SPOKEN DIGIT CLASSIFICATION

Using machine learning

HW 1 _ Assignment nr.2 _ group 13

1. OBJECTIVE AND DATA PROVIDED

1.1 OBJECTIVE OF THE PROJECT

Implement a classifier able to predict which audio is pronounced in a short audio excerpt.

1.2 TOOLS

- Python programming language,
- Jupyter Notebook (web-based environment that allows a more methodic representation of the code and its results)
- Python libraries: Numpy, Pandas, Librosa, IPython, OS, Scikit Learn, Matplotlib

1.3 DATASET

The following table describes the characteristics of the dataset:

Spoken digits	from 0 to 9
Speakers	6 different male speakers
Language	English
Repetitions	each speaker repeats each digit 50 times
Total number of files	3000
File format	mono WAV
Native sample rate	16 bits, 8 kHz
Filename format	{digit label} _ {speaker name} _ {index}.wav

The dataset is available on GitHub at the following link: https://github.com/Jakobovski/free-spoken-digit-dataset .

2. PRELIMINARY CONSIDERATIONS

1) LABELS

- Given the file name format, we are provided with the labels for our data. It is simply a matter of extracting the labels from the file name and store them in a data structure.

2) CLASSIFICATION PROBLEM

- Considering the number of digits, the problem falls into the category of *"multiclass classification problem"*: in particular we are dealing with 10 different classes and to each audio sample must correspond one single label. In order to solve these kind of problems, different classification algorithms, such as Support Vector Machine, K-Nearest Neighbors and Random Forest (among many), could be used. A method for selecting which classification algorithm works best with our dataset is to perform a cross-validation to check which one offers the best accuracy performance.

- We are provided with one single folder of files from which we should create a training set and a test set. The author of the dataset explains a possible way on how the data could be split into test and training but we decided to approach the problem in a different way.

2) PREPROCESSING

- The audio files are homogeneous in quality and they were all trimmed to eliminate silence. We don't need to preprocess them.

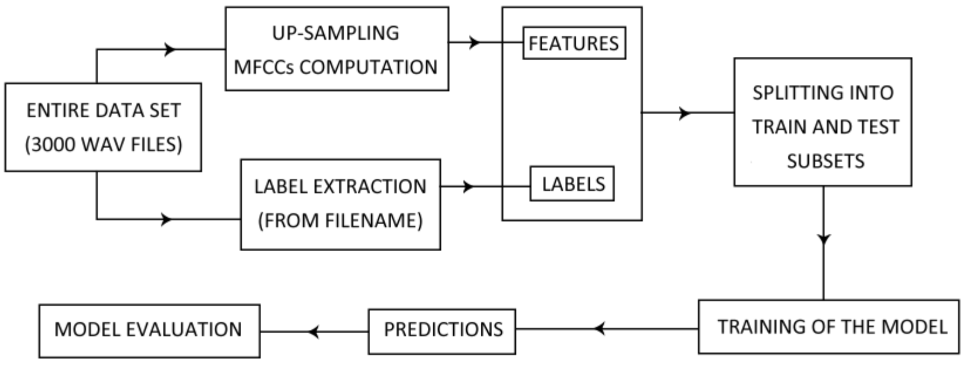
- The audio files were recorded at a low sampling rate of 8 kHz. A correct way to proceed would be to verify if up-sampling to 16 kHz leads to better results or not.

4) SIZE OF THE FILES

- The audio files duration varies and therefore also the number of samples in each file will be different. This leads to a requirement of different number of MFCCs computed for each audio file, but, in order to process all data at the same time, we need theam all to have the same number of coefficients.

- The files are small enough to be loaded in one single step each.

3. PIPELINE


ENTIRE DATA SET (3000 WAV FILES)
UP-SAMPLING MFCCs COMPUTATION
LABEL EXTRACTION (FROM FILENAME)
FEATURES
LABELS
SPLITTING INTO TRAIN AND TEST SUBSETS
TRAINING OF THE MODEL
PREDICTIONS
MODEL EVALUATION

4. IMPLEMENTATION part 1 - FEATURE EXTRACTION

4.1 DATA SET PREPARATION

1. We start by installing, through the `pip` package, the subversion control system (`svn`), which is used to manage files and directories. Subversion's function `checkout` allows us to create a copy of the GitHub repository and its content and store it in a local repository.

2. Having chosen MFCCs as audio features, we define the number of MFCC to compute from each audio file. We set this integer variable `n_mfcc` equal to 20, which is defined as an optimal value for speech analysis.

   Why 20 MFCCs?
   We noticed that computing ***20 MFCCs*** *increases noticeably the performances with respect to computing 13 MFCCs (which is a "classic" value). We also tried to compute 30 MFCCs, but the performance gain was not so significant compared to the computational cost, from which we conclude that a higher number of MFCCs would only increase the complexity of the model, while the accuracy would show little to zero variation.*

3. To improve the resolution of the audio files, we set the up-sampling frequency to 16 kHz which is exactly the double of 8 kHz, the original sampling rate.

   Why 16 kHz?
   The Nyquist Theory about up-sampling describes how, to achieve a better resolution while keeping the original informations that the audio conveys and avoid aliasing, the new sampling rate should be at least twice the value of the original bandwidth.

4. Using a for-loop that searches over the file names and directories list used to construct the path, through the functions `Path` and `glob`, which recursively retrieve file names matching a specific pattern that is fed to the function as input, we create a list of elements, called `audiofiles`, that contains the strings form of the paths of all the audio-files of the dataset.

4.2 LABELS AND FEATURE EXTRACTION

We can now proceed to loading the data and extracting from it both the labels and the MFCCs. In order to do so we implement a for-loop that iterates over each audio file, using the `elements` list we created. To store the labels and MFCC we create in advance a dictionary with two empty classes: `mfcc` and `labels`.

During **each iteration** of the for loop:

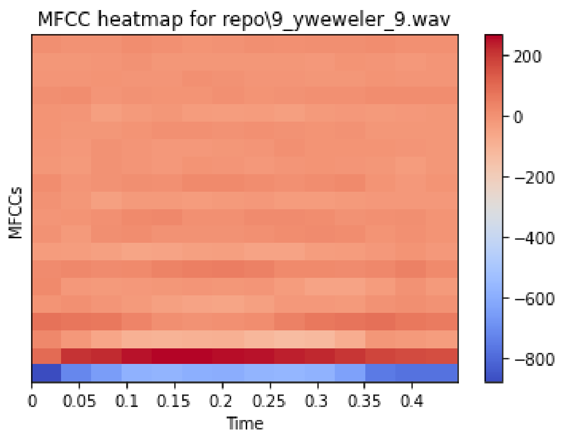
1. The audio label is extracted from the file name using the `split` function and considering "_" as separator. This function returns a list of three elements: the label, the speaker's name and the index. We select only the first element from the list, which is the only annotation we are interested in, and we store it in the "labels" class of the data dictionary.

2. Using the `load` function from Librosa, we load the audio file by passing to the function the complete path to the file. We did not modify the default parameters for this function except for the sampling rate (see 4.1).

   What does load return?
   The load function returns a tuple containing an array of floating-point numbers and the sampling rate. The array will contain (sample rate x time duration) values for each audio file where the sampling rate chosen for the case is fixed at 16 kHz.

3. Through another Librosa function, **`feature.mfcc`** we can now extract the MFCCs coefficients from the audio file. The parameters used are the following:

Keyword	Parameter	Value	Comments
`sr`	Sampling rate	16000 Hz	our choice
`n_fft`	Length of window	512 samples	recommended value for speech processing
`window`	Type of window	Hanning	by default
`hop_length`	Hop length	n_fft//4	automatically computed
`n_mels`	Number of mel filter banks	128	by default
`n_mfcc`	Number of MFCCs to compute	20	(see 4.1)



Here we include an example of the outputs obtained with the mfcc function on one of the audio files. It is visible, on the y-axis how 20 MFCCs were computed for each window. Here becomes clear how it would be useful to take the average of each MFCC coefficient over the y- axis, so that from each audio file of different lengths we can extract the same number of coefficients, precisely 20. This is done in the next passage.

4. After computing MFCCs extraction for an audio file, we create a vector of dimensions (20,) to group the average values of MFCCs along the y-axis. In the following excerpt is shown how we computed the average:

```
mfcc = librosa.feature.mfcc(audio, sr=freq, n_fft =512, n_mfcc = n_mfcc)
feature_vector =np.mean(mfcc, axis = 1)
```

5. We append the vector we created in step 4 in the dictionary class `mfcc`.

4.3 FEATURES MATRIX AND TARGET VECTOR

Once the for-loop has finished, the dictionary classes are full. The `mfcc` class is made of a list of 3000 column vectors, that we can convert into a 2D matrix of dimensions (3000,20) with the Numpy function `asarray`. The same process is applied to the `labels` class, which we convert into a vector of dimensions (3000, 1) with the same function.

5. CROSS VALIDATION – Choosing the algorithm

At this point, having available the matrix containing the data and the vector of the target labels, we can proceed with the evaluation of the classification algorithm that best suits the case. In order to do so, we perform a cross-validation, a technique that allows to compute the accuracy of classification algorithms independently from the training and test subsets, that get randomly created each time the classification program is run.

Here we include the results obtained by performing a K-Fold Cross Validation using the Scikit Learn function `cross_val_score` to which we pass (for each algorithm):

- the estimator, which is the algorithm function itself,
- the data to fit and the target variables,
- the name of the scorer we want to use, which we choose to be the accuracy,
- the CV number, which specifies in how many folds we want to split train/test sets. We choose 10.

We take the mean value of the 10 scores we get for each algorithm, to get an overall score.

algorithm (default parameters)	accuracy
Support Vector Machine	0.524
K Nearest Neighbors	0.914
Random Forest	0.871

6. IMPLEMENTATION part 2 – CLASSIFICATION

6.1 TRAIN AND TEST SUBSETS

To divide the data into train and test subsets, we use a Scikit Learn function, `train_test_split`, to which we pass the features matrix, the labels vector, a random state value, a `test_size` value and a `stratify` value. The `test_size` value represents the proportion used to split the original dataset into train and test sets and is set at 0.2. The `stratify` value consents to perform a splitting that preserves the same proportion of examples for each digit as observed in the original dataset: in order to obtain that, this value is set to work on `y`. In this code excerpt we show how we call the function:

```
y = LABELS_vector   # vector containing the labels
X = MFCC_matrix     # matrix containing the features
X_train, X_test, y_train, y_test = train_test_split(X,y, random_state=1,
test_size=0.2, stratify=y)
```

6.2 FINE TUNING OF THE HYPERPARAMETERS

The hyperparameters define the configuration of the model and, in second instance, the quality of the classification model. They can be defined by the user, but cannot be learnt from data directly. A primary selection of presumably valuable hyperparameters is usually done during the definition of the algorithm. To choose the best ones for the classification problem at hand, one should tune the hyperparameters considering the evaluation results. Luckily, Scikit Learn offers a function that, defined a grid of hyperparameters ranges, performs a K-Fold Cross Validation, picking for each fold a random combination of values from the grid.

The random forest algorithm has a large number of parameters that can be defined by the user. We studied the following (in specified ranges) and got these results:

parameter	purpose/meaning	best value
`n_estimators`	number of trees in the forest	1200
`min_samples_split`	minimum number of samples to be at a split node	2
`min_samples_leaf`	minimum number of samples to be at a leaf node	1

`max_features`	how many features a single tree should handle	auto (no restriction)
`max_depth`	limit to what depth the trees should grow	60 levels
`bootstrap`	re-using samples in a single tree	true (allowed)

6.3 FITTING OF THE RANDOM FOREST

Having found the best set of hyperparameters for our classification task, we can fit the model with the training data and labels by using the function fit. The algorithm will build a forest using the provided set. We will get a forest made of 1200 trees, each individually able to compute a prediction over the class to which the test sample belongs.

7. RESULTS REPORTING

7.1 PREDICTION OF THE VALUES

The function predict allows us, passing to it the test set (X_test), to obtain an array of predicted classes. Each tree in the forest predicts a class for each sample with a certain probability. The votes (the predictions) given by the trees, are evaluated weighing the probabilities related to each vote.

We can provide an example of how the prediction went for one test sample (`y_test[0]`):

```
The label for this sample in y_test is 8

The random forest predicted for it 8

Each class got these probabilities: [0.015    0.00166667    0.0025
0.08833333  0.00166667  0.00916667  0.07833333  0.00666667  0.7825
0.01416667]
```

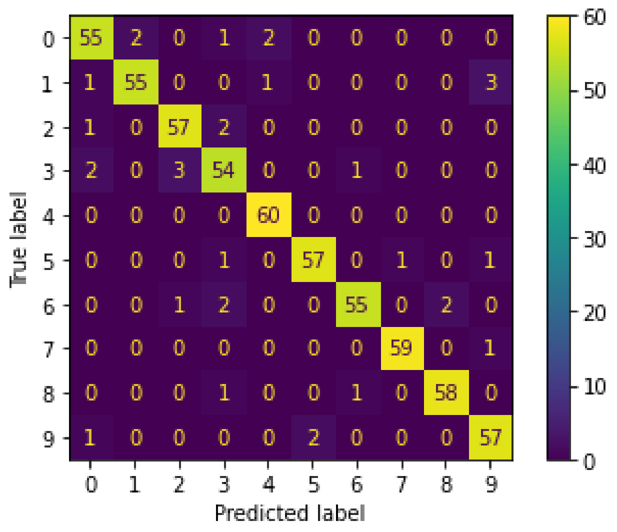
It is clear how the highest probability is related to the 9th class which corresponds to the digit 8.

7.2 CONFUSION MATRIX

The confusion matrix is a classic tool useful to represent the performance of classification models. Its dimension are NxN where N is the number of classes, so in this case it's a 10x10 matrix. The matrix compares the actual target values and the values predicted by the model.

Each element on the diagonal of the matrix represents the number of right predictions for its digit, while the other values represent the wrong prediction: an element outside of the diagonal represents the wrong prediction of its digit (Predicted label, horizontal axis) against its actual label (True label, vertical axis). The color map gives a visual hint on how high are the correct prediction values (green or yellow) and how low are instead the wrong ones (blue). The confusion matrix underlines a good behavior of the classification algorithm, which validates the quality of the built prediction model. Having chosen to split train and test subsets as explained in (6.1), it's easy to evaluate the performance of the model given the fact that 60 correct samples is the maximum score achievable by every class.

CONFUSION MATRIX _ RANDOM FOREST:



7.3 CLASSIFICATION REPORT

Just to improve the consciousness of the quality of the model's predictions, we added the Classification Report method, which gives us another point of view over the well-functioning of the algorithm. The classification report is divided in precision, recall, f1-score and support: report represents the classifier's exactness (True Positive / True Positive + False Positive), the recall is a measure of the classifier's completeness (True Positive / True Positive + False Negative) and f1-score is the weighted harmonic mean of the first two columns, while support represents the number of actual occurrences of the class in the specified dataset. This classification report in its components describes a balanced classification method , with high values in precision and recall and a high accuracy value (0.945).

Both the classification report and the Confusion Matrix indicate that the model built is able to predict a fair amount of labels without being mistaken, it is also fast and well balanced considering the referenced dataset.

```
CLASSIFICATION REPORT _ RANDOM FOREST:

              precision    recall  f1-score   support

           0       0.92      0.92      0.92        60
           1       0.96      0.92      0.94        60
           2       0.93      0.95      0.94        60
           3       0.89      0.90      0.89        60
           4       0.95      1.00      0.98        60
           5       0.97      0.95      0.96        60
           6       0.96      0.92      0.94        60
           7       0.98      0.98      0.98        60
           8       0.97      0.97      0.97        60
           9       0.92      0.95      0.93        60

    accuracy                           0.94       600
   macro avg       0.95      0.94      0.94       600
weighted avg       0.95      0.94      0.94       600

Accuracy:
 0.945
```

8. CONCLUSIONS

The model we implemented returned great results in accuracy. A number of factors may have contributed:

1. **Random forest as a powerful algorithm**: in general, the random forest works very well on large databases, even when there's some portions of data missing. It gets less biased than other classification models given the large number of decision trees working independently.

2. **High quality dataset**: the random forest we implemented worked on a homogeneous and well organized data set. The files were created ad hoc following specific guidelines.

3. **Proper sample rate**: to analyze the distinctive properties of speech, 8 kHz is sufficient and also a good choice. Consonants and vowels can be analyzed with ease and there is no distracting contribution coming from higher frequencies of no importance for our task.