# Project 9 - HTTPS Traffic

SALVATORE DI FRANCO - S346785, ANTONIO DI PONZIO - S347920, ARMAN SHEIKHHOSSEINI - S342405, and DANNY DI GIROLAMO - S331374

## 1 Introduction

The project aims to perform an analysis of the internet traffic generated by thousand of users in a single day. To achieve that, a passive packet sniffer tool (Tstat) has been used to provide statistics about different packet flows. In particular, we want to understand what kind of services are reached by users looking at the collected traffic statistics. Moreover, we want to estimate the domain name, try to aggregate together different services and try to predict the amount of bytes transmitted and the round trip time of the flows. To reach this goal, we decided to use supervised and unsupervised learning techniques to predict the domain name of the different sites, and regression techniques for transmitted bytes and round trip time. Moreover, to get more accurate results and to compare different techniques, for each learning style we decided to use different Machine Learning methods. For the supervised learning, we used decision tree, random forest and K-Nearest Neighbors. Instead, for the unsupervised learning, we tried to cluster together domain names using the K-Means and the DBSCAN. In the end, using regression techniques like linear regression, random forest and gradient boosting we tried to predict the total amount of bytes sent by a server and the average Round Trip Time values.

## 2 Data exploration and data reduction

To better understand and pre-process the training dataset, we started by dropping out features with only null values since they do not provide any meaningful contribution to our analysis. We found that 6 features correspond to our research and are: "_c_pkts_fc", "_c_pkts_unfs", "_c_syn_retx", "_s_pkts_unfs", "_s_syn_retx" and "_s_win_0". As result in removing the previous features, we remained with 117 columns out of the original 123.

### 2.1 Aggregation

After that, we processed the original dataset in two different ways to get other two levels to have different perspectives of the data: the first, was the IP level, where data points were grouped by the "c_ip" feature (the client IP of the requests), and another one which is the domain level with the entries aggregated by the 'label' feature, which is the domain name of the service. To perform that, we noticed that many features had means, maximums, minimums, counts, temporal or boolean values. Often, that was indicated in the name itself with key words like "avg", "max", "min", "first", "last", etc. For each one of the different categories, we performed a different operation. However, some features that fell under one of the previous categories do not have one of the key words in their name. For this reason, we manually inserted those features in the right category. A list of features is provided later within the table 10, for further details on those features, consult the appendix A. The only features that we decided to remove before the aggregation are the ones about server and client port (respectively,"_s_port" and "_c_port") because those information are not useful for our analysis. The relationship between category and operation is indicated in the table 1.

Authors' Contact Information: Salvatore Di Franco - s346785; Antonio Di Ponzio - s347920; Arman Sheikhhosseini - s342405; Danny Di Girolamo - s331374.

| Category | Key words | operation |
|---|---|---|
| average | avg | mean |
| maximum | max | max |
| minimum | min | min |
| count | cnt, all, data, dup, retx, count | sum |
| temporal | first, last | max |
| boolean | opt | round(mean) |
| default | (fallback category) | mean |

Table 1. Aggregation table

The original dataset instead gave us insight about the flow level and so no operations were executed other than the drop out of null features. In the end, we obtained 3 different dataset with shapes 147863 x 117 (flow level), 738x114 (ip level) and 26x114 (domain level). Notice that ip level and domain level have three features less since we decided to drop, respectively the "label" and "c_ip" features other than the port features.

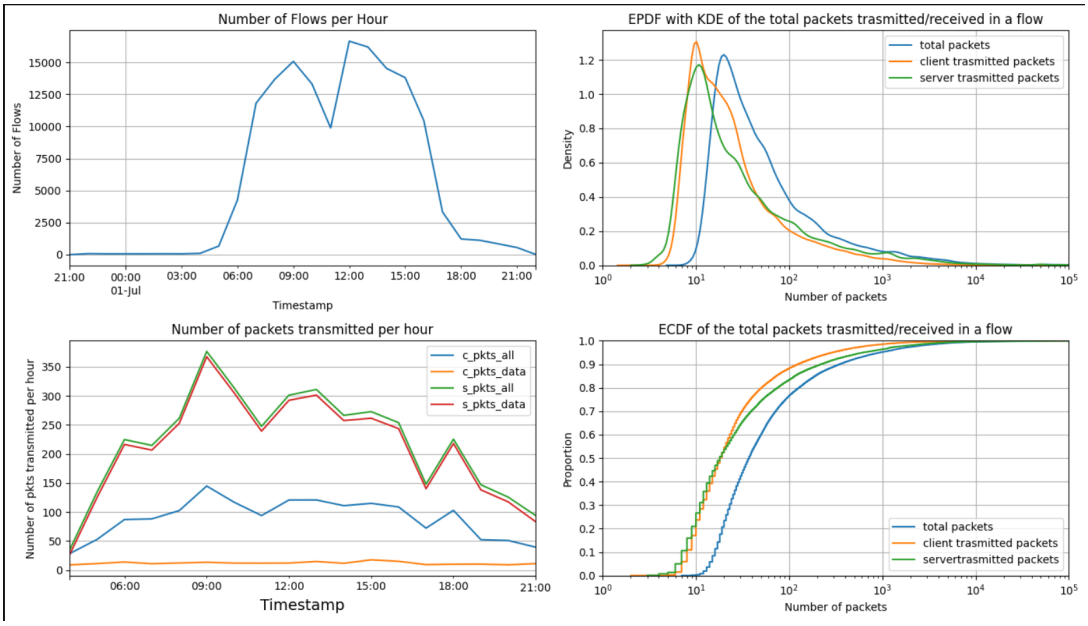



Fig. 1. Different graphics regarding the flow level analysis

## 2.2 EPDF and ECDF of the packets flow

We are now going to take a better look at the content of the dataset. First of all, we want to know the duration of the data acquisition and how many flows we have per hour. To do that, we decided to convert the "time" feature, which is the timestamp of the associated flow, from seconds to a date format. In doing that, we found that the time period goes from 2019-06-30 21:55:48 to 2019-07-01 22:03:05. After that, we wanted to see how the flows were distributed in the time period. To reach this goal, we proceeded in using the "time" feature as index of the

dataframe and then resample it in hours. Finally using the .count() method we counted the number of flows for each hour and then plotted the result in fig 1.

To further understand how many packets were transmitted, we also plotted the amount of packets per hour, plotting not only the total amount of packets sent by the servers and by the clients, but also how many of them were data packets. We can see that most of the server ones were data ones, while for the clients we had more control packets.

After that, we went back to the original dataframe. Remaining at the flow level, we decided to compute the Empirical Probability Distribution Function (EPDF) and the Empirical Cumulative Distribution Function (ECDF) of the total packets transmitted by the client per flow ("_c_pkts_all"), of the packets transmitted by the server per flow ("_c_pkts_all") and the total packet traffic per flow (summation of the previous two features). Since computing this last one and make it a feature would have generated an high correlated one, we decided to simply plot the values without storing them in the dataframe. The two graphs can be seen in 2 in the appendix.

We can immediately notice the relationship between the two graphs. In fact, peaks in the EPDF correspond to a steeper increase in the ECDF because the EPDF shows the probability to have a specific value while the ECDF considers all the occurrences before that value. In this way, a steeper increase shows that many data points have a value in that range. For example, the client transmitted packets EPDF in fig.1 has a peak at 10 packets. At the same value, the ECDF rises quicker than in other parts. The same analysis has been applied to the ip and domain level and the plotted graphs can be seen in fig.2.

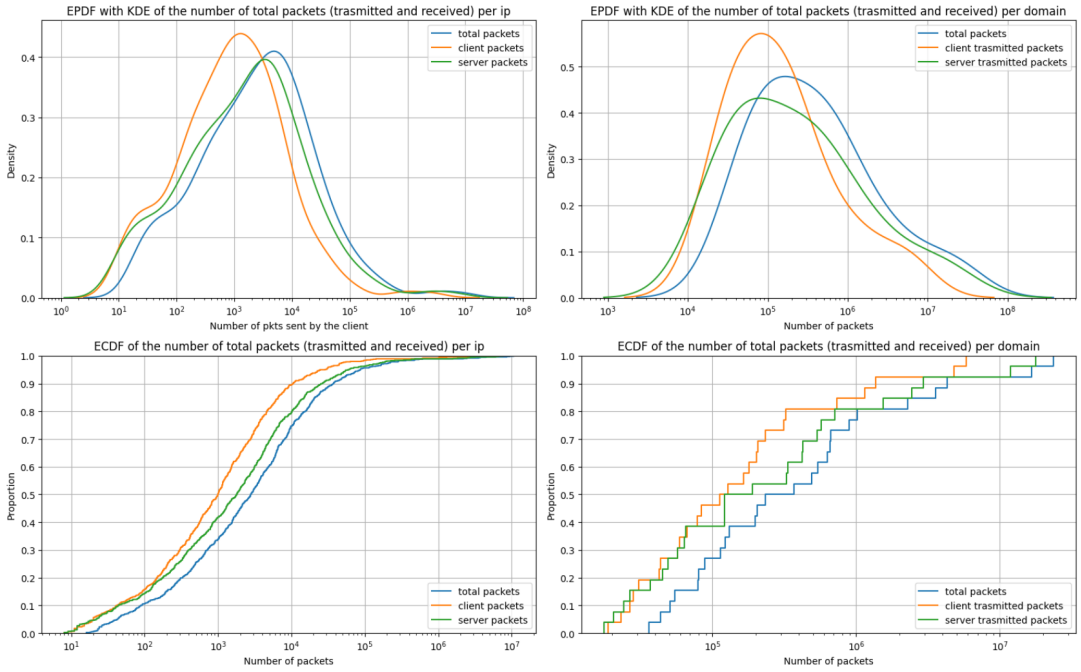



Fig. 2. EPDF and ECDF of correlation matrices for ip and domain levels

## 2.3 Correlation analysis

After exploring the distribution of the dataset features by plotting ECDF and EPDF, we implemented a correlation analysis in order to better understand the relationships between the features. The idea was to compute the correlation matrix at each level, plot it using a heatmap and then find the most correlated features. To do that, we set 0.8 as the threshold, so that every couple with a correlation above that would have been considered a "highly correlated couple". The value 0.8 has been chose as trade-off to select also not so much correlated features, but not too much not correlated ones. We decided to repeat this procedure for each level to also see how the aggregation changed the relationship between the features. For further details about the correlations, the heatmaps can be found in the appendix C For the flow level, we found that 135 couples satisfy the condition. The heatmap (aside from the main diagonal, which has correlation equal to 1 by definition) has light colours which indicates a low correlation between the features. We can see that the number is quite low compared to all the possible couples. For the ip and domain levels, we can see that the number of correlated features greatly increases, arriving at, respectively, 1365 and 1009. To conclude the search for highly correlated couples, we decided to print the three most correlated couples per level. However, those are not the only ones with that value of correlation, but just the ones that the sorted() function brought as firsts based on the correlation value. We found the following couples:
Flow Level:

- (_s_f1323_opt , _s_win_scl) : 1.0
- (_c_pkts_data , _c_pktsize_count) : 1.0
- (_c_tm_opt , _s_tm_opt) : 1.0

IP level:

- (_c_pkts_data , _c_pktsize_count) : 1.0
- (_s_pkts_data , _s_pktsize_count) : 1.0
- (_c_tm_opt , _s_tm_opt) : 1.0

Domain level:

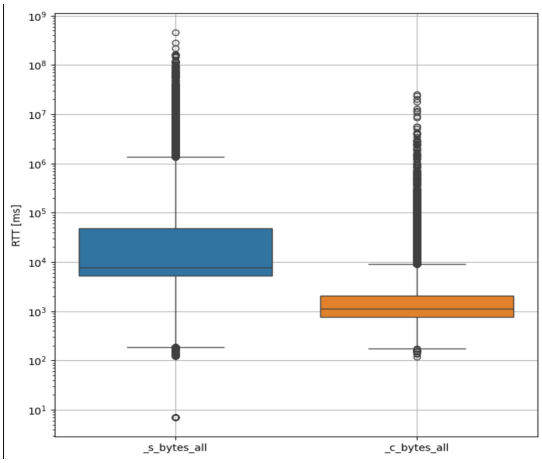
- (_c_mss , _s_win_min) : 1.0
- (_c_pkts_data , _c_pktsize_count) : 1.0
- (_s_pkts_data , _s_pktsize_count) : 1.0

Another reason to perform the correlation analysis is for the dimensionality reduction. In fact, if highly correlated features are kept in the dataset, we may run into longer computational time, overfitting of the models, and, as consequence, also worst metrics. For this reason, perform dimensionality reduction in this phase can help in improving the final result. With this idea in mind, we found out that at the three different levels we could eliminate 47 features from the flow level dataset and 68 in both IP and domain level. However, this is not the only way to perform dimensionality reduction. Other techniques, like the Principal Component Analysis (PCA) can perform better than manually removing the correlated features. Moreover, combining the two techniques, so first remove the correlated features and then perform the PCA, does not provide any improvement to the final result. Further details on the PCA implementation and associated results can be found in the dedicated subsection.

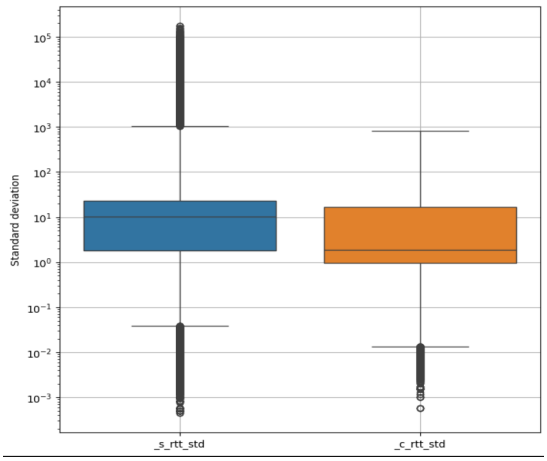
## 2.4 Bytes and RTT statistics

At the end of the data exploration part, we wanted to take a closer look at the transmitted bytes and the Round Trip Time (RTT) of the packets. Starting from the bytes, we had a total of six features associated to them, three for each device: “_bytes_all”, “bytes_retx” and “bytes_uniq”. To present as much information in the shortest amount of space, we decided to represent these features with boxplots. An advantage of this choice is the possibility to highlight the quartiles, maximum, minimum and outliers. While we were analysing the plot of “_bytes_all” and “bytes_uniq” we noticed that the two boxplots were much alike, so we assumed a low amount of retransmitted

bytes. To further investigate, we used the ".describe()" method on the bytes_retx features and we found out that 25, 50 and 75 quartiles had a 0 value. This means that most of the values were around 0 despite the means were relatively high (2697,3 for the server bytes, 43 for the client). However, this indicates the simple presence of values high enough to compensate for the huge number of zeros and that can be seen both from the standard deviation (817 for the client, 80117 for the server) and from the maximums (197307 for the client and 2 * 10$\hat{7}$ for the server). To conclude the analysis of the transmitted bytes, we provide only the boxplots of "_c_bytes_all" and "_s_bytes_all" keeping in mind that they are similar to the respective "bytes_uniq" features.




(a) EPDF of the _c_bytes_all and _s_bytes_all features




(b) EPDF of the _c_rtt_std and _s_rtt_std features

Moving on to the analysis of the RTT, for each actor, we have five features: _rtt_avg, _rtt_cnt, _rtt_max, _rtt_min and _rtt_std. For our purpose we decided to plot in boxplots all of them except the _rtt_cnt one, in this way we could show how the values were distributed in the dataset.

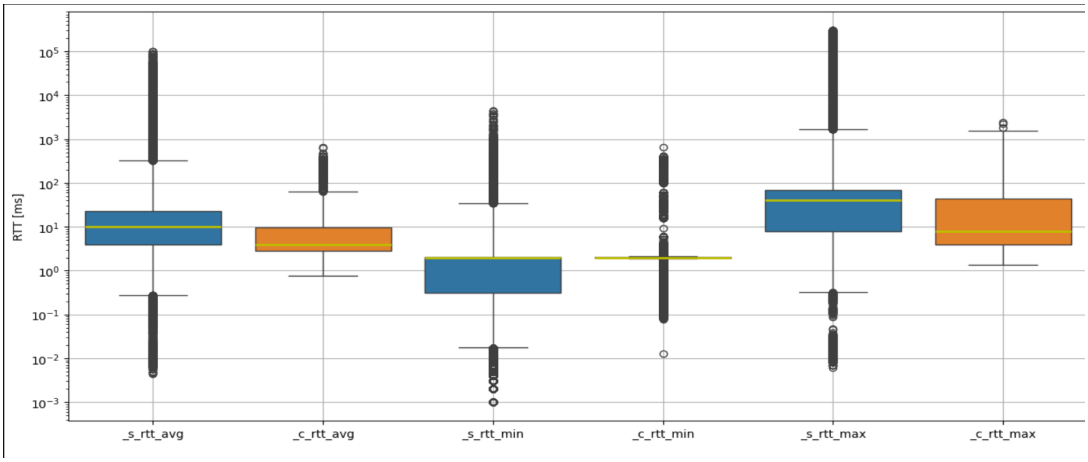


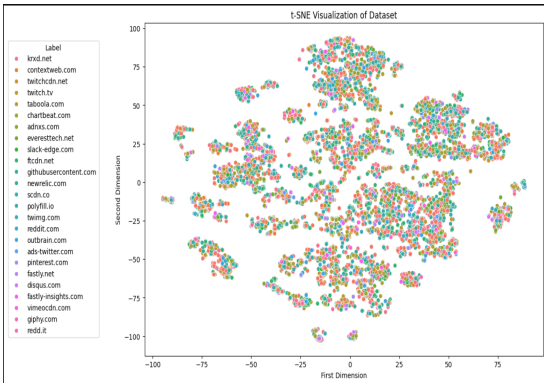

Fig. 4. EPDF of the RTT features

## 2.5 PCA and T-SNE

Finally, we proceeded to perform the data reduction through Principal Component Analysis (PCA) and T-distributed stochastic neighbour embedding (T-SNE) . First of all, we needed to standardize the dataset. To do that, we first separated numerical and categorical features, then using the StandardScaler() class, we fitted and transformed (so for each feature, we subtracted the mean and divide by the standard deviation associated) the dataset made of numerical features. In the end, we concatenated numerical and categorical features to get back the original dataset structure. After that, we were ready to perform the PCA. We made the PCA train on the standardized dataset and then we computed the explained variance of each component which represents how much of the total variance in the original dataset is captured by each principal component. In fact, we wanted to see how many principal components were needed to represent well enough the variance of the dataset. To achieve that, we calculated the cumulative sum of them and then plotted it in a graph (fig.6 in the appendix). To be sure to cover enough variance, we decided to take all the principal components which summation gave us at least 90% of variance coverage and from the graph we found that the first 41 components cover 90.27% of it.

To conclude the PCA analysis, we decided to plot the scatter plot (fig5a) of the first two principal components and see if we could use it to correctly classify the domains of the flow. From the cumulative graph, we found out that the first two components cover 19.76% of the total variance. However, two components are not enough to show any significant cluster. Since the PCA did not provide any useful cluster representation, we can suppose that we are working in a non linear space and so we need a non linear technique. A possible one is the t-SNE which can help to cluster different classes. We tried again with the classification of the domains. To be able to properly plot them, we decided to use just two components. Since we wanted to focus more on the global view than on the local one, we set the perplexity to 50 and left the "early_exageration" parameter to the default 12. For the amount of iterations, we tried to increase the maximum amount of iterations up to 3000, but despite some small improvement, only some clusters appeared. We got a similar result by disabling the max iterations and setting instead the maximum amount of iterations without progress to the default value. However, those clusters were not helpful to separate the different flows based on the domain name. The scatter plot obrtained with t-SNE can be found in fig5b.


(a) Scatter plot of the first two components of the PCA




(b) Scatter plot of the first two components of t-SNE

## 3 Supervised learning

The following sections of this paper show the development of a supervised machine learning method that classifies the flows to predict the domain names visited by clients. Supervised learning is a technique that uses labeled training data to train a model. The trained model will be then used to classify new unlabeled data.

The dataset provided for this project presents one main characteristic that influenced this work : the data are not distributed evenly among the classes, namely the dataset is imbalanced. This means that there is a huge difference between the size of the most populated class and the least populated one. Usually, in these cases, machine learning models lead to overfitting, which is when a model performs well on the training set and poorly on the validation set.

The most widespread ways to handle these types of dataset are:

- Synthetic Minority Oversampling Technique (SMOTE), which consists in duplicating the data belonging to the least populated classes to balance the dataset in such a way that no additional information is provided to the dataset.
- Undersampling technique, which consists in removing data from the most populated classes in order to balance the dataset.
- hyperparameter tuning, which consists in modifying some parameters of a model (those that are not automatically learned by the models) in order to achieve better performances.

However, the first two options have severe disadvantages.

SMOTE technique is not advised for strongly imbalanced datasets: if the minority classes contain too few elements, the "interpolated" data could completely lose significance . Moreover, this technique can create entries that do not correspond to any realistic instances, particularly in sparse datasets in which minority classes are spread without clusters.

Similarly, undersampling technique leads to an invasive modification of the data when dealing with highly imbalanced datasets: in this specific case, the dataset would be reduced to 20% of its original size. This would represent a noteworthy loss of data. Finally , both methods do not perform well with high-dimensional data.

Therefore, hyperparameter tuning seems to be the best choice to handle this dataset. Hyperparameters are settings that are not learned from the data but are setted a priori. They control the behaviour of the algorithm, impacting the performances, the computational efficiency and more.

## 3.1 Decision Tree Model

Decision Tree is a supervised machine learning algorithm that makes decisions by splitting the data into subsets based on the feature values. - each node represents a decisions - each branch represents the outcome of a decision - each leaf represents a final prediction

Decision tree is a simple model that does not require high computational time (which is an important factor due to the shape of the dataset). On the other hand, it may be biased towards the dominant class and it can easily overfit the training set.

The downsides are clearly observable from the non-tuned model performance.

```
---- ACCURACY SCORE ----
Training set accuracy: 1.0
Test set accuracy: 0.6314543671592331

---- F1 SCORE ----
Training set f1 score: 1.0
Test set f1 score: 0.6314963632076341
```

As expected, the model overfit the training set. The decision tree algorithm can be configured through several hyperparameters. The most influential are[1]:

[1]Hyperparameters consequences will be thoroughly discussed in the Random Forest chapter

- criterion: function to measure the quality of the split;
- max_depth: maximum depth of the three
- min_samples_split: minimum number of samples to split a node
- min_samples_leaf: minimum number of samples required at a leaf node

Hyperparameter tuning has been performed on the following range:

```
'criterion': ['gini','entropy'],
'max_depth': [2,3,4,5,6,7,8,9],
'min_samples_split': [2,3,4],
'min_samples_leaf': [1,2,3]
```

With a cross validation performed on two sets, the result is:

```
Best hyperparameters: {'criterion': 'entropy', 'max_depth': 9,                                    'min_s
---- ACCURACY SCORE ----
Training set accuracy: 0.33674866852650265
Test set accuracy: 0.3226929969904981

---- F1 SCORE ----
Training set f1 score: 0.3489215491581641
Test set f1 score: 0.338596740892778
```

In conclusion, hyperparameter tuning drastically reduces performances both on the training set and the validation set. This is a reasonable result: decision tree is a very simple method that is not indicated for highly imbalanced datasets.

## 3.2 Random Forest

Random Forest is an ensemble machine learning algorithm. Ensemble algorithms are techniques that combine the predictions of multiple models to achieve better performances. Aggregating the prediction of several models should improve the accuracy, reduce overfitting and improve the robustness.

Specifically, Random Forest is a bagging model (bootstrap aggregating): it trains numerous decision trees, each one with a different subset of the training set. This technique should perform well with high-dimensional data. The main disadvantages of the model are the high computational cost and memory complexity (especially when the several decision trees are computed in parallel).

First, let us check model performances without tuning.

```
---- ACCURACY SCORE ----
Training set accuracy: 1.0
Test set accuracy: 0.7521725898623745

---- F1 SCORE ----
Training set f1 score: 1.0
Test set f1 score: 0.7391030906448335
```

The model overfit the training set. Probably, the reason is that the default value for 'max_depth' hyperparameter is 'None'. Let us investigate the meaning of the several model hyperparameters.

- max_depth: it is defined as the longest path between the root node and the leaf node; if this parameter is too high (like the default case 'None', which means $+inf$) the model will overfit the training set because it is not able to generalize the data.
- min_sample_split: it is the required minimum number of observations in any given node in order to split it; if it is too low (default value is 2), the nodes are split until they contain too few elements, leading to the same overfitting issue of the previous hyperparameter.
- max_terminal_nodes: it indicates the maximum number of leaf nodes, reducing the growth of the tree; low values make the model underfit, while huge values lead to high generalization and overfit.
- min_samples_leaf: it is the minimum number of samples that a leaf node has to contain; if the split of the node were to result in another node containing fewer samples than the constraint, then the division would not be performed; if the parameter is too low, the model overfit the training set due to low generalization (too many leaves are generated); if the value is too high, the model shows underfitting.
- n_estimators: it represents the number of decision trees that are generated; a low value makes the model overfit the training set (random forest shows in this case the same disadvantages of decision tree); high values increase the computational cost.

The amount of numerical hyperparameters make the computational cost of tuning considerable. After several trials, tuning has been performed on the following ranges with the results presented below (cross validation performed on five sets).

```
'n_estimators': [100],
'max_depth': [20, 50, 100],
'min_samples_split': [15, 20, 30],
'max_leaf_nodes' : [100, 500],
'min_samples_leaf': [1,2,8]
```

```
Best hyperparameters: {'class_weight': 'balanced', 'max_depth': 20, 'max_leaf_nodes': 500,
'min_samples_leaf': 8, 'min_samples_split': 30, 'n_estimators': 100}
---- ACCURACY SCORE ----
Training set accuracy: 0.47644771324710455
Test set accuracy: 0.4488553748351537

---- F1 SCORE ----
Training set f1 score: 0.46277870832754076
Test set f1 score: 0.4439235474967166
```

The model is extremely underfitting the whole dataset: this is not the expected result. Poor performance is a direct consequence of the class imbalance. Probably a better hyperparameter tuning on wider ranges could have improved the model, but due to its significant computational cost it was not feasible in the context of this project. These hyperparameter ranges took a few hours to get computed and GPU acceleration was not a suitable solution because of the scarce memory available in the cluster[2].

### 3.3 K-Nearest Neighbors

The K-Nearest Neighbors algorithm is based on the principle of similarity: objects that are close to each other in the feature space probably belong to the same class. This model can be used thanks to the dimensionality reduction performed by the PCA: the algorithm suffers from the well-known problem of the curse of dimensionality, the

[2]Even with a Google Colab subscription, GPU ram was not enough

phenomenon that reduces the model performance in case of high dimensional data. Moreover, it is computationally expensive and prone to overfitting.

Let us see the model performance without tuning.

```
---- ACCURACY SCORE ----
Training set accuracy: 0.7780032124439936
Test set accuracy: 0.6800798025225713

---- F1 SCORE ----
Training set f1 score: 0.7705177859667081
Test set f1 score: 0.6688942886985497
```

The difference between training and validation set performances is around 9.3% for both accuracy and f1 score. The model is overfitting the data. K-NN has the following hyperparameters:

- k: indicates the number of neighbors considered for predictions
- metric: indicates the metric used for distance calculation
- weights: indicates how much objects should contribute to the prediction based on the distance

The model has been tuned with the following values:

```
'n_neighbors': [3,5,7,9,11],
'weights': ['uniform'],
'p': [1, 2],        # Distance metric (1: Manhattan, 2: Euclidean)
'metric': ['euclidean', 'manhattan', 'minkowski']
```

The 'distance' value of the hyperparameter 'weights' has actually been removed, as this value would lead to overfitting. With cross validation performed on three sets, the performances are the following:

```
Best hyperparameters: {'metric': 'manhattan', 'n_jobs': -1, 'n_neighbors': 3,
                       'p': 1, 'weights': 'uniform'}
---- ACCURACY SCORE ----
Training set accuracy: 0.8312114295375771
Test set accuracy: 0.7030061204477057

---- F1 SCORE ----
Training set f1 score: 0.826755641833954
Test set f1 score: 0.6942310316491132
```

As expected the model continues to overfit the training dataset even though the hyperparameter tuning has been performed.

Misclassification analysis could provide more informations to better understand the reasons why a model is not well performing. In this case K-NN tuned model has been chosen because other models provides too scarce performances: both classifiers lead to underfitting after hyperparameter tuning. So, K-NN has been considered the least objectionable choice despite the overfitting.

## 3.4 Misclassification Analysis

The aim of this subsection is to find a feature that presents different distributions between correctly classified samples and wrongly classified ones. In order to achieve this goal, the following modus operandi has been followed:

- prediction has been performed on the whole dataset (without training/validation splitting), giving an accuracy of around 70
- from the prediction result, the indices relative to the misclassified and correctly classified samples have been extracted
- the previously extracted indices have been used to filter the original dataset in order to create two new sets that respectively contain the misclassified samples and the correctly classified ones; this operation has been performed on the original data in order to obtain the non PCA transformed values; as a result, the features return to their original meaning
- the hypothesis is the following: features that present a considerable difference in mean may be considered discriminant for the misclassification
- to confirm that, it is necessary verify that the two features are spread differently

Actually, after analysing the features, it turns out that none of them present different distribution between truly classified data and not. This is probably a direct consequence of the overfitting: misclassification comes from the overfitting issue and is not a consequence of the influence of certain faulty features.

Note: running the code after changing a specific variable with the name of the feature will provide the plots for that attribute[3].

## 4 Introduction to Section 3: Unsupervised learning – clustering

In this section, we focus on unsupervised learning, specifically clustering, to group domain names based on similar patterns. Unsupervised learning is a type of machine learning where the algorithm is tasked with finding patterns or structures in data without predefined labels or categories. Unlike supervised learning, which relies on labeled datasets, unsupervised learning identifies hidden structures, groupings, or relationships within the data. Clustering is a prominent unsupervised learning technique used to group data points into meaningful clusters based on their similarity.

The goal of this section is to develop an unsupervised machine learning method for grouping domain names. This involves checking whether there are "families" of domain names (e.g., video streaming, news, or e-commerce). The clustering is performed independently of the labels used in Section 2, ensuring the task is entirely unsupervised.

### 4.1 Main Objectives:

(1) **Determining the optimal number of clusters:** Using methods like the Elbow Method and Silhouette Analysis.
(2) **Validating clusters using domain names:** Extracting and analyzing representative domain names from each cluster.
(3) **Reporting coarse analysis and identifying domain families:** Analyzing whether the clusters represent distinct groups of domain names, such as services for video streaming, social media, or news.

## 5 Preprocessing: Why We Used It and Its Importance

In this section, preprocessing is a crucial step that lays the foundation for effective clustering. The dataset provided contains raw numerical features derived from domain name patterns. However, raw data often includes challenges like varying scales, redundancy, or irrelevant patterns that can reduce the efficiency and accuracy of machine learning algorithms.

[3]This variable will be indicated in the 'Misclassification Analysis' section of the notebook

### 5.1 What is Preprocessing?

Preprocessing is the set of techniques applied to clean, transform, and prepare raw data to make it suitable for machine learning algorithms. In this section, preprocessing involved:

(1) **Normalization:** Scaling the numerical features so that they have a mean of 0 and a standard deviation of 1 using `StandardScaler`. This step is essential because clustering algorithms like K-Means are sensitive to the scale of data. Features with larger magnitudes could dominate smaller ones, skewing the results.
(2) **Dimensionality Reduction with PCA (Principal Component Analysis):** High-dimensional data can be challenging for clustering algorithms. PCA was applied to reduce the dataset to its most significant components, capturing most of the variance while removing noise or redundancy. This helps improve computational efficiency and enhances the interpretability of results.

### 5.2 Why Preprocessing Was Necessary?

(1) **Uniform Scale:** The original features had varying scales, making some features dominate clustering results. Normalization ensured all features contributed equally.
(2) **Noise Removal:** High-dimensional data can contain irrelevant or redundant features that obscure meaningful patterns. PCA helped us focus on the features that matter most.
(3) **Visualization:** Reducing the data to two principal components allowed us to visualize clusters, making it easier to interpret and validate results.
In Particular, PCA was applied to a dataset of 121 numeric features derived from IP traffic. The goal was to extract the most meaningful components, reduce noise, and ensure efficient clustering.

## 6 Here is the Results of PCA:

(1) **Number of Features Before PCA:** The dataset initially contained 121 numeric features, representing various metrics related to IP traffic.
(2) **Number of PCA Components:** After applying PCA, the data was reduced to 13 principal components, which together explained the majority of the variance in the dataset. This dimensionality reduction improved computational efficiency and made clustering more robust.
(3) **Explained Variance Analysis:** The cumulative explained variance was plotted against the number of PCA components. As shown in the plot in the Ipython file:
- **2 Components:** Captured ∼40% of the variance.
- **13 Components:** Captured over 90% of the variance, justifying the selection of 13 components for further analysis.

(4) **Feature Contributions to Principal Components:** The following table shows the contributions of selected features to the first few principal components:

| Feature | PC1 | PC2 | PC3 | PC4 | PC5 |
|---|---|---|---|---|---|
| _c_ack_cnt | 0.1632 | 0.0347 | 0.0374 | -0.0409 | -0.0271 |
| _c_bytes_all | 0.0667 | 0.2079 | -0.0942 | -0.0558 | 0.0306 |
| _c_cwin_max | -0.0112 | 0.0358 | -0.0625 | 0.0401 | 0.2399 |
| _s_win_max | 0.1436 | -0.0267 | 0.0306 | 0.0416 | 0.0173 |

This analysis highlights how certain features (e.g., _c_ack_cnt and _c_bytes_all) contributed more significantly to the variance in the data. The PCA results demonstrate its effectiveness in handling high-dimensional IP traffic data. By transforming the dataset into 13 principal components, we achieved:
1. Enhanced interpretability by focusing on the most critical features.
2. Improved clustering accuracy by removing noise and redundant features.

3. Simplified visualization with two-dimensional projections.

## 7 Why We Chose DBSCAN and K-Means

Here our aim was to cluster domain names based on their patterns of network traffic. For this, we selected K-Means and DBSCAN as our clustering algorithms due to their complementary strengths and suitability for unsupervised learning tasks.

### 7.1 1. K-Means Clustering, Why?

- **Simplicity and Speed:** K-Means is computationally efficient, making it a good choice for our high-dimensional domain dataset.
- **Cluster Structure:** K-Means performs well on datasets with well-separated, compact clusters. For our project, it effectively grouped domain names into distinct categories, such as social media platforms, advertising services, and video streaming services.
- **Interpretability:** The clusters produced by K-Means were easy to interpret and validate, as demonstrated by clear groupings of domains like reddit.com in the social media cluster or vimeocdn.com in the video streaming cluster.
- **Limitations for Our Dataset:** Assumes Spherical Clusters: K-Means assumes that clusters are spherical and of equal size, which may not align perfectly with the nature of domain name patterns in network traffic. However, the algorithm still provided meaningful insights in our case.

### 7.2 2. DBSCAN, why?

- **Robustness to Noise:** DBSCAN is designed to handle noise and outliers, which are common in real-world datasets like ours. This made it a logical choice to explore patterns in domain name traffic that might not fit neatly into predefined categories.
- **Non-Spherical Clusters:** DBSCAN can detect clusters of arbitrary shapes, which is useful for identifying less structured domain groups that K-Means might miss.
- **Automatic Cluster Count:** Unlike K-Means, DBSCAN does not require specifying the number of clusters beforehand, which is advantageous when the optimal cluster count is unknown.

However, for our dataset, DBSCAN did not produce valid clusters. This may have been due to the relatively uniform density of domain patterns in our data or the difficulty in fine-tuning its hyperparameters (`eps` and `min_samples`). Despite this, it remains an important approach for exploring clustering possibilities in similar datasets.

## 8 Determining the Number of Clusters Using K-Means and DBSCAN

We used the K-Means algorithm and evaluated its performance through the Elbow Method and Silhouette Analysis. We also tested the DBSCAN algorithm to explore the potential for identifying clusters in a density-based approach.

### 8.1 1. K-Means: Elbow Method and Silhouette Analysis

*8.1.1 Elbow Method.* The Elbow Method involves plotting the inertia (sum of squared distances between data points and their nearest cluster centroid) against the number of clusters. The goal is to identify the "elbow point," where the decrease in inertia starts to slow, indicating the optimal number of clusters.

- **Observation:** From the graph in the Ipython file, the elbow point is observed at $K = 4$, indicating that 4 clusters strike a balance between compactness and complexity.

*8.1.2 Silhouette Analysis.* The Silhouette Score measures how similar an object is to its own cluster compared to other clusters. A higher score indicates better-defined clusters.

- **Observation:** The Silhouette Score decreases as the number of clusters increases, but it remains relatively high at $K = 4$. This supports the conclusion from the Elbow Method.

Both the Elbow Method and Silhouette Analysis converge on $K = 4$ as the optimal number of clusters for K-Means. This aligns with our dataset's characteristics, where domain names fall into well-defined groups like social media, video streaming, and advertising services.

## 8.2 2. DBSCAN: K-Distance Graph

Unlike K-Means, DBSCAN does not require a predefined number of clusters. Instead, it uses parameters `eps` (radius) and `min_samples` (minimum points within `eps`) to form clusters. A k-distance graph helps determine a suitable value for `eps`.

- **Observation:** The k-distance graph in the Ipython file shows a gradual increase in distances without a clear threshold, indicating that the dataset lacks the density variations DBSCAN relies on to form clusters.

*8.2.1 Outcome for DBSCAN.* DBSCAN failed to produce valid clusters, likely due to:

(1) **Uniform Density:** The dataset may have lacked dense regions and clear outliers.
(2) **Parameter Sensitivity:** Despite tuning, suitable `eps` and `min_samples` values could not be identified.

## 8.3 Analysis of the Detected Clusters

*8.3.1 K-Means Clustering.* After applying K-Means clustering with 4 clusters, we performed an empirical cumulative distribution function (ECDF) analysis to understand the distribution of the domain names across the clusters. Below, we detail the results based on the first two principal components (PC1 and PC2).

(1) **ECDF Analysis of Principal Components by Clusters**
   - The ECDF plots in the Ipython file for PC1 and PC2 show the cumulative distribution of feature values within each cluster.
   - Figure 1 represents the ECDF of PC1, showing clear separations between the distributions of clusters. Clusters 0 and 1, for example, exhibit distinct cumulative patterns, indicating notable differences in feature values.
   - Figure 2 represents the ECDF of PC2, reinforcing the separation between clusters. While clusters 2 and 3 show more overlapping distributions for PC2, clusters 0 and 1 remain well-separated.

(2) **Cluster Sizes**
   - Cluster sizes indicate the number of domain names assigned to each cluster. The following table summarizes the cluster sizes:

| Cluster | Size | Examples of Domain Names |
|---|---|---|
| 0 | 100 | `giphy.com`, `reddit.com` |
| 1 | 80 | `adnxs.com`, `chartbeat.com` |
| 2 | 20 | `ftcdn.net` |
| 3 | 5 | `vimeocdn.com` |

Table 2. Cluster Sizes and Examples of Domain Names

   - The largest cluster is Cluster 0, which contains domains like `giphy.com` and `reddit.com`, likely grouped due to similar traffic patterns characteristic of social media platforms.

- The smallest cluster is Cluster 3, with domains such as `vimeocdn.com`, suggesting unique traffic patterns typically associated with video streaming services.

(3) **Example Domain Names for K-Means Clusters**
- Cluster 0: Includes domains like `giphy.com`, `reddit.com`, and `twimg.com`, representing content-sharing and social media services.
- Cluster 1: Contains `adnxs.com`, `ads-twitter.com`, and `chartbeat.com`, suggesting a strong association with advertising networks and tracking services.
- Cluster 2: Includes `ftcdn.net`, commonly associated with hosting and delivery services.
- Cluster 3: Contains `vimeocdn.com`, a domain linked to video streaming services.

*8.3.2 DBSCAN Clustering.* Unfortunately, DBSCAN failed to produce meaningful clusters for this dataset. The results showed a high proportion of noise points, likely due to the relatively uniform density of the feature space or insufficient separation between data points.

## 8.4 Domain Families Identified

From the clustering results, it is evident that domain names are grouped into distinct "families" based on their functionalities:

- **Cluster 0:** Social media and content-sharing platforms.
- **Cluster 1:** Advertising and tracking services.
- **Cluster 2:** Content delivery and hosting services.
- **Cluster 3:** Video streaming platforms.

These clusters validate the hypothesis that domain names can be grouped into families with similar services based on traffic patterns.

# 9 Introduction to Section 4: Supervised Learning - Regression

In this section, we focus on the use of supervised learning regression technique, to predict metrics related to domain-level network traffic. Supervised learning involves training a model with labeled data to establish a relationship between input features and their corresponding outputs, enabling accurate predictions for new data. Regression is particularly useful for predicting continuous values, such as network traffic volume or average round-trip times.

The primary goal of this section is to build regression models to predict important target variables, such as `_s_bytes_all` (total bytes sent) and `_s_rtt_avg` (average round-trip time). This involves several steps, starting with data preprocessing, where we clean and transform the dataset to ensure compatibility with the models. Main steps include scaling the data and reducing its dimensionality using Principal Component Analysis (PCA) to improve efficiency and focus on the most important features.

We use a variety of regression models, including Linear Regression, Random Forest, and Gradient Boosting, to capture both simple and complex relationships in the data. Each model is trained, evaluated, and fine-tuned to achieve the best performance. The performance metrics—such as Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared ($R^2$)—help us compare and understand how well the models generalize to unseen data.

Through this process, we aim to identify the most effective models for predicting network traffic metrics, analyze their strengths and weaknesses, and gain deeper insights into the data. This section sets the foundation for applying regression techniques to real-world network traffic prediction tasks.

It is worth mentioning that in the code section, the tasks for predicting `_s_bytes_all` and `_s_rtt_avg` were handled separately to account for their unique characteristics and the specific challenges they present in prediction.

## 10 Implementation for Predicting _s_bytes_all

In this section, the task was to predict `_s_bytes_all`, a feature representing the total number of bytes transmitted from the server to the client, including retransmissions. To perform this task effectively, the column `_s_bytes_uniq` was excluded as it could lead to data leakage. Additionally, other non-predictive or irrelevant columns such as `c_ip` and `time` were also excluded.

Three machine learning models were trained with default parameters: Linear Regression, Random Forest Regressor, and Gradient Boosting Regressor. The performance of these models was evaluated on both training and test datasets using standard regression metrics.

### 10.1 Step-by-Step Breakdown

**1. Data Preprocessing**

- **Feature Selection and Exclusion:** `_s_bytes_uniq` and non-predictive columns (`c_ip`, `time`) were removed from the dataset.
- **Handling Categorical Data:** The datasets were one-hot encoded to convert categorical variables into numerical form while avoiding multicollinearity by dropping the first category.
- **Feature Alignment:** Training and test datasets were aligned to ensure the same set of features in both datasets.
- **Feature Scaling:** `StandardScaler` was applied to normalize the data so that all features had a mean of 0 and a standard deviation of 1, which is critical for optimizing the performance of many machine learning models.
- **Dimensionality Reduction:** PCA was applied to reduce the number of features to 20 components. The explained variance by PCA was 59.08%, indicating that more than half of the variance in the data was retained.

**2. Models Used**

Three models were used for prediction:

| **Model** | **Train MAE** | **Test MAE** | **Train MSE** | **Test MSE** | **Train $R^2$** | **Test $R^2$** |
|---|---|---|---|---|---|---|
| Linear Regression | 214,666.37 | 221,423.38 | $7.43 \times 10^{11}$ | $2.66 \times 10^{12}$ | 0.9539 | 0.9692 |
| Random Forest | 56,432.28 | 118,437.42 | $1.89 \times 10^{11}$ | $5.17 \times 10^{13}$ | 0.9882 | 0.4021 |
| Gradient Boosting | 131,820.08 | 173,382.08 | $3.29 \times 10^{11}$ | $4.71 \times 10^{13}$ | 0.9796 | 0.4547 |

Table 3. Performance Metrics for `_s_bytes_all` Prediction

**3. Regression Metrics and Observations**

- **Linear Regression:**
  - High $R^2$ values on both training and test sets indicate it captured the general trend well.
  - The relatively high MAE suggests errors in precise predictions.
- **Random Forest:**
  - The model showed strong performance on the training set but poor generalization on the test set, a classic case of overfitting.
- **Gradient Boosting:**
  - Gradient Boosting showed moderate overfitting, with better generalization than Random Forest but still struggled on the test set.

## 10.2 4. Overfitting or Underfitting?

- **Linear Regression:** No significant overfitting or underfitting was observed. However, the high error metrics suggest that it struggles with the complexity of the data.
- **Random Forest:** Severe overfitting was evident, as shown by the significant drop in performance from training to testing.
- **Gradient Boosting:** Moderate overfitting was observed, with better generalization than Random Forest.

## 10.3 Hyperparameter Tuning and Performance Variation

In this section, hyperparameter tuning was applied to optimize the performance of two machine learning models: Random Forest Regressor and Gradient Boosting Regressor, for predicting the target variable `_s_bytes_all`. The primary goal was to explore how performance varied after tuning and to identify the model that produced the best results.

*10.3.1 1. Hyperparameter Tuning Approach.* Hyperparameter tuning was performed using `RandomizedSearchCV` with 3-fold cross-validation. This approach efficiently explores hyperparameter combinations without the computational expense of exhaustive grid searches. The parameter grids for each model included these hyperparameters:

| Model | Hyperparameter | Values |
|---|---|---|
| Random Forest | `n_estimators` | [50, 100, 200] |
| Random Forest | `max_depth` | [5, 10, 20] |
| Random Forest | `min_samples_split` | [2, 5, 10] |
| Gradient Boosting | `n_estimators` | [50, 100, 200] |
| Gradient Boosting | `learning_rate` | [0.01, 0.1, 0.2] |
| Gradient Boosting | `max_depth` | [3, 5, 7] |

Table 4. Parameter Grids for Random Forest and Gradient Boosting

*10.3.2 2. Evaluation Metrics.* The performance of the tuned models was evaluated using the following regression metrics:

- **Mean Absolute Error (MAE):** Measures the average magnitude of errors in predictions.
- **Mean Squared Error (MSE):** Penalizes larger errors more heavily than MAE.
- **R-squared ($R^2$):** Indicates how well the model explains the variance in the target variable, with a value closer to 1 representing better performance.

| Target | Model | Best Params | Train MAE | Test MAE |
|---|---|---|---|---|
| _s_bytes_all | Random Forest | {'n_estimators': 50, max_depth: 10, min_samples_split: 2} | 133,148.25 | 165,689.16 |
| _s_bytes_all | Gradient Boosting | {'n_estimators': 200, max_depth: 7, learning_rate: 0.1} | 133,165.95 | 166,609.67 |

Table 5. Hyperparameter Tuning Results for `_s_bytes_all` - Part 1

| Target | Model | Train MSE | Test MSE | Train $R^2$ | Test $R^2$ |
|---|---|---|---|---|---|
| _s_bytes_all | Random Forest | $3.57 \times 10^{12}$ | $6.20 \times 10^{13}$ | 0.7784 | 0.2827 |
| _s_bytes_all | Gradient Boosting | $3.87 \times 10^{12}$ | $6.34 \times 10^{13}$ | 0.7600 | 0.2671 |

Table 6. Hyperparameter Tuning Results for `_s_bytes_all` - Part 2

#### *10.3.3 3. Results After Hyperparameter Tuning.*

#### *10.3.4 4. Observations.*

- **Random Forest:**
  - The tuned model achieved a slightly better $R^2$ score on the test set compared to Gradient Boosting (0.2827 vs. 0.2671), indicating slightly better performance in explaining variance.
  - The MAE and MSE values for both training and test sets showed that the model struggled to generalize well, with clear signs of overfitting (high train $R^2$ but low test $R^2$).
- **Gradient Boosting:**
  - The Gradient Boosting model had comparable performance to Random Forest, with slightly higher errors and a lower $R^2$ score on the test set.
  - The model demonstrated a similar issue of overfitting, as indicated by the significant performance gap between training and test sets.
- **Performance Variation:**
  - Both models demonstrated slight improvements in training MAE and MSE after tuning compared to their default configurations.
  - However, the test set performance remained limited, indicating that hyperparameter tuning alone could not entirely address the generalization challenges posed by the dataset.
- **Why Linear Regression Was Not Hyperparameter Tuned:**
  - Linear Regression does not have hyperparameters in the traditional sense, as it involves solving a closed-form equation or using iterative optimization methods to minimize the error function.
  - Unlike Random Forest and Gradient Boosting, which have parameters such as the number of estimators or maximum tree depth, Linear Regression focuses solely on finding the optimal weights for the input features. As a result, it does not benefit from hyperparameter tuning in this context.
  - Additionally, the simplicity of Linear Regression made it a good baseline model, with its results serving as a comparison for more complex methods like Random Forest and Gradient Boosting.

### 10.4 Analysis for _s_rtt_avg

The task involved predicting the average round-trip time (_s_rtt_avg), a feature representing the time elapsed between the data segment and the corresponding ACK. Rows with zero values in the target column were removed, and columns related to other RTT measures (_s_rtt_min, _s_rtt_max, _s_rtt_stddev) were excluded to avoid data leakage. The following steps were performed:

- Non-numeric and irrelevant columns (`c_ip`, `time`) were dropped.
- Categorical variables were one-hot encoded.
- Features were aligned between training and test datasets to ensure consistency.
- Data was standardized using `StandardScaler` to bring all features to a uniform scale.
- Dimensionality reduction with PCA was applied, reducing the features to 15 principal components, explaining 53.88% of the variance.

| Target | Model | Train MAE | Test MAE | Train MSE | Test MSE | Train $R^2$ | Test $R^2$ |
|---|---|---|---|---|---|---|---|
| _s_rtt_avg | Linear Regression | 839.80 | 1085.70 | $8.79 \times 10^6$ | $9.89 \times 10^6$ | 0.057 | 0.017 |
| _s_rtt_avg | Random Forest | 405.51 | 667.61 | $2.79 \times 10^6$ | $6.57 \times 10^6$ | 0.701 | 0.347 |
| _s_rtt_avg | Gradient Boosting | 618.63 | 803.02 | $6.49 \times 10^6$ | $8.30 \times 10^6$ | 0.304 | 0.176 |

Table 7. Default Model Performance for _s_rtt_avg

#### *10.4.1 Results: Default Model Performance.*

| Target | Model | Best Params | Train MAE | Test MAE |
|---|---|---|---|---|
| _s_rtt_avg | Random Forest | {'n_estimators': 200, 'max_depth': 10, 'min_samples_split': 2} | 760.94 | 1014.80 |
| _s_rtt_avg | Gradient Boosting | {'n_estimators': 50, 'max_depth': 7, 'learning_rate': 0.1} | 816.17 | 1175.76 |

Table 8. Tuned Model Performance (MAE) for _s_rtt_avg

| Target | Model | Train MSE | Test MSE | Train $R^2$ | Test $R^2$ |
|---|---|---|---|---|---|
| _s_rtt_avg | Random Forest | $5.79 \times 10^6$ | $8.93 \times 10^6$ | 0.379 | 0.114 |
| _s_rtt_avg | Gradient Boosting | $7.16 \times 10^6$ | $1.23 \times 10^7$ | 0.232 | -0.220 |

Table 9. Tuned Model Performance (MSE and $R^2$) for _s_rtt_avg

#### *10.4.2 Results: Tuned Model Performance.*

#### *10.4.3 Observations.*

(1) **Model Comparison:**
- Random Forest performed better than Gradient Boosting in terms of all evaluation metrics, achieving lower MAE and MSE values on both training and test sets.
- Both models showed limited improvements after hyperparameter tuning, indicating the dataset's inherent challenges in predicting _s_rtt_avg.

(2) **Overfitting and Underfitting:**
- Random Forest demonstrated slight overfitting, with a noticeable drop in $R^2$ from the training set (0.379) to the test set (0.114).
- Gradient Boosting performed poorly on the test set after tuning, with a negative $R^2$ score, indicating severe underfitting or a failure to generalize.

## 11 Conclusion

(1) **Higher Mean Absolute Error (MAE):**
- _s_bytes_all has a significantly higher Mean Absolute Error compared to _s_rtt_avg:
  - MAE for _s_bytes_all (best model, Random Forest): 165,689.16 bytes.
  - MAE for _s_rtt_avg (best model, Random Forest): 1,014.80 milliseconds.

(2) **Unit of Error Measurement:**
- For _s_bytes_all, the error is measured in bytes.
- For _s_rtt_avg, the error is measured in milliseconds.

(3) **Which Problem is Easier to Solve?**
- _s_rtt_avg is easier to solve:
  - The $R^2$ score for _s_rtt_avg (0.114 for Random Forest) indicates a modest ability of the model to explain variance, albeit not ideal.
  - In contrast, the $R^2$ score for _s_bytes_all (0.283 for Random Forest) highlights greater challenges in explaining variance for this problem.
- The lower MAE and smaller variance in predictions for _s_rtt_avg suggest that it is a simpler and more predictable target compared to _s_bytes_all, which has a much larger scale and variability.

In conclusion, while both tasks pose challenges, _s_rtt_avg is comparatively easier to predict, with smaller absolute errors and less complexity in capturing its patterns.

## A Information about the features

**This part is a simple copy and paste of the "readme.txt" file provided with the project assignment.**

These attributes are related to the characteristics of a network connection. Feature prefixed with _c_ or _s_, stands for "client" and "server", respectively. Here is a list of the attributes and a brief description of each:

- **_c_ack_cnt**: The number of ACK packets sent by the client.
- **_c_ack_cnt_p**: The percentage of ACK packets sent by the client.
- **_c_appdataB**: The number of bytes of application data sent by the client.
- **_c_appdataT**: The number of packets of application data sent by the client.
- **_c_bytes_all**: The total number of bytes sent by the client.
- **_c_bytes_retx**: The number of bytes of retransmitted data sent by the client.
- **_c_bytes_uniq**: The number of unique bytes sent by the client.
- **_c_cwin_ini**: The initial congestion window size for the client.
- **_c_cwin_max**: The maximum congestion window size for the client.
- **_c_cwin_min**: The minimum congestion window size for the client.
- **_c_fin_cnt**: The number of FIN packets sent by the client.
- **_c_first**: The time of the first packet sent by the client.
- **_c_first_ack**: The time of the first ACK packet sent by the client.
- **_c_last**: The time of the last packet sent by the client.
- **_c_last_handshakeT**: The time of the last handshake packet sent by the client.
- **_c_msgsize1**: The size of the first message sent by the client.
- **_c_msgsize_count**: The number of message sizes sent by the client.
- **_c_mss**: The maximum segment size (MSS) for the client.
- **_c_mss_max**: The maximum MSS for the client.
- **_c_mss_min**: The minimum MSS for the client.
- **_c_pkts_all**: The total number of packets sent by the client.
- **_c_pkts_data**: The number of data packets sent by the client.
- **_c_pkts_data_avg**: The average number of data packets sent by the client.
- **_c_pkts_data_std**: The standard deviation of the number of data packets sent by the client.
- **_c_pkts_dup**: The number of duplicate packets sent by the client.
- **_c_pkts_fc**: Number of retransmitted segments to probe the receiver window.
- **_c_pkts_ooo**: The number of out-of-order packets sent by the client.
- **_c_pkts_push**: The number of PUSH packets sent by the client.
- **_c_pkts_reor**: The number of reordered packets sent by the client.
- **_c_pkts_retx**: Number of retransmitted packets sent by the client.
- **_c_pkts_rto**: Number of packets sent by the client that triggered a retransmission timeout.
- **_c_pkts_unfs**: Number of packets sent by the client with the FIN flag set but without ACK.
- **_c_pkts_unk**: Number of segments not in sequence or duplicate which are not classified as specific events.
- **_c_pkts_unrto**: Number of packets sent by the client that caused a retransmission timeout to be cancelled.
- **_c_pktsize1**: Size of the first packet sent by the client.
- **_c_pktsize_count**: Number of packets sizes sent by the client.
- **_c_port**: Port number used by the client.
- **_c_rst_cnt**: Number of reset (RST) packets sent by the client.
- **_c_rtt_avg**: Average round-trip time (RTT) for packets sent by the client.
- **_c_rtt_cnt**: Number of RTT measurements for packets sent by the client.

- **_c_rtt_max**: Maximum RTT for packets sent by the client.
- **_c_rtt_min**: Minimum RTT for packets sent by the client.
- **_c_rtt_std**: Standard deviation of the RTT for packets sent by the client.
- **_c_sack_cnt**: Number of selective acknowledgement (SACK) packets sent by the client.
- **_c_sack_opt**: Indicates whether the client supports SACK.
- **_c_seg_cnt**: Number of segments sent by the client.
- **_c_sit1**: Size of the first segment sent by the client.
- **_c_sit_avg**: Average size of segments sent by the client.
- **_c_sit_std**: Standard deviation of the size of segments sent by the client.
- **_c_syn_cnt**: Number of SYN (synchronize) packets sent by the client.
- **_c_syn_retx**: Number of SYN packets retransmitted by the client.
- **_c_tm_opt**: Indicates whether the client supports the TCP timeout option.
- **_c_ttl_max**: Maximum time-to-live (TTL) value used by the client.
- **_c_ttl_min**: Minimum TTL value used by the client.
- **_c_win_0**: Number of times the congestion window size was zero for the client.
- **_c_win_max**: Maximum congestion window size for the client.
- **_c_win_min**: Minimum congestion window size for the client.
- **_c_win_scl**: Window scaling factor used by the client.
- **_durat**: Duration of the connections.
- **_s_ack_cnt**: Number of acknowledgement packets sent by the server.
- **_s_ack_cnt_p**: Percentage of packets sent by the server that are acknowledgement packets.
- **_s_appdataB**: Number of bytes of application data sent by the server.
- **_s_appdataT**: Number of packets containing application data sent by the server.
- **_s_bytes_all**: Total number of bytes sent by the server.
- **_s_bytes_retx**: Number of bytes retransmitted by the server.
- **_s_bytes_uniq**: Number of unique bytes sent by the server.
- **_s_cwin_ini**: Initial congestion window size used by the server.
- **_s_cwin_max**: Maximum congestion window size used by the server.
- **_s_cwin_min**: Minimum congestion window size used by the server.
- **_s_f1323_opt**: Indicates whether the server supports the TCP options for window scaling, selective acknowledgement, and timestamps (also known as "TCP congestion control").
- **_s_fin_cnt**: Number of FIN (end-of-connection) packets sent by the server.
- **_s_first**: Timestamp of the first packet sent by the server.
- **_s_first_ack**: Timestamp of the first acknowledgement packet sent by the server.
- **_s_last**: Timestamp of the last packet sent by the server.
- **_s_last_handshakeT**: The time of the last handshake packet sent by the server.
- **_s_msgsize1**: Size of the first message sent by the server.
- **_s_msgsize_count**: Number of messages sent by the server.
- **_s_mss**: Maximum segment size used by the server.
- **_s_mss_max**: Maximum MSS value used by the server.
- **_s_mss_min**: Minimum MSS value used by the server.
- **_s_pkts_all**: Total number of packets sent by the server.
- **_s_pkts_data**: Number of data packets sent by the server.
- **_s_pkts_data_avg**: Average number of data packets sent by the server per message.
- **_s_pkts_data_std**: Standard deviation of the number of data packets sent by the server per message.
- **_s_pkts_dup**: Number of duplicate packets sent by the server.

- **_s_pkts_fc**: Number of flow control packets sent by the server.
- **_s_pkts_fs**: Number of packets with the FIN/SYN flag set sent by the server.
- **_s_pkts_ooo**: Number of out-of-order packets sent by the server.
- **_s_pkts_push**: Number of push packets sent by the server.
- **_s_pkts_reor**: Number of packets sent by the server that caused reordering.
- **_s_pkts_retx**: Number of retransmitted packets sent by the server.
- **_s_pkts_rto**: Number of packets sent by the server that caused a retransmission timeout.
- **_s_pkts_unfs**: Number of unfragmented packets sent by the server.
- **_s_pkts_unk**: Number of unknown packets sent by the server.
- **_s_pkts_unrto**: Number of packets sent by the server that caused a retransmission timeout to be cancelled.
- **_s_pktsize1**: Size of the first packet sent by the server.
- **_s_pktsize_count**: Number of packets sent by the server.
- **_s_port**: Port number used by the server.
- **_s_rst_cnt**: Number of reset (RST) packets sent by the server.
- **_s_rtt_avg**: Average round-trip time (RTT) for packets sent by the server.
- **_s_rtt_cnt**: Number of RTT measurements for packets sent by the server.
- **_s_rtt_max**: Maximum RTT for packets sent by the server.
- **_s_rtt_min**: Minimum RTT for packets sent by the server.
- **_s_rtt_std**: Standard deviation of the RTT for packets sent by the server.
- **_s_sack_cnt**: Number of selective acknowledgement (SACK) packets sent by the server.
- **_s_sack_opt**: Indicates whether the server supports SACK.
- **_s_seg_cnt**: Number of segments sent by the server.
- **_s_sit1**: Size of the first segment sent by the server.
- **_s_sit_avg**: Average size of segments sent by the server.
- **_s_sit_std**: Standard deviation of the size of segments sent by the server.
- **_s_syn_cnt**: Number of SYN (synchronize) packets sent by the server.
- **_s_syn_retx**: Number of SYN packets retransmitted by the server.
- **_s_tm_opt**: Indicates whether the server supports the TCP timestamp option.
- **_s_ttl_max**: Maximum time-to-live (TTL) value used by the server.
- **_s_ttl_min**: Minimum TTL value used by the server.
- **_s_win_0**: Number of zero-sized window updates sent by the server.
- **_s_win_max**: Maximum window size used by the server.
- **_s_win_min**: Minimum window size used by the server.
- **_s_win_scl**: Indicates whether the server uses window scaling.
- **_tls_session_stat**: Status of the TLS (Transport Layer Security) session between the client and the server.
- **c_ip**: IP address of the client.
- **time**: Timestamp of the flow.
- **label**: Domain name.

## B Categories of features with no key word in the name for the aggregation

| Category | Features |
|---|---|
| count | '_c_bytes_uniq', '_c_pkts_ooo', '_c_pkts_push', '_c_pkts_reor', '_c_pkts_unrto', '_c_pkts_rto', '_c_pkts_unk', '_c_win_0', '_s_bytes_uniq', '_s_bytes_uniq', '_s_pkts_fc', '_s_pkts_fs', '_s_pkts_ooo', '_s_pkts_push', '_s_pkts_reor', '_s_pkts_rto', '_s_pkts_unk', '_s_pkts_unrto', '_c_pkts_fc', '_c_pkts_unfs', '_s_pkts_unfs', '_c_syn_retx','_s_syn_retx', '_s_win_0' |
| maximum | '_c_mss', '_s_mss' |
| boolean | _s_win_scl |

Table 10. Features with no keyword

## C Graphs of the pre-processing part

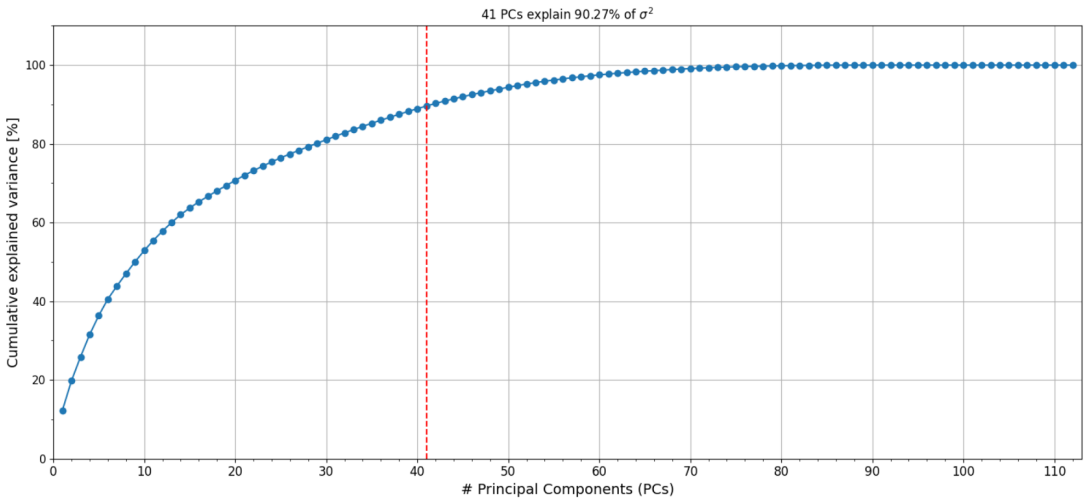



Fig. 6. Cumulative explained variance graph

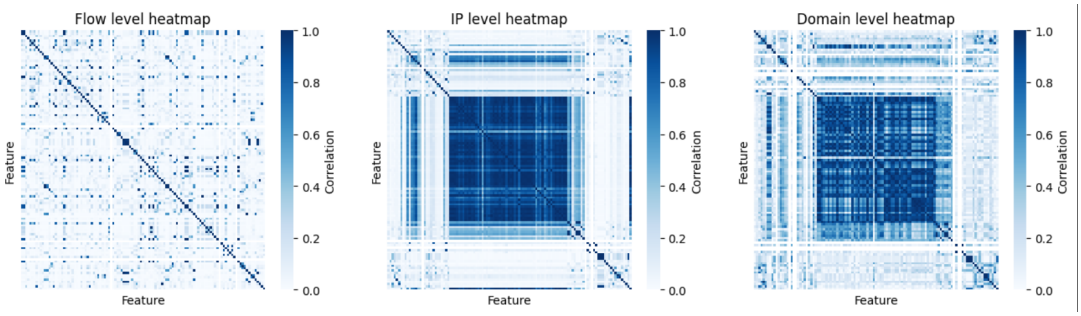



Fig. 7. Correlation heatmaps of the three levels