Machine learning for networking

Encrypted Traffic Data Collection and Classification

Group 28

Project Tutor:

Ciravegna Gabriele

Group members:

Bedini Niccolò

Lombardi Giovanni

Oldani Edoardo

Perna Fabrizio

ACADEMIC YEAR 2023/2024

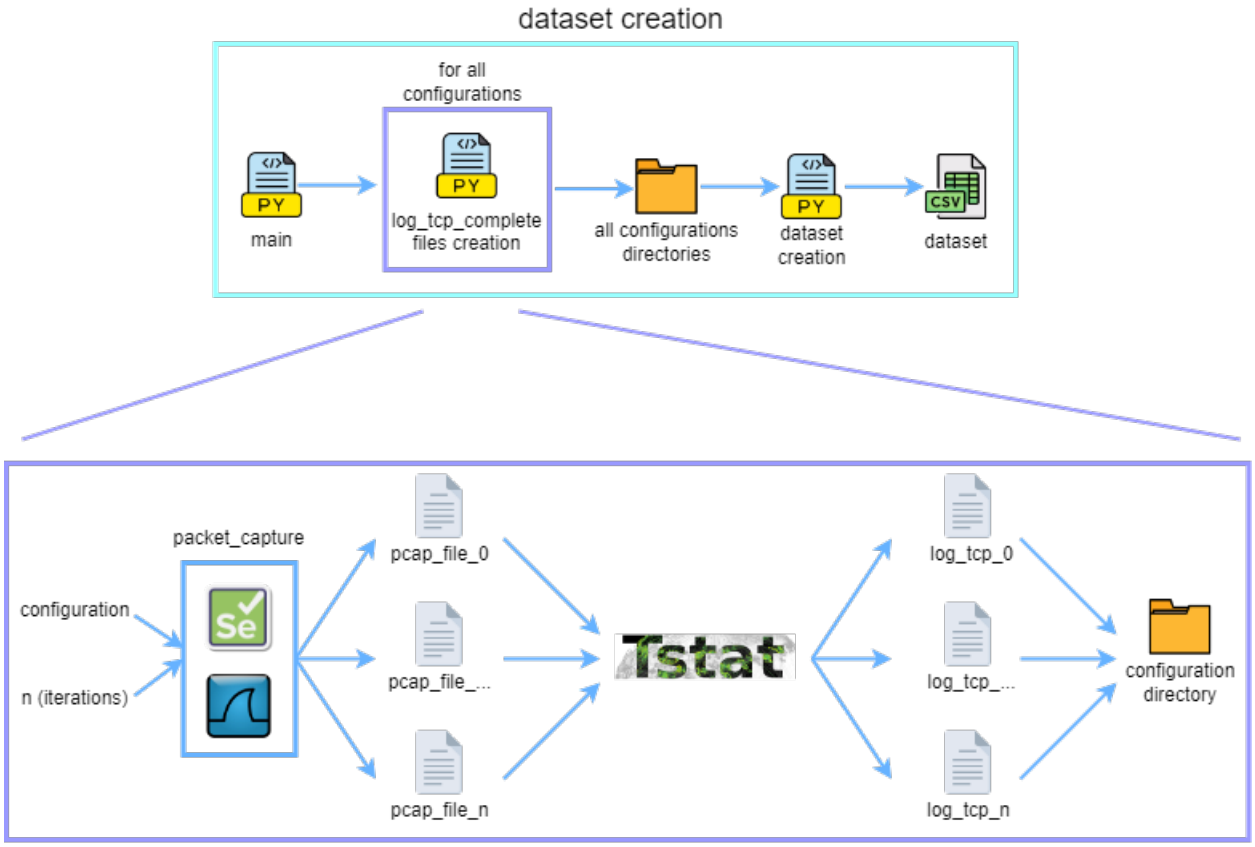
Contents

1 Introduction

In the world of cybersecurity, the adoption of secure communication channels has become crucial in securing user interactions and web traffic. Although encryption brings enormous security advantages, this creates many difficulties for monitoring this data. This project stems from the need to address these challenges and explore the potential of Deep Packet Inspection methods in recognizing information within encrypted packets. The scope of this project is centered around enhancing the monitoring capabilities of encrypted communication. Traditional methods, such as traffic classification based on port numbers, have faced limitations, particularly with the dynamic nature of ports and the rise of encrypted traffic. Our approach involves the use of machine learning techniques to improve the monitoring and identification throughout the network traffic. This exploration transcends traditional boundaries within monitoring systems, with the objective of providing insights into the feasibility and efficacy of machine learning in addressing the traffic. The methods implemented will attempt to identify patterns and information within encrypted packets headers, thus enhancing the overall monitoring capabilities. Through this research, our aim is to provide clarity on the feasibility and effectiveness of machine learning in enhancing monitoring capabilities. In doing so, we contribute to the ongoing debate on promoting advanced security practices and addressing challenges associated with encrypted communication.

2 Dataset Acquisition and Preprocessing

The first step of the project involves acquiring the dataset that will be used in subsequent phases. To achieve this goal, a Python module has been developed, its high-level flow is described in the following image:



log_tcp_complete files creation for one configuration

2.1 Browser simulation

The simulation of browser behavior was made possible through the use of Selenium WebDriver, a wide-spread library used to record user interactions such as clicks, selections, etc. in the browser and plays them back automatically. Every simulation involves the combination of three elements:

- browser;
- website;
- behaviour;

together, they define the concept of configuration. For the dataset construction, 60 configurations were defined, each simulated 100 times to achieve greater data consistency. Subsequently, a table displaying the configurations:

Browser	**Website**	**Behaviour**	**Associated Websites**
Chrome	Google	ML	Google, Youtube
Firefox	Youtube	Polito	Google, Youtube
Edge	Amazon	Trucebaldazzi	Google, Youtube
	Ebay	Amazon	Google, Youtube
		Selenium	Google, Youtube
		Xbox	Amazon, Ebay
		wii	Amazon, Ebay
		ps5	Amazon, Ebay
		gta6	Amazon, Ebay
		switch	Amazon, Ebay

2.1.1 The three main functions to achieve the goal

The module created for dataset construction consists of three main functions:

- **firefoxSimulation(function, number_iterations, website, key, csv_path, tstat_path);**
- **chromeSimulation(function, number_iterations, website, key, csv_path, tstat_path);**
- **edgeSimulation(function, number_iterations, website, key, csv_path, tstat_path);**

They differ only in the Selenium driver they refer to, for this reason, we will analyze just one of them. On the next page, the code for firefoxSimulation() is available.

```
def firefoxSimulation(function, number_iterations, website, key, csv_path, tstat_path):
  """Simulates the firefox browser and creates the tstat files related to the packets captured during the simulations


    function: the function that represents the behaviour of a website(ex. googleSearch)
    number_iterations: represents the number of iterations of the choosen function
    website: string that represents the website name associated to the function
    key: string that represents the key used inside the function
    csv_path: string that represents the path to the csv file which contains the metadata about the simulations
    tstat_path: string that represents the path to the folder which will contain all the tstat files after the simulation"""

  if not os.path.exists(tstat_path) and not os.path.isdir(tstat_path):
    os.mkdir(tstat_path)
  try:
    tstat_directory_path = os.path.join(os.path.abspath(tstat_path), "firefox_" + website + "_" + key)
    os.mkdir(tstat_directory_path)
    current_directory = os.path.abspath("./")
    options = Options()
    options.add_argument("--headless")
    options.add_argument("--no-sandbox")
    i = 0
    while i < number_iterations:
      driver = webdriver.Firefox(options = options)
      filename = "firefox_" + website + "_" + key + str(i) + ".pcap"
      full_path = os.path.join(tstat_directory_path, filename)
      try:
        packetCapture(function, driver, key, full_path)
      except:
        driver.quit()
        continue
      update_csv_file(csv_path, "firefox", website, key, full_path)
      driver.quit()
      i += 1
    tstatExtraction(os.path.join(current_directory, "labels.csv"), tstat_directory_path, number_iterations)
  except Exception as e:
    logging.error(f'Firefox error: {e}', extra={'website': website, 'key': key})
```

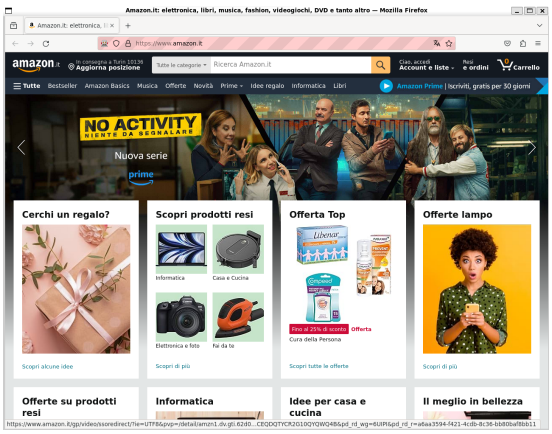
The function aims to simulate a configuration(for a given number of iterations) and create a folder containing all the necessary files for creating the dataset related to that configuration. Currently, we won't describe the roles of the functions **packetCapture()** and **tstatExtraction()** as they will be discussed in the subsequent sections. As mentioned earlier, one of the fundamental components of a configuration is the website. For each website that you want to automate, it is necessary to define a separate function because you need to identify the elements you want to interact with based on the HTML code of the page. For this reason, one of the fundamental parameters is **function**. Thanks to this parameter, the behavior of the function is independent of the website being visited. This allows extending the functionality of this module to websites different from those chosen for our configurations. For example, below there is the code for the function designed to simulate the behavior on YouTube:

```
1  def youtubeSearch(driver, key):
2    """Searches the key in the search bar of youtube and watches the first
     ↪  video for 2 seconds
3
4
5      driver: selenium driver which is used for the simulation
6      key: string that represents the key used inside the function"""
7
8    try:
9      driver.get("https://www.youtube.com")
10     WebDriverWait(driver,
       ↪  10).until(EC.presence_of_element_located((By.XPATH,
       ↪  "/html/body/ytd-app/ytd-consent-bump-v2-lightbox/tp-yt-paper-dial
       ↪  og/div[4]/div[2]/div[6]/div[1]/ytd-button-renderer[1]/yt-button-s
       ↪  hape/button/yt-touch-feedback-shape/div/div[2]"))).click()
11     time.sleep(2)
12     search_box = driver.find_element(By.XPATH,
       ↪  '/html/body/ytd-app/div[1]/div/ytd-masthead/div[4]/div[2]/ytd-sea
       ↪  rchbox/form/div[1]/div[1]/input')
13     search_box.click()
14     search_box.send_keys(key)
15     search_box.submit()
16     WebDriverWait(driver,
       ↪  10).until(EC.presence_of_element_located((By.ID,
       ↪  "video-title"))).click()
17     time.sleep(1)
18   except Exception as e:
19     logging.error(f'Errore youtubeSearch: {e}', extra={ 'key': key})
20     raise
21   return
```

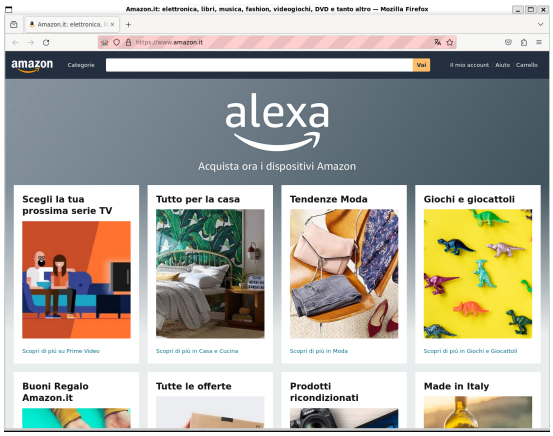
In this case, the function accesses the YouTube web page, performs a search with the specified keyword, and watches the first indexed video for two seconds.

2.1.2 Exception handling

Another relevant aspect is exception handling. Specifically, during the testing phase, situations were identified where the displayed page was a placeholder for the actual web page, likely used to handle requests made to the server. The presence of such placeholder pages leads to the capture of incorrect packets during an iteration, as they do not reflect the data traffic related to the actual page. Additionally, since the interaction with elements on a web page heavily depends on the HTML code, the driver may not be able to locate the specified element in the code (e.g., the search bar). Without proper exception handling, this could result in a script crash during execution. Below is a comparison between the placeholder page and the actual Amazon page.



(a) Actual Amazon page



(b) Placeholder

Given that the captured packets in the exception handling context are not relevant for creating a consistent dataset, the exception handling involves reattempting the same iteration. Below is a focus on the code snippet related to exception handling:

```
while i < number_iterations:
  driver = webdriver.Firefox(options = options)
  filename = "firefox_" + website + "_" + key + str(i) + ".pcap"
  full_path = os.path.join(tstat_directory_path, filename)
  try:
    packetCapture(function, driver, key, full_path)
  except:
    driver.quit()
    continue
  update_csv_file(csv_path, "firefox", website, key, full_path)
  driver.quit()
  i += 1
```

2.2 Data recording

In order to capture packets related to the simulations is used Tshark, a terminal extension of Wireshark. Since the Python module was created to automatically generate the dataset, the following function has been implemented to initiate packet capture with TShark and save them in a .pcap file:

```
1  def packetCapture(function, driver, key, pcap_path):
2    """Initiates packet capture through tshark with a terminal command,
     ↪  starts the individual simulation, and saves the pcap file in the
     ↪  pcap_path
3
4    function: the function that represents the behaviour of the a website
     ↪  (ex. googleSearch)
5    driver: is the Selenium driver of the relative browser
6    key: string that represents the key used in the function
7    pcap_path: string that represents the path where will be saved the pcap
     ↪  file that contains the recorded traffic (used in the temporary
     ↪  csv)"""
8
9    open(pcap_path, "w")
10   tshark_command = ["tshark", "-i", "eth0", "-F", "pcap", "-w",
     ↪  pcap_path, "-f", "tcp port 80 or tcp port 443" ]
11   process = subprocess.Popen(tshark_command)
12   try:
13     function(driver, key)
14   except:
15     process.terminate()
16     os.remove(pcap_path)
17     raise
18   process.terminate()
```

As can be observed from line 13, the simulation of a website's behavior is initialized within this function. The reason is that, in order to develop a fully automated tool, it was necessary to create a layering of the steps required for building the dataset. In this way, all components can be combined within the three main macro functions presented in the previous section. Another important consideration is the presence of the **pcap_path** parameter. Since the .pcap files outputted by TShark are needed for extracting TCP information in the subsequent step, it's crucial to keep track of the path where these files are stored. Metadata about the simulation and the .pcap file's path are stored in a .csv file, which is essential for extracting information related to the captured traffic. In the event of an anomalous simulation, such as the one shown in the figure 30b, the exception involves deleting the created .pcap file and terminating TShark. By using the "raise" command, it's possible to defer the exception handling to the macro functions that utilize packetCapture().

2.2.1 Isolation with Docker

During the execution of the data acquisition of the network packets it is possible to record data from other connections. To mitigate this, the script was executed using a Docker container. In addition to the isolation, the choice to container the execution of the script was given by the fact that Tstat runs only on a Linux environment that we could achieve with Docker. The container configuration file (dockerfile) contains only the commands necessary to execute the script and to record the network's packets. The initial commands to be included in the configuration file, to enable the execution of the developed Python script, are the commands associated with running the Python script, with the requisite libraries, shown subsequently:

```
RUN apt-get update && apt-get install -y \
    wget \
    unzip \
    curl \
    subversion \
    python3-pip

RUN pip3 install selenium

RUN pip3 install pandas
RUN pip3 install scikit-learn
```

In addition to the commands for executing the Python script, the installation of drivers was pivotal to enable dynamic utilization of the browsers employed within the script. The following code was used to install the drivers and all the associated dependencies:

```
# Install Firefox
RUN apt-get update && apt install -y firefox-esr \
    && wget https://github.com/mozilla/geckodriver/releases/download/v0.3
    ↪  3.0/geckodriver-v0.33.0-linux64.tar.gz
    ↪  \
    && tar -xvf geckodriver-v0.33.0-linux64.tar.gz \
    && rm geckodriver-v0.33.0-linux64.tar.gz \
    && mv geckodriver /usr/local/bin

## Install Chrome
RUN apt-get update \
   && wget https://dl.google.com/linux/direct/google-chrome-stable_curren
   ↪  t_amd64.deb
   ↪  \
   && apt-get install -y ./google-chrome-stable_current_amd64.deb \
   && apt-get install -f \
   && wget https://edgedl.me.gvt1.com/edgedl/chrome/chrome-for-testing/12
   ↪  0.0.6099.71/linux64/chromedriver-linux64.zip
   ↪  \
   && unzip chromedriver-linux64.zip \
   && mv chromedriver-linux64/chromedriver /usr/local/bin \
```

```
16      && rm -r c*
17
18  ### Install Edge
19  RUN apt-get update \
20      && curl https://packages.microsoft.com/keys/microsoft.asc | gpg
        ↪  --dearmor > microsoft.gpg \
21      && install -o root -g root -m 644 microsoft.gpg
        ↪  /etc/apt/trusted.gpg.d/ \
22      && sh -c 'echo "deb [arch=amd64]
        ↪  https://packages.microsoft.com/repos/edge stable main" >
        ↪  /etc/apt/sources.list.d/microsoft-edge-dev.list' \
23      && rm microsoft.gpg \
24      && apt update \
25      && apt-get install -y microsoft-edge-dev \
26      && wget https://msedgedriver.azureedge.net/121.0.2277.4/edgedriver_li
        ↪  nux64.zip
        ↪  \
27      && unzip edgedriver_linux64.zip \
28      && mv msedgedriver /usr/local/bin \
29      && rm edgedriver_linux64.zip  \
30      && rm google*
```

Finally, the following commands detail the installation of TShark and TStat along with all their dependencies:

```
1   # Install tshark
2   RUN apt-get install -y tshark
3
4   #### Install tstat
5   RUN apt-get update \
6       && apt-get install -y autoconf automake libtool \
7          libpcap0.8-dev \
8          libcap2 \
9       && svn checkout http://tstat.polito.it/svn/software/tstat/trunk tstat
        ↪  \
10      && cd tstat \
11      && ./autogen.sh \
12      && ./configure --enable-libtstat --enable-zlib \
13      && make \
14      && make install
```

Upon completion of the Docker configuration file, given its considerable advantages in terms of portability, we were able to execute the container on all team members' computers. This Docker feature obviated the need for a more complex and resource-intensive virtual machine, enabling us to parallelize the execution of the data acquisition script across our PCs. Furthermore, the container, operating as an isolated environment, ensures that the execution of the script and the capture of network packets remain isolated from all potential connections within a complete computer or virtual machine, including background-running programs.

2.3 TCP statistics extraction

Once all the .pcap files are saved using Tshark it's needed to get the most important TCP data of those packets. To achieve this, it was used Tstat, a tool developed by the Polytechnic University of Turin that gives the possibility to extract all statistics regarding the client-server connection. Among the various options provided by tstat, the selection was made to report TCP statistics through the Log Tcp-Complete with reporting all TCP traces found in the packets. The Tstat execution on a .pcap file results in few rows of data derived by the packets recorded which are stored in a log_tcp_complete file. Below is a screenshot of the structure of a log_tcp_complete file. For simplicity, only 7 out of 134 attributes are shown with their respective values:

To extract all the information using Tstat, a function has been devised, and its code is presented below.

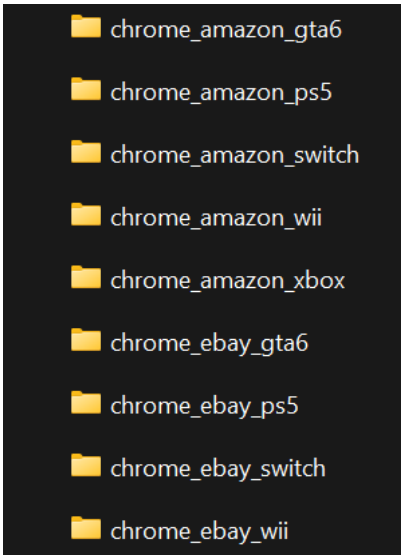
```
def tstatExtraction(csv_path, tstat_path, number_iterations):
  """ The purpose of this function is to extract, from all the pcap files
  ↪  of a configuration obtained through tshark, the corresponding tstat
  ↪  files containing TCP information of the captured traffic.
  From these, the dataset will then be created.

  csv_path: this is the path that indicates the location of the CSV file
  tstat_path: is the path indicating where the tstat file will be saved,
  ↪  extracted from its corresponding pcap file.
  number_iterations: is the count of times a configuration is executed.
  ↪  """

  current_directory = os.getcwd()
  df = pd.read_csv(csv_path, header=None)
  len_index = len(df.index)
  start_index = len_index - number_iterations
  for i in range(start_index, len_index):
    try:
      tstat_command = ["tstat", df.iloc[i,-1]]
    except Exception as e:
      continue
    process = subprocess.Popen(tstat_command)
    time.sleep(0.1)
    os.remove(df.iloc[i,-1])
    time.sleep(2)
    process.terminate()
  os.chdir(tstat_path)
  list_dir = os.listdir()
  for i in range(len(list_dir)):
    os.chdir(list_dir[i])
    sub_list = os.listdir()
    os.chdir(sub_list[0])
    os.rename("./log_tcp_complete","./log_tcp_complete_" + str(i))
    shutil.move("./log_tcp_complete_" + str(i), tstat_path)
    os.chdir(tstat_path)
    shutil.rmtree(list_dir[i])
  os.chdir(current_directory)
```

The function has three parameters:

1 **csv_path:** This is the path to the CSV file containing all the simulation metadata. Among the values, there is a field containing the path to the .pcap file related to the specific configuration, which is crucial in this phase. Below is an example of the CSV file format:

browser_name, website, key, pcap _path

2 **tstat_path:** This is the path to the folder where all the log_tcp_complete files will be saved after extraction with Tstat. The folder is named after the configuration it refers to and is created within the **firefoxSimulation()** function. This folder will be stored inside another directory, facilitating the organization of folders containing all the log_tcp_complete files in one place for easier processing during the dataset creation phase. Below is an image of the root directory:



3 **number_iterations:** This represents the number of iterations performed for each configuration. Since the CSV file containing metadata is unique, this parameter allows proper indexing during this phase.

As can be observed from the code, the extraction process and its subsequent saving to the directory are fully automated. This is a crucial requirement, especially considering the high number of configurations (and associated iterations) needed for building a comprehensive dataset.

2.4 Data Preprocessing

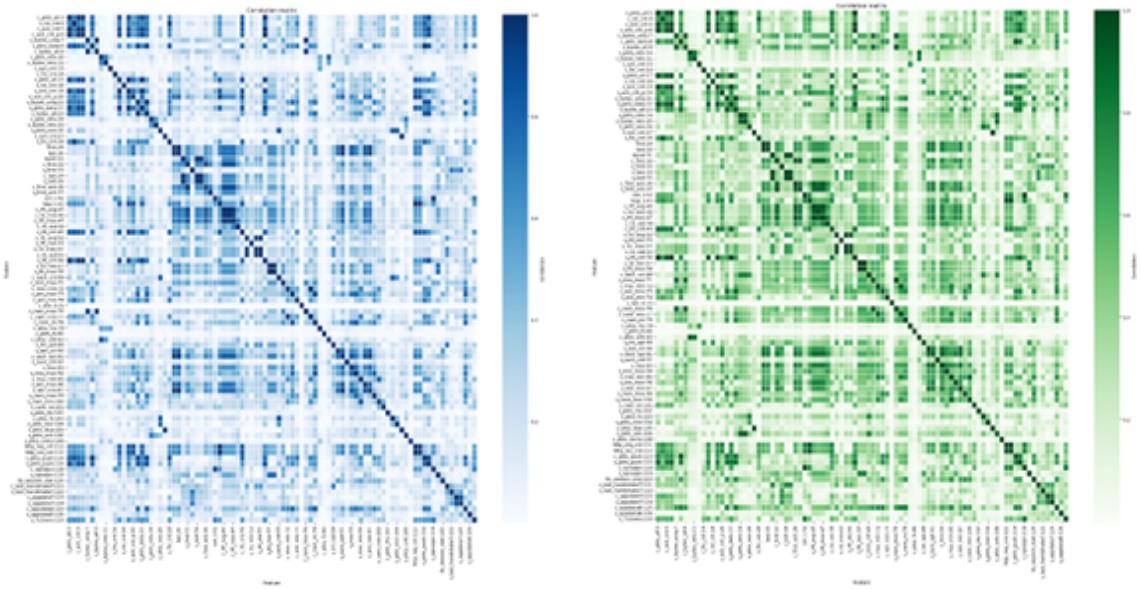
After recording the packets and analyzing them with Tstat the data acquired have to be pre-processed before actually using them to create the dataset. First of all, since the data in the log_tcp_complete file could contain more than one line, the mean of the rows is calculated, in this way, a single row is returned. To calculate the mean, all features containing strings have been set to 0. This does not compromise the validity of the dataset, as after analysis, it was determined that these string features were not deemed essential for the calculations. Therefore, the IP addresses and ports of both the client and server are then removed. Also, for each row of data the labels to recognize the browser, website and behaviour are added. The three labels are directly extracted from the name of the folder containing the log_tcp_complete file. This allowed us to eliminate the .csv file containing all the metadata and simplify this phase of the process.

```
def datasetCreation(path):
  """Creates the dataset from the log_tcp_complete files taken from each
  ↪  simulation, each log_tcp_complete file contains more than one line
  ↪  so in order to
     obtain a single object, the mean of the lines inside the file is
     ↪  computed


    path: string that represents the path to the folder which contains
    ↪  all the directories with the log_tcp_complete files taken from
    ↪  each simulation"""

  os.chdir(path)
  list_dir = os.listdir()
  dataset = pd.DataFrame()
  for dir in list_dir:
    os.chdir(dir)
    labels = dir.split("_")
    list_files = os.listdir()
    for log_tcp_complete in list_files:
      df = pd.read_csv(log_tcp_complete, delimiter=" ")
      columns_to_drop = ['#15#c_ip:1', 's_ip:15', 'c_port:2', 's_port:16']
      df = df.drop(columns=columns_to_drop)
      to_remove = df.select_dtypes(include=object)
      df[to_remove.columns] = 0
      column_mean = df.mean()
      mean_df = pd.DataFrame(column_mean).transpose()
      mean_df = pd.concat([mean_df, pd.DataFrame({'browser': [labels[0]],
      ↪  'website': [labels[1]], 'behaviour': [labels[2]]})], axis=1)
      dataset = dataset._append(mean_df)
    os.chdir("..")
  dataset.to_csv("../dataset.csv", index = False)
```

3 Unsupervised data analysis

3.1 Dimensionality Reduction

Starting with the unsupervised data analysis, our focus was on performing a series of preprocessing operations on the dataset. In the initial phase, we aimed to enhance the dataset's coherence and interpretability. To achieve this, we implemented standardization using StandardScaler, ensuring a consistent scale across the various features. Additionally, we filtered the dataset by eliminating columns where all values were zero, thereby simplifying the data representation. Afterwards, we created a 'type' column to uniquely label each data type following the multi-class approach. This strategic approach not only refined the dataset's structure but also laid the groundwork for a more comprehensive understanding of the underlying patterns and relationships within the data. Following the normalization of our dataset, our attention shifted towards gaining deeper insights into the interrelationships among features. To achieve this, we computed both the correlation matrix and the Spearman matrix, exposing intricate patterns within the data.



(a) Correlation matrix (b) Spearman matrix

In the subsequent stage of our analysis, we harnessed the power of dimensionality reduction techniques, specifically employing PCA (Principal Component Analysis) and KPCA (Kernel Principal Component Analysis). These techniques transform the original features into a new set of uncorrelated variables, called principal components, with the goal of capturing the maximum variance in the data. Following the application of PCA, we calculated the cumulative explained variance. This metric, when expressed as a percentage, serves as a key indicator for identifying the optimal number of components.

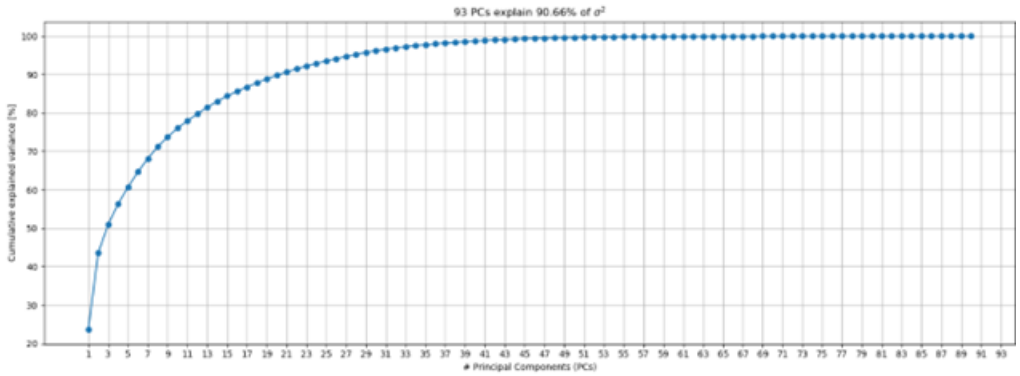


Figure 3: Cumulative explained variance

From the previously presented graph, it is evident that the curve exhibits exponential growth as the number of components increases. Based on this curve and considering all possible component values, we opted to utilize 21 components. This choice is informed by the fact that with this specific number, we can retain 90% of the variance. In essence, employing these 21 components enabled us to have a smaller number of components, different from the original dataset, that are informative or "explanatory" at a high percentage of the dataset itself. To prove this, we computed the correlation matrix for the dataset derived from PCA, retaining only 21 features. As we thought, the new matrix is shows that all components exhibit a correlation value of zero, except on the diagonal.

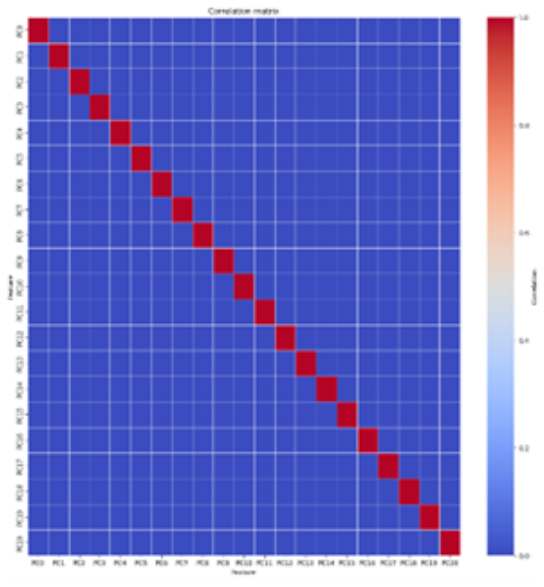


Figure 4: Correlation matrix after PCA

Following the verification on the PCA results, we conducted an additional PCA analysis, aiming to unveil the relationships between features and components. This involved calculating the loading scores for each feature across the identified components. The graph 5 represent the most significant feature loadings for the initial five components.

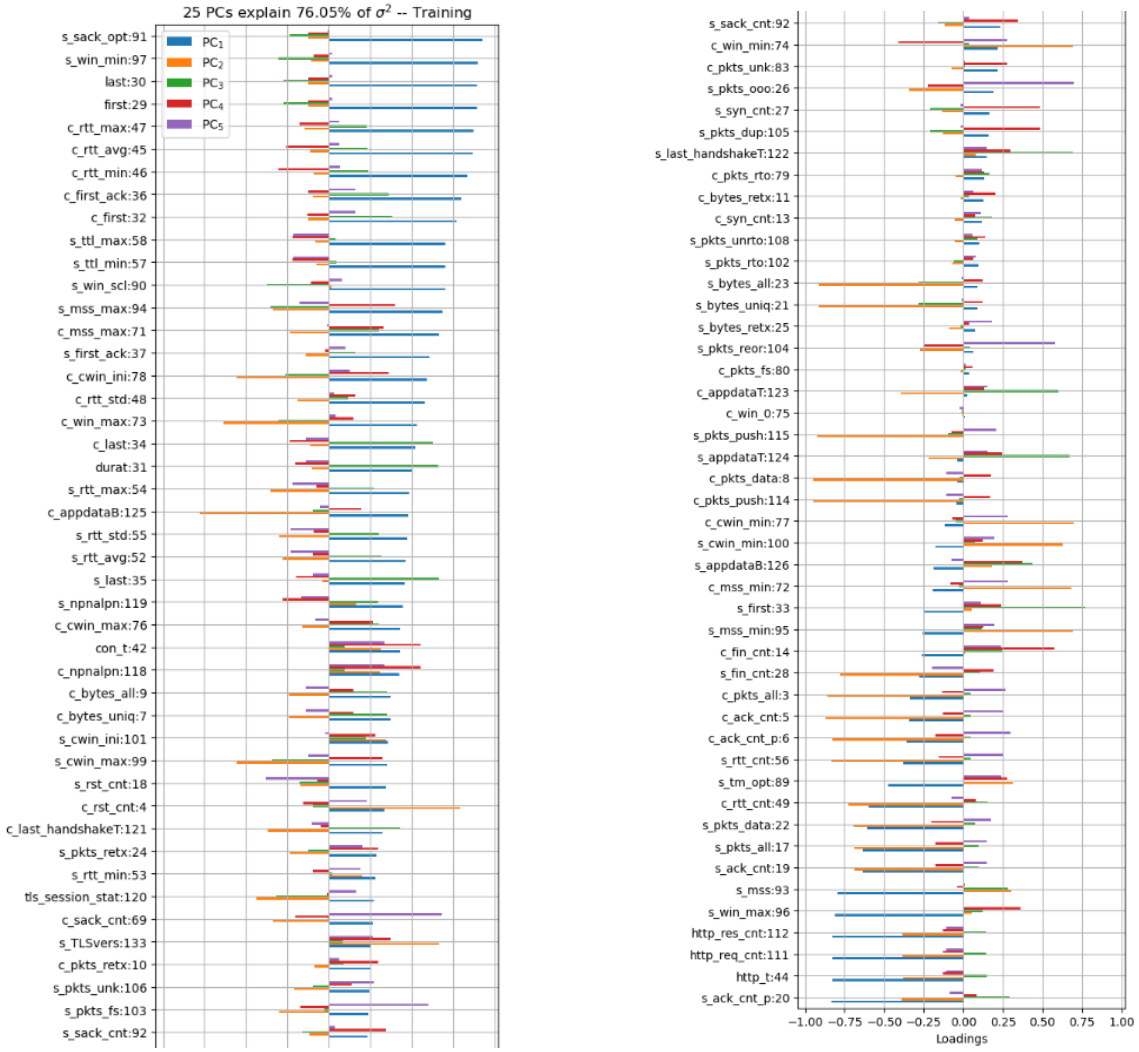


Figure 5: Loadings

In pursuit of comprehensive validation through diverse dimensionality reduction techniques, we extended our analysis to include Kernel PCA (KPCA). While the fundamental process aligns with PCA, a distinct approach is employed in this instance to ascertain the optimal number of components. To achieve this, we established a designated list, "reconstruction errors," designed to encapsulate the reconstruction errors associated with each tested number of KPCA components by selecting a range of components, specifically from 10 to 19, aligned with our project objectives. The iterative process unfolded as follows: for each chosen number of components, a new dataset was generated employing KPCA, transforming the data to capture its inherent structure. This transformed dataset then underwent a reconstruction phase to provide an approximate representation of the original dataset. The reconstruction error graph displayed is the graph 6.

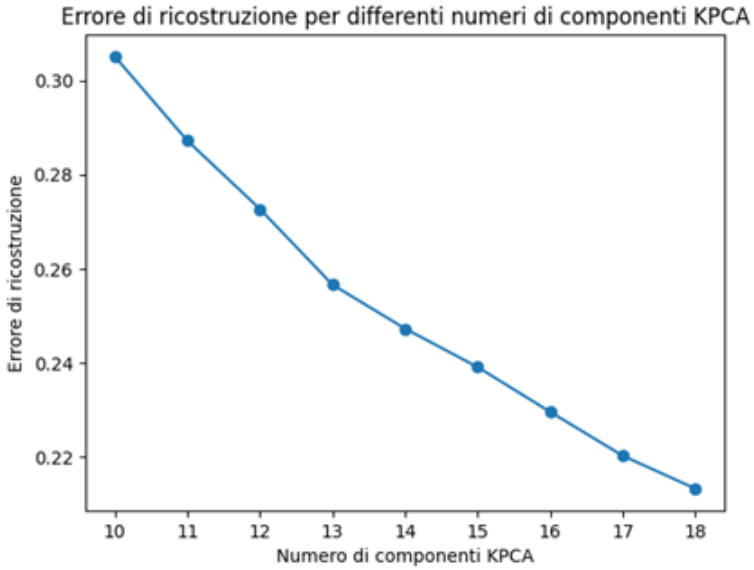


Figure 6: Reconstruction error for different number of components of KPCA

Then, using the elbow method, we chose a number of components equal to 13, representing a balanced tradeoff between the number of components used and the reconstruction error of the original dataset. At this point, we applied KPCA with 13 components to transform the dataset.With both transformed datasets with the two techniques of dimensionality reduction, we compared the impact on the inherent structure of the original data in the graph 7.

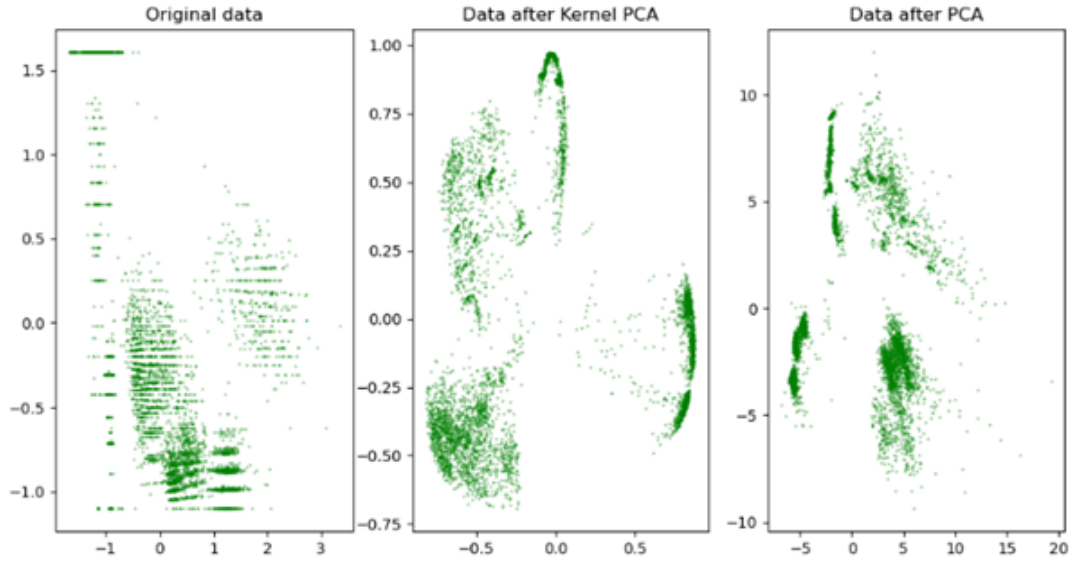
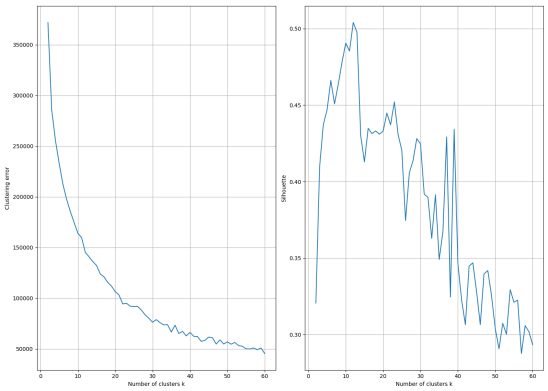


Figure 7: Comparison between original dataset, KPCA and PCA

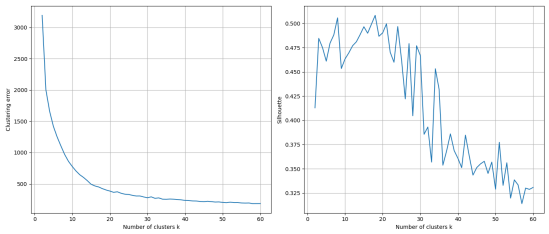
In conclusion, we examined the original dataset and normalized it to facilitate the application of machine learning techniques, enabling a more in-depth analysis of its contents. Subsequently, we employed two distinct dimensionality reduction techniques, both of which yielded interesting results. Since PCA is a linear technique it is effective in capturing linear relationships among features. On the other hand, KPCA, with its ability to address nonlinear relationships, is optimal in handling complex or nonlinear patterns present in network packet data that the standard PCA could not capture. Given the significance of these findings, we opted to extend our analysis further by utilizing both techniques for cluster identification. This strategic approach aims to provide a comprehensive overview of the collected data.

3.2 Clustering Algorithms

Our goal following the dimensionality reduction was to explore the relationships within the TCP flow data of our dataset using unsupervised learning methods. In the first step, this analysis was based only on the features' information. Later, we aimed to compare these findings with the original labels to identify if there are direct correlations and determine which labels are particularly distinctive. We initially decided to examine the results for both the dataset post-PCA and KPCA, as these two reduction methods might have preserved different relationships compared to the original dataset. We think that this approach is particularly interesting as it acknowledges the unique capabilities of PCA and KPCA in capturing different aspects of data variance and structure. By evaluating the performance of both methods, we aim to determine which dimensionality reduction technique more effectively reveals the proximity of the data as highlighted by the clustering in comparison to the original labels. In our analysis, we initiated a methodical search for discovering the most promising number of clusters based on our thoughts by focusing on the silhouette metric during hyper-parameter tuning. This approach was designed to balance cluster cohesion and separation effectively. Simultaneously, we contemplated the inherent label combinations within the dataset, aiming to align our unsupervised learning with the data's underlying structure. Regarding the hyper-parameters, we considered that a suitable range for k (number of clusters) would be from 2 to 61, as there are 60 potential configurations in our dataset and anyway there could be clusters also for low values.



(a) PCA



(b) KPCA

Silhoutte consideration: After examining the silhouette score graphs, we observed that the values were not exceptionally high. This outcome is likely attributable to data overlaps, a consequence of how the dataset was constructed. This method of data generation inherently leads to a degree of overlap in the dataset, which in turn affects the distinctiveness of clusters and reduces the silhouette scores. This overlap suggests the complexity of the relationships within our data and the challenges in clearly delineating distinct groups or patterns. It is worth noting that these observations may be independent of the labels assigned to the data and are solely based on inherent characteristics. In addition, the decreasing trend of the silhouette score with increasing k may substantiate this hypothesis, as one might expect a peak at k = 60 due to the combination of browser, website, and label, which is likely absent due to overlap.

Clustering error consideration: As evident from the graphs, KPCA exhibits a significantly lower clustering error compared to PCA. Consequently, it is essential to consider the implications of the curse of dimensionality in the context of clustering analysis. In the case of PCA, with its utilization of 21 dimensions, the curse of dimensionality may exacerbate the clustering error. The different number of components implies that the distance calculation from centroids is more penalizing for the PCA rather than KPCA.

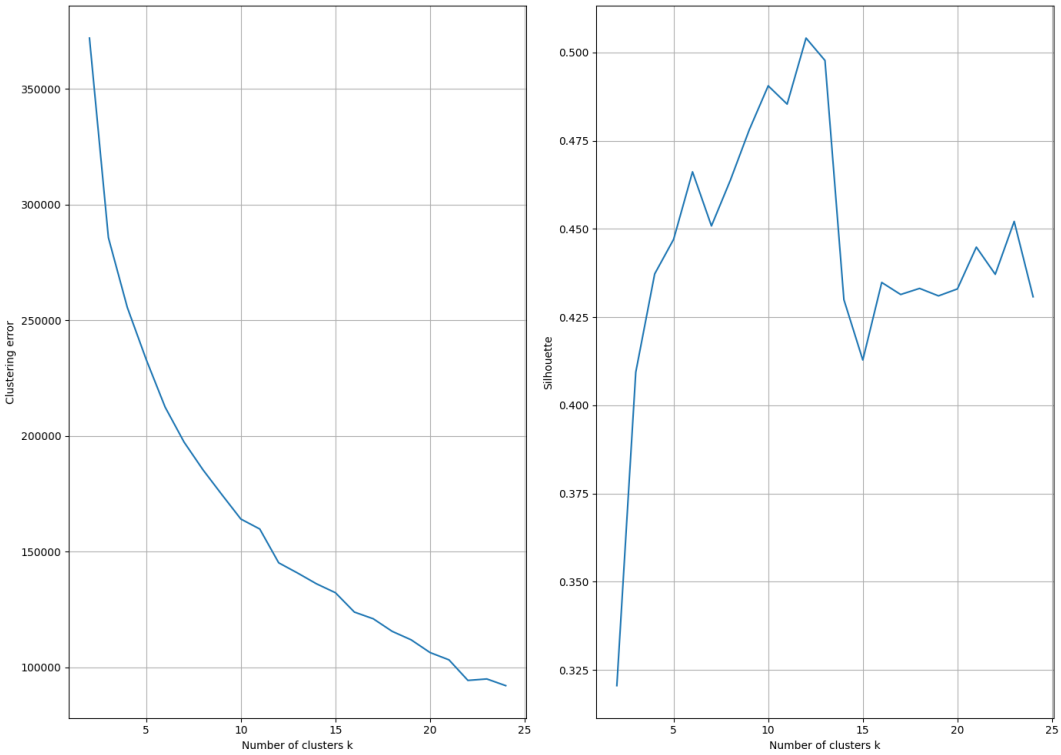
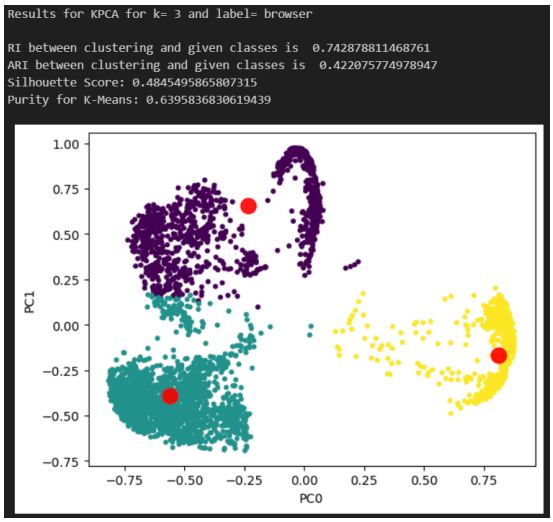


Figure 9: PCA silhouette

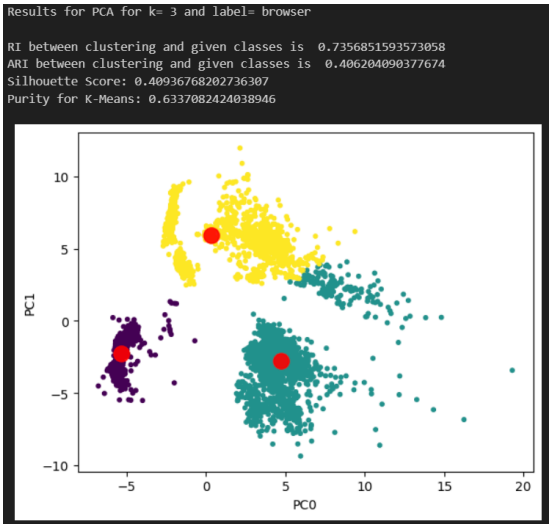
PCA consideration: The graph 9 resulting from the PCA-reduced dataset initially suggests the best silhouette values and a relatively low clustering error, as indicated by the elbow visualization around 12 clusters, with performance progressively deteriorating as the number of groups increases. While 12 is a plausible number, corresponding possibly to the *browser-website* label combination, which we believe could be most distinctive, we decided to delve deeper into the analysis. This decision stems from our understanding that the values emerging from clustering might be influenced by relationships not directly related to the labels.

KPCA consideration: Regarding KPCA, there appear to be no substantial differences in terms of silhouette compared to PCA. For this reason, our approach was to employ external metrics such as ARI and purity to assess which of the two dimensionality reduction methods was more suitable for unsupervised learning. To accomplish this, we encode each label combination as distinct classes within a unified label, following the multi-class approach.

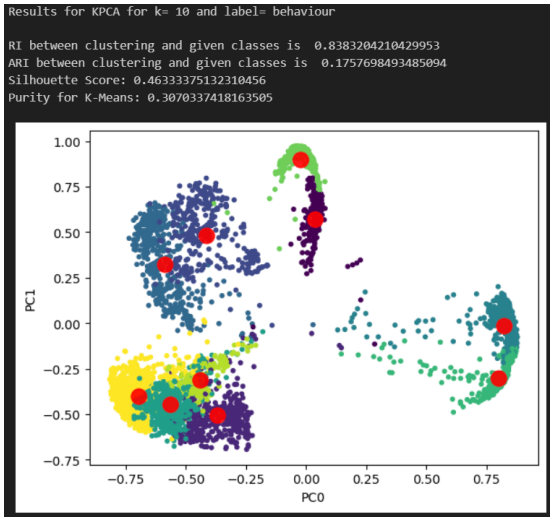
Our subsequent step involved a deeper exploration of clusters with favorable silhouette scores, to assess their purity and ARI in relation to potential label-dependent divisions. Below are shown the graphs and associated results:



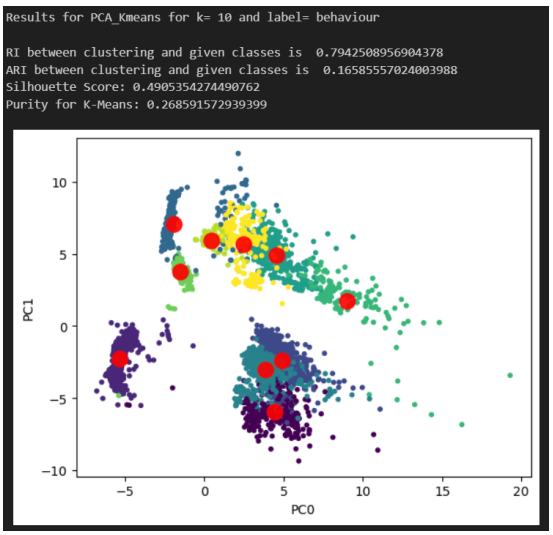
(a) KPCA-3 clusters



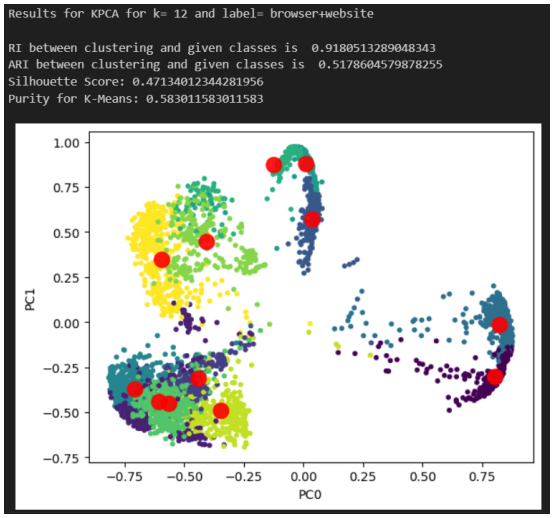
(b) PCA-3 cluster



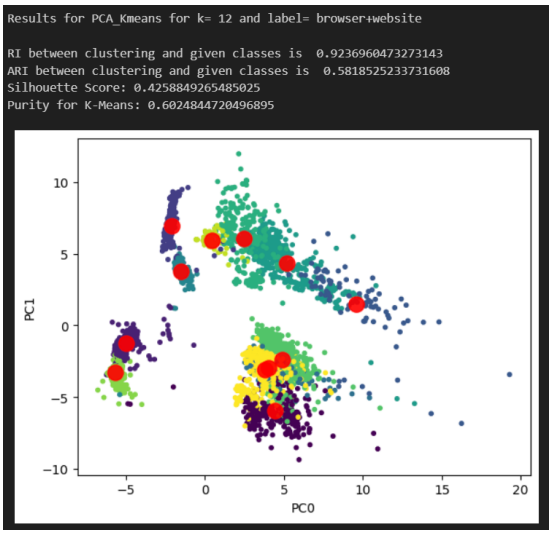
(a) KPCA-10 clusters



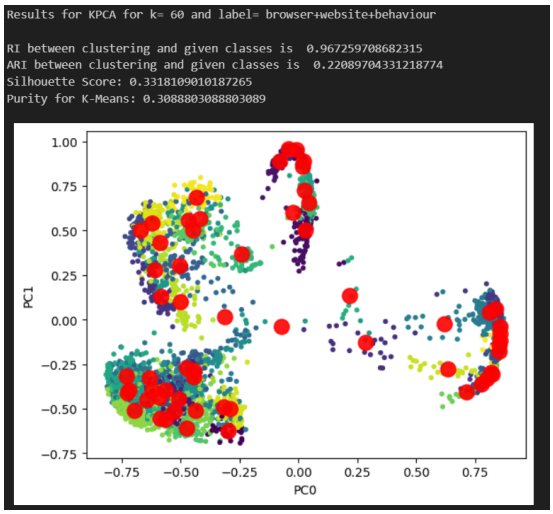
(b) PCA-10 clusters



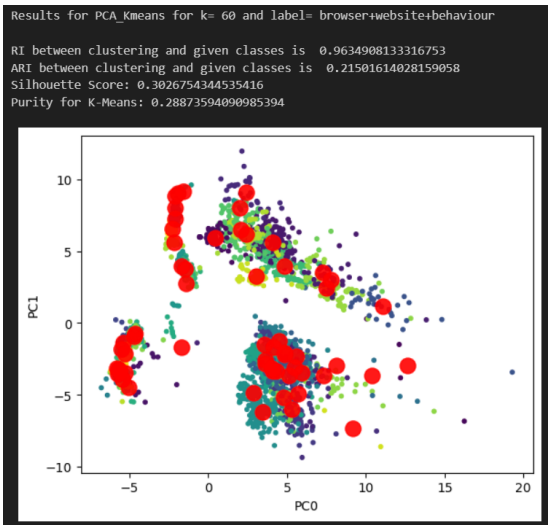
(a) KPCA-12 clusters



(b) PCA-12 clusters



(a) KPCA-60 clusters



(b) PCA-60 clusters

It is noteworthy that, despite KPCA having a lower explained variance of the original dataset, it yields results in unsupervised learning comparable to those of PCA, even concerning metrics referencing ground truth. Nevertheless, we observed overall better results in terms of purity and ARI for the case of 12 clusters with PCA, confirming the findings obtained from the silhouette graph presented initially. For this reason, we opted to perform anomaly detection on the dataset transformed through PCA. It is also imperative to consider that excellent results were obtained for k = 3, suggesting that data extracted from different browsers may exhibit distinctiveness.

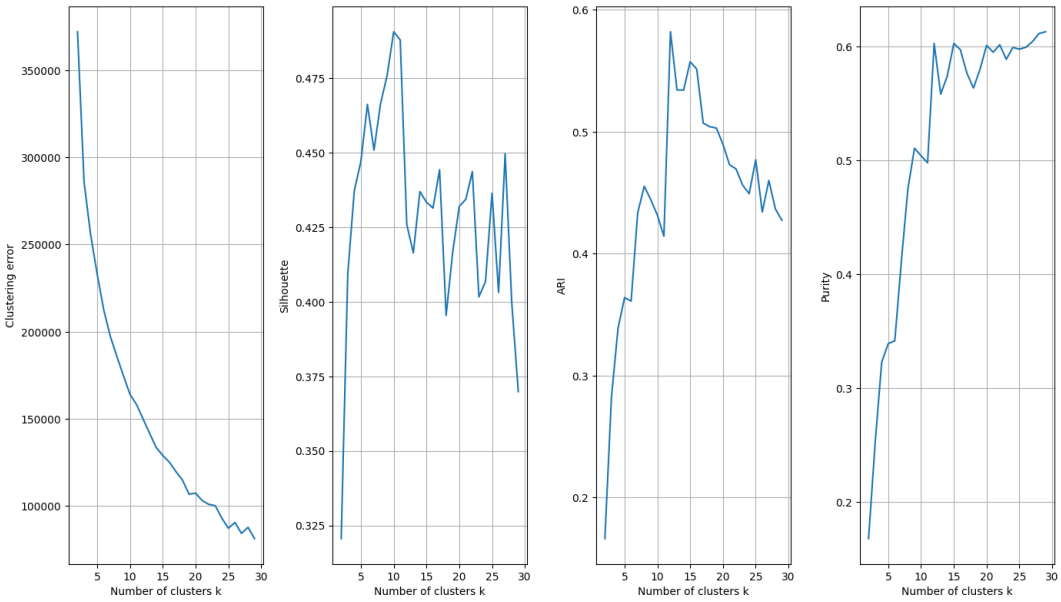
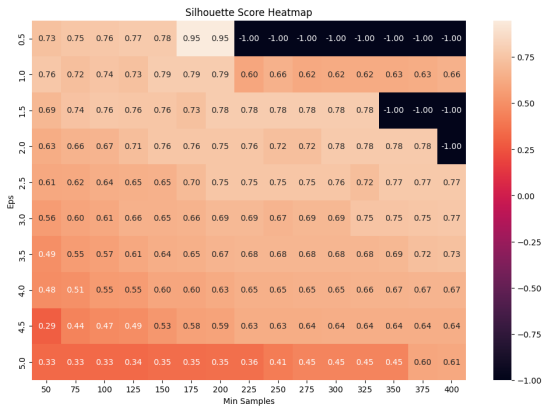


Figure 14: Evaluation of cluster with label=browser-website

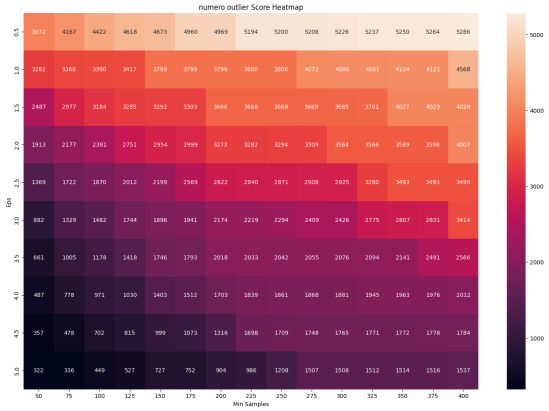
3.3 Anomaly Detection

As a next step, we aim to assess whether the presence of noise or outliers in our specific case of interest can be significant in the division of clusters, and whether it can 'improve' the results in the case of our primary focus, k = 12. Indeed, evaluating the impact of noise and outliers is a crucial aspect of any cluster analysis, especially in complex datasets like network traffic. To accomplish this, we will utilize the DBSCAN clustering algorithm. In our exploration with DBSCAN, we initially focused on optimizing *min_samples* and *eps* to balance between a high silhouette score and a manageable outlier count. We prioritized the silhouette score as the metric for outlier identification and removal, ensuring that this process remained true to the core principles of unsupervised clustering: to uncover and understand the natural groupings within the data, independent of any pre-assigned labels. This approach allows us to first refine our dataset without any biases introduced by the labels, potentially leading to a more authentic representation of the underlying data patterns. Subsequently, we plan to revisit the cleaned data by reusing label-dependent metrics like ARI and purity in a k-means context, comparing these new results with those previously found. This subsequent step is designed to assess the impact of outlier removal on the congruence between our clusters and the original labels, offering a comprehensive view of how our unsupervised learning efforts align with the ground truths.

Here the images related to the hyper-parameter tuning:



(a) Heatmap silhouette



(b) Heatmap outliers

Consideration: We can observe that for many hyperparameter configurations, the number of possible outliers found by DBSCAN is very high. We attribute this event to the mutable nature of the network and to the fact that the packet capture representing the dataset was carried out simultaneously but separately by the four members of the group. It became apparent that an enhanced silhouette was often accompanied by an increased number of outliers. However, completely discarding these outliers would strip away valuable insights, rendering the dataset less representative of our project's goals (as exemplified in the DBSCAN plot with 0.5 eps and 200 min_samples). Therefore, the solution was to seek a tradeoff between a good silhouette score and a reasonable number of outliers, so we chose 3 eps e 200 min_samples.

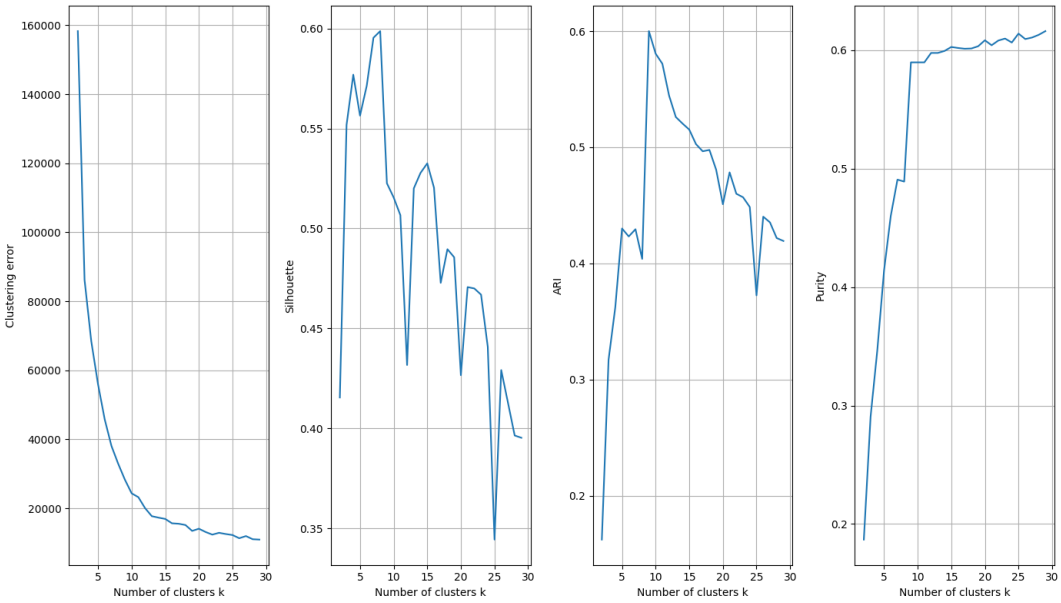


Figure 16: DBSCAN PCA

Upon re-evaluating the clustering for k=12 on the dataset from which points identified as outliers by DBSCAN were removed, it is evident that the results remain unchanged. This could be attributed to the possibility that these outliers represent isolated points or exhibit characteristics that do not substantially impact the relationships between the data or the cluster structure. As a result, the removal may not significantly alter the outcomes.

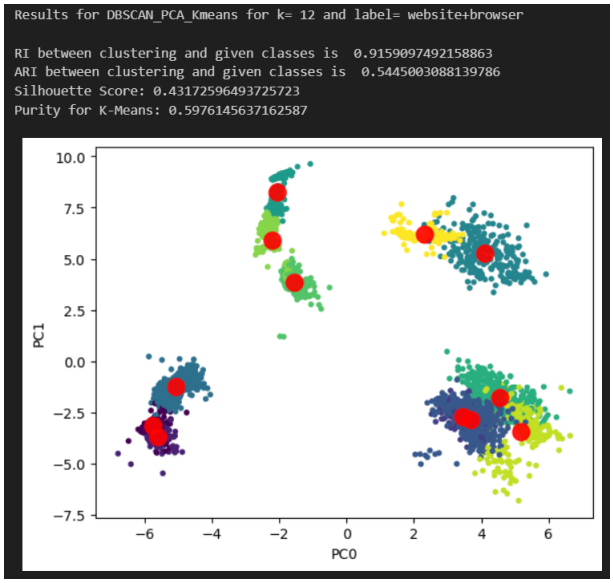


Figure 17: KMeans DBSCAN

In conclusion, it has been observed that, for unsupervised learning, the values of k yielding noteworthy results are 3 and 12. Consequently, in the supervised learning segment, we will build upon these observations, focusing our analysis particularly on the respective label combinations:

- **browser;**
- **browser-website.**

4 Supervised data analysis

The initial step involves splitting the dataset into training and validation sets ensuring that they are distributed in a balanced manner. Following this, we have opted for two models that employ distinct approaches for classifying the samples:

- Decision Tree;
- KNN(K-Nearest Neighbours).

Regarding label combinations, we follow the multi-class approach, reminiscent of the methodology utilized in unsupervised learning (e.g., grouping browsers and websites under a consolidated label). For simplicity, graphs will be presented for a single label combination. However, metrics pertaining to other combinations will also be provided, facilitating subsequent considerations and comparisons. Furthermore, given that the outcomes using datasets transformed with Kernel Principal Component Analysis (KPCA) and Principal Component Analysis (PCA) were found to be highly similar, our report will exclusively consider those pertaining to reduction performed with KPCA. These results will be contrasted with the dataset prior to reduction, as discernible differences have been identified.

4.1 Decision Tree

We initiated the process by conducting cross-validation and hyperparameter tuning with the aim of identifying the optimal parameters for:

- max_depth;
- min_samples_leaf;
- min_samples_split.

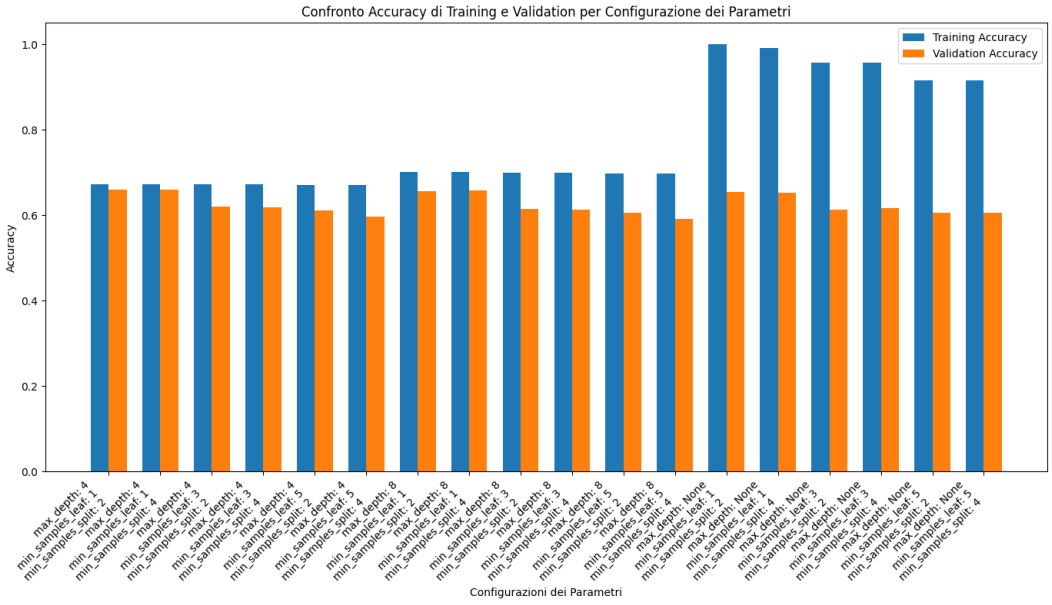


Figure 18: Hyper-parameter tuning and cross validation

Having identified the optimal hyper-parameters, we endeavor to discern, through the learning curve, the training set partition that yields the most favorable results in terms of validation accuracy.

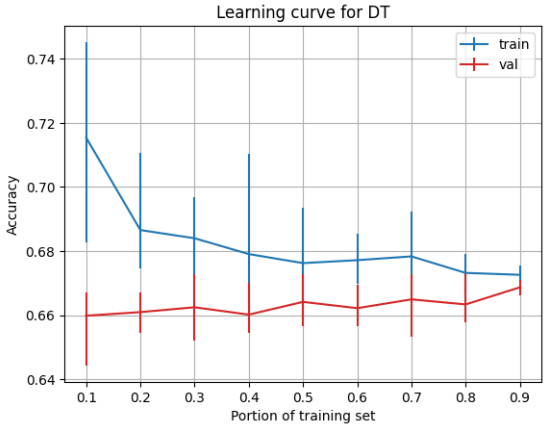
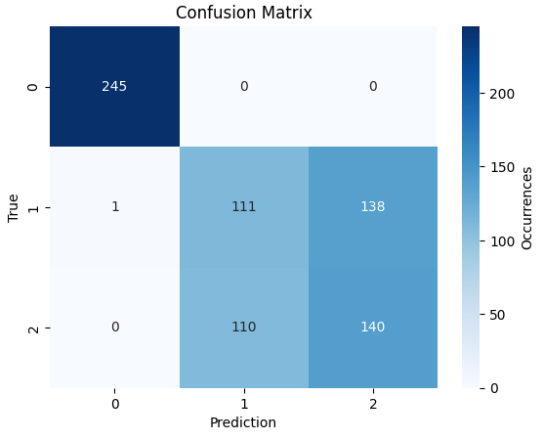


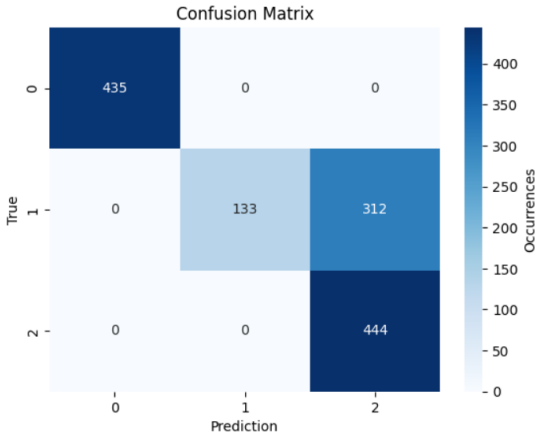
Figure 19: Validation curve

Below are the results obtained by training the model with the dataset before and after transformation with KPCA.



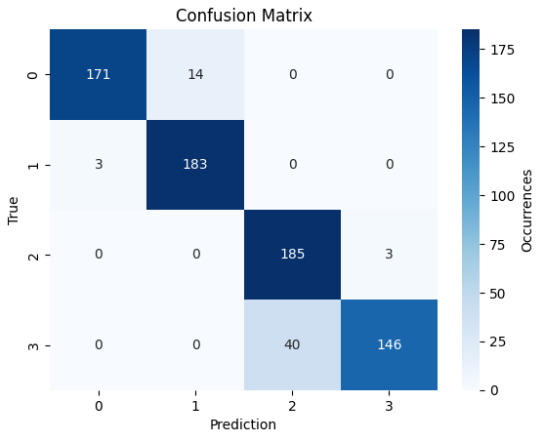
(a) KPCA

- browser
  - accuracy training:1.0
  - accuracy validation:0.66



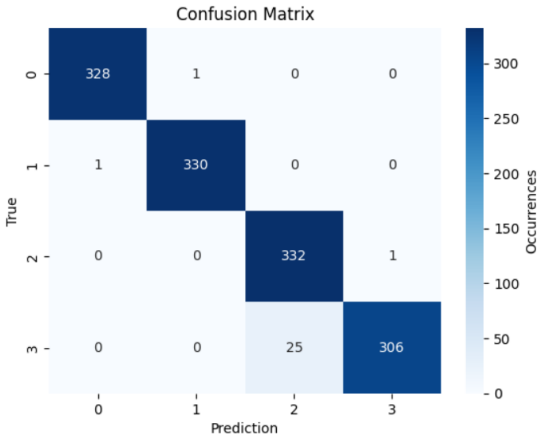
(b) Original

- browser
  - accuracy training: 0.76
  - accuracy validation: 0.76



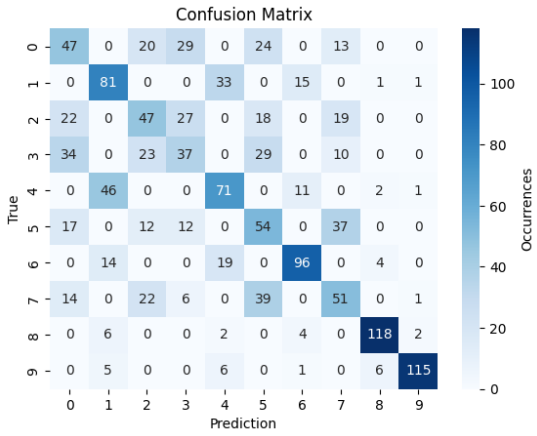
(a) KPCA

- website
  - accuracy training:0.94
  - accuracy validation:0.92



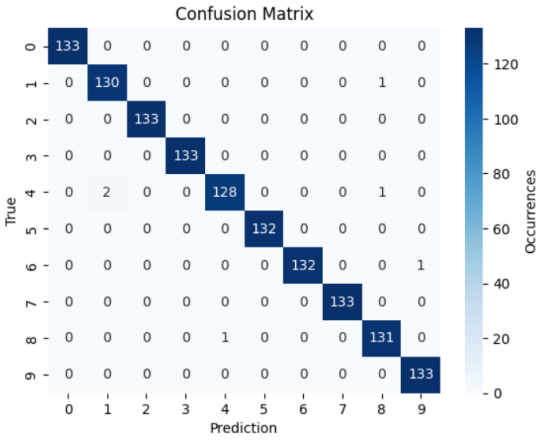
(b) Original

- website
  - accuracy training: 0.98
  - accuracy validation: 0.97



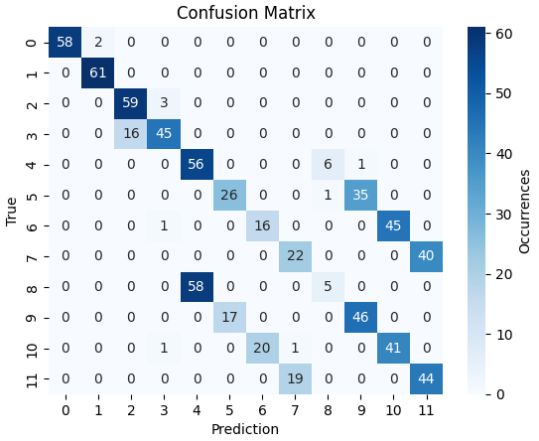
(a) KPCA

- behaviour
  - accuracy training:0.88
  - accuracy validation:0.54



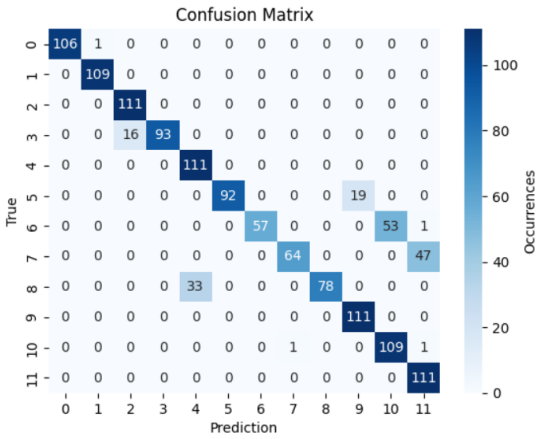
(b) Original

- behaviour
  - accuracy training: 1.0
  - accuracy validation: 0.99



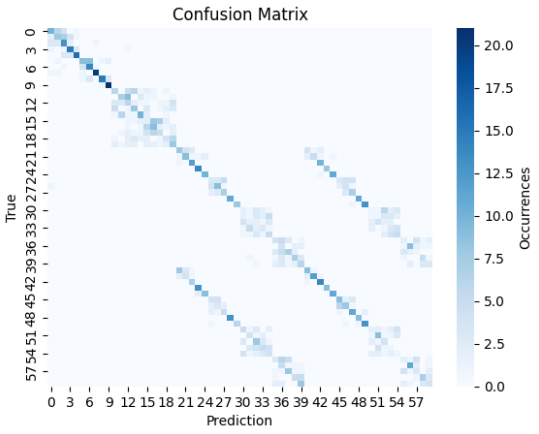
(a) KPCA

- browser+website
  - accuracy training:0.70
  - accuracy validation:0.64



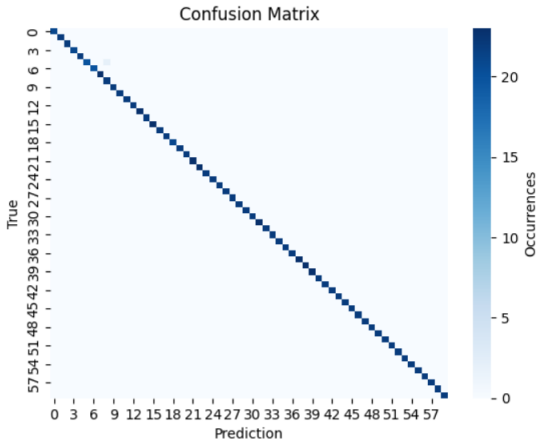
(b) Original

- browser+website
  - accuracy training: 0.88
  - accuracy validation: 0.87



(a) KPCA

- browser+website+behaviour
  - accuracy training:1.0
  - accuracy validation:0.37



(b) Original

- browser+website+behaviour
  - accuracy training: 0.99
  - accuracy validation: 0.99

Considerations: From the analysis of the confusion matrix graphs and their corresponding accuracy metrics, a clear distinction emerges in the model's capability to predict different types of labels when using principal component analysis (KPCA) compared to the dataset prior to dimensionality reduction. This could be a direct consequence of the 20% variance loss incurred post-reduction, or potentially due to other factors associated with the specific model utilized. Furthermore, it is also important to note that, as inferred from the confusion matrices, even when the accuracy is not high, the model does not make predictions randomly. Instead, it struggles to discern differences between specific classes. For clarity, let's take the prediction of the 'browser' category as an example. The Chrome browser is predicted with 100% precision, while the model encounters difficulties in distinguishing between Firefox and Edge, which may exhibit similar characteristics in terms of the specific TCP flows recorded. It should be taken into account that the only label predicted with an accuracy close to 100%, regardless of the dataset, is 'website.' This could be the primary distinguishing factor among the TCP flows that have been utilized as the foundation of the project. To further investigate and evaluate these assumptions with greater discernment, we proceeded with an analysis conducted using K-Nearest Neighbors (KNN), which operates differently compared to the Decision Tree approach.

4.2 KNN

We initiated the process by conducting cross-validation and hyperparameter tuning through the validation curve with the aim of identifying the optimal parameter for k.

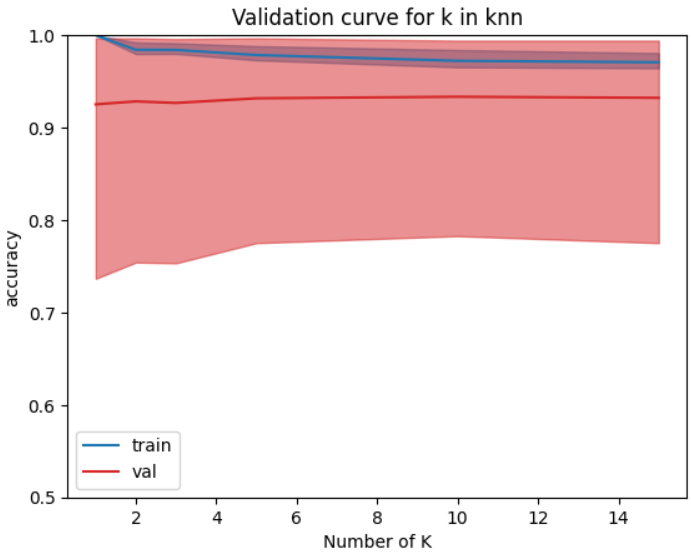
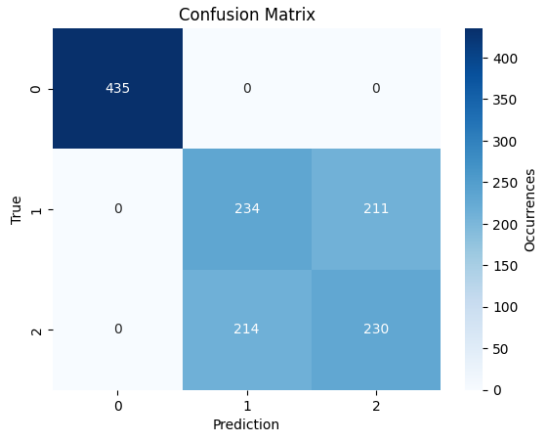


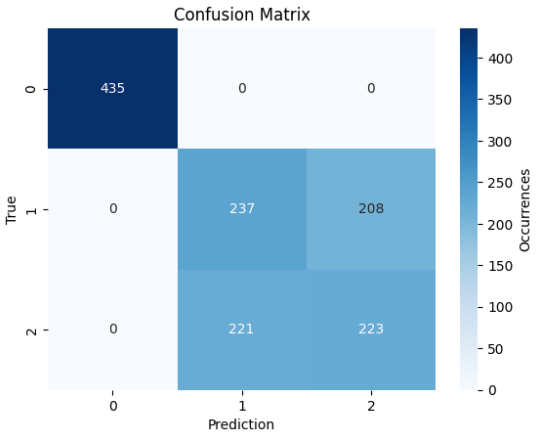
Figure 25: Validation curve

Once the optimal number of neighbors is determined for predicting the specific label combination, the model performs his prediction, and results for the particular configuration are obtained. Below are the confusion matrices for various label combinations:



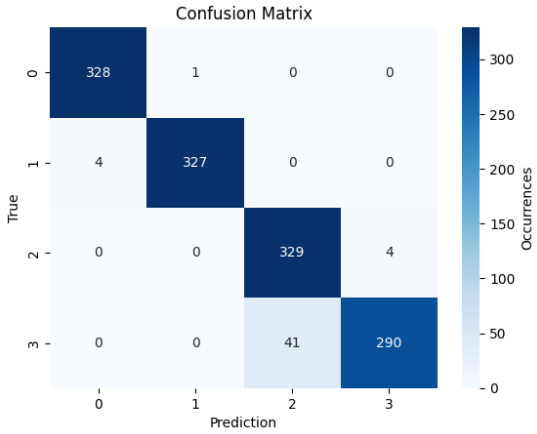
(a) KPCA

- browser
  - accuracy training:1.0
  - accuracy validation:0.67



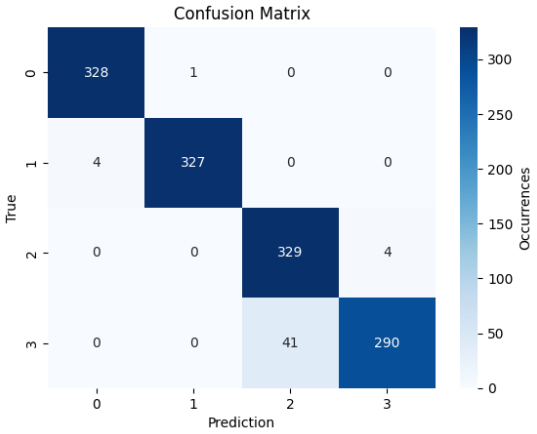
(b) Original

- browser
  - accuracy training: 0.80
  - accuracy validation: 0.67



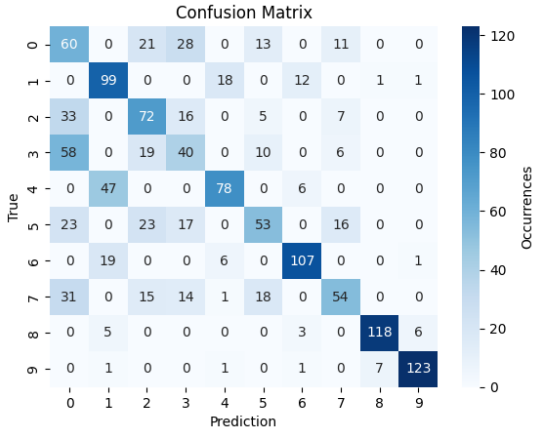
(a) KPCA

- website
  - accuracy training:0.97
  - accuracy validation:0.96



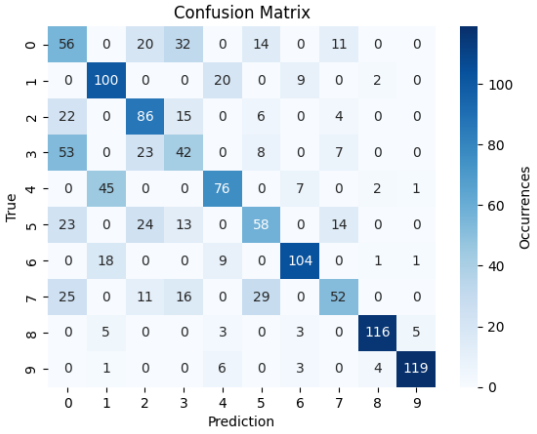
(b) Original

- website
  - accuracy training: 0.97
  - accuracy validation: 0.96



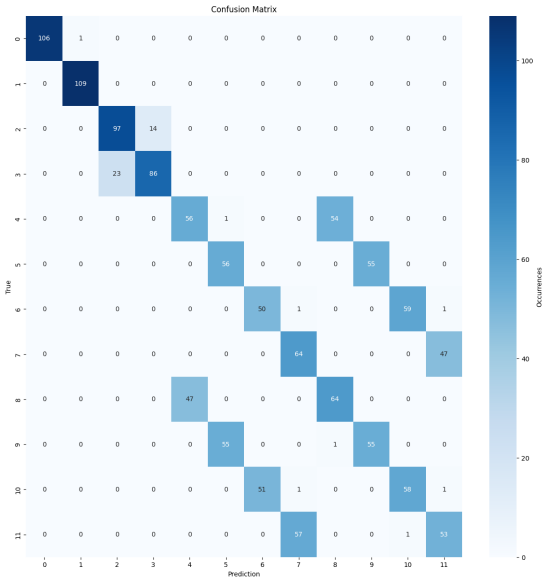
(a) KPCA

- behaviour
  - accuracy training:0.77
  - accuracy validation:0.60



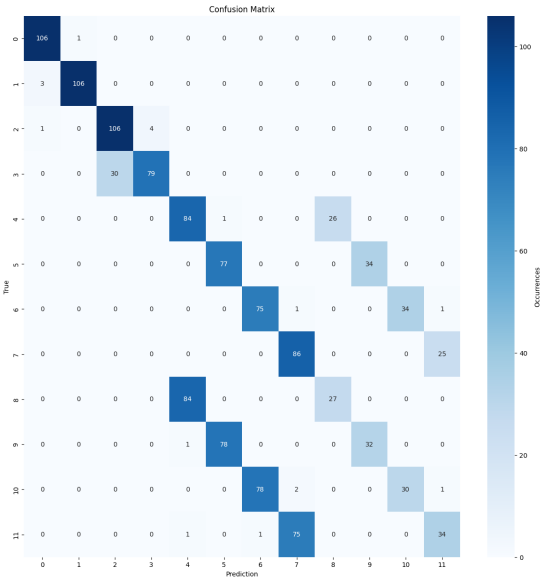
(b) Original:

- behaviour
  - accuracy training: 0.73
  - accuracy validation: 0.61



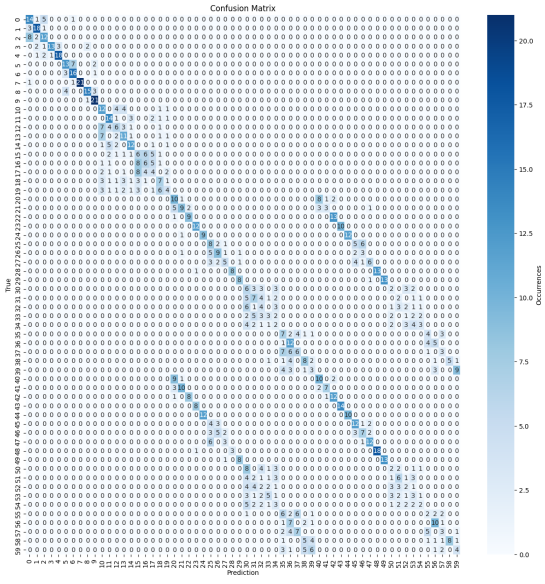
(a) KPCA

- browser+website
  - accuracy training:1.0
  - accuracy validation:0.64



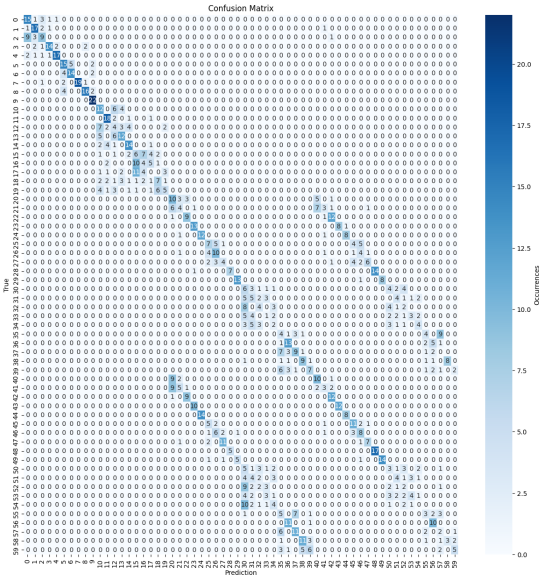
(b) Original

- browser+website
  - accuracy training: 0.82
  - accuracy validation: 0.63



(a) KPCA

- browser+website+behaviour
  - accuracy training:0.60
  - accuracy validation:0.41



(b) Original

- browser+website+behaviour
  - accuracy training: 0.60
  - accuracy validation: 0.40

Considerations: It is interesting to note that, unlike the Decision Tree model, the K-Nearest Neighbors (KNN) does not predict with the same level of accuracy for the labels related to the dataset without dimensional reduction. However, the predictions for the dataset processed with Kernel PCA (KPCA) remain substantially unchanged. This suggests that the KNN model may be less suited for predicting samples in our specific case of interest. Nevertheless, in this scenario as well, the 'website' label is consistently predicted with high accuracy. This supports our hypothesis that, compared to 'browser' and 'behavior', the 'website' is the most distinguishing ground truth of the recorded TCP flows. Additionally, it can be observed from the confusion matrices that, in this case too, the errors are not random. For instance, with the combination browser+website the error occur predominantly between only two classes. We believe this is related to how the data, being combinations of the same ground truths, can appear similar and are thus easily misclassified into a class that shares a ground truth.

4.3 Interpretation of Results

At the conclusion of our analysis, we observed distinct differences in the findings during the Unsupervised and Supervised learning phases. Initially, through clustering metrics, the predominant component appeared to be the browser, or anyway a combination of browser and website, in the allocation of points across various clusters. However, upon delving deeper in the Supervised learning phase, the only label that consistently demonstrated high accuracy, regardless of the dimensionality reduction techniques employed, was the website. This leads us to believe that the website is the label that most significantly differentiates the various TCP flows captured at the beginning of the project. From this observation, we can infer that among the various labels, the one least characteristic seems to be the one related to behavior. This conclusion aligns with our reasoning, as the information in TCP packet headers should be minimally influenced by changes in behavior, which instead have a more significant impact on the payload content. Furthermore, it is interesting to note that regardless of the Supervised learning model used, the errors in classification are not random but follow a specific pattern, as can be deduced from the visualization of the confusion matrices. We observed that these errors occur when making assignments between different classes that share some common ground truths. This insight could be pivotal in further refining our classification model and understanding the underlying dynamics of the data.