CMLS - Spoken digit classification

Group: Euclidean space cowboys
Personal codes: 10801101, 10560345, 10801270

April 26, 2021

Abstract

Our assignment consisted in implementing a classifier able to predict which digit is pronounced in a short audio excerpt. The dataset used (**Free Spoken Digit Dataset**) consisted of 3000 audio recordings, each one containing a single digit (0-9) pronounced by one of 6 different English speakers fifty times. All the $.Wav$ files were sampled at $Fs = 8kHz$ and they were trimmed so that they have minimal silence at the beginnings and ends. Each recording is named in this format: digitLabel_speakerName_index.wav (e.g. the label "4_nicholas_26.wav" means that this is the 26th file in which the speaker Nicholas says the number four). So, our task was to extract a set of features from each audio track, cluster them in the feature space in different classes and produce a confusion matrix to analyze the performance of our model.

Link to Github repository: CMLS-homework1.

1 Import Libraries

In the first part of the code we have imported all the libraries that we will need in different parts of the code. In particular, we used "sklearn" for different purposes: implementing all the classification methods, train and test splitting, feature scaling, plotting confusion matrices. In addition, we imported "pandas" for working with the 3000 files that we have in input. Finally, we used "matplotlib" and "seaborn" for plotting all the graphs.

2 Import recordings

In this part of the code we had to define the dictionary containing all the classes. Considering our data structure (ten classes, 3000 audio tracks), we decided to create ten classes collected in "class_train_files": each class will contain the corresponding audio files. To check the process was done correctly, we used "librosa.load" and "IPython.display.audio" for listening to a random audio file.

3 Split dataset

We decided to use "pandas" library for converting "class_train_files" into a DataFrame object, which is composed by 300 rows corresponding to the audio file for each class (e.g 0_george_0.wav for class 0, 2_george_1 for class 2) and 10 columns that obviously corresponds to the classes. After that, we splitted the data using "train_test_split", imported from sklearn, which divides the data in a one-liner. The arguments of this function are, respectively:

1. the **DataFrame**
2. **test_size**, that represent the proportion of the split (if $test\_size = 0.1$, $train\_size = 0.9$)
3. **random_state** that controls the shuffling applied to the data
4. **shuffle**, which was set **True**, meaning that the function shuffles the data before splitting.

4 Feature extraction

Before applying the Mel-frequency cepstrum method, we tried to use other feature extraction methods like "zero crossing rate" and "spectral centroid". The first one is used for counting the number of times in which the audio waveform crosses the zero axis and it is mostly useful for understanding when the signal is voiced or unvoiced. The second one is a measure of the tendency of the spectrum. After computing those features, we noticed that the values distribution between different classes was almost the same, consequently we decided to discard.

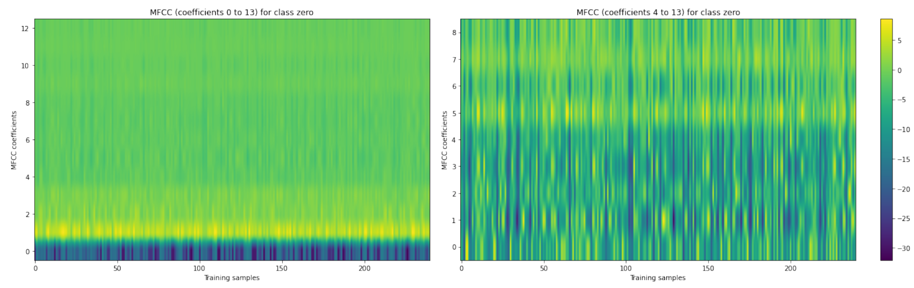
On the other hand, MFCC method is generally considered one of the best features for speech recognition and we extracted 13 mel-frequency cepstrum coefficients (MFCC) from each audio file (12 or 13 is the recommended number of coefficients for this task). These coefficients can be used as an optimal vector for representing the human voice.

4.1 MFCC scheme

Briefly, the scheme for the extraction of MFCC vectors is focused on a Mel Filter Bank which is a series of triangular filters with the center frequencies spaced according to the mel scale. These filters are applied on the FFT version of the signal which was previously sampled and windowed. After that, using the discrete cosine transformation we can obtain the mfcc vectors.

4.2 MFCC implementation

Coming back to the code: firstly, we have defined two different dictionaries that will contain the mfc coefficients for training and testing and the number of coefficients for each speech signal. Therefore, with the librosa function “mfcc”, we computed the processing of the mel-coefficients and we placed them into the dictionaries. The final result for class ”zero” was:



The figures show the MFC coefficients, the right image is trimmed from 4 to 13, for each class. The x-axis represents the samples, while the y-axis contains the coefficients.

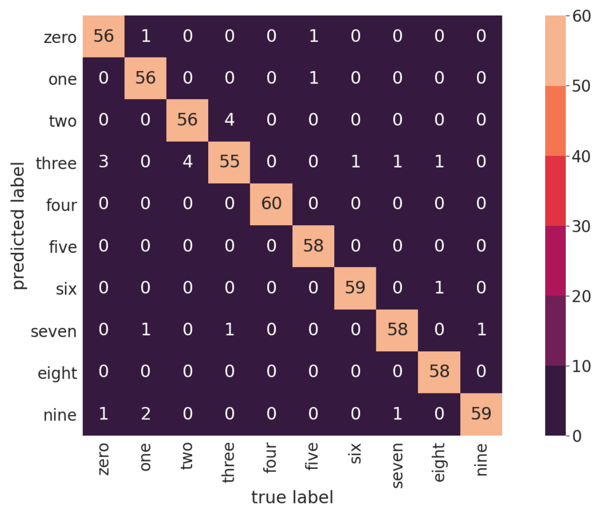
5 Gaussian Mixture Model

A Gaussian mixture model is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. Mixture models can be seen as a generalization of k-means clustering to incorporate information about the covariance structure of the data. In fact, a GMM finds a mixture of multi-dimensional Gaussian probability distributions that best model any input dataset. The GaussianMixture object (from sklearn) implements the expectation-maximization (EM) algorithm for fitting mixture-of-Gaussian models estimating the unknown parameters.

5.1 GMM implementation

For the implementation of GMM model we chose to set the number of mixture components as $n\_components = 3$. Then, we defined GMM through sklearn’s BayesianGaussianMixture, setting random_state = 2, and we trained it for each class, fitting with the dictionary containing the trained Mel-frequency cepstrum coefficients. For each class we initialized a multivariate normal variable through scipy.stats.multivariate_normal, setting the mean and the covariance matrix of the distribution like ones of the train mfccs corresponding to that class. We then appended in a list the weights that have been learnt from the same gaussian class, for each components times the pdf computed for the test features. We also sampled the multivariate normal distribution in order to visualize the gaussians in a 3D plot. In particular we sampled 500 points weighted by the corresponding gaussian weight. The next step was to define the PDFs summing for each mixture component the mixture PDFs referring to a specific class, then we concatenated them.

5.2 GMM metrics



Listing 1: Metrics

```
Results:
accuracy = 0.9583333333333334
precision = 1.0
recall = 0.9824561403508771
F1 score = 0.9911504424778761
```

With GMM, we built a model of *generative classification*, since we first determined the distribution of each class and then for each sample we assigned the corresponding label, based on the probability density function (pdf). In the next section we followed a different approach, called *discriminative classification*: instead of modeling each class as before, we can try to divide classes from each other by drawing lines, curves (2 dimensions) or manifolds (3+ dimensions).

6 Support Vector Machine

Support vector machines (SVM) are a powerful class of supervised Machine Learning algorithms used for classification and regression analysis. This means that, compared to GMM which is unsupervised, this model needs to know in advance the expected class for each sample.

6.1 Data preparation

First of all, we created X_train and X_test by concatenating respectively all dict_train_mfcc and dict_test_mfcc fields. After that, since SVMs are very sensitive to data scaling, we imported StandardScaler() from sklearn.preprocessing, we fitted this method with our X_train data and then we applied a scaling to both train and test features.

6.2 SVM implementation

In order to define a classifier, we set up a few parameters:

1. **C** (regularization parameter), it determines the penalty for misclassifying a data point. Starting from the default value ($C = 1$), we tried to decrease (determining more *bias* and less *variance* in our model) or increase (less *bias*, more *variance*) this value, choosing $C = 2$ as definitive value.

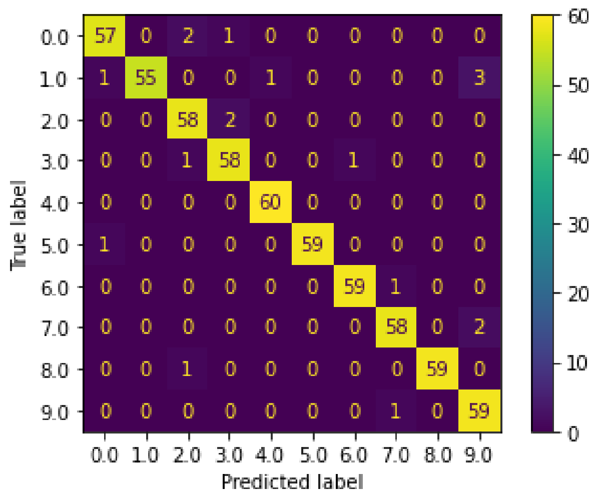
2. **Kernel**, it specifies the kernel type to be used in the algorithm. In other words, we can generate different type of boundaries between classes, based on the chosen kernel. after trying all the options, Radial Basis Function ('rbf') was clearly the best choice.

3. **Random state**, it's the seed for the pseudo random number generator. If set up, it ensures that multiple function calls will return the same result.

After adjusting those parameters (and leaving all the others as default), we fitted our model with the scaled Train features and the corresponding labels.

6.3 SVM metrics

Finally, we were able to predict the class of Y_test, generating a confusion matrix and some basic metrics:



Listing 2: Metrics

```
Results:
```

```
accuracy = 0.97
precision = 1.0
recall = 0.9821428571428571
F1 score = 0.9909909909909909
```

7 Multi-layer Perceptron

Our third approach consisted of implementing a deep learning method: our choice was the Multi-layer perceptron, since sklearn provides an easy and efficient implementation. A MLP is a class of feed-forward artificial neural networks characterized by at least three layers of nodes: an input layer (set of input features), one or more hidden layers and an output layer (final choice). Each node (except for the first layer) is a neuron that uses a nonlinear activation function. While a single Perceptron (only two layers) can only work with linearly separable classes (e.g., it can emulate an AND, but not a XOR), adding one or more hidden layers allows us to overcome this limit and deal with far more complex problems, as in our case. Like SVM, this model requires the prior knowledge of each sample's label, since during training phase it relies on a supervised learning technique called back-propagation.

7.1 Data preparation

The same data preprocessed for SVM was used for this section.

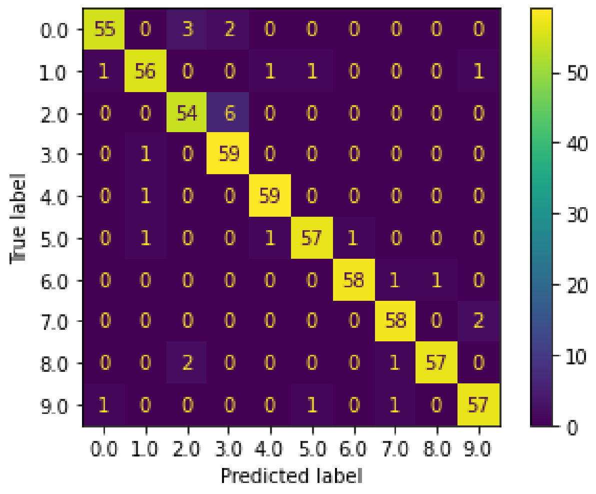
7.2 MLP implementation

As before, we needed to set up some parameters:

1. **activation**, activation function for the hidden layer, 'logistic' provided the best performance.
2. **solver**, the solver for weight optimization. Also in this case, the default option 'lbfgs' was the best choice.
3. **alpha**, it's the L2 penalty parameter (regularization term). The default value ($alpha = 1e - 5$) was the optimal.
4. **hidden_layer_sizes**. After trying to change both the number of hidden layers and their size, we set just one layer with 90 nodes. Adding more than one hidden layer seriously decreased the accuracy of our model.
5. **Random state**, see previous sections. In addition, this

After defining the classifier, the process is the same as for the SVM.

7.3 MLP metrics



Listing 3: Metrics

```
Results:
accuracy = 0.95
precision = 1.0
recall = 0.9824561403508771
F1 score = 0.9911504424778761
```

8 Conclusion

All three classifiers provided good performances, with always *accuracy* $\geq$ 95%. Moreover, each confusion matrix showed the worst performances in distinguing between classes 'two' and 'three', probably because they presented similar MFCCs. In conclusion, each method proved to be a solid solution to our problem, but SVM provided slightly better results.