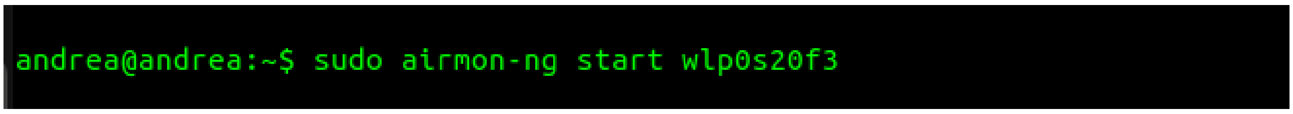
Wi-Fi Encrypted traffic classification

by Andrea Amore (10710643), Alberto Ceresoli (10680462)

The goal of the project is to sniff traffic from a Wi-Fi network, to process the captured traffic in order to extract some features and to use these features with an ML algorithm which would allow us to classificate the traffic in a supervised fashion. The project can be split in three main parts :
1) sniffing
2) processing the data
3) test the algorithm and provide some results

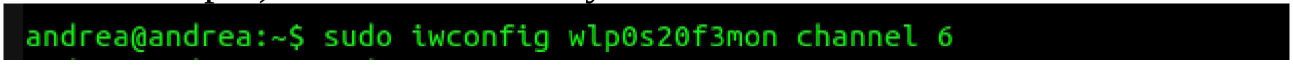
1- Sniffing the traffic

Let's start from the used devices: a Linux laptop able to perform the sniff in monitor mode, an Apple phone used to provide connectivity (in AP mode) and eventually an Apple laptop which will "create" the traffic.
Configuring the sniffer was actually easy due to the Linux OS which simplify the job : we used the airmon-ng command to set the interface in monitor mode



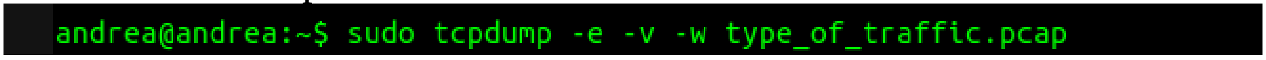
```
andrea@andrea:~$ sudo airmon-ng start wlp0s20f3
```

Then, we set the interface at the correct channel (the one used by the mobile hotspot) in order to actually see the traffic



```
andrea@andrea:~$ sudo iwconfig wlp0s20f3mon channel 6
```

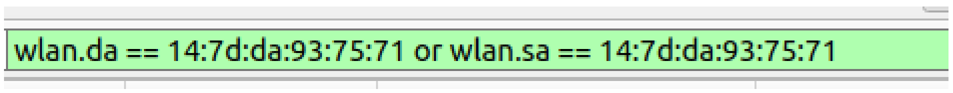
which corresponds to 2.437 GHz.
The sniffer is ready to perform the catches.
After attaching the Apple laptop to the hotspot connection we can create traffic with it and we use the command tcpdump on the sniffer to catch all the traffic in that specific channel:



```
andrea@andrea:~$ sudo tcpdump -e -v -w type_of_traffic.pcap
```

where :
-e is used to get information related to MAC addresses
-w to write the sniffed traffic in a .pcap file
-v for verbosity, helpful to see if we are catching something
Instead of type_of_traffic there will be the name of the used application.
We used in particular four application : *whatsapp web, wikipedia, youtube and zoom.*

To end the first phase we need two other things : filter the .pcap file in order to see only our traffic (the sniff was made in a flat apartment hence there were a lot of other traffic sources and AP) and convert the results into a .csv file to be able to to work with it.
The first one is performed using Wireshark, selecting only the traffic (and then exporting it in a new .pcap file) we get after applying a filter on the source (or destination) of the traffic:



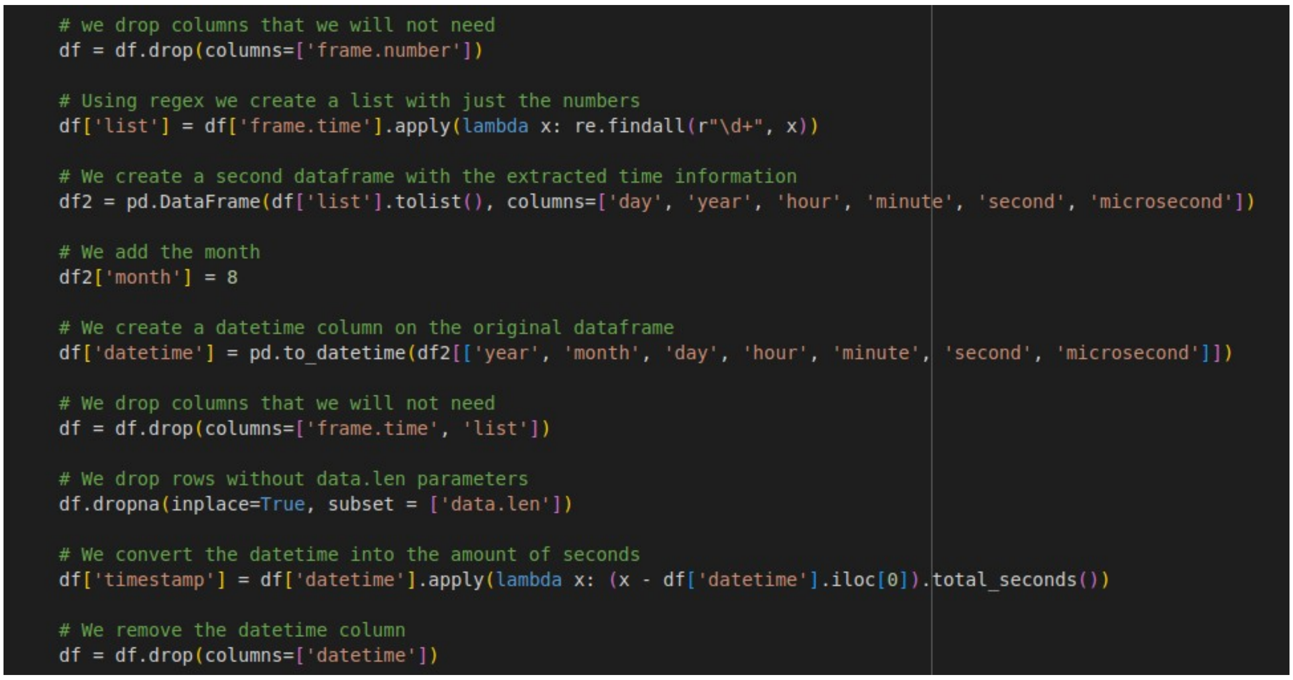
```
wlan.da == 14:7d:da:93:75:71 or wlan.sa == 14:7d:da:93:75:71
```

where **14:7d:da:93:75:71** is our Apple laptop MAC address.
The second one is done by running the following command on the terminal:
tshark -r <application>.pcap -2 -T fields -e frame.number -e frame.time -e wlan.sa -e wlan.da -e data.len -E header=y -E separator=, -E quote=d -E occurrence=f > <application>.csv
where :
-e is used to select only those fields we are interested in

All the filtered .pcap and .csv files can be found at the following link: https://drive.google.com/drive/folders/1ssyLrOENFVpBwwHYK0ktiDvbQMR7cEaR in the folders “**Catture**” and “**CSV**”.

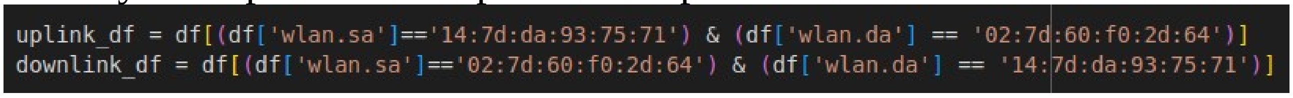
2- Data Cleaning and Processing

The second phase is an intermediate step toward the final and most interesting part but it is nonetheless very important. We used Google Colab to write and run the code since it is online and simple to use. After importing the .csv files with our captures and the necessary libraries, we need to work the files in order for them to suit our goal : a group of datasets, each one described by a given time step (in seconds), where all traffic are collected together and represented by their statistical features.
The first thing we need to do is to modify the **Time** field (necessary for calculating the inter-arrival time): in the .csv file is represented as “*Aug 10, 2024 12:57:44.935991000 CEST*”. This was done using the **re** library (extracting all the numbers from the date). We also get rid of unused column from the file such as the “frame.number”. The following is the code used for this “Cleaning” part and is performed for each .csv file:



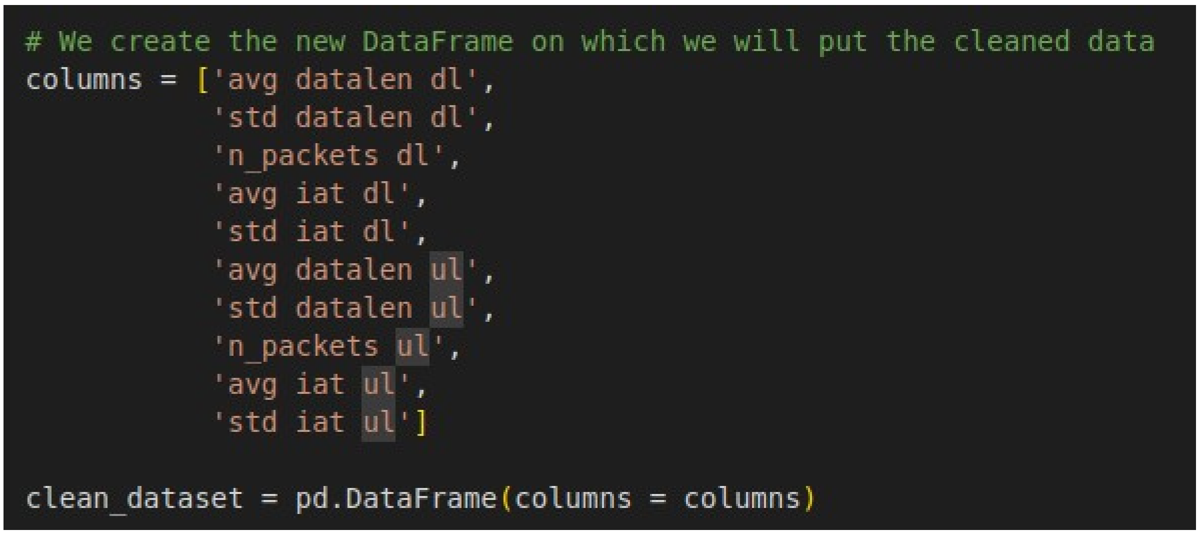
```
    # we drop columns that we will not need
    df = df.drop(columns=['frame.number'])

    # Using regex we create a list with just the numbers
    df['list'] = df['frame.time'].apply(lambda x: re.findall(r"\d+", x))

    # We create a second dataframe with the extracted time information
    df2 = pd.DataFrame(df['list'].tolist(), columns=['day', 'year', 'hour', 'minute', 'second', 'microsecond'])

    # We add the month
    df2['month'] = 8

    # We create a datetime column on the original dataframe
    df['datetime'] = pd.to_datetime(df2[['year', 'month', 'day', 'hour', 'minute', 'second', 'microsecond']])

    # We drop columns that we will not need
    df = df.drop(columns=['frame.time', 'list'])

    # We drop rows without data.len parameters
    df.dropna(inplace=True, subset = ['data.len'])

    # We convert the datetime into the amount of seconds
    df['timestamp'] = df['datetime'].apply(lambda x: (x - df['datetime'].iloc[0]).total_seconds())

    # We remove the datetime column
    df = df.drop(columns=['datetime'])
```

Eventually we can create the datasets.
Initially we separate all the packets in uplink and downlink ones:



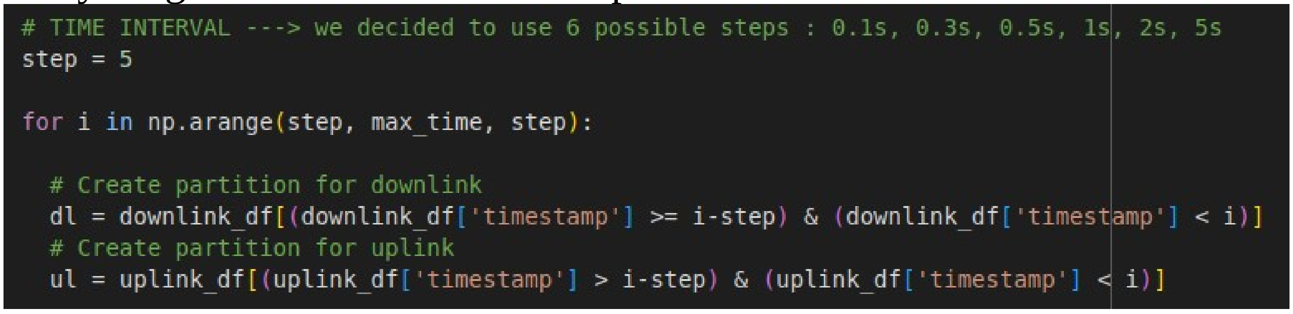
```
uplink_df = df[(df['wlan.sa']=='14:7d:da:93:75:71') & (df['wlan.da'] == '02:7d:60:f0:2d:64')]
downlink_df = df[(df['wlan.sa']=='02:7d:60:f0:2d:64') & (df['wlan.da'] == '14:7d:da:93:75:71')]
```

and we initialize a new dataframe where columns are the statistical features we need:



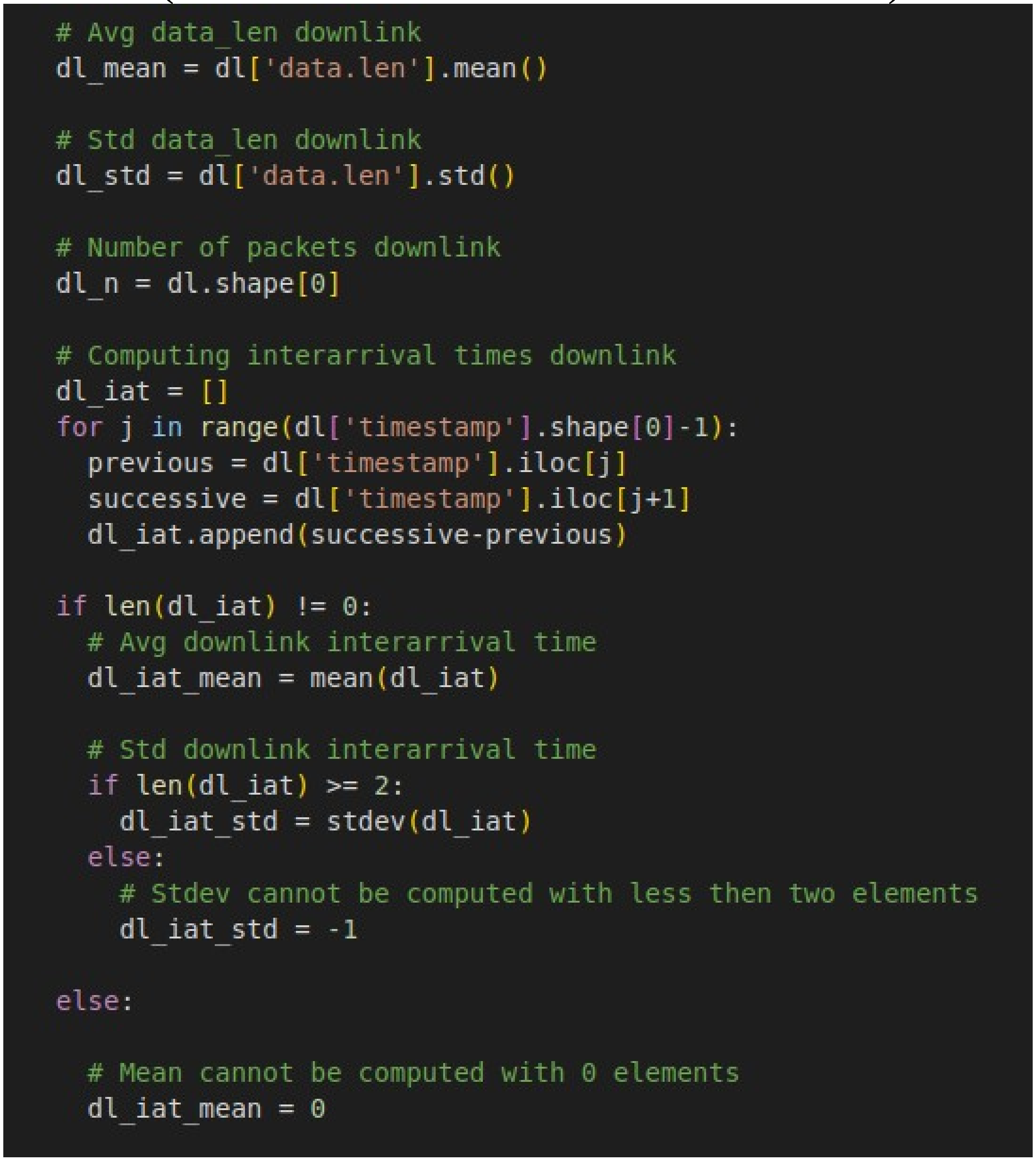
```
# We create the new DataFrame on which we will put the cleaned data
columns = ['avg datalen dl',
           'std datalen dl',
           'n_packets dl',
           'avg iat dl',
           'std iat dl',
           'avg datalen ul',
           'std datalen ul',
           'n_packets ul',
           'avg iat ul',
           'std iat ul']

clean_dataset = pd.DataFrame(columns = columns)
```

Then we have to fill the dataset: we define the step and consider everything which is “inside” the step for calculation



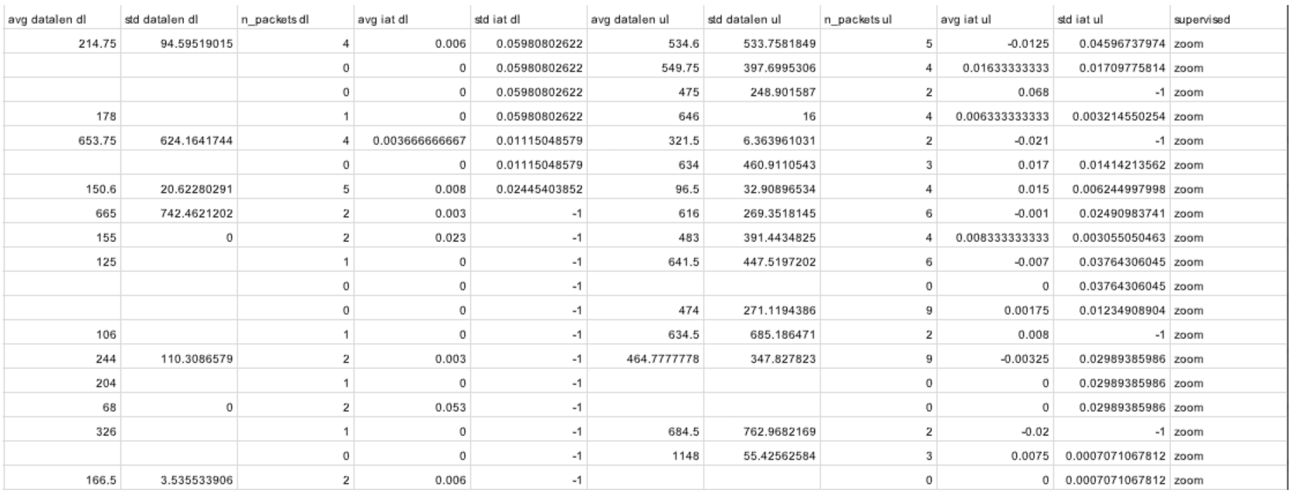
```
# TIME INTERVAL ---> we decided to use 6 possible steps : 0.1s, 0.3s, 0.5s, 1s, 2s, 5s
step = 5

for i in np.arange(step, max_time, step):

  # Create partition for downlink
  dl = downlink_df[(downlink_df['timestamp'] >= i-step) & (downlink_df['timestamp'] < i)]
  # Create partition for uplink
  ul = uplink_df[(uplink_df['timestamp'] > i-step) & (uplink_df['timestamp'] < i)]
```

Then we perform the calculation to extrapolate *Average* and *Standard Deviation* of the IAT (inter-arrival time), both for the uplink and the downlink (commands are identical with ul instead of dl):



```
# Avg data_len downlink
dl_mean = dl['data.len'].mean()

# Std data_len downlink
dl_std = dl['data.len'].std()

# Number of packets downlink
dl_n = dl.shape[0]

# Computing interarrival times downlink
dl_iat = []
for j in range(dl['timestamp'].shape[0]-1):
  previous = dl['timestamp'].iloc[j]
  successive = dl['timestamp'].iloc[j+1]
  dl_iat.append(successive-previous)

if len(dl_iat) != 0:
  # Avg downlink interarrival time
  dl_iat_mean = mean(dl_iat)

  # Std downlink interarrival time
  if len(dl_iat) >= 2:
    dl_iat_std = stdev(dl_iat)
  else:
    # Stdev cannot be computed with less then two elements
    dl_iat_std = -1

else:

  # Mean cannot be computed with 0 elements
  dl_iat_mean = 0
```

Lastly, we re-add the label to identify the application/source of traffic and we create the final dataset. These passages are repeated for each one of the six time-steps.
The result should look like the following sample taken from the dataset_0.1: as anyone can notice there are some empty space and strange values (such as -1 in std iat dl). This is due to the probability ditribution of the traffic which is not uniform (of course) and there are moments where a lot of traffic can be seen as moments where it looks like there is no active source, even though the application is running. This will be discussed lately in the ML part where these features are treated and processed to provide some interesting results.

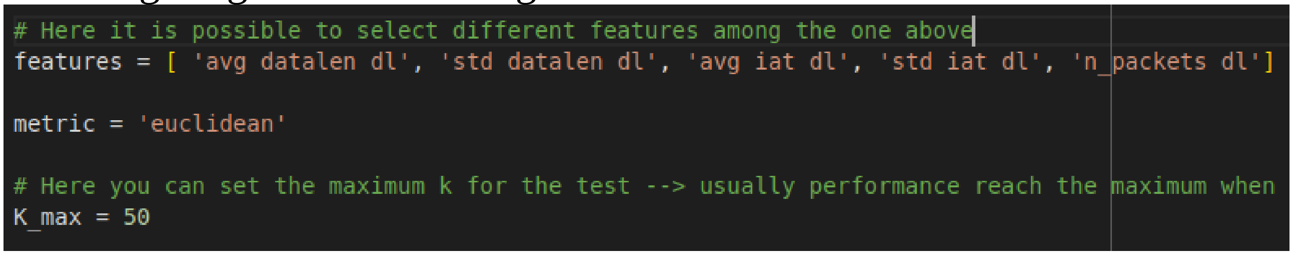


avg datalen dl	std datalen dl	n_packets dl	avg iat dl	std iat dl	avg datalen ul	std datalen ul	n_packets ul	avg iat ul	std iat ul	supervised
214.75	94.59519015	4	0.006	0.05980802622	534.6	533.7581849	5	-0.0125	0.04596737974	zoom
		0	0	0.05980802622	549.75	397.6995306	4	0.01633333333	0.01709775814	zoom
		0	0	0.05980802622	475	248.901587	2	0.068	-1	zoom
178		1	0	0.05980802622	646	16	4	0.006333333333	0.003214550254	zoom
653.75	624.1641744	4	0.003666666667	0.01115048579	321.5	6.363961031	2	-0.021	-1	zoom
		0	0	0.01115048579	634	460.9110543	3	0.017	0.01414213562	zoom
150.6	20.62280291	5	0.008	0.02445403852	96.5	32.90896534	4	0.015	0.006244997998	zoom
665	742.4621202	2	0.003	-1	616	269.3518145	6	-0.001	0.02490983741	zoom
155	0	2	0.023	-1	483	391.4434825	4	0.008333333333	0.003055050463	zoom
125		1	0	-1	641.5	447.5197202	6	-0.007	0.03764306045	zoom
		0	0	-1			0	0	0.03764306045	zoom
		0	0	-1	474	271.1194386	9	0.00175	0.01234908904	zoom
106		1	0	-1	634.5	685.186471	2	0.008	-1	zoom
244	110.3086579	2	0.003	-1	464.7777778	347.827823	9	-0.00325	0.02989385986	zoom
204		1	0	-1			0	0	0.02989385986	zoom
68	0	2	0.053	-1			0	0	0.02989385986	zoom
326		1	0	-1	684.5	762.9682169	2	-0.02	-1	zoom
		0	0	-1	1148	55.42562584	3	0.0075	0.0007071067812	zoom
166.5	3.535533906	2	0.006	-1			0	0	0.0007071067812	zoom

At the link : https://drive.google.com/drive/folders/1ssyLrOENFVpBwwHYK0ktiDvbQMR7cEaR all the datasets can be found in the “**Dataset**” folder and the complete code with all the comments in the “**Python Code**” folder under the name ‘***DataCleaning’***

3- Training, Testing

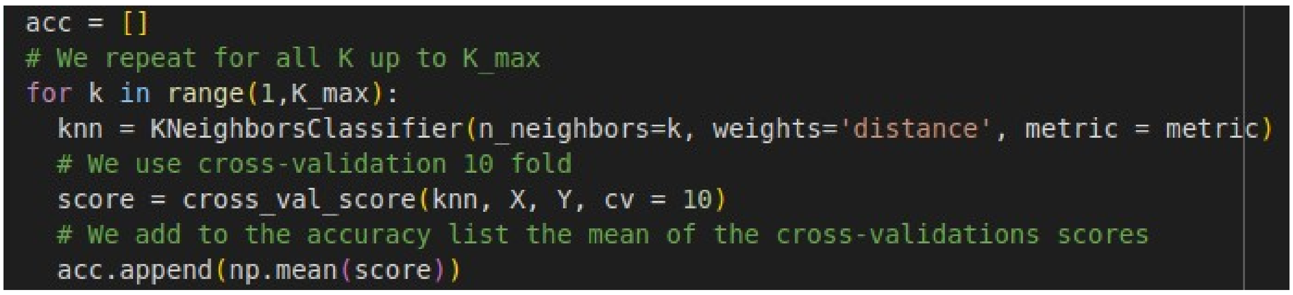
This third and last phase is made up of two main parts: the testing part and the result one. In the first part we essentialy tested an algorithm which goal was to identify the source of traffic given the statistical features. But let us look it in more details.
The algorithm we decided to use is KNN with cross-validation 10-fold.
First, we defined: an array where we can select the parameters that will be used by the algorithm, the metric and a K max. The metric selected was ‘euclidean’ after some research on the topic and some testing, since was the one giving more interesting results:



```
# Here it is possible to select different features among the one above
features = [ 'avg datalen dl', 'std datalen dl', 'avg iat dl', 'std iat dl', 'n_packets dl']

metric = 'euclidean'

# Here you can set the maximum k for the test --> usually performance reach the maximum when
K_max = 50
```

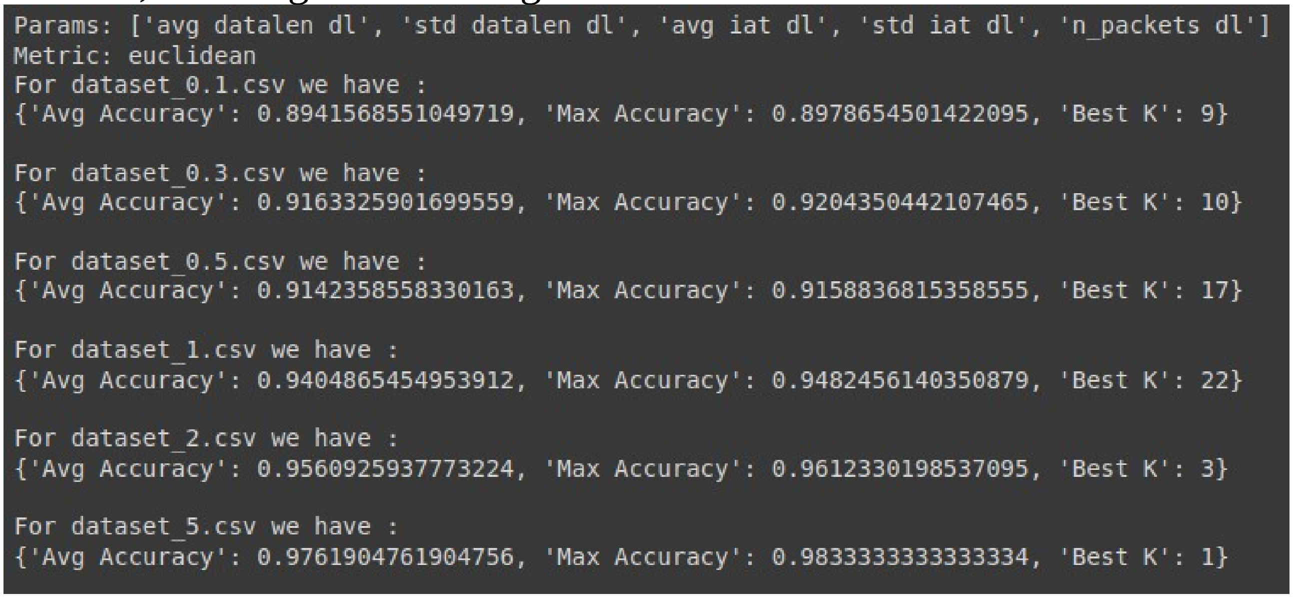
Then there is the actual testing part which can be briefely described as such:
-get rid of unusable parameters (i.e. std = -1)
-separate the dataset in two: one with values and one with only the names of the application, which will be used as the desired output value. The

valued one is filtered with only the selected parameters and then randomly scrambled
-run the algorithm for K ranging from 1 to K Max: the steps are
	-use the KneighborsClassifier with the two dataset, the metric and the weight (set as ‘distance’)
	-compute the score using cross-validation 10-fold (a tenth of the dataset for testing and 9/10 for training): this result in an array of score of which we will the perform the average to find the accuracy (on avg) of the classifier



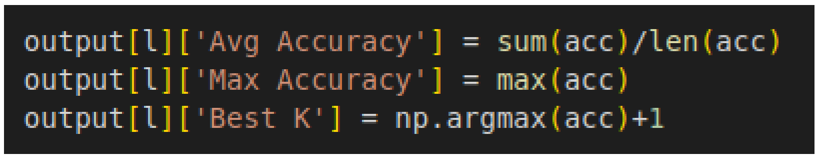
```
acc = []
# We repeat for all K up to K_max
for k in range(1,K_max):
  knn = KNeighborsClassifier(n_neighbors=k, weights='distance', metric = metric)
  # We use cross-validation 10 fold
  score = cross_val_score(knn, X, Y, cv = 10)
  # We add to the accuracy list the mean of the cross-validations scores
  acc.append(np.mean(score))
```

This process is done for each k from 1 to K Max and for each of the 6 datasets, resulting in something like this:



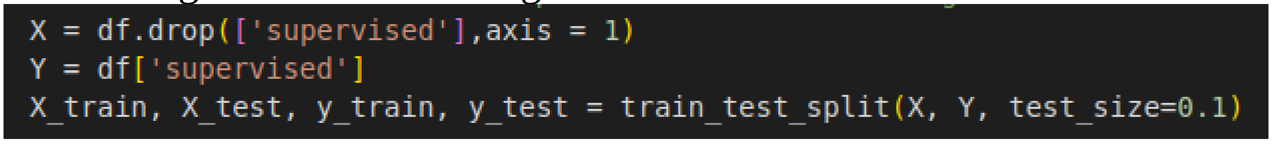
```
Params: ['avg datalen dl', 'std datalen dl', 'avg iat dl', 'std iat dl', 'n_packets dl']
Metric: euclidean
For dataset_0.1.csv we have :
{'Avg Accuracy': 0.8941568551049719, 'Max Accuracy': 0.8978654501422095, 'Best K': 9}

For dataset_0.3.csv we have :
{'Avg Accuracy': 0.9163325901699559, 'Max Accuracy': 0.9204350442107465, 'Best K': 10}

For dataset_0.5.csv we have :
{'Avg Accuracy': 0.9142358558330163, 'Max Accuracy': 0.9158836815358555, 'Best K': 17}

For dataset_1.csv we have :
{'Avg Accuracy': 0.9404865454953912, 'Max Accuracy': 0.9482456140350879, 'Best K': 22}

For dataset_2.csv we have :
{'Avg Accuracy': 0.9560925937773224, 'Max Accuracy': 0.9612330198537095, 'Best K': 3}

For dataset_5.csv we have :
{'Avg Accuracy': 0.9761904761904756, 'Max Accuracy': 0.9833333333333334, 'Best K': 1}
```

where the results we are interested in are calculated in this way:



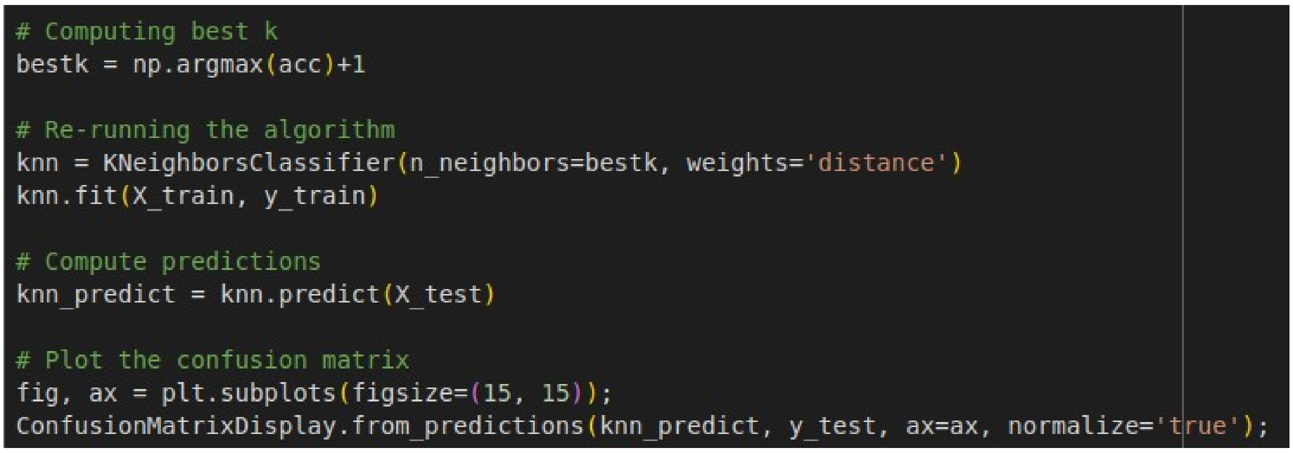
```
output[l]['Avg Accuracy'] = sum(acc)/len(acc)
output[l]['Max Accuracy'] = max(acc)
output[l]['Best K'] = np.argmax(acc)+1
```

The result section is structured in the same way of the one we just discussed with few exceptions (the code is ran for only one dataset, some images are produced describing graphically the results) and an important difference: now we want to focus on the best performing K hence we split the analyzed dataset in two (valued and supervised) and then each of them in a training one and in a testing one.



```
X = df.drop(['supervised'],axis = 1)
Y = df['supervised']
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.1)
```

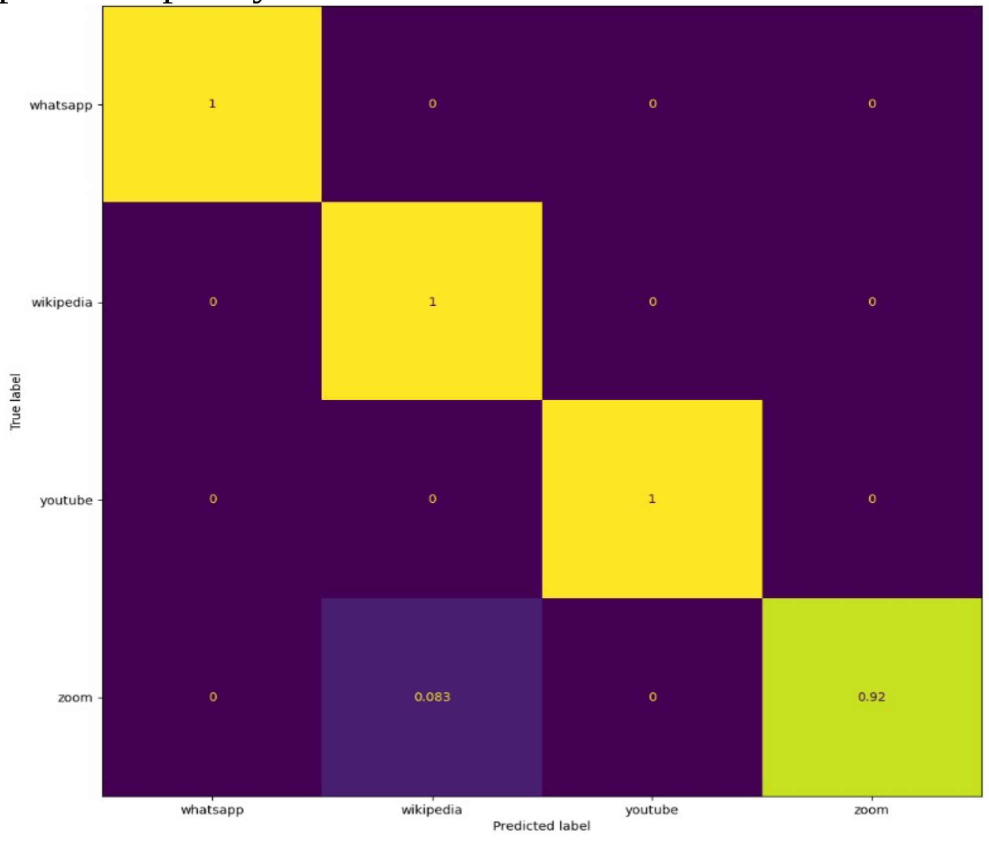
This final separation is used to plot the confusion matrix and to see how well the algorithm is performing for each of the traffic’ sources:



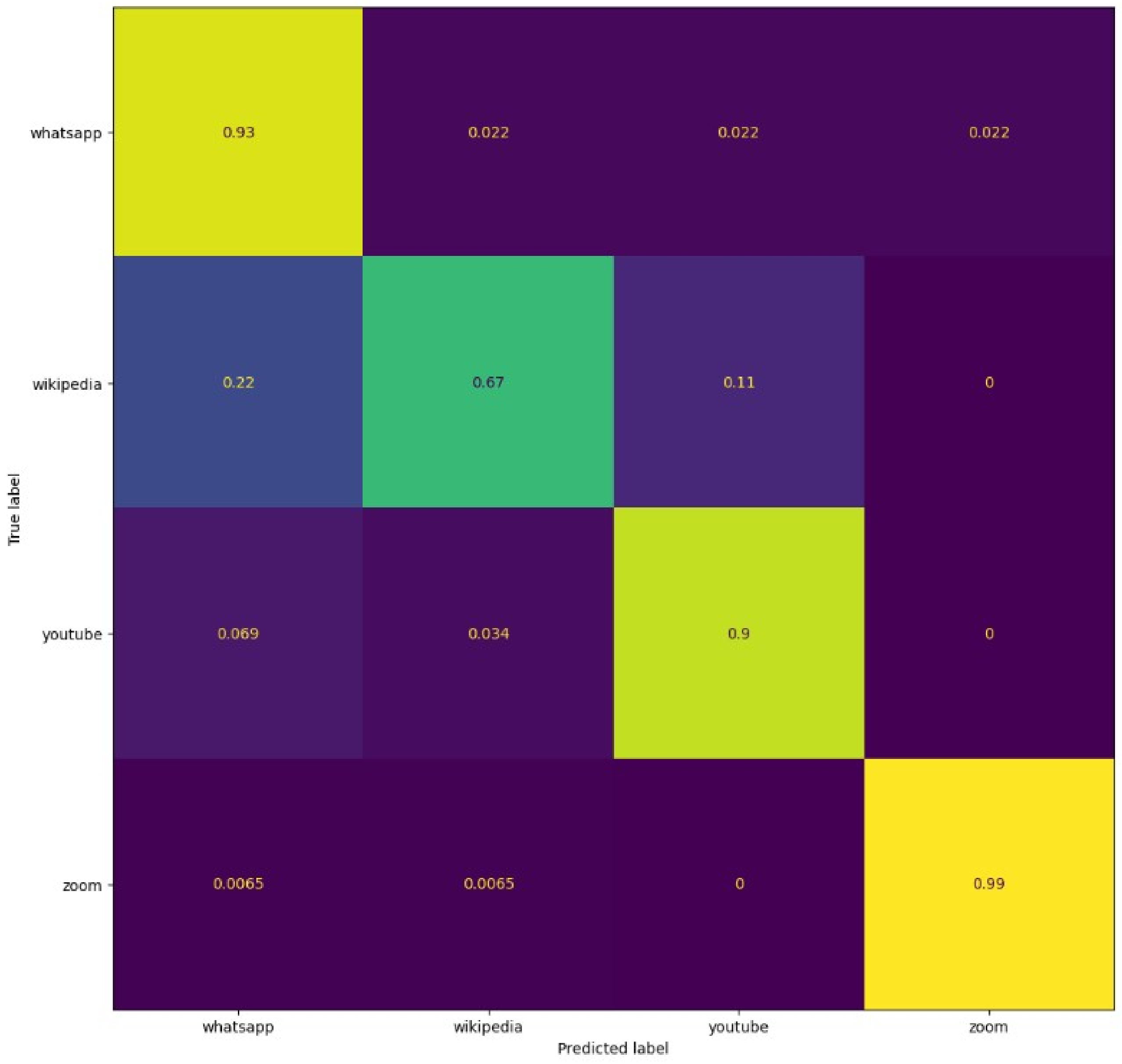
```
# Computing best k
bestk = np.argmax(acc)+1

# Re-running the algorithm
knn = KNeighborsClassifier(n_neighbors=bestk, weights='distance')
knn.fit(X_train, y_train)

# Compute predictions
knn_predict = knn.predict(X_test)

# Plot the confusion matrix
fig, ax = plt.subplots(figsize=(15, 15));
ConfusionMatrixDisplay.from_predictions(knn_predict, y_test, ax=ax, normalize='true');
```

4- <u>Results and Comments</u>

In order to have a more complete view of the results we tested different combination of parameters and different values of K. Some notes about this can also be found on the comments in the code. As we tested we found out some combinations that give very good results and other which performs poorly.

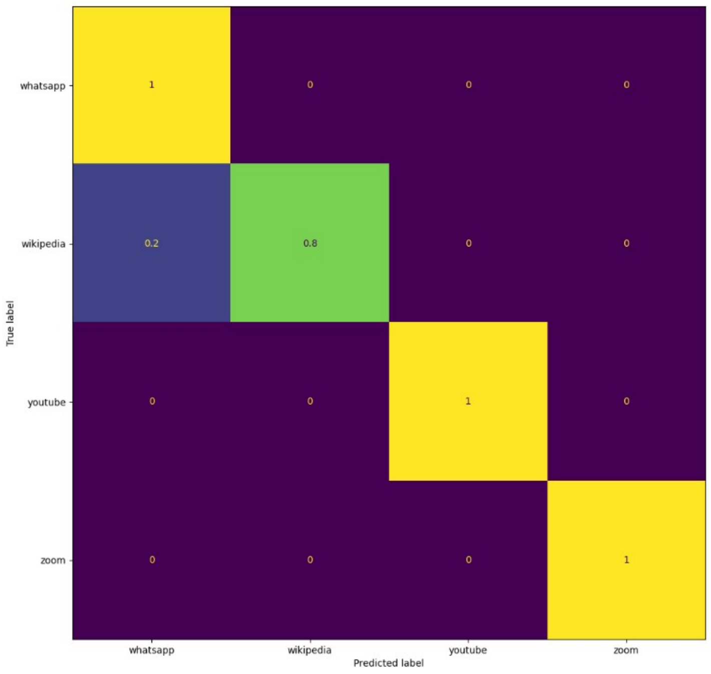


Dataset: 5s steps
Params: All

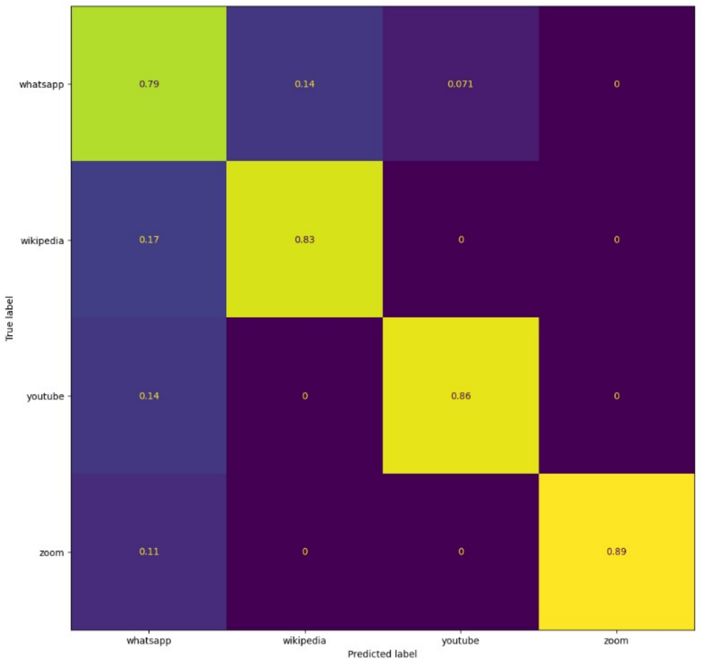


Dataset: 0.1s
Params: All

The best set of parameters are the one considering all possible features and the considering only downlink one: this is probably due to the fact that the application used tend to create very little traffic in uplink while the downlink one is way bigger. Also features relative to datalen are way better in terms of prediction with respect to iat relative features, which performed way worst: this is probably related to the fact that traffic distribution affects more the inter-arrival time, which tend to look completely randomic wrt the source (lowly correlated we can say) while the relation between datalen and application is way stronger.

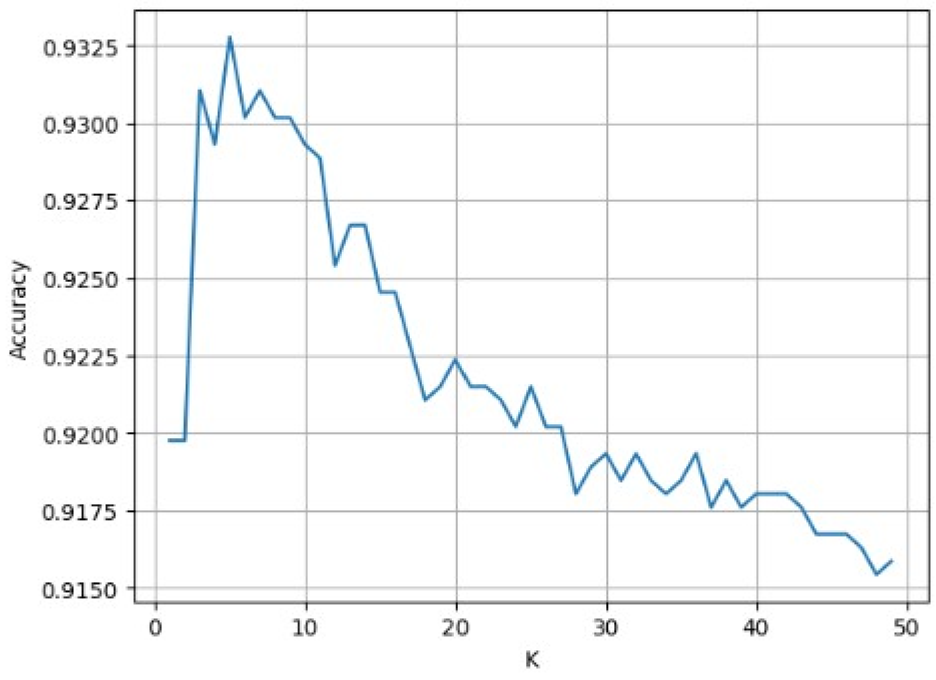


Dataset: 5s steps
Params: only downlink



Dataset: 5s steps
Params: only uplink

Last two considerations before showing the actual results: the first one is relative to the value of the best K, which usually stays between 1 and 25, and the best performing dataset. This last one is probably the weirdest of the results: while one should expect that when reducing the time step the result should get better since we have more fine-grained datas and a greater number of them, the best accuracy is obtained when using the 5 seconds step dataset. The result is not as expected but not completely senseless: when reducing a lot the time step we may expect to find a lot of empty spaces or useless values in the dataset, since it is possible that "nothing happened" in that little span of time (or at least it seems so from the sniffer perspective). Instead, the dataset_5 is full and with only insightful values. The confirmation of this hypotesis is given from the fact that Accuracy gets better the greater the step gets (this is not always true but in most of the tried cases it is). Zoom is the "best recognized" application, probably due to the high amount of traffic and its characteristics, such as the usually high data length. Instead, wikipedia and whatsapp web are often mistaken one from the other: as for Zoom, this is probably due to the similarity of the two traffics in terms of the features used.
On average the Accuracy reach pretty high values (with peaks of 98%) within the first tried Ks, usually between 1 and 20 but more tilted towards lower values as can be seen in this example accuracy plot:



Dataset: 0.5s steps
Params: All
Best K: 5

These examples are useful to give us an idea of how the algorithm performs, but we must notice that at each iteration the algorithm will give slightly different results.
To conclude all the code can be found at the usual link, in the folder "**Python Code**" under the name "**Training_and_Testing**".