Homework #1

Spoken Digit Recognition

Group 11, Assignment 2

Simone Marcucci, Marco Donzelli, Andrea Ravizzotti, Lina Bolshakova

1. INTRODUCTION

In this report we are going to explain how we implemented a classifier able to predict which digit is pronounced in a short audio excerpt.

Dataset
We used Free Spoken Digit Dataset (FSDD), an open dataset consisting of recorded of spoken digits in .wav files at 8kHz sample-rate.

The total number of files is 2000, all recorded from 4 different speakers in English (50 of each digit per speaker).

Library
As libraries we used

- ***Librosa***: a python package for music and audio analysis. It provides the building blocks necessary to create music information retrieval systems. We used it for computing features.
- ***Scikit*****:** a free machine learning library for Python that we used for *Support Vector Machine* (SVM).
- ***Numpy, Scipy, IPython.display***

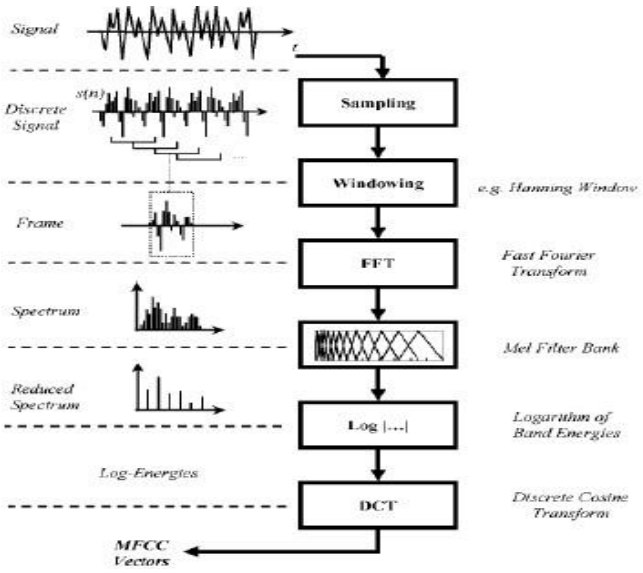
Classifier
In order to design a spoken digit classifier, we exploited *Support Vector Machines* (SVM), a binary classifier that learns the boundary between items belonging to two different classes.

In our case, we divided the database in 10 classes (one for each digit pronounced) and implemented a multi-class SVM classifier using the *"one vs. one"* strategy that splits a multi-class classification into one binary classification problem per each pair of classes.

In detail: an SVM classifier is built for every combination of two classes to be compared. This is done for every possible combination.

The result of these binary classification step is then fed to another stage of the model that implements an algorithm called *Majority Voting*: it simply takes the value that occurs the most in the binary predictions for a given file and gives it as the output. This step is embedded in the SVM classifier featured in *Scikit.*

Features
Features extraction was done using Librosa to compute the Mel-Frequency Cepstrum Coefficients (MFCC) since they are the most commonly used attributes in speech processing tasks such as speaker and speech recognition. We computed 13 MFCC coefficients for each file both for test and training set.
The procedure to extract them is the following:



2. CODE

Features computation

MFCC computation

We defined a function called *compute_mfcc* in which we first compute the Short Time Fourier Transform (STFT) using Librosa with a 1024 sample Hamming window and a hop size of 512 as suggested during the class. Then, we computed the Mel filters in order to apply them to the spectrogram. Afterwards, we took the logs of the power spectrum and we applied the Discrete Cosine Transform to the logarithmic Mel spectrogram to obtain the coefficients.

```
def compute_mfcc(audio, fs, n_mfcc):
    # Compute the spectrogram of the audio signal
    X = np.abs(librosa.stft(
        audio,
        window='hamming',
        n_fft=1024,
        hop_length=512,)
        )

    # Find the weights of the mel filters
    mel = librosa.filters.mel(
        sr=fs,
        n_fft=1024,
        n_mels=40,
        fmin=133.33,
        fmax=6853.8,
    )
    # Apply the filters to spectrogram
    melspectrogram = np.dot(mel, X)
    # Take the logarithm
    log_melspectrogram = np.log10(melspectrogram + 1e-16)

    # Apply the DCT to log melspectrogram to obtain the coefficients
    mfcc = sp.fftpack.dct(log_melspectrogram, axis=0, norm='ortho')[1:n_mfcc+1]
    return mfcc
```

Compute training features

Once we defined the classes we then proceeded to compute the training feature vector. We first started by initializing the vector and then splitting the dataset in training and test set. The files are named in the following format: {digitLabel}_{speakerName}_{index}.wav (e.g. *7_jackson_32.wav*).

We decided to create a cycle that automatically sorts the files in the test and training part.

As suggested by the creators of the dataset we put 10% of our files in the test and 90% in the training set.

Files with an index within the range 0-4 are assigned to the test set while the others are assigned to the training set. Once we split the dataset in train and test sets, we proceeded to assign the single files according to their class in a dictionary variable named *dict_train_files*.

Subsequently, we calculated the features for each key in *dict_train_files* using the *compute_mfcc* function we defined in the feature computation section. The feature matrices are stored in *dict_train_features* that has the same structure as *dict_train_files.*

```
classes = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
test = []
train = []
n_mfcc = 13
dict_train_files = {'0': [], '1': [], '2': [], '3': [], '4': [], '5': [], '6': [
], '7': [], '8': [], '9': []}
dict_train_features = {'0': [], '1': [], '2': [], '3': [], '4': [], '5': [], '6'
: [], '7': [], '8': [], '9': []}

train_root = 'free-spoken-digit-dataset/recordings'
class_train_files = [f for f in os.listdir(train_root) if f.endswith('.wav')]
#outputs a list with the filename of every file in train_root

#Splits dataset in test and train
for f in class_train_files:
  first_split = f.rsplit("_", 1)[1]
  second_split = first_split.rsplit(".", 1)[0]
  if int(second_split) <= 4:
      test.append(f)
  else:
      train.append(f)

#Puts files name in dict_train_files according to class
for c in classes:
  for f in train:
    split = f.rsplit("_", 2)[0]
    if int(c) == int(split):
      dict_train_files[c].append(f)

#Computes feature matrix according to class and puts it into dict_train_features
for c in dict_train_files:

  train_features = np.zeros((len(dict_train_files[c]), n_mfcc)) #

  for index, f in enumerate(dict_train_files[c]):
    audio, fs = librosa.load(os.path.join(train_root, f), sr=None)
    mfcc = compute_mfcc(audio, fs, n_mfcc)
    train_features[index, :] = np.mean(mfcc, axis=1)

    dict_train_features[c] = train_features
```

Compute test feature

The procedure applied for the training was repeated for the test.

```
#same as above for test set
dict_test_files = {'0': [], '1': [], '2': [], '3': [], '4': [], '5': [], '6': []
, '7': [], '8': [], '9': []}
dict_test_features = {'0': [], '1': [], '2': [], '3': [], '4': [], '5': [], '6':
 [], '7': [], '8': [], '9': []}

for c in classes:
  for f in test:
    splitto = f.rsplit("_", 2)[0]
    if int(c) == int(splitto):
      dict_test_files[c].append(f)


for c in dict_test_files:

  n_test_samples = len(dict_test_files[c]) #outputs length of class_train_files

  test_features = np.zeros((n_test_samples, n_mfcc))

  for index, f in enumerate(dict_test_files[c]):
    audio, fs = librosa.load(os.path.join(train_root, f), sr=None)
    mfcc = compute_mfcc(audio, fs, n_mfcc)
    test_features[index, :] = np.mean(mfcc, axis=1)

  dict_test_features[c] = test_features
```

Feature Visualization

In this section we plot the feature matrices for each class using *Matplotlib.pyplot* and displayed them in a graphical fashion.

Classifier (SVM)

In this section we generated the annotation vector.

```
#Creating the annotation row-vectors with as many
#columns as the feature matrices
y_train_0 = np.zeros((X_train_0.shape[0]))
y_train_1 = np.ones((X_train_1.shape[0]))
y_train_2 = np.full((X_train_2.shape[0]), 2)
y_train_3 = np.full((X_train_3.shape[0]), 3)
y_train_4 = np.full((X_train_4.shape[0]), 4)
y_train_5 = np.full((X_train_5.shape[0]), 5)
y_train_6 = np.full((X_train_6.shape[0]), 6)
y_train_7 = np.full((X_train_7.shape[0]), 7)
y_train_8 = np.full((X_train_8.shape[0]), 8)
y_train_9 = np.full((X_train_9.shape[0]), 9)

#Creating the annotation matrix by concatenation

y_train_mc = np.concatenate((y_train_0, y_train_1, y_train_2, y_train_3,
                             y_train_4, y_train_5, y_train_6, y_train_7,
                             y_train_8, y_train_9), axis=0)
```

Normalize Features

Since SVM algorithms are not scaling variant, we applied to our train and test set a normalization in order to have data ranging from 0 to 1.

```
#finding Max and Min value for each MEL frequency coefficient and
#normalizing it along the time axis

feat_max = np.max(np.concatenate((X_train_0, X_train_1, X_train_2,
                                  X_train_3, X_train_4, X_train_5,
                                  X_train_6, X_train_7, X_train_8,
                                  X_train_9), axis=0), axis=0)
feat_min = np.min(np.concatenate((X_train_0, X_train_1, X_train_2,
                                  X_train_3, X_train_4, X_train_5,
                                  X_train_6, X_train_7, X_train_8,
                                  X_train_9), axis=0), axis=0)

X_train_0_normalized = (X_train_0 - feat_min) / (feat_max - feat_min)
X_train_1_normalized = (X_train_1 - feat_min) / (feat_max - feat_min)
X_train_2_normalized = (X_train_2 - feat_min) / (feat_max - feat_min)
X_train_3_normalized = (X_train_3 - feat_min) / (feat_max - feat_min)
X_train_4_normalized = (X_train_4 - feat_min) / (feat_max - feat_min)
X_train_5_normalized = (X_train_5 - feat_min) / (feat_max - feat_min)
X_train_6_normalized = (X_train_6 - feat_min) / (feat_max - feat_min)
X_train_7_normalized = (X_train_7 - feat_min) / (feat_max - feat_min)
X_train_8_normalized = (X_train_8 - feat_min) / (feat_max - feat_min)
X_train_9_normalized = (X_train_9 - feat_min) / (feat_max - feat_min)

X_train_mc_normalized = np.concatenate((X_train_0_normalized,
                                X_train_1_normalized, X_train_2_normalized,
                                X_train_3_normalized, X_train_4_normalized,
                                X_train_5_normalized, X_train_6_normalized,
                                X_train_7_normalized, X_train_8_normalized,
                                X_train_9_normalized), axis=0)
```

Define and train a model for each couple of classes

In this section we set the *SVM* parameters. We implemented it with an *RBF* kernel function and a "one vs one" decision scheme, then we trained the model using the method fit and giving both the normalized train features matrix and the annotation vector as inputs.

```
#Builds SVM classifier
SVM_parameters={
    'C': 1,
    'kernel': 'rbf',
    'decision_function_shape' : 'ovo',
    'probability' : True
}

clf = sklearn.svm.SVC(**SVM_parameters)

#fits the model
clf.fit(X_train_mc_normalized, y_train_mc)
```

Make a prediction

We called the method predict on our trained model *clf* giving the normalized test set as input, the output will be a vector with the resulting prediction for each file of the training set.

```
y_test_predicted = clf.predict(X_test_mc_normalized)
```

3. Results

Confusion Matrix

The confusion matrix depicts the performance of a classification model. It is a square matrix with as many column and rows as the number of classes present in the classification problem. On the columns we can find the actual predicted values that the classifier returns whereas on the rows we find the awaited values. On the diagonal the CM reports the number of correct predictions the model performed in a particular class.

The quality of the classification method is also described by the accuracy, which is an aggregate value that can be obtained by calling the method *score* on the classifier we used .

The accuracy corresponding to our confusion matrix is 0.98 which means that on average 98% of the time the predicted value matches the actual value. The achieved result shows that the implemented classification model is indeed reliable.

```
def compute_cm_multiclass(gt, predicted):
    classes = np.unique(gt)

    CM = np.zeros((len(classes), len(classes)))

    for i in np.arange(len(classes)):
        pred_class = predicted[gt==i]

        for j in np.arange(len(pred_class)):
            CM[i, int(pred_class[j])] = CM[i, int(pred_class[j])] + 1
    print(CM)

compute_cm_multiclass(y_test_mc, y_test_predicted_mv)
#returns and prints the classifier accuracy
sc = clf.score(X_test_mc_normalized, y_test_mc)*100
print ("The classifier accuracy is: "+str(round(sc, 2)) +"%")
```

		PREDICTED VAL									
		Zero	One	Two	Three	Four	Five	Six	Seven	Eight	Nine
REAL VAL	Zero	20									
	One		19			1					
	Two			20							
	Three	1		1	18						
	Four					20					
	Five						20				
	Six							20			
	Seven								20		
	Eight									20	
	Nine						1				19

Table 1: CM of our classifier. Considering 20 test files for each class we can notice e.g. that files in class 'Three' were incorrectly classified two times: one in which the model predicted a 0 and one in which a 2 was predicted. Class 'Four', instead, scored perfectly at 20/20.