Αναγνώριση Προτύπων
2η Εργαστηριακή Άσκηση

Αναγνώριση φωνής με Κρυφά Μαρκοβιανά Μοντέλα και Αναδρομικά Νευρωνικά Δίκτυα

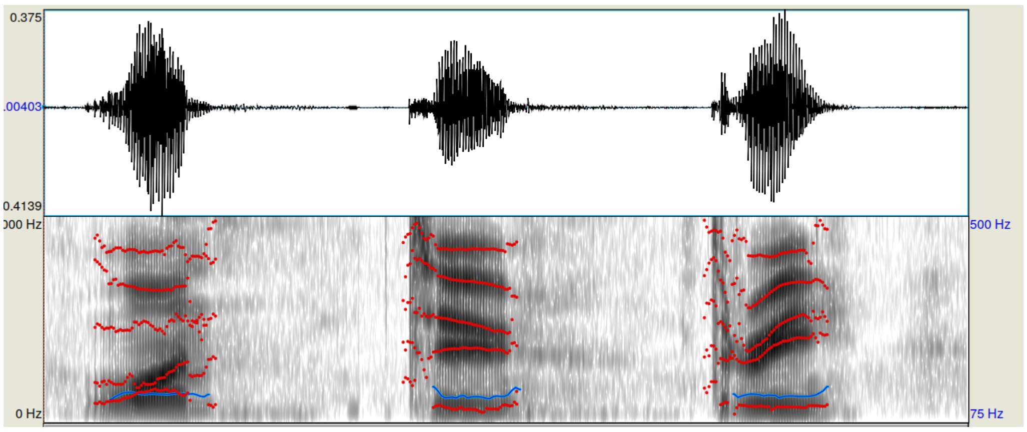
Παρέλλη Μαρία
03115155
9ο Εξάμηνο

Χαρδούβελης Γεώργιος-Ορέστης
03115100
9ο Εξάμηνο

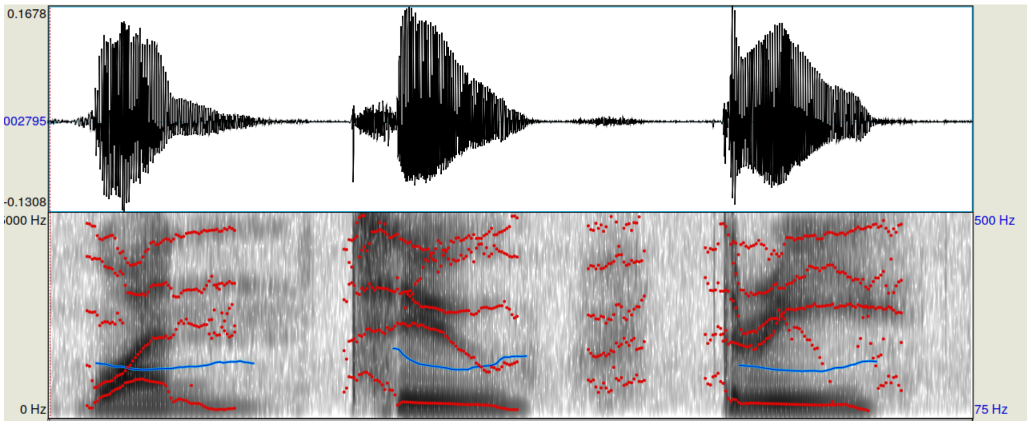
This exercise was developed using Praat and python code. The implementations in the code are mostly vectorized.

Step 1

At first, we open the files with the pronunciations of the digits one, two and three from speakers one and five. The waveforms and spectograms can be seen in the images below.



Waveform (up) and spectogram (down) of onetwothree pronunciations of 1st speaker



Waveform (up) and spectogram (down) of onetwothree pronunciations of 8th speaker

The blue lines represent the pitch whereas the red lines the formants. We can already notice that the pitch of the female speaker is greater. In addition, the range of their waveform is much smaller.

The mean of the pitch and the first three formants for each of the dominant vowels in every digit are:

1. For the 1st speaker

    - 'a': mean pitch is 134.42733386994914 Hz and the formants in consecutive order are 789.9258329626538 Hz, 1142.1050545297396 Hz, 2349.7383679800746 Hz
    - 'ou': mean pitch is 131.59196663807782 Hz and the formants in consecutive order are 429.4474336028136 Hz, 1805.5039497824487 Hz, 2438.1528683485 Hz
    - 'i': mean pitch is 132.366020778073 Hz and the formants in consecutive order are 395.68298002295734 Hz, 2050.258798967754 Hz, 2538.241125142736 Hz

2. For the 8th speaker

   - ’a’: mean pitch is 176.73171329852156 Hz and the formants in consecutive order are 879.0818077878397 Hz, 1853.9354980006233 Hz, 3156.5450161561917 Hz
   - ’ou’: mean pitch is 182.2980390763653 Hz and the formants in consecutive order are 355.2092032875961 Hz, 1806.4261845165402 Hz, 2661.465910007983 Hz
   - ’i’: mean pitch is 178.56012264571524 Hz and the formants in consecutive order are 315.92586175520273 Hz, 2153.684638493432 Hz, 2996.289513834071 Hz

As we can see, the pitch is moslty related to the speaker, whereas the formants as closer to describing the digit enunciated, even for speakers of different sex.

Step 2

Starting from this step, we utilized the Notebook provided in Google colab.
At first, we created a parser function. The reading of the files was realized with the google.colab.files.upload command.
After some editing, the function returns three python lists, one with the wav files as they where read by librosa, one with the respective speaker (1 to 15) and one with the digit pronounced in every respective wav file.

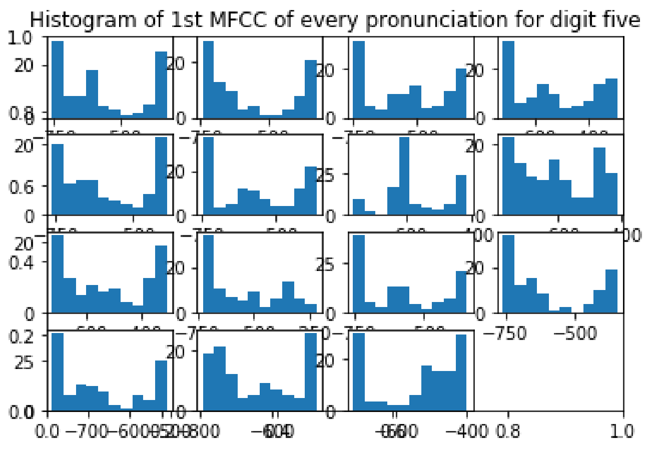
Step 3

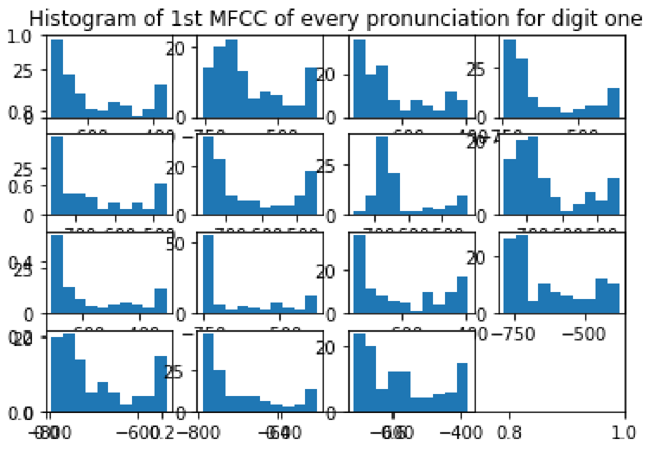
Here, we extract for each sound file the MFCC’s (Mel-Frequency Cepstral Coefficients). We only use the first 13 coefficients. From that we extract the local derivatives of first and second order (aka deltas and deltas-deltas). Thus we create three lists with the above elements. In order to achieve that, we used the feature.mfcc and feature.delta functions from the librosa library.

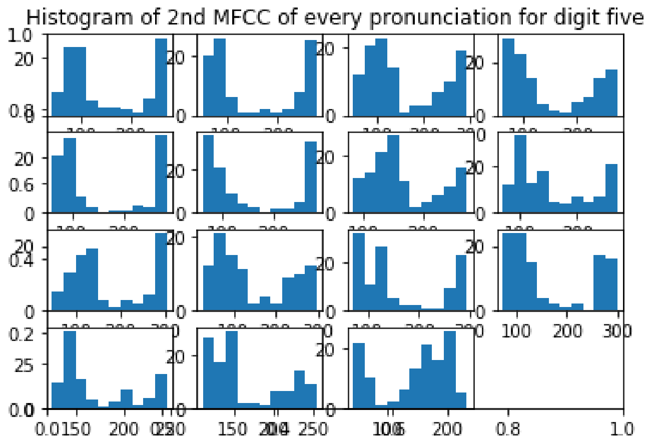
The MFCCs are very commonly used as characteristics in speech recognition and have proved to be reliable. In order to calculate them, we must assume that on short time scales the signal does not change (and so we can frame it). Then the power spectrum of each frame is calculated, which values are connected with the function of the human cochlea. Then the mel filterbank is applied and we sum the energy in each filter, in order to isolate only the necessary information. After that, we take the logarithm of the filterbank energies, an action also motivated by the human ear since we perceive loudness on a logarithmic scale. At last the DCT is computed which decorrelates the energies, and we keep the first ones because the rest represent the faster changes in the filterbank energies, which have been shown to degrade classifiction performance.

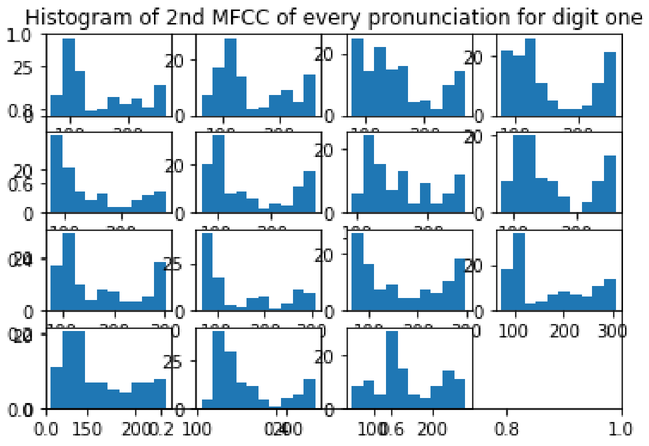
Step 4

As we have registration numbers 03115100 and 03115155, we defined n1 and n2 as ’five’ and ’one’ (third from end).
Below we can see the histograms for the first and second MFCC’s for every pronunciation for n1 and n2.



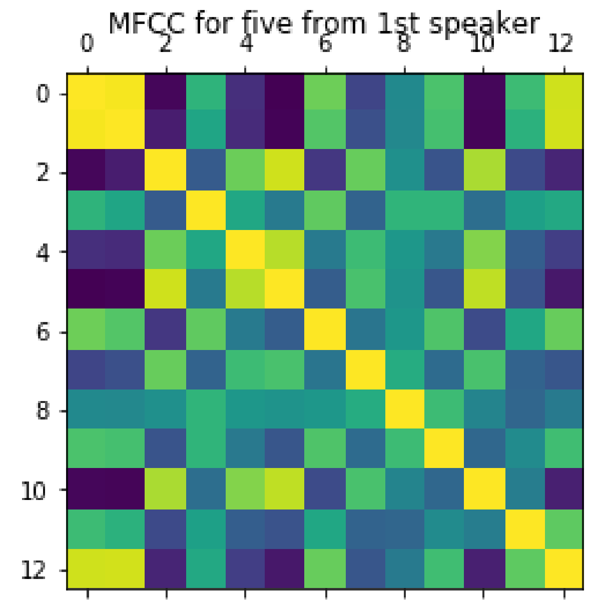


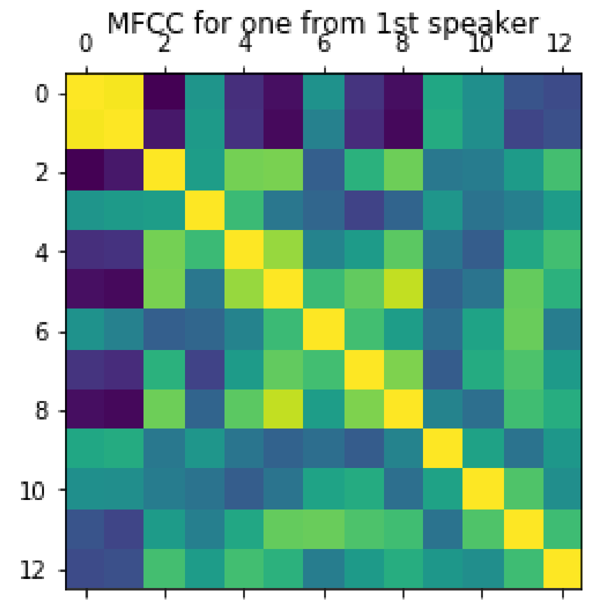


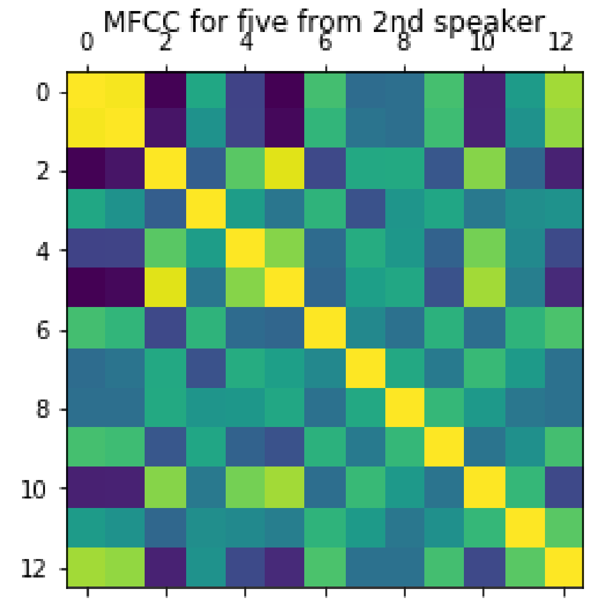


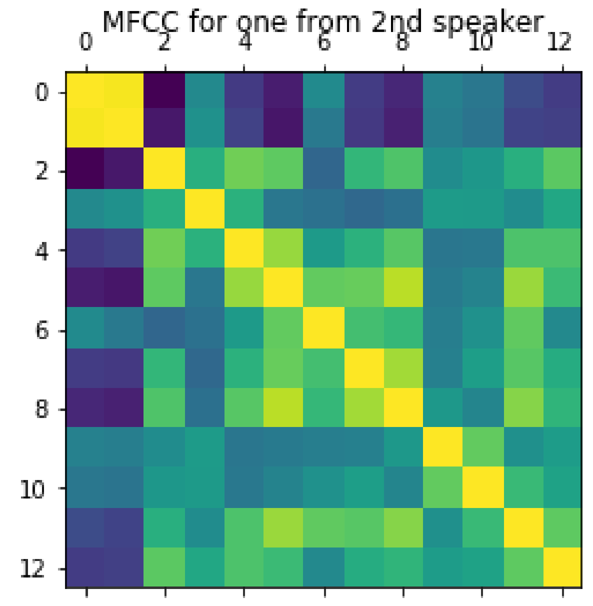
As we can see, the deviation is not always tangible, but there are many distinct differences.

In the diagrams below, we can also see the correlation matrix of the MFCC's of the first two speakers.





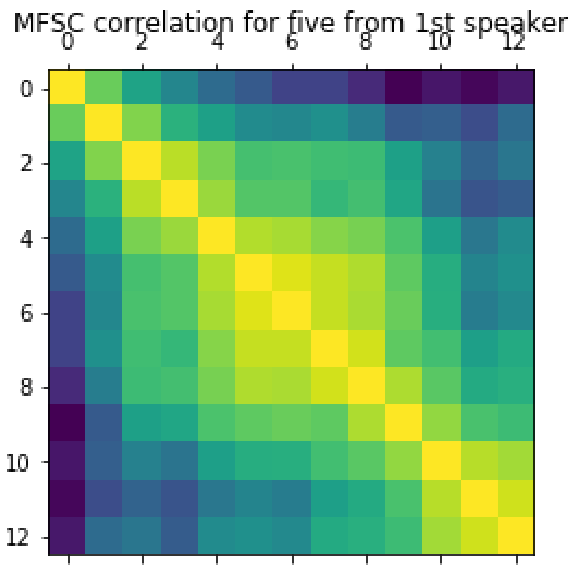


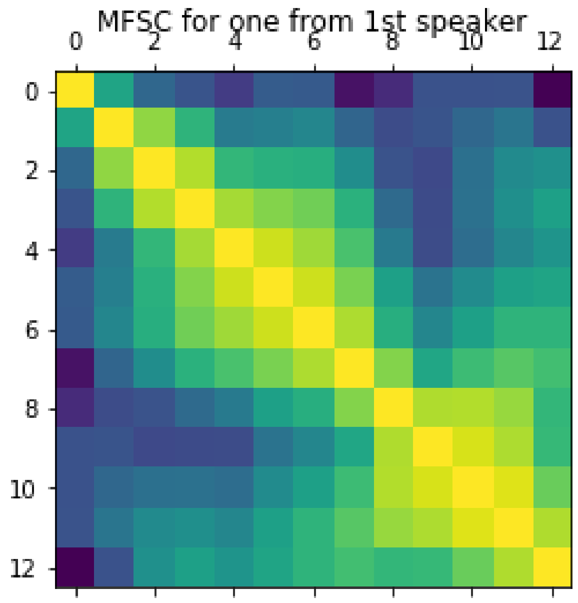


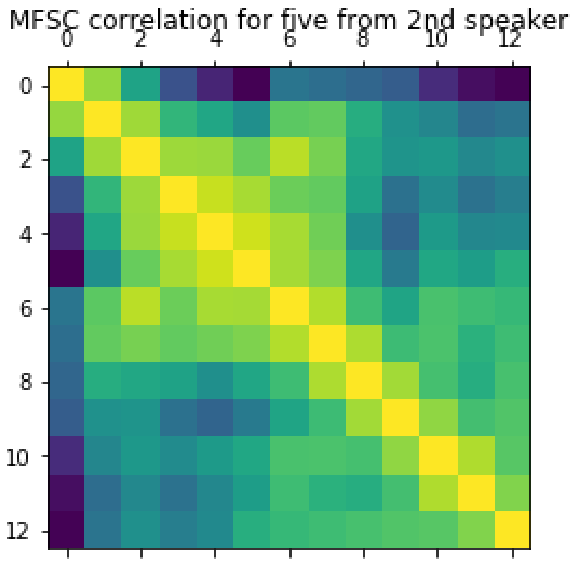
It is clear that, apart from the diagonal, the correlation between the coefficients is mostly low. To achieve that with MFCC's, a key step is the DCT (Discrete Cosine Transform) of the mel log powers, as also mentioned before.

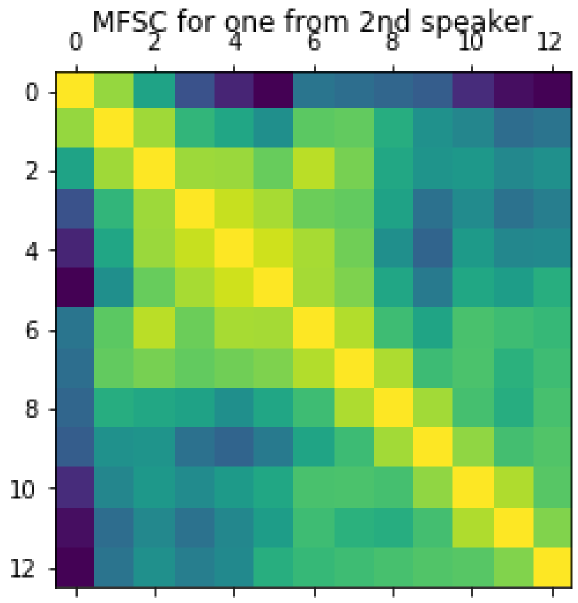
In order to prove that, we produced the correlation matrix of MFSC's (Mel-Filterbank **Spectral** Coefficients), which are produced similarly to MFCC's, but without DCT.

The MFSC's where produced with the logfbak function imported from python_speech_features. The results have as followed:





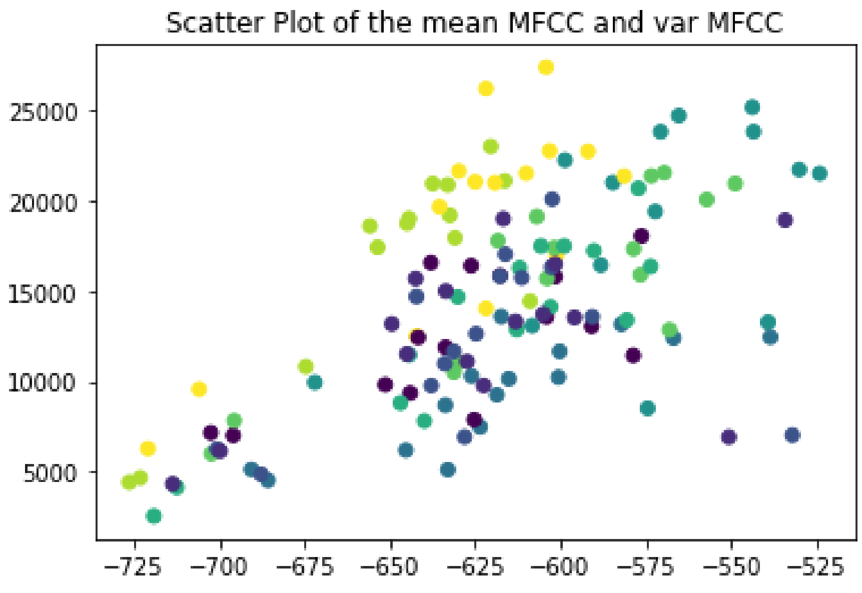




It is apparent that between many coefficients there is a high correlation observed, which is not suitable for our purpose. That was an expected results because the filterbanks are all overlapping, producing correlated results. Hence, MFCC's are preferred.

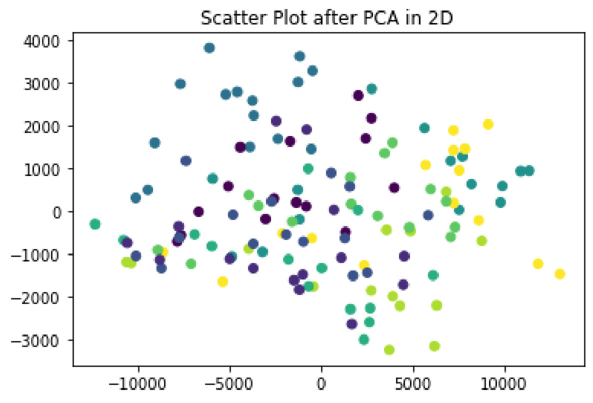
Step 5

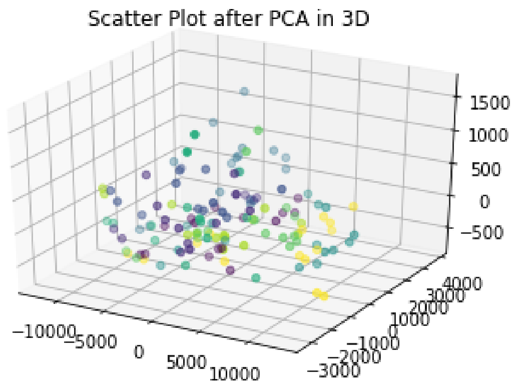
We created an array, in which, for each of the 133 sound files and each of 13 components, we have stored the means and the variances MFCC's, deltas and deltas-deltas (6*13 values for each file).
At first we created a 2D scatter plot of the first 2 dimensions of this array (the mean of the MFCC's and the variances of MFCC's) depicting the different digits pronounced, which is shown below.
As we can see, bearing in mind that the the classification of this data set is too complicated to be computed with linear ways, the separation of digits is not successful. Nevertheless, we could locate some vague clusters of digits.



Step 6

Another approach to visualise the above multi-dimensional vector is with PCA (Principal Analysis Component). In this case, using this method, we managed to reduce the dimension, computing 2 and 3 principal dimensions, making it possible to depict our data. The results our shown below.





Here, similarly as above, we are not close to a proper separation of the digits. However, the created clusters are a bit more distinct than above, especially in the 3D model.

Calculating the ratio of the variance of the first 3 principal dimensions in relation with the initial variance we get 0.93319456, for the first one, 0.06126316 for the second one and 0.00554229 for the third one.
We can easily observe that for the first principal dimension the algorithm achieves a really high proportion of the initial variance, contrary to the other two where the results are really low. This means that the latter two components don't offer much to the representation of our data, where as the first one holds most of the information.

Step 7

After dividing our data randomly in a training and testing set (with a 70%-30% ratio), we tested some classifiers.

At first, we examined the Naive Bayes classifier (both the one implemented in the last exercise and the scikit-learn one), which proved to have accuracy 0.6 and 0.675 respectively.
In addition, we tested a Gaussian Process Classifier, a Nearest Neighbors Classifier (checking 3 neighbours) and an SMV Classifier, with respective accuracies 0.125, 0.375 and 0.125.

As we can see the best results are met with the Bayes Classifiers whereas GPC and SVM produce really poor results.

We repeat the process after adding to our characteristics the mean and the variance of the zero crossing rate of our sound files. The accuracies we get then for our estimators are: 0.7 for our implementation of Naive Bayes, 0.8 for the scikit Naive Bayes, 0.15 for GPC, 0.4 for 3NN and 0.15 for SVM.

Overall, as expected, we got better results, since we added characteristics to our classifiers. Even though the ratio of the characteristics added is small, they are quite different for the ones we used so far, giving the classifiers an edge.

Step 8

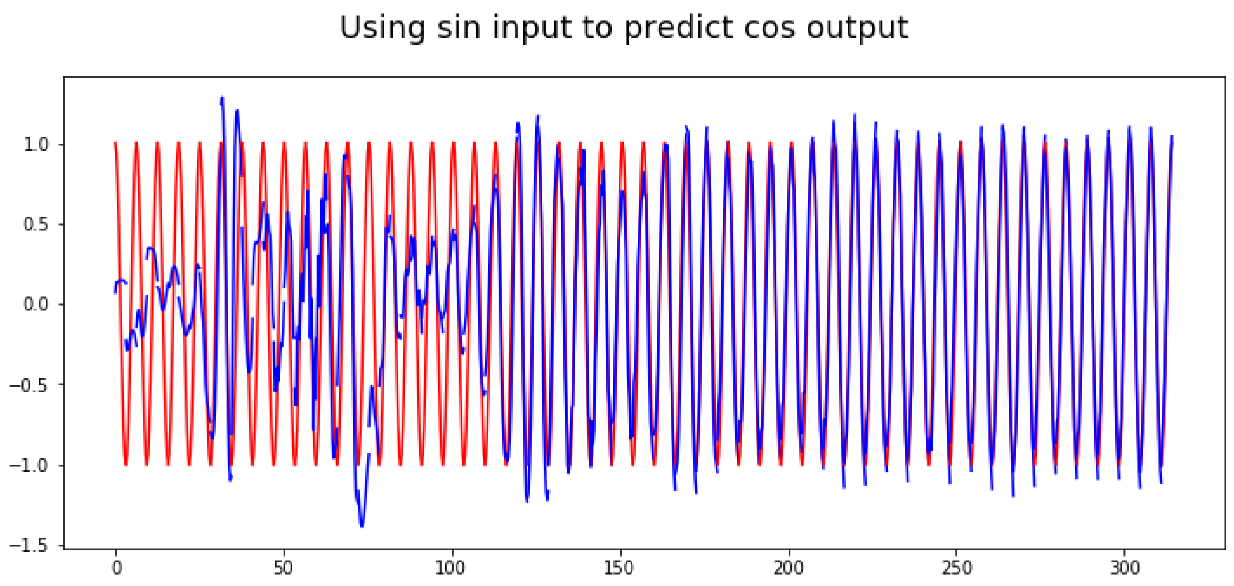
In this step, we tried to predict a cos curve having sin as an input by training a Recurrent Neural Network (RNN). Basically, it calculates output for each time step and put it into the Liner to compute predicted output. The learning rate was chosen at 0.02, and a time step of 10.

To begin with, we define the RNN by defining the forward function and the layers. It was implemented with one layer and 36 rnn hidden units was chosen.

Afterwards we implemented the training of the model. In each epoch we gradually moved to the next π time units and generate 10 equidistant points in this half period of sin. The sin of these elements are our input data and the cos the output we want to fit. After each prediction, we repack the hidden state in order to break the connection from the last iteration. Then we compute the loss and compute the gradients again with backpropagation. At the end we apply the gradients to optimize the result of each iteration.

Overall, the RNN is able to understand the relationship between sin and cos, and use the parameters inside the RNN to decide which point on cos curve is corresponding to a point on sin curve at each time step.

After doing a continuous plot, we can see the progress of our RNN model in the figure below.



As we can see, by the end, the cos predictions (blue) are stabilized with a loss of 0.0052 after 100 epochs.

Step 9

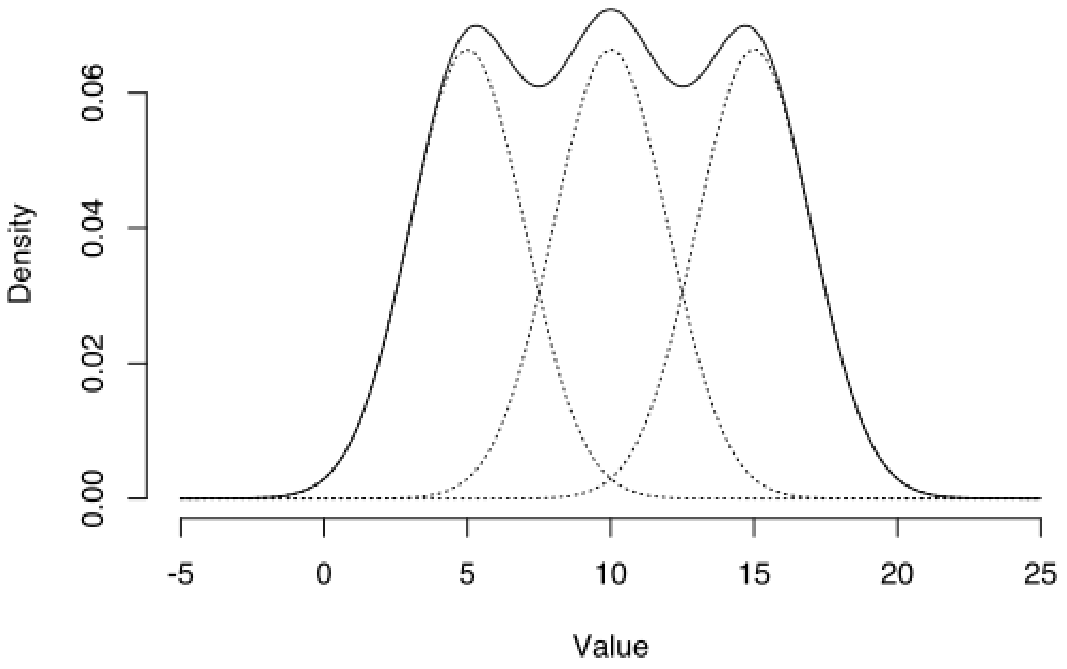
From this step onward,we move on to the main part. We get to utilize a much bigger dataset (https://github.com/Jakobovski/free-spoken-digit- dataset). In order to use it, we use the given *parser.py* function, in which we extract the MFCCs, normalize the data and divide in train and test datasets. In order to get the data, we uploaded them in a zip file google drive and then extracted them in a folder named *recordings*.

Aftewards, using the built in function StratifiedShuffleSplit, we split our training set in training and validation set (80%-20% proportion) as described.

Step 10

Here we try digit recognition using GMM-HMM models.

As we already know, Gaussian models are easy to use in Machine Learning and produce great results. But usually (as in our case) the data are too complex to be described with a Gaussian Model. That's why Gaussian Mixture Models are used, which combine a number of Gaussian distributions, giving the flexibility needed in speech recognition. More specifically, in terms of speech recognition, an array of characteristics is a probable observation in a state, and since constant changes of such observation are allowed, the modelization with GMM is possible.



Example of GMM

The HMM model is useful in order to represent speech as a sequence of observations. In our case, we use the phonemas as sequences to observe the pronounced digits.

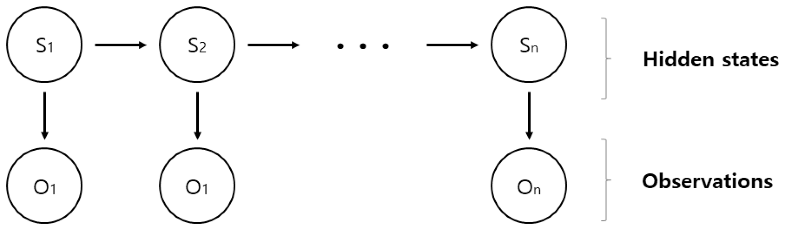
In our case, we fit 10 models, one for each digit, and in order to predict values, we calculate the possibility of each model using the EM algorithm, and pick as the prediction the digit that corresponds to the model with the highest probability. More specifically, if W is the digit (word) whose probability we are trying to predict, we have to calculate

$$argmax_w\ P(W|X) = argmax_w\ P(X|W)P(W)/P(X) = argmax_w\ P(X|W)P(W)$$

since P(X) is a constant.

For each model, we get the MFCCs (/frames) for the equivalent digit in the GMM. That is because GMMs perform density estimation, which is an unsupervised method. Hence, it tries to approximate the density probability function from the frames of a digit.

The HMM model is composed of hidden variables and observables. The bottom nodes are the audio features and the top ones are the phonemas (hidden variables).



Example of HMM model

The transition probabilities in the Markov Chains are defined in our transition matrix. As described, the only non-zero values are $a_{i,i}$ and $a_{i,i+1}$ $\forall i$. We chose 1/2 for each value, and for the emission (ending) probability, we have 1/2 for the last state and zero for the rest. In addition, our starting state is always the first one.

Step 11

In order to train each model (fit), the EM algorithm was used. We chose 50 numbers of max iterations (defined iterations before stopping -if the model doesn't converge sooner-).

At the beginning of the EM-algorithm in general, we initialize our parameters with arbitrary values for the means, variances and weights of the Gaussians forming the GMM. Then the following two steps are repeated:

- Expectation step: Based on the current parameters, the density function of each Gaussian Distribution is modeled, multiplied by its prior probability. In the end we divide using the summation of the computed density functions (normalization)
- Maximazation Step: With the calculated density function, we compute the most suitable parameters (optimizing the log-likelihood), and update our predictions.

In general we are trying to converge by maximizing the sum of the log-likelihood of each observation (the probability of the data with a given model -calculated each time in the E-step).

Step 12

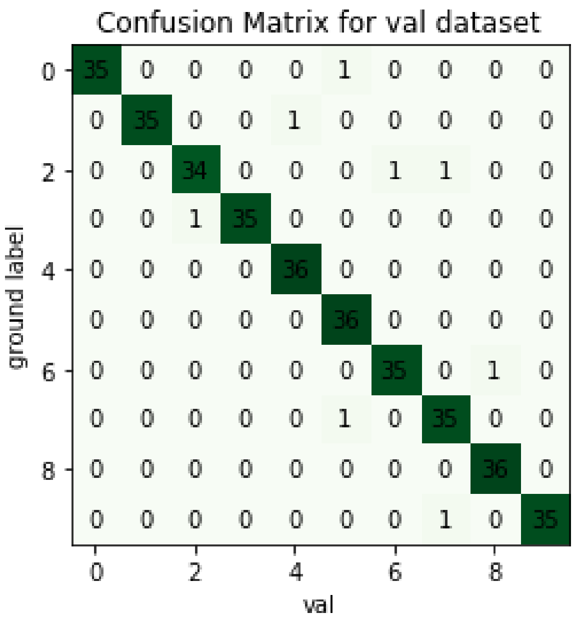
Here we have utilized the gridsearch method. More specifically, we test the model on our validation set using different parameters for the number of HMM states and the number of the Gaussians distributions. Then, we pick the one with the highest accuracy score and use it on our test dataset.

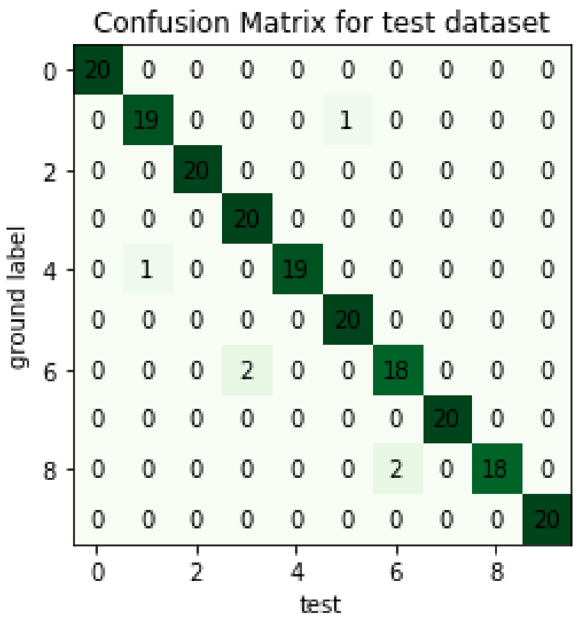
As mentioned in a previous step, in order to make a prediction, we calculate the log probability for every model (one for each digit) called Viterbi Probability and pick the one with the highest one.
We test all the combinations with 1 to 3 HMM states (exhibited worse results) and 2 to 5 Gaussians.
We conclude that the best model is the one with 3 HMM States and 5 Gaussians, with accuracy 0.97778.
Using this model, we also try digit recognition in our test set, achieving a similar accuracy score of 0.97.

As explained in the issue 15 in github, sometimes an error occurs due to an exception caused by the changes in numpy after the 1.7 edition. Nevertheless, the code keeps running and the models are fitted, correctly (as we can later observe) so the error in the .ipynb file is to be ignored.
Defining the hyperparameters using the validation set is a necessary procedure. First of all, the validation set has resulted from a stratified split (and thus preserving the percentages of the different digits) and that makes it a good reference set for our training set.
More primarily though, we couldn't have used the test set, since the set used to determine the best hyperparameters becomes part of the fitting of the model. So it would be futile and produce overoptimistic results. The same would happen in case we used the same validation set to measure the performance.

Step 13

Here we can see the confusion matrixes for both the validation and the test datasets. Hence, we can see for each of our predictions, how many were correctly classified, and how many were incorrectly classified (and the misclassified label)





In general, we observe a small portion of random mistakes. We could say (for the test dataset) that was a bigger confusion with the the number 6 (either wrongly classifying it as 8 or wrongly classifying 3 as 6).

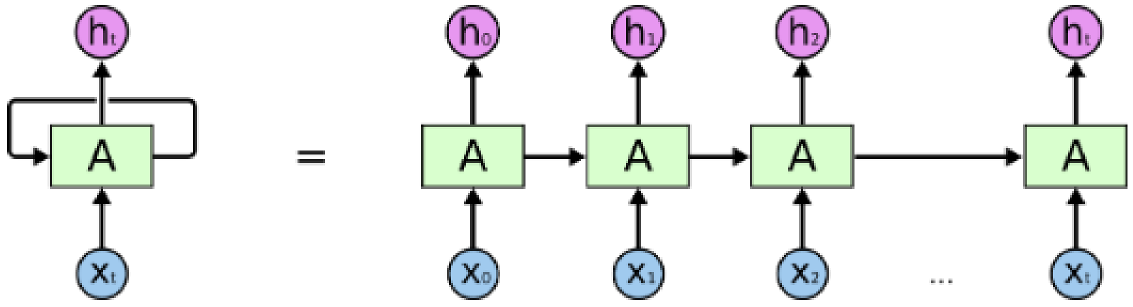
Concerning the accuracies, as mentioned in the previous step, we achieved 0.97778 and 0.97 in the validation and test datasets respectively.

Step 14

Recurrent Neural Networks

In traditional neural networks, all the inputs and outputs are independent of each other, however in many cases there is the need that the network “remembers” things learnt from prior input(s) while generating output(s).

The main difference between RNNs and feedforward networks is the existence of feedback loops, that produce the recurrent connection in the unfolded network.



Unrolled RNN

Given an input sequence $x = (x^0, ..., x^{T-1})$,the hidden states of a recurrent layer $h = (h^0, ..., h^{T-1})$ and the output of a single hidden layer RNN $y = (y^0, ..., y^{T-1})$ can be computed as follows:

$$h^t = H(W_{xh}x^t + W_{hh}h^{t-1} + b_h)$$

$$y^t = O(W_{ho}h^t + b_o)$$

where W_{xh}, W_{hh}, W_{ho} denote the connection weights from the input layer x to the hidden layer h , the hidden layer h to itself and the hidden layer to the output layer y, respectively, b_h and b_o are two bias vectors, $H()$ and $O()$ are the activation functions in the hidden layer and the output layer.

The hidden state captures the relationship that neighbors might have with each other in a serial input and it keeps changing in every step, and thus every input undergoes a different transition.

Long Short Term Memory Network (LSTM)

LSTM is an advanced RNN architecture, which overcomes the vanishing gradient problem and effectively captures long term dependencies. Vanishing gradients is when the gradients of loss functions become too small (approaches zero), then the network becomes increasingly hard to train, as the weights and biases of the initial layers are not updated effectively with the training sessions. In LSTM networks non linear units in RNN are replaced by LSTM memory blocks.

The LSTM memory block contains one self connected memory cell c_t and three multiplicative units (gates), e.g the input gate i_t, the forget gate f_t and the output gate o_t. It has the ability to remove or add information to the cell state, carefully regulated by the gates. The input gate layer determines which values we'll update and the forget gate governs the flow of information, by deciding which information will be excluded.The output gate controls how much information from the cell is passed to the output. The memory cell has a self connected edge of weight 1, so that the gradient does not vanish or explode. The activation of the memory cell and the three gates are as follows:

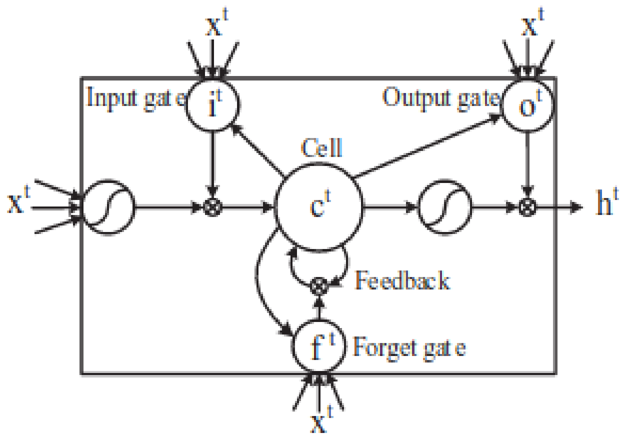
$$i^t = \sigma(W_{xi}x^t + W_{hi}h^{t-1} + W_{ci}c^{t-1} + b_i)$$

$$f^t = \sigma(W_{xf}x^t = W_{hf}h^{t-1} + W_{cf}c^{t-1} + b_f)$$

$$c^t = f_t c_{t-1} + i_t tanh(W_{xc}x^t + W_{hc}h^{t-1} + b_c)$$

$$o^t = \sigma(W_{xo}x^t + W_{ho}h^{t-1} + W_{co}c^t + b_o)$$

$$h^t = o_t tanh(c^t)$$



LSTM block with one cell

0.1 Steps 1-6

We implemented an LSTM network which aims to classify spoken digits. The input consist of sequences of variable length, which we padded with 0s. In each time step the input is 6 features(in this speech recognition task mel-frequency cepstrum coefficients are used). After carefully handtuning the hyperparameters and observing the accuracy we fixed them as follows : $hiddensize = 16$ and $numberoflayers = 2$,$epochs = 25$.

In our LSTM we introduced a new hyperparameter, dropout probability that specifies the probability at which outputs of the layer are dropped out, or inversely, the probability at which outputs of the layer are retained. We set it as 0.4. Probabilistically dropping out nodes in a network is a simple and effective regularization method, that reduces overfitting and improves generalization error. In our case we have limited training data and many of the complicated relationships between inputs and outputs will be the result of sampling noise, so they will exist in the training set but not in the test set. This may lead to overfitting. The obvious idea of averaging the outputs of many separately trained nets is prohibitively expensive. Applying dropout to a neural network amounts to sampling a “thinned” network from it. The thinned network consists of all the units that survived dropout. So training a neural network with dropout can be seen as training a collection of exponentially many thinned networks with extensive weight sharing and thus can be interpreted as a form of model averaging.

Apart from dropout another regularization technique, we applied is L2 regularization. We introduced a weight decay term and set it as 0.001. L2 regularization (ridge regression) adds “squared magnitude” of coefficient as penalty term to the loss function and thus ’penalizes’ large weights.

$$Loss = Error(y, \hat{y}) + \sum_{i=1}^{N} w_i^2$$

Regularization attempts to reduce the variance of the estimator by simplifying it, something that will increase the bias, in such a way that the expected error decreases.

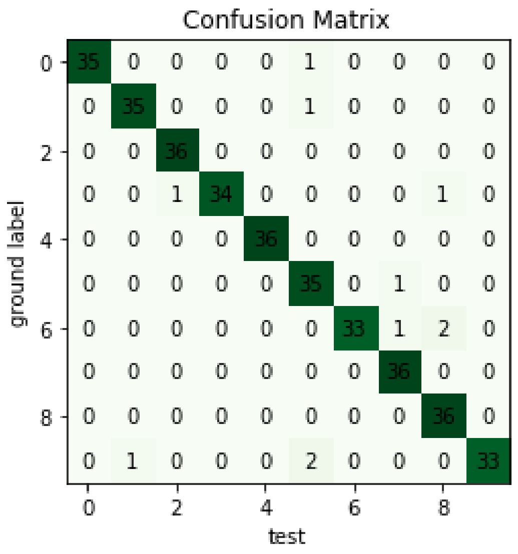
As a last step we implemented early stopping, with a patience of 10 epochs. If validation loss shows no improvement in these epochs the training phase stops. Early stopping is a form of regularization used to avoid overfitting. Up to a point the learner's performance on data outside of the training set is improved. Past that point, improving the learner's fit to the training data comes at the expense of increased generalization error. Early stopping rules provide guidance as to how many iterations can be run before overfitting occurs.

Using the above model we achieved the following accuracy and loss:

Test data : Accuracy 95% Loss: 0.157976

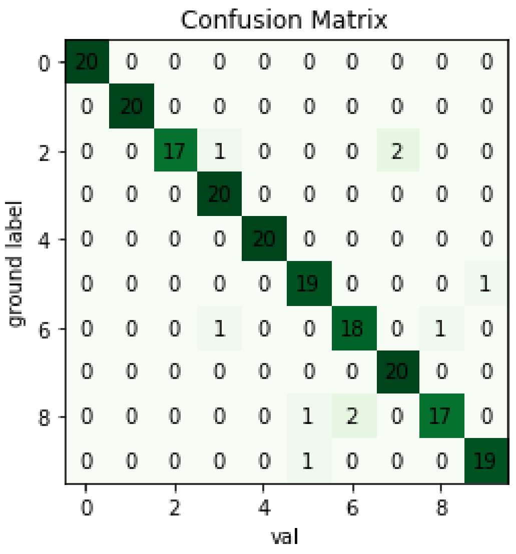
Validation data : Accuracy 97% Loss: 0.150924

We also present the confusion matrices and the loss plot:


Confusion Matrix
ground label
test

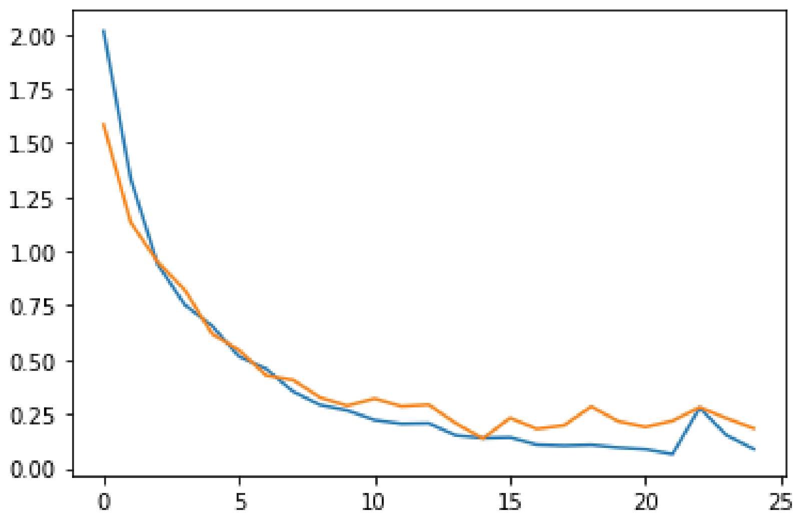
	0		2		4		6		8	
0	35	0	0	0	0	1	0	0	0	0
	0	35	0	0	0	1	0	0	0	0
2	0	0	36	0	0	0	0	0	0	0
	0	0	1	34	0	0	0	0	1	0
4	0	0	0	0	36	0	0	0	0	0
	0	0	0	0	0	35	0	1	0	0
6	0	0	0	0	0	0	33	1	2	0
	0	0	0	0	0	0	0	36	0	0
8	0	0	0	0	0	0	0	0	36	0
	0	1	0	0	0	2	0	0	0	33

(α′) Confusion matrix for test data


Confusion Matrix
ground label
val

	0		2		4		6		8	
0	20	0	0	0	0	0	0	0	0	0
	0	20	0	0	0	0	0	0	0	0
2	0	0	17	1	0	0	0	2	0	0
	0	0	0	20	0	0	0	0	0	0
4	0	0	0	0	20	0	0	0	0	0
	0	0	0	0	0	19	0	0	0	1
6	0	0	0	1	0	0	18	0	1	0
	0	0	0	0	0	0	0	20	0	0
8	0	0	0	0	0	1	2	0	17	0
	0	0	0	0	0	1	0	0	0	19

(β′) Confusion matrix for validation data



Validation loss (orange) and train loss (blue) plot

We note that in our notebook validation and test loss for each epoch is shown.

0.2 Step 7

Bidirectional Neural Network

In many cases it is desirable to overcome the limitations to utilize information from both past (backwards) and future (forward) states simultaneously. The idea of Bidirectional Recurrent Neural Networks (BRNNs) is to split the state neurons of a regular RNN in a part that is responsible for the positive time direction (forward states) and a part for the negative time(backward states). These two recurrent hidden layers share the same output layer. In speech recognition providing the sequence bi-directionally is justified because there is evidence that the context of the whole utterance is used to interpret what is being said rather than a linear(in time) interpretation.

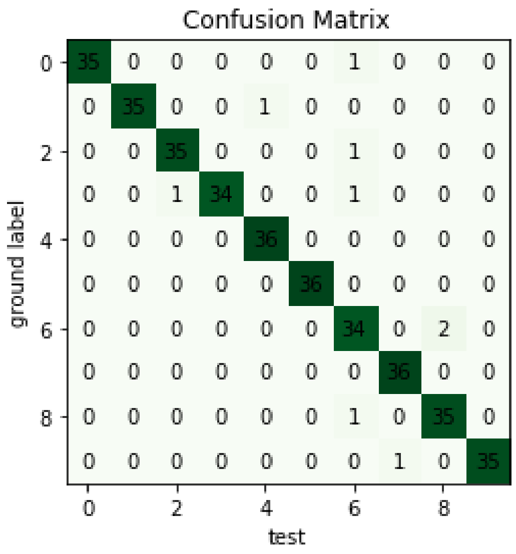
In this step we utilized a bidirectional LSTM with the same hyperparameters as the previous model, changing the epochs to 45. We observed slightly higher accuracy in the test dataset. We present our results below.

Using the above model we achieved the following accuracy and loss:

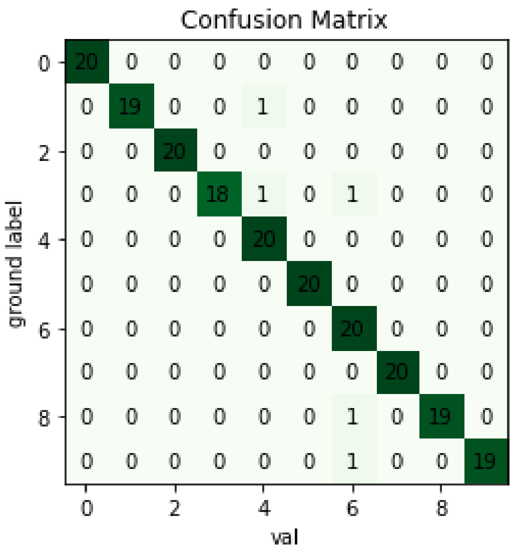
Test data : Accuracy: 98%   Loss: 0.154444

Validation data : Accuracy: 98%   Loss: 0.077564

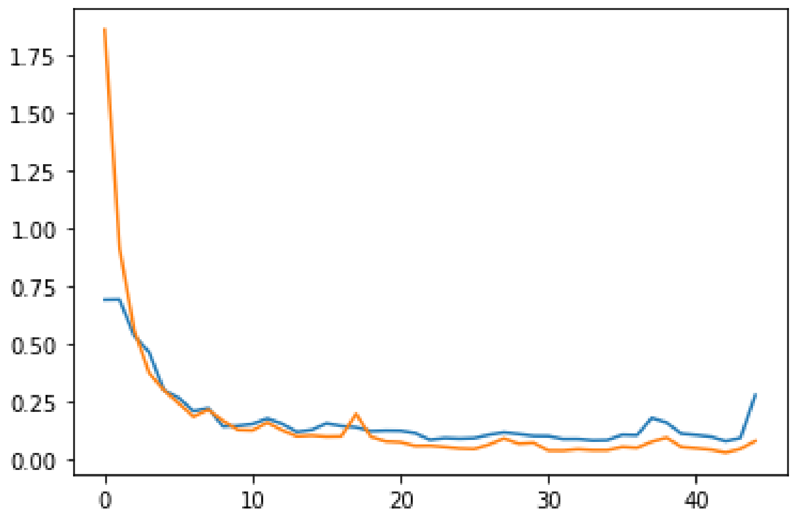
We also present the confusion matrices and the loss plot:



(α′) Confusion matrix for test data



(β′) Confusion matrix for validation data



Validation loss (orange) and train loss (blue) plot

We note that in our notebook validation and test loss for each epoch is shown.

Αναφορές

[1] Duda, Richard O. and Hart, Peter E. and Stork, David G. *"Pattern Classification", 2000*

[2] Nitish Srivastava et al. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*

[3] Ian Goodfellow, Yoshua Bengio, Aaron Courville. *"Deep Learning", MIT, 2017*