# American Sign Language alphabet recognition using the Tap Strap hand-gestures device

Guillem Cornella I Barba, Eudald Sangenís Rafart, Miguel Antonio Salazar del Río

**Abstract**—The aim of this project is to present the Tap Strap hand-gestures device as an American Sign Language alphabet interpreter, by applying Machine Learning and Pattern Recognition techniques. A dimensionality reduction of the data set has been performed by PCA. Then, four classifiers have been trained using different techniques: LDA, QDA, SVM and Random Forest. Finally, the performance analysis of each classifier has been carried out, considering precision indices and time performances of the process of training and classification. The results obtained lead to the conclusion that for the scope of this project, SVM tends to be a more complete classifier. In addition to this, a Python script has been designed to test the alphabet recognition in real time in Ubuntu, with very good results.

**Index Terms**—Tap Strap, Machine Learning, American Sign Language, SVM

## 1 Introduction

SINCE the invention of the typewriter by Christopher Sholes in 1867, the QWERTY keyboard layout has been considered a constant feature in every computer available in the market. Although it was initially designed for English writing, multiple variations of this standard have been implemented in order to make the keyboards suitable for its use in other languages.

The TAP company developed a new alternative to the classical keyboard, creating the TAP Strap. Defined by the company as an All-In-One, Wearable Keyboard, Mouse and Air Gesture Controller, the TAP Strap is formed by a set of sensors that measure accelerations and displacements of each one of the fingers of the hand. However, the main application of this device is to control any virtual environment by using a mouse and a keyboard, which can be applied to a wide range of applications, such as virtual reality or air gesture control. Furthermore, the SDK is provided by the company, which makes TAP Strap fully customizable [2].

The aim of this project is to classify and recognise all the different symbols of the sign language alphabet using Machine Learning and Pattern Recognition Techniques [1]. For that purpose, a data set has been created containing all the sensor's information provided by the Tap Strap finger sensors. The previously mentioned SDK has been used to extract the raw data from these sensors and to transform it into inputs for different Machine Learning algorithms.

## 2 Development

### 2.1 Data-acquisition and data-handling

TAP Python SDK allows to build Python apps that can establish BLE connections with the Tap Strap, send commands and receive events and data. This SDK implements two basic interfaces with the Tap Strap:

- Setting the operation mode of the Tap strap as:
  - *Raw data mode* - the strap will stream raw sensors data to the SDK.
- Subscribing to the event:

- *Raw data* - whenever new raw data sample is being created.

In raw sensors mode, the Tap continuously sends raw data from the following sensors:

1) Five 3-axis accelerometers (one per each finger).
2) IMU (3-axis accelerometer + gyro) located on the thumb

The sensor's measurements are given with respect to the reference system, shown in fig. 1.

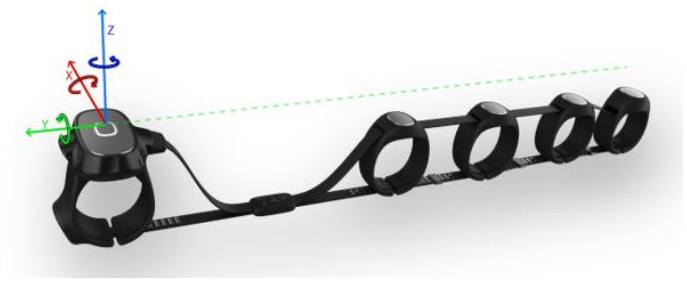


Fig. 1: TAP Strap reference system.

The first platform tested was Windows, yet Linux was chosen since it allowed access to each sample's timestamp. Every sample of raw data is sent with a frequency of 200 Hz (5 ms) thanks to an internal Tap clock. The structure of the raw data that the Linux platform receives is the following:

- timestamp; type (IMU, device); values

By accessing the timestamp of the data, we reduced the reading rate to 20Hz (50 ms), in this manner the classifier has enough time to predict the received data.

A first analysis on the raw data proved that the 6 IMU values were not needed for this kind of classification (static). Therefore, the observations went down from 21 features to 15 features (xyz coordinates of each finger w.r.t. the thumb). To generate the data set, a code was written to gather all the raw data in a .txt file. The data-acquisition procedure consisted in putting on the TAP device on our right hand and representing the alphabet letters so that every observation could be assigned to its class/label/letter.

The alphabet data set contains 4910 observations. The first five rows/observations are shown in table 1. We can see how all the letters are considered as labels in the last column.

## 2.2 Data standardization

This section aims to standardize features by removing the mean and scaling to unit variance using sklearn.preprocessing.StandardScaler.

We performed a standardization of our data set because it is a common requirement for many machine learning estimators: they might behave badly if the individual features do not look like standard normally distributed data. Table 2 shows the first five rows of the standardized data set.

## 2.3 PCA - Dimensionality Reduction

PCA was used to reduce the dimensions of the dataset by projecting each point onto the first few principal components to obtain lower-dimensional data while preserving as much of the data variability as possible. The plot represented in figure 2 was used for choosing the number of principal components.

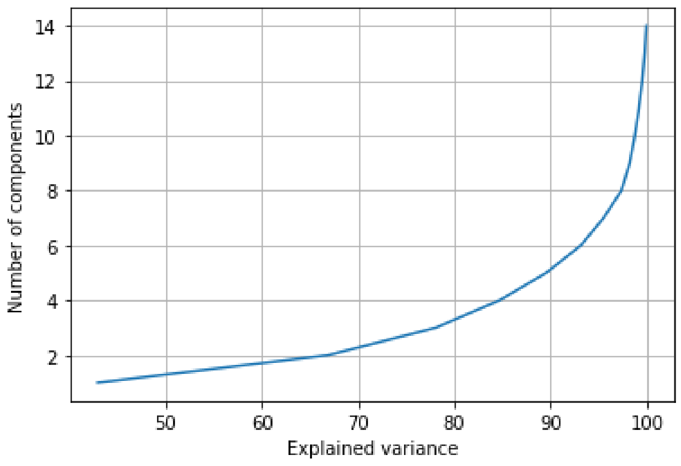


Fig. 2: Explained variance vs # of PCA.

The previous plot gives the following information:

- The 4th PCA component explains 86.69% of the variability of the data.
- The 5th PCA component explains 91.03% of the variability of the data.
- The 6th PCA component explains 94.86% of the variability of the data.

Hence, all the classifiers have been trained with all these 3 options to observe which reduction performs better. All the explanations on this document will be done considering 5 components:

We reduced our data set features dimensions from 21 to 15 (eliminating the IMU values), and afterwards from 15 to 5 (performing a PCA). With 5 dimensions we can compress more than 90% of the variance of our data.

Then we obtained the projected data, which can be seen on the following table. It is important to mention that the data set was shuffled to avoid any elements of bias/patterns in the split data sets before training the ML model.

| | 0 | 1 | 2 | 3 | 4 | symbols |
|---|---|---|---|---|---|---|
| 4230 | 2.24 | 1.44 | -1.26 | -1.23 | 1.28 | X |
| 2925 | 5.04 | 2.58 | 2.2 | -0.68 | -0.75 | Q |
| 4894 | -2.11 | -1.04 | -0.38 | 0.28 | -1.18 | * |
| 19 | -4.24 | 3.26 | 1.23 | 0.57 | -1.29 | A |

TABLE 3: Suffled dataset

## 2.4 Classifiers

The first step before classification is to divide the data in four different subsets: {data_train, label_train, data_test, label_test}, where the training data contains the 70% of all the data, and the testing data comprises the 30% of the remaining data.

Then, four classifiers were trained (LDA, QDA, SVM and RandForest). The best classifier was SVM as it will be seen in the next section. The Python library used was sklearn.svm.SVC, where the multi-class support is handled according to a one-vs-one scheme. A key to this classifier's success is that for the fit, only the position of the support vectors matter. Any points further from the margin which are on the correct side do not modify the fit. Technically, this is because these points do not contribute to the loss function used to fit the model, so their position and number do not matter so long as they do not cross the margin.

The advantages of SVM are:

* Effective in high dimensional spaces.

* Still effective in cases where number of dimensions is higher than the number of samples.

* Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient. Table 4 shows the localization of some support vectors, from a total of 1.183

| | acc1 | acc2 | acc3 | acc4 | acc5 | acc6 | acc7 | acc8 | acc9 | acc10 | acc11 | acc12 | acc13 | acc14 | acc15 | simbology |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 32 | 6 | -8 | 11 | -7 | 30 | 6 | 7 | 32 | 8 | 7 | 30 | 9 | -7 | 31 | A |
| 1 | 33 | 6 | -9 | 12 | -8 | 29 | 6 | 7 | 32 | 8 | 7 | 30 | 8 | -7 | 31 | A |
| 2 | 33 | 5 | -8 | 11 | -7 | 30 | 6 | 8 | 32 | 8 | 7 | 31 | 8 | -6 | 31 | A |
| 3 | 33 | 5 | -8 | 11 | -7 | 30 | 5 | 7 | 32 | 8 | 7 | 30 | 8 | -7 | 31 | A |
| 4 | 33 | 5 | -8 | 11 | -7 | 30 | 5 | 7 | 32 | 7 | 8 | 31 | 7 | -6 | 32 | A |

TABLE 1: Head of the dataset

| | acc1 | acc2 | acc3 | acc4 | acc5 | acc6 | acc7 | acc8 | acc9 | acc10 | acc11 | acc12 | acc13 | acc14 | acc15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1.00 | 1.74 | -0.61 | -0.44 | -1.51 | 1.23 | -0.50 | -1.02 | 1.66 | -0.29 | -2.08 | 1.75 | -0.33 | -2.66 | 1.52 |
| 1 | 1.09 | 1.74 | -0.68 | -0.37 | -1.60 | 1.15 | -0.50 | -1.02 | 1.66 | -0.29 | -2.08 | 1.74 | -0.40 | -2.6691 | 1.52 |
| 2 | 1.09 | 1.65 | -0.61 | -0.44 | -1.51 | 1.23 | -0.50 | -0.90 | 1.66 | -0.29 | -2.08 | 1.82 | -0.40 | -2.54 | 1.52 |
| 3 | 1.09 | 1.65 | -0.61 | -0.44 | -1.51 | 1.23 | -0.56 | -1.02 | 1.66 | -0.29 | -2.08 | 1.74 | -0.40 | -2.66 | 1.52 |
| 4 | 1.0 | 1.65 | -0.61 | -0.44 | -1.51 | 1.2397 | -0.56 | -1.02 | 1.66 | -0.35 | -1.91 | 1.82 | -0.47 | -2.54 | 1.62 |

TABLE 2: Head of the standardized dataset

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | -2.87 | 4.07 | 0.76 | 0.04 | -1.06 |
| 1 | -3.26 | 3.28 | 0.69 | -0.39 | -0.84 |
| 2 | -2.13 | 4.17 | 0.43 | 0.32 | -1.06 |
| 3 | -3.54 | 3.53 | 0.88 | -0.10 | -1.07 |

TABLE 4: Caption

The number of support vectors for each class is: [17, 23, 24, 33, 66, 28, 37, 32, 56, 48, 43, 40, 71, 74, 71, 38, 31, 20, 47, 78, 61, 46, 56, 19, 42, 31, 51].

### 2.5 Prediction and Performance Analysis

The accuracy score for each classifier can be observed in the following table, considering the 3 possible PCA reductions (4,5 and 6 components).

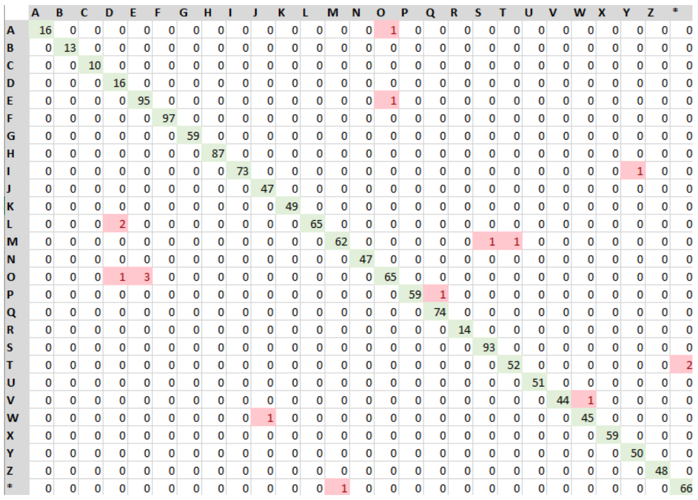
| PCA dimensions | LDA | QDA | SVM | RF |
|---|---|---|---|---|
| **4** | 91.17 | 97.21 | 94.84 | 96.87 |
| **5** | 97.02 | 98.98 | 99.18 | 98.84 |
| **6** | 97.14 | 99.11 | 98.98 | 99.11 |

TABLE 5: Accuracies depending on the method.

The highest accuracy is obtained by the SVM classifier with 5 components, and the worst by LDA with 4 components.

A more detailed breakdown of the classification result is provided by the confusion matrices. These reflect in a clear manner the number of samples miss-classified, and it is an especially useful method to analyze the performance. The next table represent the Confusion Matrix of the SVM classifier.

Not only the results of the confusion matrices are important when studying the performance of a classifier. The time each classifier takes to finish the process of training and classifying a given data set is also essential to decide which one to use. For that reason, a time performance analysis is carried out using the timeit() Python method from the timeit library. This method measures the execution time (in seconds) given by a certain code snippet. The number of iterations needed to run the desired portion of code will be set to 1000 to avoid extremely large execution times.



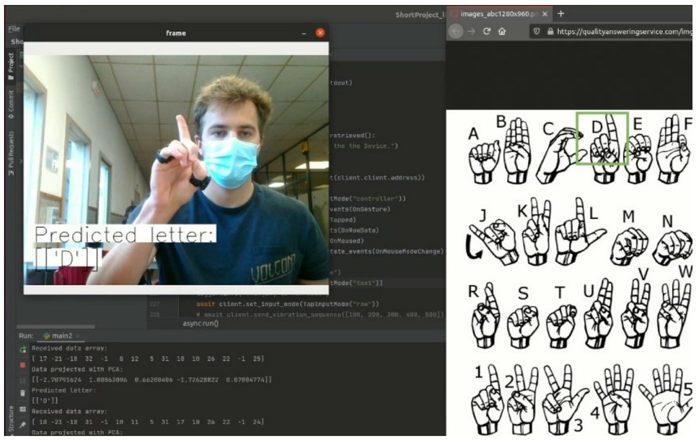
| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | * |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **A** | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **B** | 0 | 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **C** | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **D** | 0 | 0 | 0 | 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **E** | 0 | 0 | 0 | 0 | 95 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **F** | 0 | 0 | 0 | 0 | 0 | 97 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **G** | 0 | 0 | 0 | 0 | 0 | 0 | 59 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **H** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 87 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **I** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 73 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| **J** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 47 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **K** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 49 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **L** | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 65 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **M** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 62 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **N** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 47 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **O** | 0 | 0 | 0 | 1 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 65 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **P** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 59 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Q** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 74 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **R** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **S** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 93 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **T** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 52 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| **U** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 51 | 0 | 0 | 0 | 0 | 0 | 0 |
| **V** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 44 | 1 | 0 | 0 | 0 | 0 |
| **W** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 45 | 0 | 0 | 0 | 0 |
| **X** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 59 | 0 | 0 | 0 |
| **Y** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 50 | 0 | 0 |
| **Z** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 48 | 0 |
| ***** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 66 |

Fig. 3: Confusion matrix for SVM method.

| Classifier | Execution time x 1000 |
|---|---|
| **LDA** | 5.95 |
| **QDA** | 8.79 |
| **SVM** | 182 |
| **RF** | 864 |

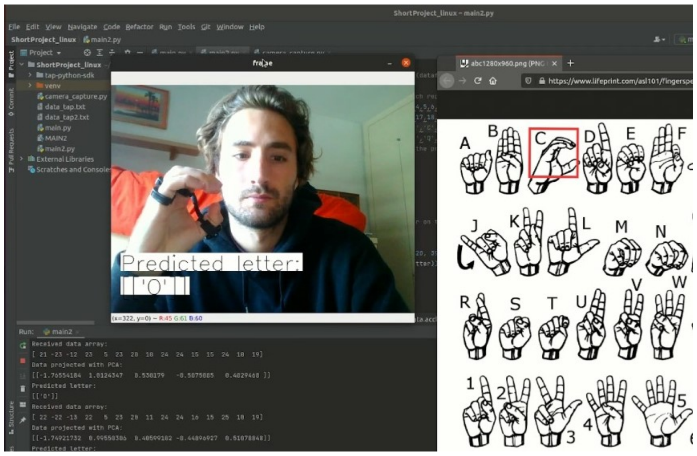
TABLE 6: Execution time of each method x1000.

## 3 REAL TIME APPLICATION



```
await client.set_input_mode(TapInputMode("raw"))
# await client.send_vibration_sequence([100, 200, 300, 400, 500])
asyncrun()
```

```
Received data array:
[ 17 -21 -18  32  -1   8  12   5  31  18  10  26  22  -1  25]
Data projected with PCA:
[[-2.78791624  1.80863086  0.66288486 -1.72628822  0.87804774]]
Predicted letter:
[['D']]
Received data array:
[ 18 -21 -18  31  -1  10  11   5  31  17  18  26  22  -1  24]
Data projected with PCA:
```



```
Received data array:
[ 21 -23 -12  23   5  23  20  10  24  24  15  15  24  10  19]
Data projected with PCA:
[[-1.76554184  1.0124347   0.530179   -0.5075085   0.4029468 ]]
Predicted letter:
[['O']]
Received data array:
[ 22 -22 -13  22   5  23  20  11  24  24  16  15  25  10  19]
Data projected with PCA:
[[-1.74921732  0.99558386  0.40599182 -0.44896927  0.51078848]]
Predicted letter:
```

Fig. 4: Correct/Incorrect tap prediction using SVM classifier.

## 4 Conclusions

- The PCA analysis reduces the dimensions of the data from 15 dimensions to 5 dimensions, where the 91% of variability of data is explained.

- The accuracies of the classifiers are very good, in fact all the methods perform similarly. However, the best classifier is SVM with 5 components, achieving 99.18% of accuracy.

- From the confusion matrices and performance scores we can validate that there are few errors and few incorrectly detected classes.

- Analyzing the execution time for each algorithm, we see that the SVM, which we considered as the best one regarding accuracy has the second slowest execution time. The random forest classifier is awfully slow. The classifiers that use discriminant analysis (LDA, QDA) are the fastest ones, and since the accuracy is also good, we could use them if we required minimization of time.

- The code written can predict the alphabet letters on real-time with almost no delay, which would be very helpful for its possible users.

## References

[1] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[2] TapWithUs. *TapWithUs/tap-python-sdk*. Mar. 2021. URL: https://github.com/TapWithUs/tap-python-sdk.

## APPENDIX A
## CODE - DATA PROCESSING & CLASSIFIERS ANALYSIS

```
# Basic libraries
import math
import numpy as np
import pandas as pd
import seaborn as sns
from matplotlib import *
import matplotlib.pyplot as plt
from sklearn.utils import shuffle

# Dimensionality Reduction
from sklearn.preprocessing import StandardScaler
from sklearn import decomposition

# Performance Metrics
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report

# Classifiers
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn import svm
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier

# Other libraries
import plotly.graph_objects as go
import timeit

df = pd.read_csv('data_tap2.txt', delimiter = ",")
#df = pd.read_csv('/content/drive/MyDrive/Machine Learning & Pattern
    Recognition/short_project/data_tap.txt',delimiter=",")
df.head()
print(df.shape)

# Eliminate the nan values
df.dropna()

# Change the string values in simbology to int values:
df['simbology'] = df['simbology'].replace(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
    'I', 'J4', 'K', 'L', 'M', 'N','O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
    'Y', 'Z', '*'],[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
    19, 20, 21, 22, 23, 24, 25, 26, 27])
true_simbology = df['simbology']
true_simbology

# 2. DIMENSIONALITY REDUCTION
# 1) Normalize the data
D = df.iloc[:,:-1].to_numpy()
D_st= StandardScaler().fit_transform(D)
pd.DataFrame(D_st)

# 2) Apply a decomposition analysis to all the features to observe the variance
```

```
pca = decomposition.PCA(n_components=14).fit(D_st)
exp_var = 100*pca.explained_variance_ratio_.cumsum()

# Plot the explained variance as a function of the components
plt.plot(exp_var,np.arange(1, 15, 1))
plt.grid()
plt.xlabel('Explained variance')
plt.ylabel('Number of components')

# Explained variance taking (4, 5 or 6) components
print('The 4th PCA component explains {} % of the variability of the
    data'.format(exp_var[3]))
print('The 5th PCA component explains {} % of the variability of the
    data'.format(exp_var[4]))
print('The 6th PCA component explains {} % of the variability of the
    data'.format(exp_var[5]))

pca_4comp = decomposition.PCA(n_components=4).fit(D_st)
pca_5comp = decomposition.PCA(n_components=5).fit(D_st)
pca_6comp = decomposition.PCA(n_components=6).fit(D_st)

# Chose the amount of components to visualize the accuracies later
pca = pca_5comp

# Data projected
Xproj = pca.transform(D_st)
print(Xproj.shape)
Xproj = pd.DataFrame(Xproj)
Xproj['symbols'] = df['simbology']
Xproj = shuffle(Xproj)

# 3. CLASSIFIERS
# Without PCA -> consider df
# With PCA -> consider Xproj
# keep only numeric variables
df_num = Xproj.select_dtypes(include='number')

# Define the train dataset (70%) and the test dataset (30%)
[row, col] = df_num.shape
training_rows = math.ceil(row*.7)
testing_rows = math.floor(row*.3)

# trainX, testX, trainY, testY = train_test_split( X, Y, test_size = 0.3)
data_train = df_num.iloc[:training_rows,:col-1]
label_train = df_num.iloc[:training_rows,col-1:]
data_test = df_num.iloc[training_rows:,:col-1]
label_test = df_num.iloc[training_rows:,col-1:]
label_test

# 3.1 Linear Discriminant Analysis
# Training the model
#clf_lda = LinearDiscriminantAnalysis(n_components=27,priors=None)
clf_lda = LinearDiscriminantAnalysis(n_components=5,priors=None)
clf_lda.fit(data_train, label_train['symbols'].values)

# Predict with test data
label_predicted_lda = clf_lda.predict(data_test)
```

```
# Accuracy score
accur1 = accuracy_score(label_test, label_predicted_lda)
print('The accuracy of the LDA is: {} %'.format(accur1*100))


#3.2 Quadratic Discriminant Analysis
# Training the model
clf_qda = QuadraticDiscriminantAnalysis(priors=None)
#clf_qda.fit(data_train, label_train)
clf_qda.fit(data_train, label_train['symbols'].values)

# Predict with test data
label_predicted_qda = clf_qda.predict(data_test)

# Accuracy score
accur2 = accuracy_score(label_test, label_predicted_qda)
print('The accuracy of the QDA is: {} %'.format(accur2*100))

# Support Vector Machine
support = svm.SVC()
support.fit(data_train, label_train['symbols'].values)
print('The accuracy of the SVM is: {} %'.format(100*support.score(data_test,
   label_test)))

label_predicted_svm = support.predict(data_test)

# The localization of the support vectors
pd.DataFrame(support.support_vectors_)
# Only x points are important in order to build the classifier
support.dual_coef_.sum()
# If he sum is equal to 0; then it is well balanced
# indices of support vectors
support.support_
# get number of support vectors for each class
support.n_support_

# 3.4 Random forest
forest = RandomForestClassifier()
#forest.fit(data_train, label_train)
forest.fit(data_train, label_train['symbols'].values)
print('The accuracy of the RandomForest is: {} %'.format(100*forest.score(data_test,
   label_test)))

label_predicted_rf = forest.predict(data_test)

# 4. PERFORMANCE ANALYSIS
# Accuracy of each classifier depending on the dimensions used for the PCA
fig = go.Figure(data=[go.Table(header=dict(values=['PCA dimensions','LDA', 'QDA',
   'SVM', 'RF']),cells=dict(values=[['4', '5', '6'], [LDA_4comp, LDA_5comp,
   LDA_6comp], [QDA_4comp, QDA_5comp, QDA_6comp], svm_4comp, svm_5comp, svm_6comp],
   [rf_4comp, rf_5comp, rf_6comp]]))])

# LDA Classification report
print(classification_report(label_test, label_predicted_lda))
# QDA Classification report
print(classification_report(label_test, label_predicted_qda))
# SVM Classification report
print(classification_report(label_test, label_predicted_svm))
```

```
# RF Classification report
print(classification_report(label_test, label_predicted_rf))

# LDA Confusion Matrix
CM_LDA = confusion_matrix(label_test, label_predicted_lda)
CM_LDA = pd.DataFrame(CM_LDA)
CM_LDA.index = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
    'N','O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '*']
# QDA Confusion matrix
CM_QDA = confusion_matrix(label_test, label_predicted_qda)
CM_QDA = pd.DataFrame(CM_QDA)
CM_QDA.index = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
    'N','O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '*']
# SVM Confusion matrix
CM_SVM = confusion_matrix(label_test, label_predicted_svm)
CM_SVM = pd.DataFrame(CM_SVM)
CM_SVM.index = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
    'N','O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '*']
# RF Confusion matrix
CM_RF = confusion_matrix(label_test, label_predicted_rf)
CM_RF= pd.DataFrame(CM_RF)
CM_RF.index = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
    'N','O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '*']

# LDA time performance analysis
s_lda = """\
clf_lda = LinearDiscriminantAnalysis(n_components=5,priors=None)
clf_lda.fit(data_train, label_train['symbols'].values)
label_predicted_lda = clf_lda.predict(data_test)
"""
time_lda = timeit.timeit(stmt=s_lda, number=1000,globals=globals())
print(time_lda)
# QDA time performance analysis
s_qda = """\
clf_qda = QuadraticDiscriminantAnalysis(priors=None)
clf_qda.fit(data_train, label_train['symbols'].values)
label_predicted_qda = clf_qda.predict(data_test)
"""
time_qda = timeit.timeit(stmt=s_qda, number=1000,globals=globals())
print(time_qda)
# SVM time performance analysis
s_svm = """\
support = svm.SVC()
support.fit(data_train, label_train['symbols'].values)
label_predicted_svm = support.predict(data_test)
"""
time_svm = timeit.timeit(stmt=s_svm, number=1000,globals=globals())
print(time_svm)
# RF time performance analysis
s_rf = """\
forest = RandomForestClassifier()
forest.fit(data_train, label_train['symbols'].values)
label_predicted_rf = forest.predict(data_test)
"""
time_rf = timeit.timeit(stmt=s_rf, number=10,globals=globals())
print(time_rf)
```

## APPENDIX B
## CODE - REAL TIME APPLICATION

```
# TAP libraries
from bleak import _logger as logger
from tapsdk.models import AirGestures
from tapsdk import TapSDK, TapInputMode
import asyncio

# Basic libraries
import math
import numpy as np
import pandas as pd
from matplotlib import *
from sklearn.utils import shuffle

# Dimensionality Reduction libraries
from sklearn.preprocessing import StandardScaler
from sklearn import decomposition

# Performance Metrics libraries
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix

# Classifier's libraries
from sklearn import svm

# Webcam libraries
import cv2

# Define global variables
data_array = []
data_dataframe = pd.DataFrame()
classifier = svm.SVC()

############ DATA ACQUISITION ##############
# Upload the dataset from a txt file(the txt file must be edited previously to
    eliminate the[ and] characters)
df = pd.read_csv('data_tap.txt', delimiter=",")
print(df)
# Shuffle the complete dataset before training
# df = shuffle(df)
# Eliminate the nan values (there are non nan-values, therefore it is not severely
    handled)
df.dropna()
# Change the string values (the labels) from the column 'symbology' into int values:
df['simbology'] = df['simbology'].replace(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H',
    'I', 'J4', 'K', 'L', 'M', 'N',
                                        'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
                                            'Y', 'Z', '*'],
                                       [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
                                        14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
                                            27])
true_symbology = df['simbology']
```

```
############ PCA Dimensionality Reduction ##############
# Normalize the data
D = df.iloc[:, :-1].to_numpy()
scaler = StandardScaler()
scaler.fit(D)
D_st = scaler.transform(D)
print(D_st)
# Obtain the means of the input data
D_mean = D.mean(axis=0)
print('The meaens are:' + str(D_mean))
# OBtain the covariances od the input data
D_std = D.std(axis=0)
print('The covariances are' + str(D_std))
# Explained variance from the 5th component
pca = decomposition.PCA(n_components=5).fit(D_st)
# Data projected
Xproj = pca.transform(D_st)
print('The shape of the projected data is:' + str(Xproj.shape))
Xproj = pd.DataFrame(Xproj)
print(Xproj)
print(type(Xproj))
Xproj = pd.DataFrame(Xproj)
Xproj['symbols'] = true_symbology
Xproj = shuffle(Xproj)
print('The projected data in dataframe format is:' + '\n' + str(Xproj))
# Keep only the numeric variables to perform the fitting
df_num = Xproj.select_dtypes(include='number')

############ Create the Support Vector Machine classifier ##############
# Define the train dataset (70%) and the test dataset (30%)
[row, col] = df_num.shape
training_rows = math.ceil(row * .7)
testing_rows = math.floor(row * .3)
data_train = df_num.iloc[:training_rows, :col-1]
label_train = df_num.iloc[:training_rows, col-1:]
data_test = df_num.iloc[training_rows:, :col-1]
label_test = df_num.iloc[training_rows:, col-1]
# The following line of code can also be used to do the same data partition:
# trainX, testX, trainY, testY = train_test_split( X, Y, test_size = 0.3)
# Training the model with the specified classifier
# Choose from the proposed ones [clf_lda, clf_qda, svm, rf...]
classifier.fit(np.array(data_train), np.array(label_train))
# Predict with the test data
label_predicted = classifier.predict(np.array(data_test))
# print(label_predicted)
# Calculate and show the Confusion Matrix
CM_svm = confusion_matrix(label_test, label_predicted)
print('Confusion Matrix:' + '\n' + str(pd.DataFrame(CM_svm)))
# Calculate and shoe the Accuracy score
accur2 = accuracy_score(label_test, label_predicted)
print('The accuracy of the SVM is: {} %'.format(accur2 * 100))
# Predict just for a simple array of data
print('\n' + 'Predict just for a simple array of data:')
# Without PCA -> test_array = np.array([13, -27, -9, -15, -2, 34, 0, 9, 33, 10, 19,
    25, 7, 12, 30])
# With PCA ->
test_array = np.array([56.353360, 36.461163, -9.610339, -3.687739, 5.370382])
print('Observation' + str(test_array))
```

```
test_array_pred = classifier.predict([test_array])
print('Predicted class:' + str(test_array_pred))

# Define the function that will generate a box containing the predicted letter
def __draw_label(img, text, pos, bg_color):
   font_face = cv2.FONT_HERSHEY_SIMPLEX
   scale = 1.5
   color = (0, 0, 0)
   thickness = cv2.FILLED
   margin = 5
   txt_size = cv2.getTextSize(text, font_face, scale, thickness)
   end_x = pos[0] + txt_size[0][0] + margin
   end_y = pos[1] - txt_size[0][1] - margin

   cv2.rectangle(img, pos, (end_x, end_y), bg_color, thickness)
   cv2.putText(img, text, pos, font_face, scale, color, 1, cv2.LINE_AA)

# Libraries, SDK downloaded from the TapStrap device
def notification_handler(sender, data):
   """Simple notification handler which prints the data received."""
   print("{0}: {1}".format(sender, data))
def OnMouseModeChange(identifier, mouse_mode):
   print(identifier + " changed to mode " + str(mouse_mode))
def OnTapped(identifier, tapcode):
   print(identifier + " tapped " + str(tapcode))
def OnGesture(identifier, gesture):
   print(identifier + " gesture " + str(AirGestures(gesture)))
def OnTapConnected(self, identifier, name, fw):
   print(identifier + " Tap: " + str(name), " FW Version: ", fw)
def OnTapDisconnected(self, identifier):
   print(identifier + " Tap: " + identifier + " disconnected")
def OnMoused(identifier, vx, vy, isMouse):
   print(identifier + " mouse movement: %d, %d, %d" % (vx, vy, isMouse))

# The function that will be used to generate raw data from the hand-movements
# (xyz axis coordinates from every finger with respect to the thumb)
def OnRawData(identifier, packets):
   # Uncomment to acquire data_ Open a txt file where all the data stream from the
       device will be saved
   # txt = open('data_tap2.txt', 'a')
   # Read only the accelerometer values from all the information send in 1 packet
       (lots of compressed data)
   for m in packets:
      # If the information received from the packet comes from an accelerometer, then
          use it
      if m["type"] == "accl":
         # Process the stream of data every 35ms (this value can be modified)
         OnRawData.accl_cnt += 1
         if OnRawData.accl_cnt == 35:
            OnRawData.accl_cnt = 0
            # Save the received data (15 values = xyz * 5 fingers) into an array
            data_array = np.array(m["payload"])
            #The means are:
            means =
               np.array([21.51181263,-13.28105906,1.1311609,16.97311609,9.96863544,
                          14.86191446,14.16558045,15.31731161, 9.9790224,12.56639511,
                          19.10753564,8.14806517,13.82708758,15.22423625,15.64663951])
            # The covariances are
            stds = np.array([10.46414511, 11.02906859,14.72906386, 13.39172048,
               11.21374602,
                        12.21047911, 16.17396756,8.08367176,13.20777744,15.46618317,
                        5.80464771, 12.52223567,14.47146358,8.32648604,10.06518388])
            # Normalize the new data packet received
            DS = (data_array - means)/stds
            # Apply PCA to the normalized data
            data_proj = pca.transform([DS])
            # To perform the data acquisition, uncomment the next line to write the
               data into a new line on the .txt
            # txt.write(str(m["payload"]) + ',' + 'R' + '\n')
            # Predict the corresponding number using the previously chosen classifier
            number_pred = classifier.predict(data_proj)
            # Convert the number predicted (array) into a (dataframe) to be able to
               use the replace function
            number_pred_df= pd.DataFrame(number_pred)
            # Replace the numeric values into strings, which represent the predicted
               letter from the alphabet
            number_letter = number_pred_df.replace([1,2,3,4,5,6,7,8,9,10,11,12,13,14,
                                          15,16,17,18,19,20,21,22,23,24,25,26,27],
                                         ['A','B','C','D','E','F','G','H','I','J','K','L','M',
                                          'O','P','Q','R','S','T','U','V','W','X','Y','Z','*']
            # Print all the desired variables to check if the program works properly
            print('Received data array:')
            print(data_array)
            print('Data projected with PCA:')
            print(data_proj)
            print('Predicted letter:')
            print(str(np.array(number_letter)))
            # Open the webcam and show the predicted letter on the screen to check
               the classifier's functionality
            ret, frame = cap.read()
            if ret == True:
               __draw_label(frame, 'Predicted letter:', (20, 390), (255, 255, 255))
               __draw_label(frame, str(np.array(number_letter)) , (20, 440), (255,
                  255, 255)) # str(np.array(number_letter))
            cv2.imshow('frame', frame)
            k = cv2.waitKey(5) & 0xff
            # Press the 'esc' key to close the webcam
            if k == 27:
               break

OnRawData.imu_cnt = 0
OnRawData.accl_cnt = 0
OnRawData.cnt = 0

# Internal function from the Tap device
async def run(loop=None, debug=False):
   if debug:
      import sys

      loop.set_debug(True)
      h = logging.StreamHandler(sys.stdout)
      h.setLevel(logging.WARNING)
      logger.addHandler(h)

   client = TapSDK(None, loop)
   if not await client.client.connect_retrieved():
      logger.error("Failed to connect the the Device.")
      return

   logger.info("Connected to {}".format(client.client.address))

   await client.set_input_mode(TapInputMode("controller"))
   await client.register_air_gesture_events(OnGesture)
   await client.register_tap_events(OnTapped)
   await client.register_raw_data_events(OnRawData)
   await client.register_mouse_events(OnMoused)
   await client.register_air_gesture_state_events(OnMouseModeChange)

   # logger.info("Changing to text mode")
   await client.set_input_mode(TapInputMode("text"))
   logger.info("Changing to raw mode")
   await client.set_input_mode(TapInputMode("raw"))
   # await client.send_vibration_sequence([100, 200, 300, 400, 500])
   await asyncio.sleep(50.0, loop=loop)

# Main function from the program
if __name__ == "__main__":
   cap = cv2.VideoCapture(0)
   loop = asyncio.get_event_loop()
   loop.run_until_complete(run(loop, True))
   cap.release()
   cv2.destroyAllWindows()
```