ECG Heartbeat Classification

In this practice we will detect cases of cardiovascular disease througth the analysis of heardbeats

This exercise is based on [paper] that solves the problem we are facing.

1. Data Analysis

Library Import

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

Dataset Import

```
# Connect with Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
data = pd.read_csv('/content/drive/MyDrive/datasets/DL1_ECD/mitbih_train.csv', header=N
```

```
data.shape
```

```
(87554, 188)
```

```
data.head()
```

Each column represents an elctrocardiogram reading (at 125hz). In total thera are 187 readings, in this columns we have about a second and half off keystrokes. The last column contains the category to which these keystrokes belong. In total there are five, each represented by a number:

- Normal:0
- Premature arrhythmia:1
- Ventricular premature contraction or Ventricular escape:2
- Fusion of ventricular and normal contraction:3
- Resuscitation, fusion of normal and resucitation or unclassifiable:4

Data Distribution

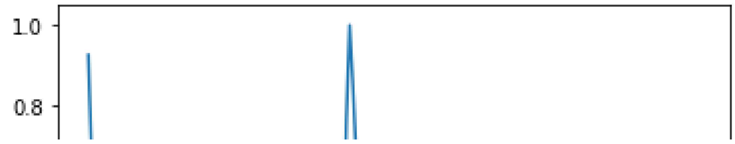
```
data.describe()
```

	0	1	2	3	4	
count	87554.000000	87554.000000	87554.000000	87554.000000	87554.000000	87554.0
mean	0.890360	0.758160	0.423972	0.219104	0.201127	0.2
std	0.240909	0.221813	0.227305	0.206878	0.177058	0.1
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.0
25%	0.921922	0.682486	0.250969	0.048458	0.082329	0.0
50%	0.991342	0.826013	0.429472	0.166000	0.147878	0.1
75%	1.000000	0.910506	0.578767	0.341727	0.258993	0.2
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.0

8 rows × 188 columns

```
plt.plot(data.iloc[3])
```

```
[<matplotlib.lines.Line2D at 0x7f59e43072d0>]
```



2. Data Pocessing

Now that we have visualized our data, let is work with it. Firs to divide them to imput and output.

```
# Converting the Dataset to a Numpy array
M = data.values
X = M[ : ,:-1] # Matriz M without the last column
y = M[ : , -1].astype(int) # The last column of M
y
```

```
array([0, 0, 0, ..., 4, 4, 4])
```

Arrays are created with the indices of the examples that belong to each category. can be used [np.argwhere] and [np.flatten]

```
C0 = np.argwhere(y == 0).flatten()
C1 = np.argwhere(y == 1).flatten()
C2 = np.argwhere(y == 2).flatten()
C3 = np.argwhere(y == 3).flatten()
C4 = np.argwhere(y == 4).flatten()
C4
```

```
array([81123, 81124, 81125, ..., 87551, 87552, 87553])
```

```
# count how many examples we have of each category
u = {'N': C0, 'S': C1, 'V': C2, 'F': C3, 'Q': C4}
www = []
for k in u:
    print('Hay {} muestras de la categoría {}'.format(len(u[k]), k))
    www.append(len(u[k]))
```

```
Hay 72471 muestras de la categoría N
Hay 2223 muestras de la categoría S
Hay 5788 muestras de la categoría V
Hay 641 muestras de la categoría F
Hay 6431 muestras de la categoría Q
```

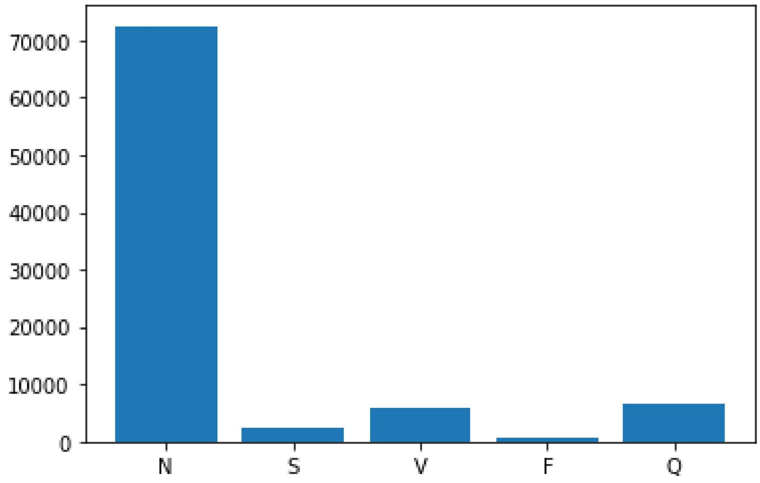
```
www
```

```
[72471, 2223, 5788, 641, 6431]
```

To better see how many of each type we have, we are going to make a bar graphand a plot

```
labels = list(u.keys())
plt.bar(labels, www)
```

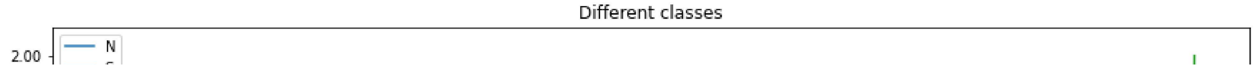
```
<BarContainer object of 5 artists>
```



We can print an electrocardiogram of each type

```
plt.figure(figsize=(15,8))
plt.plot(M[C0[20]], label='N')
plt.plot(M[C1[20]], label='S')
plt.plot(M[C2[20]], label='V')
plt.legend()
plt.title('Different classes')
```

```
Text(0.5, 1.0, 'Different classes')
```



Data preparation

[ ] ↳ *12 celdas ocultas*

3. AI Model

```
from sklearn import model_selection
from sklearn.metrics import confusion_matrix

import tensorflow
import keras
from keras.layers import Dense, Dropout, Activation, Flatten, Conv1D, Conv2D, MaxPoolir
from keras.utils import np_utils

from keras import models, layers, optimizers
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.utils import class_weight

from tensorflow.keras.optimizers import SGD, RMSprop, Adam, Adagrad, Adadelta, RMSprop
from keras.models import Sequential, model_from_json
from keras.preprocessing.image import ImageDataGenerator
from keras.callbacks import ReduceLROnPlateau, ModelCheckpoint
from keras import backend as K
from keras.applications.vgg16 import VGG16
from keras.models import Model

from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, la
import itertools
```

Define:

- Input length.
- Neurons in the last layer
- Batch_size for training with SGD.

```
signal_length = 187
batch_size = 256
n_classes = 5
```

Model

```
model = Sequential()

model.add(Conv1D(32, kernel_size=(5), input_shape=(signal_length, 1)))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Activation('relu'))

model.add(Conv1D(32, (4)))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling1D(pool_size=(2)))

model.add(Conv1D(32, (4)))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling1D(pool_size=(2)))

model.add(Conv1D(32, (4)))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Activation('relu'))
model.add(MaxPooling1D(pool_size=(2)))

model.add(Flatten())

model.add(Dense(128, activation='sigmoid'))
model.add(Dropout(0.5))
model.add(Dense(n_classes, activation='softmax'))

model.summary()

model.compile(loss=keras.losses.categorical_crossentropy,
              optimizer=tensorflow.keras.optimizers.Adadelta(),
              metrics=['accuracy'])

```

```
 conv1d (Conv1D)             (None, 183, 32)           192

 dropout (Dropout)           (None, 183, 32)           0

 batch_normalization (BatchN  (None, 183, 32)          128
 ormalization)

 activation (Activation)     (None, 183, 32)           0

 conv1d_1 (Conv1D)           (None, 180, 32)           4128

 dropout_1 (Dropout)         (None, 180, 32)           0

 batch_normalization_1 (Batc  (None, 180, 32)          128
 hNormalization)

 activation_1 (Activation)   (None, 180, 32)           0
```

```
 max_pooling1d (MaxPooling1D  (None, 90, 32)           0
 )

 conv1d_2 (Conv1D)           (None, 87, 32)            4128

 dropout_2 (Dropout)         (None, 87, 32)            0

 batch_normalization_2 (Batc  (None, 87, 32)           128
 hNormalization)

 activation_2 (Activation)   (None, 87, 32)            0

 max_pooling1d_1 (MaxPooling  (None, 43, 32)           0
 1D)

 conv1d_3 (Conv1D)           (None, 40, 32)            4128

 dropout_3 (Dropout)         (None, 40, 32)            0

 batch_normalization_3 (Batc  (None, 40, 32)           128
 hNormalization)

 activation_3 (Activation)   (None, 40, 32)            0

 max_pooling1d_2 (MaxPooling  (None, 20, 32)           0
 1D)

 flatten (Flatten)           (None, 640)               0

 dense (Dense)               (None, 128)               82048

 dropout_4 (Dropout)         (None, 128)               0

 dense_1 (Dense)             (None, 5)                 645

=================================================================
Total params: 95,781
Trainable params: 95,525
Non-trainable params: 256
_________________________________________________________________
```

To compile the model, .compile() is called. Here we specify which loss function we use, which optimizer and which metrics we want to keep for each epoch.

```
model.compile(loss='categorical_crossentropy', optimizer=Adam(), metrics=['accuracy'])
```

We train the model a number of epochs and with a specified batch_size. This returns us a history object with the accuracy of all the training phases.

Training

```
history = model.fit(X_train, y_train,
                    epochs=75,
                    batch_size=batch_size,
```

```
                    batch_size=batch_size,
                    verbose=1,
                    validation_data=(X_test, y_test))
```

```
274/274 [==============================] - 7s 25ms/step - loss: 0.0968 - accuracy:
Epoch 47/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0965 - accuracy:
Epoch 48/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0948 - accuracy:
Epoch 49/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0951 - accuracy:
Epoch 50/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0969 - accuracy:
Epoch 51/75
274/274 [==============================] - 7s 24ms/step - loss: 0.0950 - accuracy:
Epoch 52/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0958 - accuracy:
Epoch 53/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0953 - accuracy:
Epoch 54/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0937 - accuracy:
Epoch 55/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0928 - accuracy:
Epoch 56/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0935 - accuracy:
Epoch 57/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0956 - accuracy:
Epoch 58/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0901 - accuracy:
Epoch 59/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0919 - accuracy:
Epoch 60/75
274/274 [==============================] - 7s 24ms/step - loss: 0.0930 - accuracy:
Epoch 61/75
274/274 [==============================] - 7s 24ms/step - loss: 0.0925 - accuracy:
Epoch 62/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0887 - accuracy:
Epoch 63/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0897 - accuracy:
Epoch 64/75
274/274 [==============================] - 7s 26ms/step - loss: 0.0901 - accuracy:
Epoch 65/75
274/274 [==============================] - 7s 24ms/step - loss: 0.0900 - accuracy:
Epoch 66/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0885 - accuracy:
Epoch 67/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0909 - accuracy:
Epoch 68/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0889 - accuracy:
Epoch 69/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0900 - accuracy:
Epoch 70/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0888 - accuracy:
Epoch 71/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0883 - accuracy:
Epoch 72/75
274/274 [==============================] - 7s 25ms/step - loss: 0.0875 - accuracy:
Epoch 73/75
274/274 [==============================] - 7s 24ms/step - loss: 0.0882 - accuracy:
Epoch 74/75
274/274 [==============================] - 7s 24ms/step - loss: 0.0857 - accuracy:
```

```
Epoch 75/75
```

Access to the historical accuracy of the model (with the history attribute)

```
history.history
```

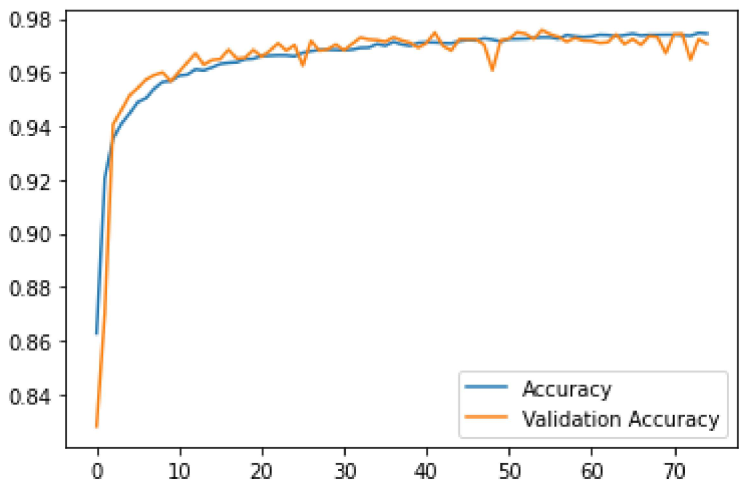
```
        0.11635860055685043,
        0.12120477110147476,
        0.11769509315490723,
        0.1110141202807426 5,
        0.11513130366802216,
        0.1069787815213203 4,
        0.10440034419298172,
        0.10927128791809082,
        0.10051724314689636,
        0.12406480312347412,
        0.09744863212108612,
        0.10840492695569992,
        0.10468432307243347,
        0.10148876160383224,
        0.10558591037988663,
        0.09736981987953186,
        0.09891257435083389,
        0.09821584075689316,
        0.09725581854581833,
        0.10032787919044495,
        0.09103266149759293,
        0.0982784628868103,
        0.09699684381484985,
        0.10702674835920334,
        0.09707674384117126,
        0.08697402477264404,
        0.10201660543680191,
        0.10459234565496445,
        0.09405051916837692,
        0.09666819125413895,
        0.091598279774189,
        0.09911223500967026,
        0.14012511074543,
        0.09638926386833191,
        0.09862349182367325,
        0.0911770761013031,
        0.08987992256879807,
        0.09612362831830978,
        0.08746001869440079,
        0.09349197149276733,
        0.09117641299962997,
        0.10377916693687439,
        0.09539712220430374,
        0.10085691511631012,
        0.09267361462116241,
        0.09856083989143372,
        0.09981495141983032,
        0.09145855158567429,
        0.10609102994203568,
        0.09644510596990585,
        0.10398600995540619,
        0.09261532127857208,
```

```
0.09731055051088333,
0.11862365156412125,
0.08503658324480057,
0.09194368124008179,
0.12440980225801468,
0.10360081493854523,
```

To see if our model is overfitting, a graph is drawn with the accuracy in train and in validation using the data from the history object.

```
plt.plot(history.history['accuracy'], label='Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
```

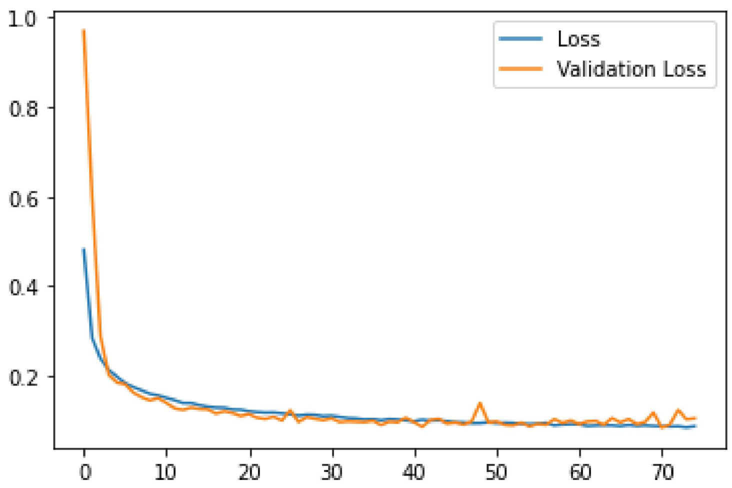
```
<matplotlib.legend.Legend at 0x7f58f19ee810>
```



And the Loss

```
plt.plot(history.history['loss'], label='Loss')
plt.plot(history.history['val_loss'], label='Validation Loss' )
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7f58f31f7390>
```



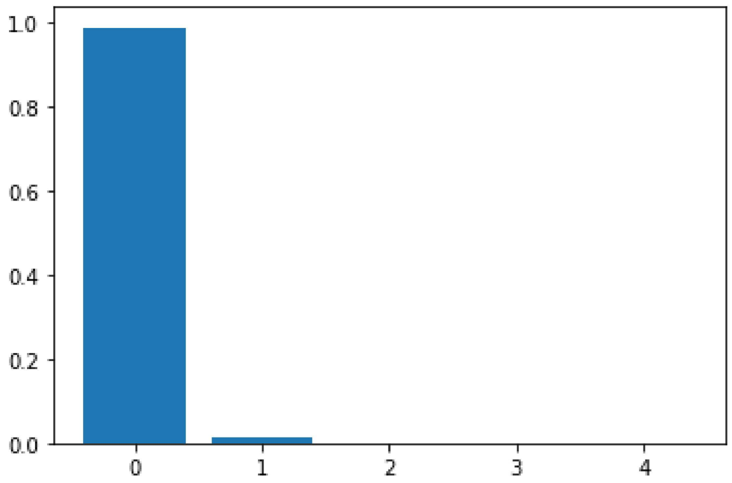
Model Test

```
y_pred = model.predict(X_test, batch_size=1000)
y_pred.shape
```

```
(17511, 5)
```

```
print(y_test[299])
plt.bar(range(n_classes), height=y_pred[4])
```

```
[1. 0. 0. 0. 0.]
<BarContainer object of 5 artists>
```



This code snippet generates a model report, and the next a confusion matrix.

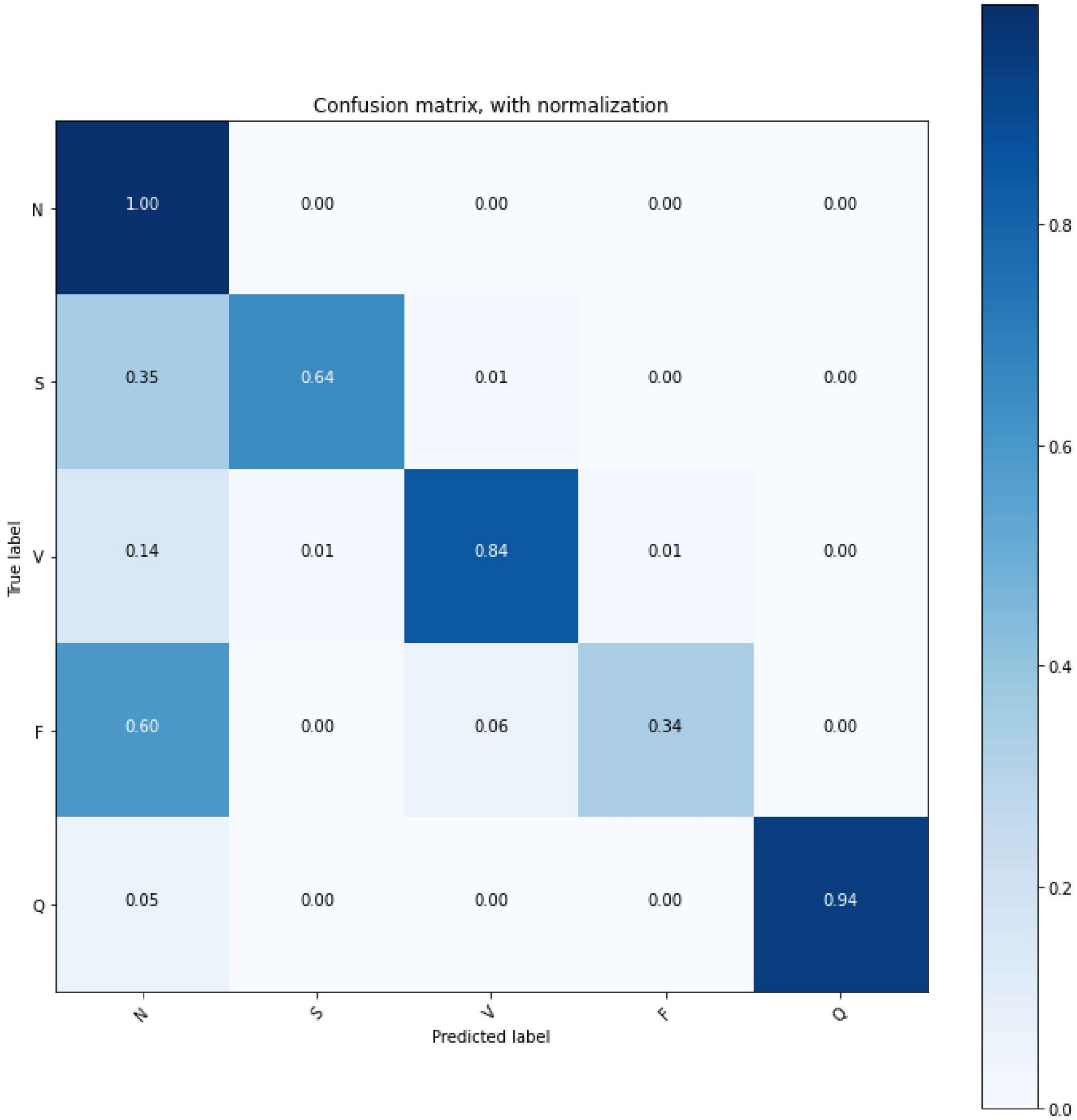
```
print(classification_report(y_test.argmax(axis=1), y_pred.argmax(axis=1)))
```

```
              precision    recall  f1-score   support

           0       0.97      1.00      0.98     14500
           1       0.96      0.64      0.77       441
           2       0.97      0.84      0.90      1137
           3       0.81      0.34      0.48       124
           4       1.00      0.94      0.97      1309

    accuracy                           0.97     17511
   macro avg       0.94      0.75      0.82     17511
weighted avg       0.97      0.97      0.97     17511
```

```
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
```

```
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

# Compute confusion matrix
cnf_matrix = confusion_matrix(y_test.argmax(axis=1), y_pred.argmax(axis=1))
np.set_printoptions(precision=2)

# Plot non-normalized confusion matrix
plt.figure(figsize=(10, 10))
plot_confusion_matrix(cnf_matrix, classes=['N', 'S', 'V', 'F', 'Q'],
                      title='Confusion matrix, with normalization',
                      normalize=True)
plt.show()
```

```
Normalized confusion matrix
```



✓ 0 s completado a las 15:55