow(df_combined))
train <- df_combined[train_index, ]
test <- df_combined[-train_index, ]
```

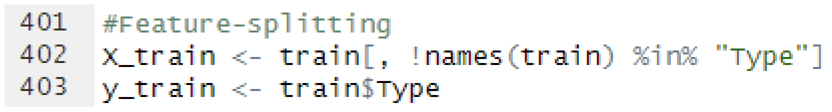
I follow up by randomly splitting the dataset into train/test splits with a ratio of 7:3. This is achieved by utilizing various functions: set.seed(), sample(). The set.seed() function is employed to ensure the random split is reproducible, specifically by setting a set.seed value (42 in our case); ensuring consistency as the same results will be produced. Next, I use the sample() function to generate random indices for the training set. The 1:nrow(df_combined) argument creates a sequence of integers from 1 to the total number of rows in the df_combined dataset, where each integer corresponds to a row index. The sample() function then randomly selects 70% of these indices, computed as 0.7 * nrow(df_combined), and stores them in the train_index variable. This variable now holds the indices of the rows that will be assigned to the training set – ensuring 70% of the dataset to the training split.

I create the train and test splits with a 7:3 ratio by assigning 70% of the dataset to the training set and the remaining 30% to the testing set. By employing the train_index variable we created previously to the train and test variables, specifically for the training set, the train_index is applied to select 70% of the rows from df_combined to train (df_combined[train_index,]). For the testing set, the remaining 30% of the dataset is selected by excluding these indices, using the minus sign before train_index (df_combined[-train_index,]).

Output:



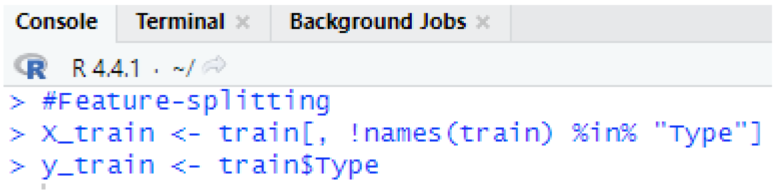
```
> #Split dataset randomly into train/test with 7:3 ratio
> set.seed(42)
> train_index <- sample(1:nrow(df_combined), 0.7 * nrow(df_combined))
> train <- df_combined[train_index, ]
> test <- df_combined[-train_index, ]
```



```
#Feature-splitting
X_train <- train[, !names(train) %in% "Type"]
y_train <- train$Type
```

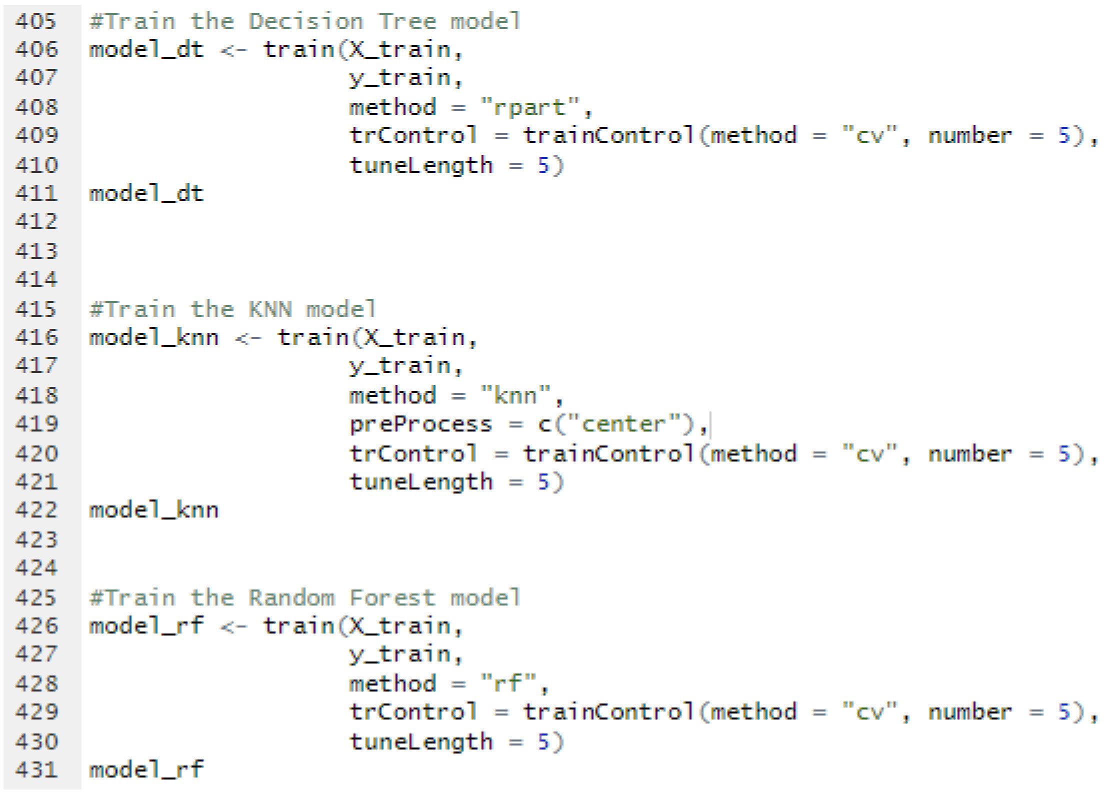
I split the features into X_train and y_train for the training set. X_train represents every feature used to classify the outcome (predictor variables) – all features besides Type. In contrast, y_train represents the Type feature which is the target feature (dependent variable) – indicates whether a website is malicious or benign. This target feature is what the models will be trained to classify based on the predictor variables. This was achieved by using certain functions/arguments; X_train <- train[, !names(train) %in% "Type"]: This line extracts all the columns from the train dataset, excluding the Type column (which is the target variable). The resulting X_train contains only the predictor variables. The y_train <- train$Type line extracts the target variable for the training set and stores it in the y_train variable.

Output:



```
> #Feature-splitting
> X_train <- train[, !names(train) %in% "Type"]
> y_train <- train$Type
```

- **Decision Tree, K-Nearest Neighbor, Random Forest Training:**



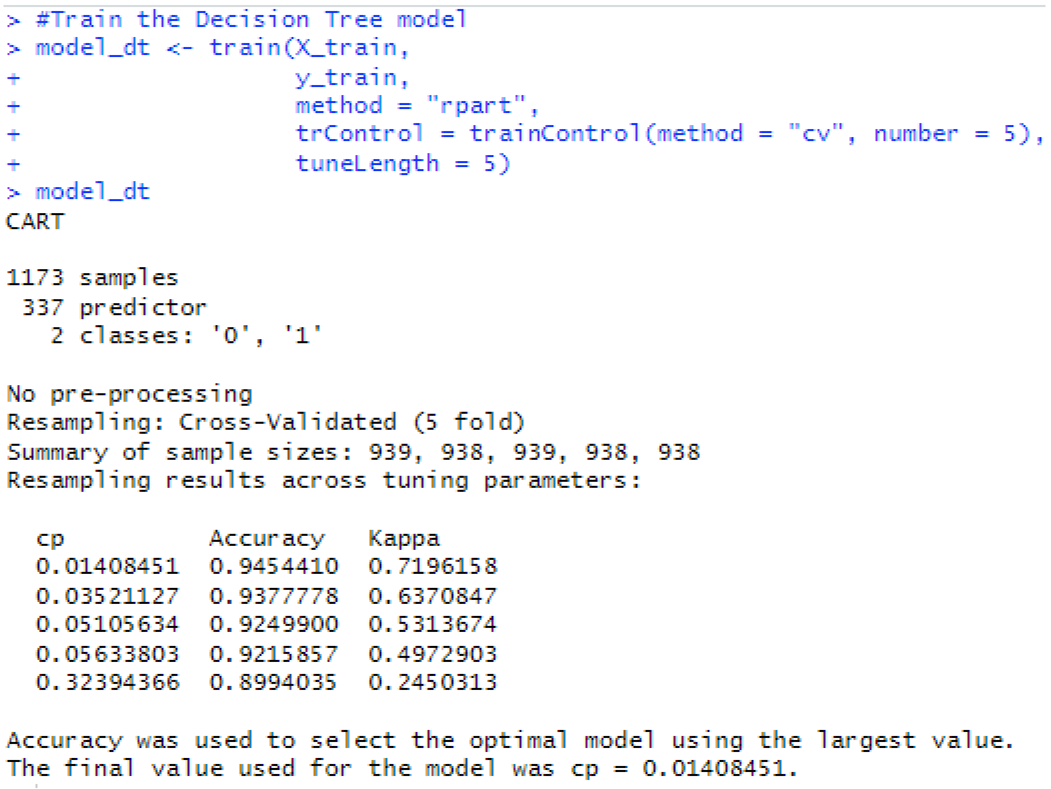
```
#Train the Decision Tree model
model_dt <- train(X_train,
                  y_train,
                  method = "rpart",
                  trControl = trainControl(method = "cv", number = 5),
                  tuneLength = 5)
model_dt



#Train the KNN model
model_knn <- train(X_train,
                  y_train,
                  method = "knn",
                  preProcess = c("center"),
                  trControl = trainControl(method = "cv", number = 5),
                  tuneLength = 5)
model_knn


#Train the Random Forest model
model_rf <- train(X_train,
                  y_train,
                  method = "rf",
                  trControl = trainControl(method = "cv", number = 5),
                  tuneLength = 5)
model_rf
```

Finally, we train our chosen models by using the training set. The training data enables the models to learn patterns and relationships within the predictor variables (X_train) that are indicative of the target variable (y_train). This process equips the models with the ability to classify websites as malicious or benign. The train() function is used to build our models by leveraging the predictor variables in X_train and the corresponding target labels in y_train.

Specifically, the method = "rpart", "knn", and "rf" parameters specifies that each of our chosen algorithms should be conducted (rpart: decision tree, knn: k-nearest neighbor, rf: random forest). The trControl = trainControl(method = "cv", number = 5) argument calls for cross-validation, specifically k-fold cross-validation (default) with 5 folds – splits the training data into 5 subsets, training the model 5 times, and validating on each fold ensuring a reliable model evaluation. The tuneLength = 5 argument specifies that the model should be tuned over 5 different values of parameters. Basically, it streamlines the process of finding the optimal model configuration by selecting the configuration that yields the highest performance based on a chosen metric (accuracy by default) – hyperparameter tuning essentially, eliminating the need for manual experimentation, making it extremely convenient.

The KNN model has one different parameter from the other two models: preprocess = c("center"). This parameter centers the features by subtracting their mean, which standardizes the data and ensures that each feature contributes equally to the distance calculations. By doing so, it can improve the performance of the KNN model by making the distance metrics more accurate and ensuring that features with different scales do not disproportionately affect the results. In contrast, Decision Trees and Random Forests do not require centering because they use splitting criteria based on feature values and are not affected by feature scales.

Decision Tree Training results:



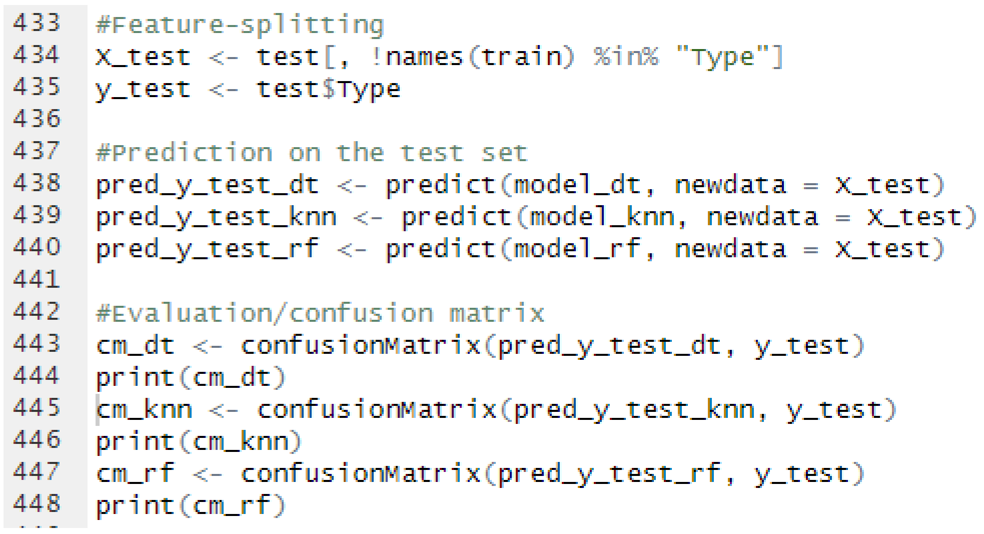
```
> #Train the Decision Tree model
> model_dt <- train(X_train,
+                   y_train,
+                   method = "rpart",
+                   trControl = trainControl(method = "cv", number = 5),
+                   tuneLength = 5)
> model_dt
CART

1173 samples
 337 predictor
   2 classes: '0', '1'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 939, 938, 939, 938, 938
Resampling results across tuning parameters:

  cp          Accuracy   Kappa
  0.01408451  0.9454410  0.7196158
  0.03521127  0.9377778  0.6370847
  0.05105634  0.9249900  0.5313674
  0.05633803  0.9215857  0.4972903
  0.32394366  0.8994035  0.2450313

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was cp = 0.01408451.
```

After training the model, you can access the results of the training process by referencing the models variable, in my case:'model_dt'; model_dt contains performance metrics from the cross-validation process. This includes metrics like accuracy, kappa, or other evaluation criteria based on the metric argument specified in the train function.

When you print model_dt, it shows a detailed summary of the training process, including:

- **Best Model**: The configuration of the Decision Tree model that performed the best according to cross-validation.
- **Performance Scores**: Metrics such as accuracy or kappa, indicating how well the model is expected to perform on unseen data.
- **Tuning Results**: Information about how different parameter settings impacted model performance.

This output provides a comprehensive view of the model's training performance and helps in selecting the best model configuration based on cross-validation results.



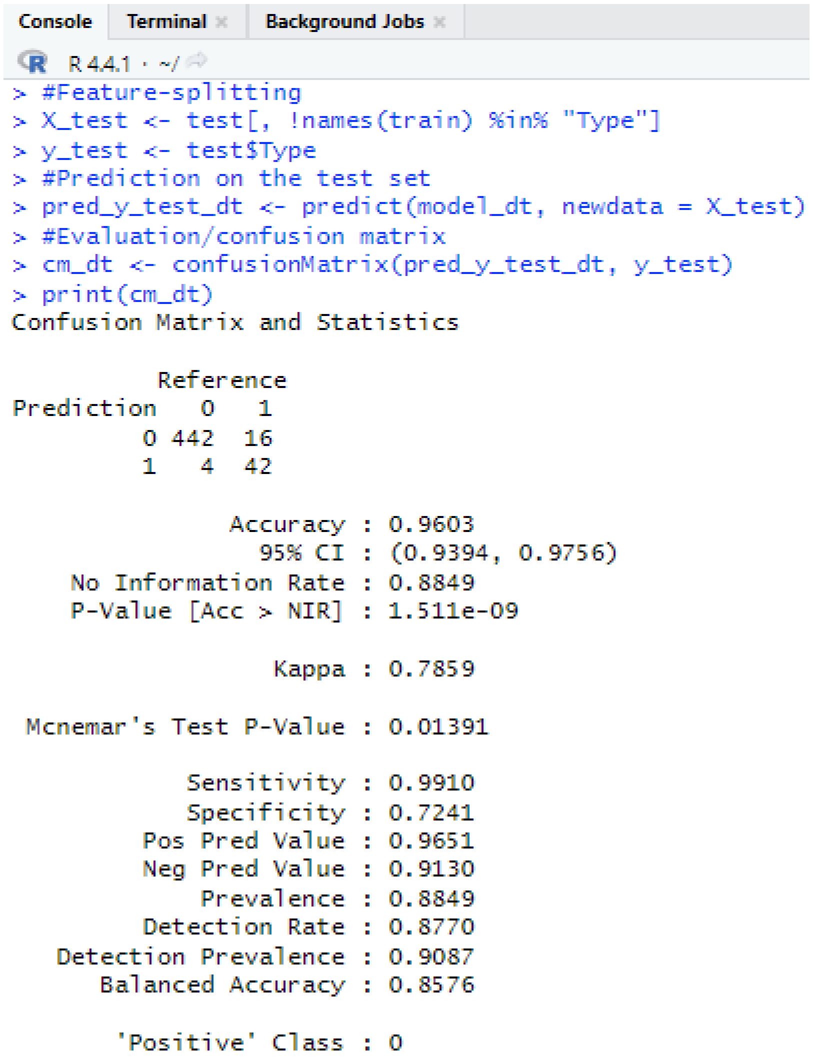
```
#Feature-splitting
X_test <- test[, !names(train) %in% "Type"]
y_test <- test$Type

#Prediction on the test set
pred_y_test_dt <- predict(model_dt, newdata = X_test)
pred_y_test_knn <- predict(model_knn, newdata = X_test)
pred_y_test_rf <- predict(model_rf, newdata = X_test)

#Evaluation/confusion matrix
cm_dt <- confusionMatrix(pred_y_test_dt, y_test)
print(cm_dt)
cm_knn <- confusionMatrix(pred_y_test_knn, y_test)
print(cm_knn)
cm_rf <- confusionMatrix(pred_y_test_rf, y_test)
print(cm_rf)
```

In this last task to show the test and performance calculations, I split the test dataset into predictor variables (X_test) and the target variable (y_test). This separation is essential for evaluating the model's performance on unseen data. X_test contains all the predictor variables, excluding the target variable (label). These predictors are the features that our models use to make classifications. y_test represents the target variable, which indicates whether a website is malicious or benign. Our models were trained to classify this target variable based on the patterns learned from the training data. Essentially, it is the exact same protocol as the previous splitting we applied for train, only this time its for test.

To assess the model's performance on the test data, I use the predict() function to generate predictions for the test set. Specifically, predict(model_dt, newdata = X_test) applies the trained decision tree model (model_dt) to the test set features (X_test). The function returns the predicted class labels (benign and malicious), which are stored in the variable pred_y_test_dt.

Finally, I evaluate the decision tree model's performance by comparing the predicted labels (pred_y_test_dt) with the actual labels (y_test) from the test set. This comparison is done using the confusionMatrix() function, which produces a confusion matrix and various performance metrics (e.g., accuracy, sensitivity, specificity). The confusion matrix provides a detailed summary of the model's classification performance, showing how many instances were correctly and incorrectly classified as benign and malicious.

Decision Tree Testing results:

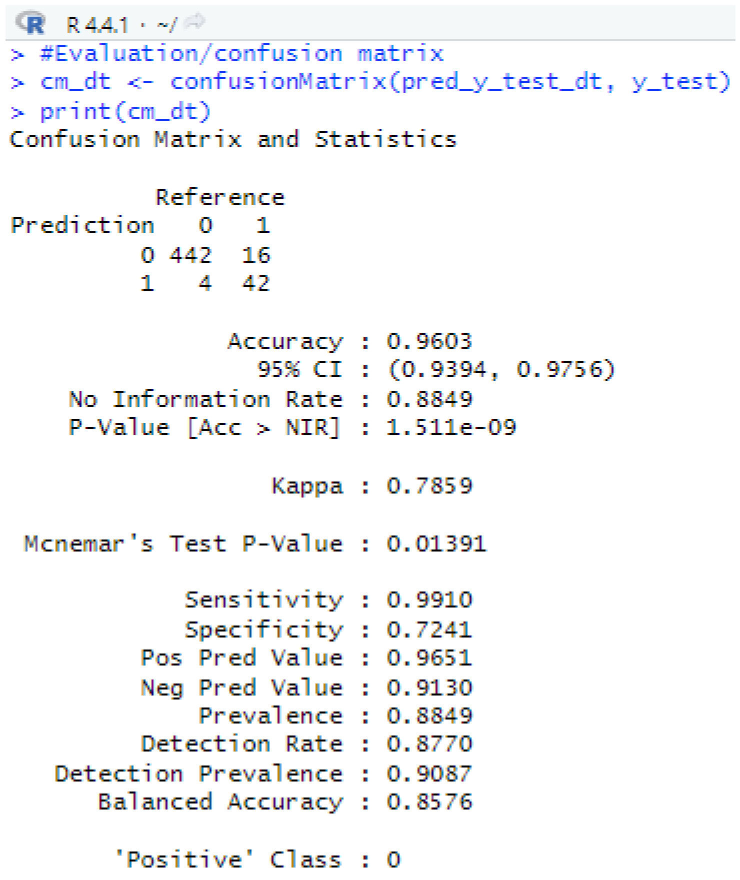


```
> #Feature-splitting
> X_test <- test[, !names(train) %in% "Type"]
> y_test <- test$Type
> #Prediction on the test set
> pred_y_test_dt <- predict(model_dt, newdata = X_test)
> #Evaluation/confusion matrix
> cm_dt <- confusionMatrix(pred_y_test_dt, y_test)
> print(cm_dt)
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 442  16
         1   4  42

               Accuracy : 0.9603
                 95% CI : (0.9394, 0.9756)
    No Information Rate : 0.8849
    P-Value [Acc > NIR] : 1.511e-09

                  Kappa : 0.7859

 Mcnemar's Test P-Value : 0.01391

            Sensitivity : 0.9910
            Specificity : 0.7241
         Pos Pred Value : 0.9651
         Neg Pred Value : 0.9130
             Prevalence : 0.8849
         Detection Rate : 0.8770
   Detection Prevalence : 0.9087
      Balanced Accuracy : 0.8576

       'Positive' Class : 0
```

Besides this decision tree algorithm, I am using the k-nearest neighbour and random forest algorithms. Essentially, the entire process for classification for all my models is basically the exact same, the only difference between them are the variables. Similarly, if I were to run the other two machine learning algorithms – it would be done in the exact same fashion as the decision tree model here.

Performance Evaluation

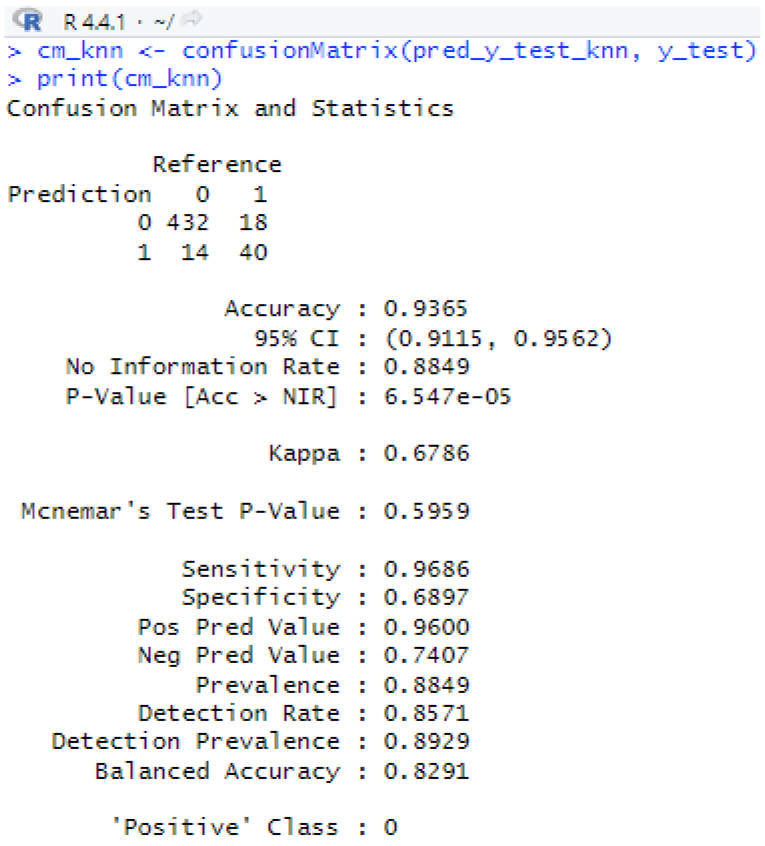
Before I begin with measuring the performance of each model, I'm solely focusing on the testing set metrics as they represent the model's performance on predicting unseen data, simulating real-world scenarios on classifying the website type in unseen websites — benign or malicious; immense importance placed on the testing set.



```
R 4.4.1 · ~/
> #Evaluation/confusion matrix
> cm_dt <- confusionMatrix(pred_y_test_dt, y_test)
> print(cm_dt)
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 442  16
         1   4  42

               Accuracy : 0.9603
                 95% CI : (0.9394, 0.9756)
    No Information Rate : 0.8849
    P-Value [Acc > NIR] : 1.511e-09

                  Kappa : 0.7859

 Mcnemar's Test P-Value : 0.01391

            Sensitivity : 0.9910
            Specificity : 0.7241
         Pos Pred Value : 0.9651
         Neg Pred Value : 0.9130
             Prevalence : 0.8849
         Detection Rate : 0.8770
   Detection Prevalence : 0.9087
      Balanced Accuracy : 0.8576

       'Positive' Class : 0
```

Decision Tree model:

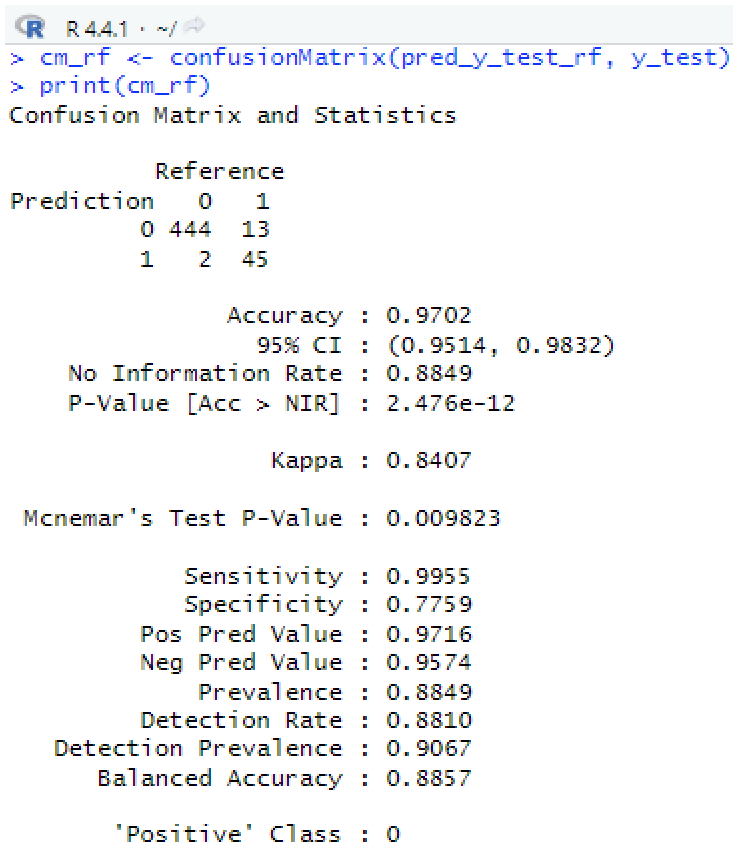
- **Accuracy:** 96.03%
- **Recall/Sensitivity:** 99.10%
- **Specificity:** 72.41%
- **Precision/Pos Pred Value:** 96.51%
- **Balanced Accuracy:** 85.76%



```
R 4.4.1 · ~/
> cm_knn <- confusionMatrix(pred_y_test_knn, y_test)
> print(cm_knn)
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 432  18
         1  14  40

               Accuracy : 0.9365
                 95% CI : (0.9115, 0.9562)
    No Information Rate : 0.8849
    P-Value [Acc > NIR] : 6.547e-05

                  Kappa : 0.6786

 Mcnemar's Test P-Value : 0.5959

            Sensitivity : 0.9686
            Specificity : 0.6897
         Pos Pred Value : 0.9600
         Neg Pred Value : 0.7407
             Prevalence : 0.8849
         Detection Rate : 0.8571
   Detection Prevalence : 0.8929
      Balanced Accuracy : 0.8291

       'Positive' Class : 0
```

K-Nearest Neighbor model:

- **Accuracy:** 93.65%
- **Recall/Sensitivity:** 96.86%
- **Specificity:** 68.97%
- **Precision/Pos Pred Value:** 96.00%
- **Balanced Accuracy:** 82.91%


R 4.4.1 · ~/

```
> cm_rf <- confusionMatrix(pred_y_test_rf, y_test)
> print(cm_rf)
Confusion Matrix and Statistics

          Reference
Prediction   0   1
         0 444  13
         1   2  45

               Accuracy : 0.9702
                 95% CI : (0.9514, 0.9832)
    No Information Rate : 0.8849
    P-Value [Acc > NIR] : 2.476e-12

                  Kappa : 0.8407

 Mcnemar's Test P-Value : 0.009823

            Sensitivity : 0.9955
            Specificity : 0.7759
         Pos Pred Value : 0.9716
         Neg Pred Value : 0.9574
             Prevalence : 0.8849
         Detection Rate : 0.8810
   Detection Prevalence : 0.9067
      Balanced Accuracy : 0.8857

       'Positive' Class : 0
```

Random Forest model:

- **Accuracy:** 97.02%
- **Recall/Sensitivity:** 99.55%
- **Specificity:** 77.59%
- **Precision/Pos Pred Value:** 97.16%
- **Balanced Accuracy:** 88.57%

	Accuracy	Recall	Specificity	Precision	Balanced Accuracy
Decision Tree Model	**96.03%**	**99.10%**	**72.41%**	**96.51%**	**85.76%**
K-Nearest Neighbor Model	**93.65%**	**96.86%**	**68.97%**	**96.00%**	**82.91%**
Random Forest Model	**97.02%**	**99.55%**	**77.59%**	**97.16%**	**88.57%**

Ultimately, by evaluating the performance metrics, the Random Forest model stands out as the superior choice for malicious website detection. It excels across all key metrics, demonstrating its robustness and effectiveness in classifying websites accurately.

- **High Accuracy**: The Random Forest model achieves an impressive accuracy of 97.02%. This high level of accuracy indicates that the model performs exceptionally well in distinguishing between malicious and benign websites overall. It correctly classifies the vast majority of websites, providing a reliable assessment of the model's performance in real-world scenarios.

- **Exceptional Recall/Sensitivity**: With a recall of 99.55%, the Random Forest model exhibits an outstanding ability to detect benign websites. This high recall ensures that the model identifies almost all true benign websites (positive class), minimizing the risk of missing potentially safe sites. This is crucial for security applications where detecting as many benign sites as possible is essential to avoid falsely labelling them as threats.

- **Balanced Specificity**: The model's specificity of 77.59% is particularly noteworthy. Specificity measures the model's ability to correctly identify malicious websites (negative class). A high specificity score indicates that the Random Forest model effectively distinguishes malicious sites from benign ones, ensuring that legitimate websites are not incorrectly flagged as threats. This balance is vital for maintaining user trust and avoiding unnecessary disruptions – this score is excellent considering the major class imbalance.

- **High Precision**: With a precision of 97.16%, the Random Forest model also demonstrates strong performance in correctly identifying benign websites. This means that when the model classifies a website as benign, there is a very high probability that it is indeed benign, further reducing the risk of false positives.

- **Balanced Accuracy**: The model's balanced accuracy of 88.57% reflects its overall performance in identifying both malicious and benign websites. By averaging recall and specificity, this metric provides a comprehensive view of the model's ability to perform well across both classes. It highlights the model's effectiveness in maintaining a strong performance in both detecting threats and minimizing false positives.

Given these metrics, the Random Forest model is highly recommended for malicious website detection. Its exceptional recall ensures that most benign websites are accurately identified, while its strong specificity means it effectively detects malicious sites without flagging benign ones incorrectly. The model's high specificity reduces false positives, minimizing user inconvenience and preventing unnecessary blocks of legitimate sites. Overall, the Random Forest model's balance of high recall and specificity makes it an excellent choice for robust and user-friendly malicious website detection.

Conclusion

Vigorous testing and analysis were conducted to identify the most effective model for detecting malicious websites, and the Random Forest algorithm emerged as the best choice. The decisive factor in selecting Random Forest was its exceptional performance in identifying malicious websites, especially in the context of an imbalanced dataset. While traditional metrics such as accuracy, precision, and recall provided some insight, they were less informative due to the imbalance between benign and malicious websites. Accuracy, in particular, was misleading as it was skewed by the high number of correct predictions for benign websites. The Random Forest model demonstrated superior performance in addressing this imbalance, making it a standout choice for this classification problem. The Random Forest algorithm's ability to maintain high recall ensures that most benign websites are detected, while its strong specificity confirms its effectiveness in correctly identifying malicious websites. This balance is crucial for minimizing false positives and ensuring that legitimate sites are not incorrectly flagged as threats. In summary, the Random Forest model offers a robust solution for malicious website detection. Its effectiveness in handling imbalanced data, along with its high recall and balanced specificity, makes it an excellent choice for this application. The model’s performance justifies its deployment for detecting and classifying malicious websites, ensuring a high level of accuracy and reliability in cybersecurity efforts.

Bibliography

PWG, 2023. Anti-Phishing Working Group. Available at: https://apwg.org/ [Accessed 16 August 2024].

Gaur, D., 2021. A Guide to Any Classification Problem. Kaggle. Available at: https://www.kaggle.com/code/durgancegaur/a-guide-to-any-classification-problem [Accessed 16 August 2024].

Lewis et al., 2020. The Hidden Costs of Cybercrime. Available at: https://www.csis.org/analysis/hidden-costs-cybercrime [Accessed 16 August 2024].