# CMLS Homework 1

Group 2: 10804339,10796569,10528650,10622373

April 25, 2021

*https://github.com/FrancescoBorgna/CMLS_HW1_VoiceSpokenDigits*

## 1 Data Collection

The aim of this homework is to implement a classifier able to predict which digit is pronounced in a short audio excerpt. To achieve this, we have collected our data from the folders of the original dataset in the vector *train root* :

```
! git clone https://github.com/Jakobovski/free-spoken-digit-dataset.git\\
train_root = ('free-spoken-digit-dataset/recordings')\\
```

The dataset consists of 3000 recordings of spoken digits in wav files at 8kHz.
The recordings are trimmed so that they have near minimal silence at the beginnings and ends.

Furthermore files are named in the following format:

```
{digitLabel}_{speakerName}_{index}.wav      Example: 7_jackson_32.wav
```

## 2 Feature selection

In the second place we have thought which set of features can be useful for our task and that can guarantee best classification results. We came to the conclusion that we can use as main feature the mel-frequency cepstrum coefficients. In fact MFCCs can be used as an excellent feature vector for representing the human voice and musical signals. In particular we know that the MFCCs parametrization of speech has been proven to be beneficial for speech recognition. The MFCCs are based on a subband filtering by means of a series of triangular filters whose center frequencies are equally spaced according to the mel scale.
In order to compute the filters we have defined the function that takes in input the audio array, the frequency rate and the number of MFCCs:

```
def compute_mfcc(audio, fs, n_mfcc)
```

In the code we computed the spectrogram of the audio signal, this by taking the absolute value of the Short Time Fourier Transform. The setting of the lengh *n fft* is of 512 samples because we did not need a high resolution for an audio file like those in the dataset would be sufficient. After that, we have calculated the weights of the mel filters with the *filters.mel* function of librosa and applied them to spectrogram:

```
melspectrogram = np.dot(mel, X)
```

We took the logarithm:

```
log_melspectrogram = np.log10(melspectrogram + 1e-16)
```

At the end we have computed the DCT to log melspectrogram to obtain the coefficients:

```
mfcc = sp.fftpack.dct(log_melspectrogram, axis = 0, norm='ortho')[1:n_mfcc+ 1]
```

# 3 Prepare data

Now we must organize and prepare our data by splitting our dataset in training set and testing set. The test set officially consists of the first 10% of the files.
Recordings numbered 0-4 (inclusive) are in the test set and 5-49 are in the training set. In this way we can build a dataset that is balanced and has an equal number of samples for each class. Furthermore, training set and testing set are as different as possible, which is a necessary condition in order to have a more efficient classifier.

Taking these considerations into account, we have defined our classes, one for each digit, and the number of MFCCs:

```
classes = [0,1,2,3,4,5,6,7,8,9]
n_mfcc = 13
```

We took 13 as the number of coefficients. This will guarantee us excellent accuracy in predicting the digits spoken by the various speakers. An interesting consideration that can be made is the following: if we use a smaller number of coefficients the precision of our code decreases as I take into account a smaller number of frequency bands. On the contrary, if I increase the number of coefficients, the precision of our code increases for the opposite reason. The cons of this second consideration will be that, for a large number of registrations and mel coefficients, the computational time of the code will be higher. We therefore felt that a standard number (13) was sufficient for our purposes.

Later we defined an empty dictionary, where each subclass is the digit:

```
dict_train_features = {0: [], 1: [], 2: [],3: [],4: [],5: [],6: [],7: [],8: [],9:[]}
dict_test_features = {0: [], 1: [], 2: [],3: [],4: [],5: [],6: [],7: [],8: [],9:[]}
```

We have allocated in the

```
 class_train_files
```

variable all the train root files.
After that, for every file contained in *class train files* we have splitted the string of the name of each file in a vector containing 3 strings: digit, speaker, number of recording of the speaker. We computed this by using a temporal variable tmp that will be rewritten at every iteration. With *librosa* at every iteration we obtained the audio array of the current *.waw* file at the original frequency. We calculated the MFCC's and compute the mean value for every frequency band along the axix=1 (columns). At the end, as said before, if the recording is numbered from 5 to 49 we have put the n-th audio in the *dict train*, by placing it in the correct digit class of our dictionary. Otherwise, if the recording is numbered from 0 to 4 we have put it in the dict test features.

# 4 Support Vector Machines

SVMs are commonly used algorithms in supervised machine learning. In essence, given a training set, SVMs identify a separating hyper plane between clusters of elements of two classes. Such plane is then used with a test set to make a prediction on whether the given sample belongs to one class or the other.

SVMs differ from perceptron algorythms in that they are designed to find the hyperplane which maximizes the discriminability between to classes.

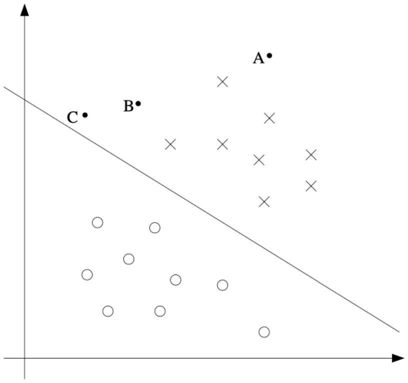


Figure 1: SVM

# 5 RBF Kernel

The kernel used for this implementation is the radial basis function kernel, which is useful for the classification of non linear data sets; the distribution of the RBF kernel is very similar to the Gaussian distribution.

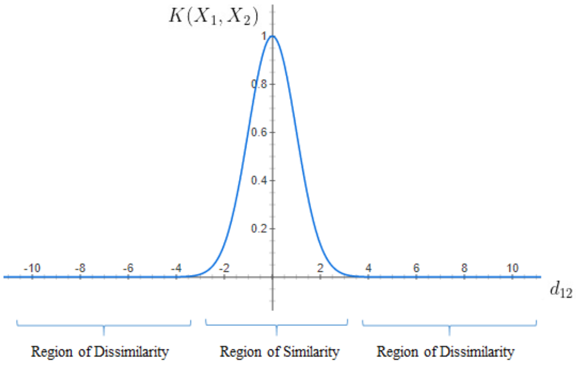


Figure 2: rbf

# 6 Implementation

After the MFCCs are computed for each audio excerpt, the training set as well as the test data is normalized; the maximum and minimum values of each data set are found and a simple normalization algorithm is applied.

```
  X_train_normalized[i] = (X_train[i] - feat_min) / (feat_max - feat_min);
```

In this implementation, SVMs are computed for each of the 45 combinations of digits, one against another (0 vs 1, 0 vs 2, etc).

In other words, we are interested in evaluating whether a test sample is more similar to 0 or 1, to 0 or 2, to 1 or 2, etc. Once all of the combinations are tested, majority voting will be applied, as described in the next paragraph.

A cycle that iterates over the combinations above, and the SVM is computed by means of the sklearn.svm library. The function returns a binary value corresponding to either class.

```
for i in np.arange(Len_comb):
  clf_vec = np.append(clf_vec, sklearn.svm.SVC(**SVM_parameters, probability=True));
```

# 7 Fit

In this section we are going to fit our model. Let's start by just analyzing the case of a binary classifier. To fit a model means to *train* it such that,provided some training input belonging to two different classes, it will be able to distinguish between the two classes based on their probability distribution. For doing so, we took advantage of the **fit** function provided by the library **sklearn**, which, given the values of the training set in form of a vector with the corresponding class of belonging (??), returns us the model for our data. Since our input training data are in separate vectors, we need to concatenate them in a unique one in order to have the funciton working.

In the case of multiple classes, we can build our model by creating a binary classifier for each pair of different classes, so that in our case, having 10 different classes corresponding to the digits from 0 up to 9, we will have $\mathrm{N}_{\mathrm{possibleComb}}$ different binary calssifiers, where $N_{\mathrm{possibleComb}} = \frac{1}{2}N_{classes}(N_{classes} - 1) = 45$.

[utf8]inputenc natbib graphicx multicol float

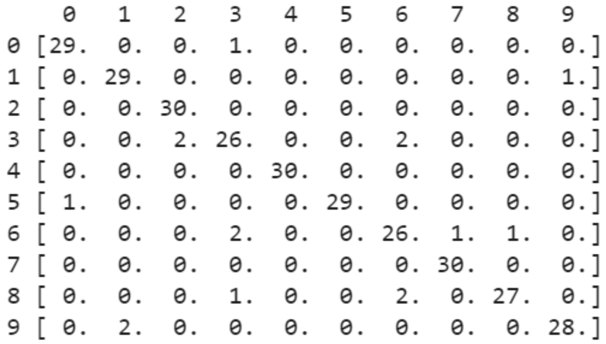
# 8 Predict and majority voting

In the predict section we prepare the test data and then we predict each digit in a binary OneVsOne fashion. Namely, there is a binary SVM for each combination of 2 digits (ex. 1 SVM predicts if an audio file is a '0' or '1', another if it's a '0' or '2' and so on...) and every test file goes through them.

Now, having 45 predictions for each file, we have to select which digit is the most guessed to have a single prediction and we do that in the majority voting section.

# 9 Confusion matrix

We put our predicted values into a matrix so we can evaluate how the classifier worked. Each row's index indicates the true class of the files, while each column the guessed class.

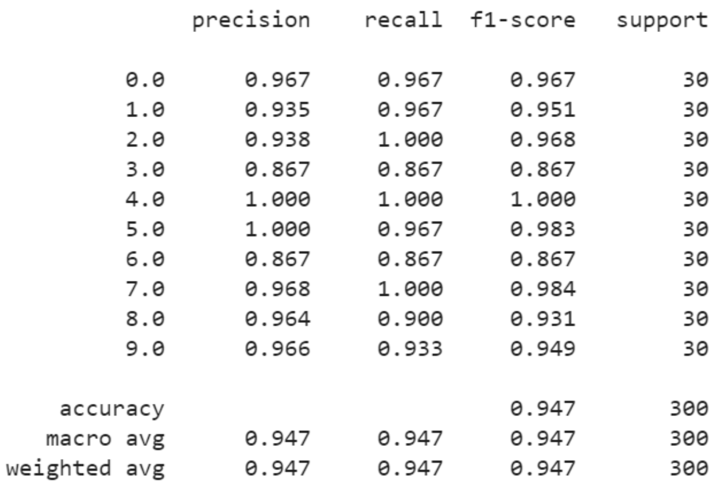


```
     0   1   2   3   4   5   6   7   8   9
0 [29.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
1 [ 0. 29.  0.  0.  0.  0.  0.  0.  0.  1.]
2 [ 0.  0. 30.  0.  0.  0.  0.  0.  0.  0.]
3 [ 0.  0.  2. 26.  0.  0.  2.  0.  0.  0.]
4 [ 0.  0.  0.  0. 30.  0.  0.  0.  0.  0.]
5 [ 1.  0.  0.  0.  0. 29.  0.  0.  0.  0.]
6 [ 0.  0.  0.  2.  0.  0. 26.  1.  1.  0.]
7 [ 0.  0.  0.  0.  0.  0.  0. 30.  0.  0.]
8 [ 0.  0.  0.  1.  0.  0.  2.  0. 27.  0.]
9 [ 0.  2.  0.  0.  0.  0.  0.  0.  0. 28.]
```

Figure 3: OneVsOne matrix

Looking at the results we can see that the classifier worked well: almost all the digits were correctly recognised. Only some digits like '3' and '6' had the most trouble being recognised, maybe sharing letters with other digits or just some frequencies (even with each other).

Metrics: computing precision, recall and f1-score we can corroborate our observation about having precise results, with a total of 94,7% of accuracy.



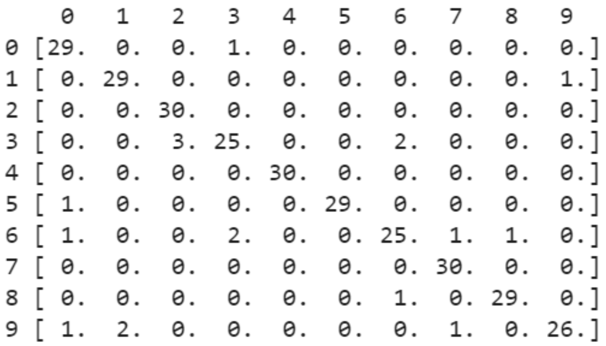
```
              precision    recall  f1-score   support

         0.0      0.967     0.967     0.967        30
         1.0      0.935     0.967     0.951        30
         2.0      0.938     1.000     0.968        30
         3.0      0.867     0.867     0.867        30
         4.0      1.000     1.000     1.000        30
         5.0      1.000     0.967     0.983        30
         6.0      0.867     0.867     0.867        30
         7.0      0.968     1.000     0.984        30
         8.0      0.964     0.900     0.931        30
         9.0      0.966     0.933     0.949        30

    accuracy                          0.947       300
   macro avg      0.947     0.947     0.947       300
weighted avg      0.947     0.947     0.947       300
```

Figure 4: OneVsOne metrics

## 10 OneVsRest SVM

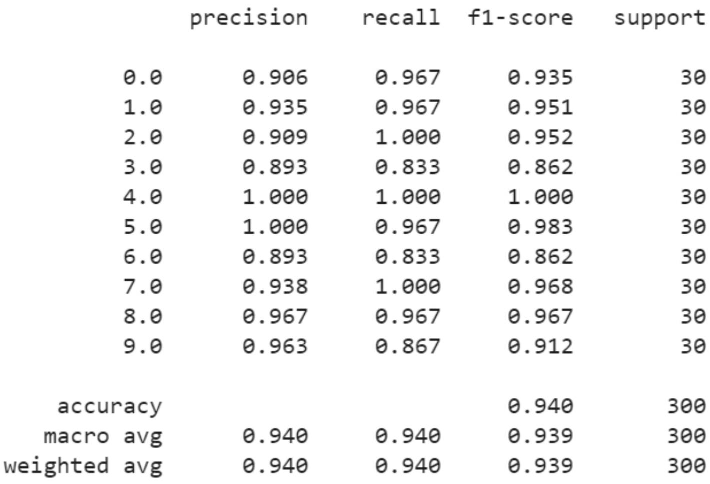
To have more than one only model of classification we tried to see how the results changed using binary SVMs in a OneVsRest fashion. Namely, this time there is a binary SVM for each class that checks if the input data belongs to its class or to one of the other. This approach requires that each model predicts a class membership with a probability score. The maximum of these scores is then used to predict a class, so that a majority voting algorithm is not required.

Therefore in the OneVsRest SVM section we fit, test the model and then compute confusion matrix and metrics for this classifier.



```
    0   1   2   3   4   5   6   7   8   9
0 [29.  0.  0.  1.  0.  0.  0.  0.  0.  0.]
1 [ 0. 29.  0.  0.  0.  0.  0.  0.  0.  1.]
2 [ 0.  0. 30.  0.  0.  0.  0.  0.  0.  0.]
3 [ 0.  0.  3. 25.  0.  0.  2.  0.  0.  0.]
4 [ 0.  0.  0.  0. 30.  0.  0.  0.  0.  0.]
5 [ 1.  0.  0.  0.  0. 29.  0.  0.  0.  0.]
6 [ 1.  0.  0.  2.  0.  0. 25.  1.  1.  0.]
7 [ 0.  0.  0.  0.  0.  0.  0. 30.  0.  0.]
8 [ 0.  0.  0.  0.  0.  0.  1.  0. 29.  0.]
9 [ 1.  2.  0.  0.  0.  0.  0.  1.  0. 26.]
```

Figure 5: OneVsRest matrix



```
              precision    recall  f1-score   support

         0.0      0.906     0.967     0.935        30
         1.0      0.935     0.967     0.951        30
         2.0      0.909     1.000     0.952        30
         3.0      0.893     0.833     0.862        30
         4.0      1.000     1.000     1.000        30
         5.0      1.000     0.967     0.983        30
         6.0      0.893     0.833     0.862        30
         7.0      0.938     1.000     0.968        30
         8.0      0.967     0.967     0.967        30
         9.0      0.963     0.867     0.912        30

    accuracy                          0.940       300
   macro avg      0.940     0.940     0.939       300
weighted avg      0.940     0.940     0.939       300
```

Figure 6: OneVsRest metrics

## 11 Comparison

The computation time results below show a total of:

- 4500 * 1,19ms (mean value) = 5355ms, approximately 5s for 1vs1
- 10* 39ms = 390ms for 1vsRest

We can see that this methodology is slightly less accurate and precise than the OneVsOne approach, but computation time is much faster and accuracy is still very high at 94%. In fact having just a few classes and only one SVM per class (10 SVMs), instead of 45, it is an easily understandable behaviour.

The advantage of this approach is clearly computation speed, so it may be more suitable for larger data sets, while for smaller data sets or a need for more accurate predictions the 1vs1 may be the best solution.

## 12 Computation time results

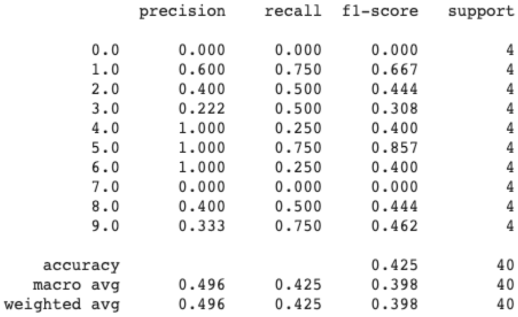
```
Execution time 1vs1 for each SVM:
1000 loops, best of 5: 1.19 ms per loop
1000 loops, best of 5: 1.89 ms per loop
1000 loops, best of 5: 1.9 ms per loop
1000 loops, best of 5: 1.08 ms per loop
1000 loops, best of 5: 1.08 ms per loop
1000 loops, best of 5: 1.23 ms per loop
1000 loops, best of 5: 977 µs per loop
1000 loops, best of 5: 925 µs per loop
1000 loops, best of 5: 1.33 ms per loop
1000 loops, best of 5: 1.03 ms per loop
1000 loops, best of 5: 954 µs per loop
1000 loops, best of 5: 1.32 ms per loop
1000 loops, best of 5: 1.54 ms per loop
1000 loops, best of 5: 918 µs per loop
1000 loops, best of 5: 1.24 ms per loop
1000 loops, best of 5: 783 µs per loop
100 loops, best of 5: 2.37 ms per loop
100 loops, best of 5: 2.37 ms per loop
1000 loops, best of 5: 799 µs per loop
1000 loops, best of 5: 849 µs per loop
1000 loops, best of 5: 1.24 ms per loop
1000 loops, best of 5: 853 µs per loop
1000 loops, best of 5: 851 µs per loop
1000 loops, best of 5: 985 µs per loop
1000 loops, best of 5: 845 µs per loop
1000 loops, best of 5: 944 µs per loop
100 loops, best of 5: 2.21 ms per loop
```

```
1000 loops, best of 5: 1.11 ms per loop
1000 loops, best of 5: 1.38 ms per loop
1000 loops, best of 5: 1.01 ms per loop
1000 loops, best of 5: 1.02 ms per loop
1000 loops, best of 5: 735 µs per loop
1000 loops, best of 5: 845 µs per loop
1000 loops, best of 5: 727 µs per loop
1000 loops, best of 5: 850 µs per loop
1000 loops, best of 5: 880 µs per loop
1000 loops, best of 5: 1.32 ms per loop
1000 loops, best of 5: 799 µs per loop
1000 loops, best of 5: 1.5 ms per loop
1000 loops, best of 5: 1.3 ms per loop
1000 loops, best of 5: 1.87 ms per loop
1000 loops, best of 5: 972 µs per loop
1000 loops, best of 5: 860 µs per loop
1000 loops, best of 5: 1.84 ms per loop
1000 loops, best of 5: 870 µs per loop
```

Execution time 1vsRest for all SVMs: 10 loops, best of 5: 39 ms per loop

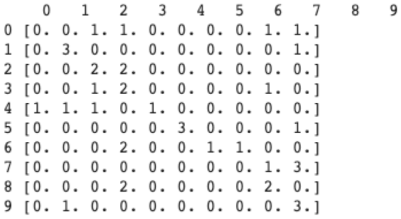
## 13 Additional Test Set

To further evaluate the performance of the classifier, the authors have recorded a small collection of audio test files (one set of digits per person). The recordings were trimmed to an appropriate length and down sampled to 8 kHz. The test files the were uploaded to a Git Hub repository and were imported similarly to what described earlier.



```
              precision    recall  f1-score   support

         0.0      0.000     0.000     0.000         4
         1.0      0.600     0.750     0.667         4
         2.0      0.400     0.500     0.444         4
         3.0      0.222     0.500     0.308         4
         4.0      1.000     0.250     0.400         4
         5.0      1.000     0.750     0.857         4
         6.0      1.000     0.250     0.400         4
         7.0      0.000     0.000     0.000         4
         8.0      0.400     0.500     0.444         4
         9.0      0.333     0.750     0.462         4

    accuracy                          0.425        40
   macro avg      0.496     0.425     0.398        40
weighted avg      0.496     0.425     0.398        40
```

Figure 7: Accuracy of the classifier for the authors' test set



```
    0   1   2   3   4   5   6   7   8   9
0 [0. 0. 1. 1. 0. 0. 0. 0. 1. 1.]
1 [0. 3. 0. 0. 0. 0. 0. 0. 0. 1.]
2 [0. 0. 2. 2. 0. 0. 0. 0. 0. 0.]
3 [0. 0. 1. 2. 0. 0. 0. 0. 1. 0.]
4 [1. 1. 1. 0. 1. 0. 0. 0. 0. 0.]
5 [0. 0. 0. 0. 0. 3. 0. 0. 0. 1.]
6 [0. 0. 0. 2. 0. 0. 1. 1. 0. 0.]
7 [0. 0. 0. 0. 0. 0. 0. 0. 1. 3.]
8 [0. 0. 0. 2. 0. 0. 0. 0. 2. 0.]
9 [0. 1. 0. 0. 0. 0. 0. 0. 0. 3.]
```

Figure 8: Confusion Matrix

We attribute the lower accuracy metric to the small size of the training set (only six subjects are included), to the difference in the methodology of recording and to the difference in accent and tone. Also, our own test set is only comprised of one sample per subject.

In order to improve the classifier, one would have to supply a larger training set, potentially including different accents, different tones, and recording modalities.