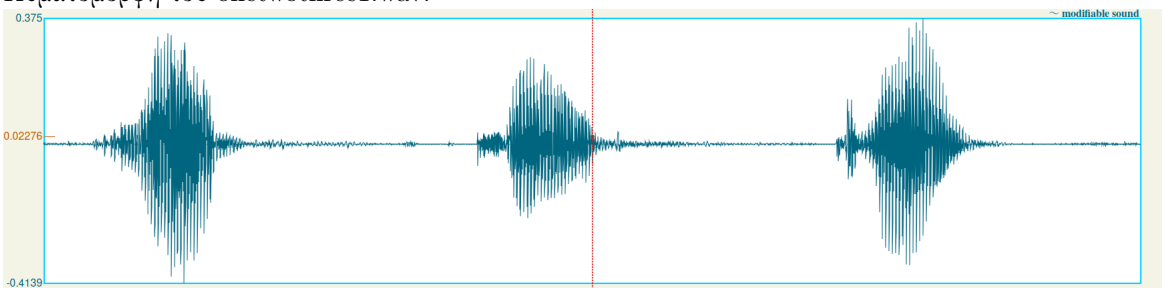
1η Εργαστήριακή Άσκηση Αναγνώριση Προτύπων Αναγνώριση φωνής με Κρυφά Μαρκοβιανά Μοντέλα και Αναδρομικά Νευρωνικά Δίκτυα

Άρης Μαρκογιαννάκης: 03120085

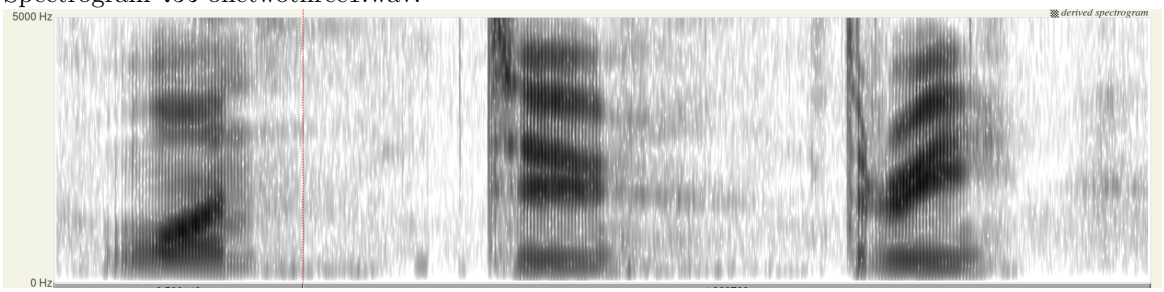
Προπαρασκευή

Βήμα 1

Κυματομορφή του onetwothree1.wav:



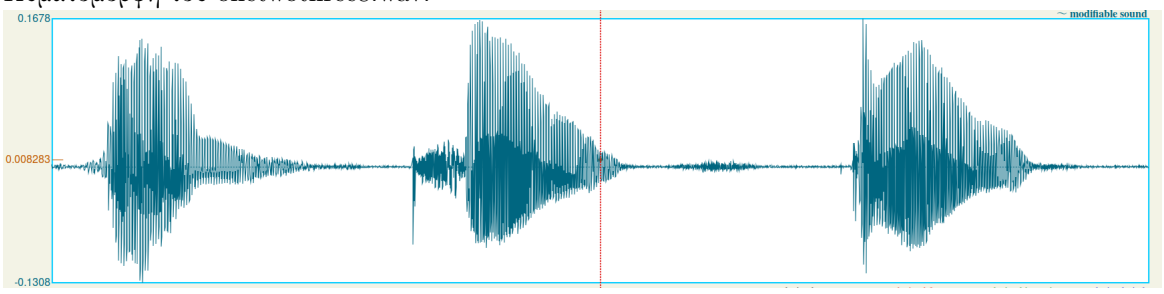
Spectrogram του onetwothree1.wav:



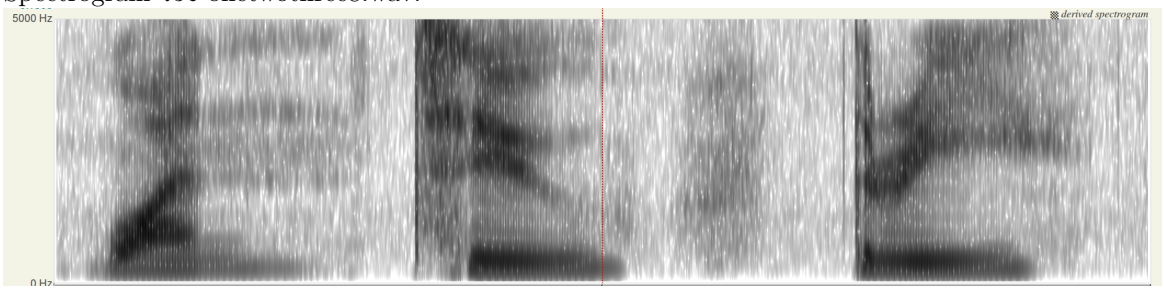
Φώνημα	pitch(Hz)	Formant 1 (Hz)	Formant 2 (Hz)	Formant 3 (Hz)
"α"	134	764	1227	2465
"ου"	130	320	1809	2385
"ι"	128	385	1955	2402

Table 1: onetwothree1.wav, άντρας

Κυματομορφή του onetwothree8.wav:



Spectrogram του onetwothree8.wav:



Φώνημα	pitch (Hz)	Formant 1 (Hz)	Formant 2 (Hz)	Formant 3 (Hz)
”α”	176	880	1249	3050
”ου”	186	390	2312	2876
”ι”	174	350	2346	2618

Table 2: onetwothree8.wav, γυναίκα

Βήμα 2

Ο κώδικας που υλοποιεί τον data parser βρίσκεται στο αρχείο *scripts/step2.py*. Επίσης φαίνεται και παρακάτω:

```
import os
import librosa
import re

def text_to_number(text):

  number_map = { 'zero': 0,'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6, 'seven': 7

  if  text.lower() in number_map:
    return number_map[text.lower()]
  else:
    return None

def data_parser(data_dir = '/home/arismarkog/patrec1/pr_lab2_data/digits'):
    wav_data = []
    speakers = []
    digits = []

    # Regular expression to capture speaker and digit from filename
    pattern = re.compile("([a-zA-Z]+)([0-9]+)")

    for filename in os.listdir(data_dir):

        if filename.endswith('.wav'):

            filepath = os.path.join(data_dir, filename)

            # Load the audio file with librosa
            wav, _ = librosa.load(filepath, sr=None)
            wav_data.append(wav)
```

```
            match = pattern.match(filename)
            if match:
                speaker = match.group(2)
                digit = match.group(1)
                speakers.append(speaker)
                digits.append(text_to_number(digit))

    return wav_data, speakers, digits
```

Βήμα 3

Ο κώδικας που εξάγει τα Mel-Frequency Cepstral Coefficients (MFCCs) για κάθε αρχείο ήχου, συγκεκριμένα 13 MFCCs ανά αρχείο, με μήκος παραθύρου 25 ms και βήμα 10 ms και υπολογίζει τις deltas και delta-deltas βρίσκεται στο αρχείο *scripts/step3.py*. Επίσης φαίνεται και παρακάτω:

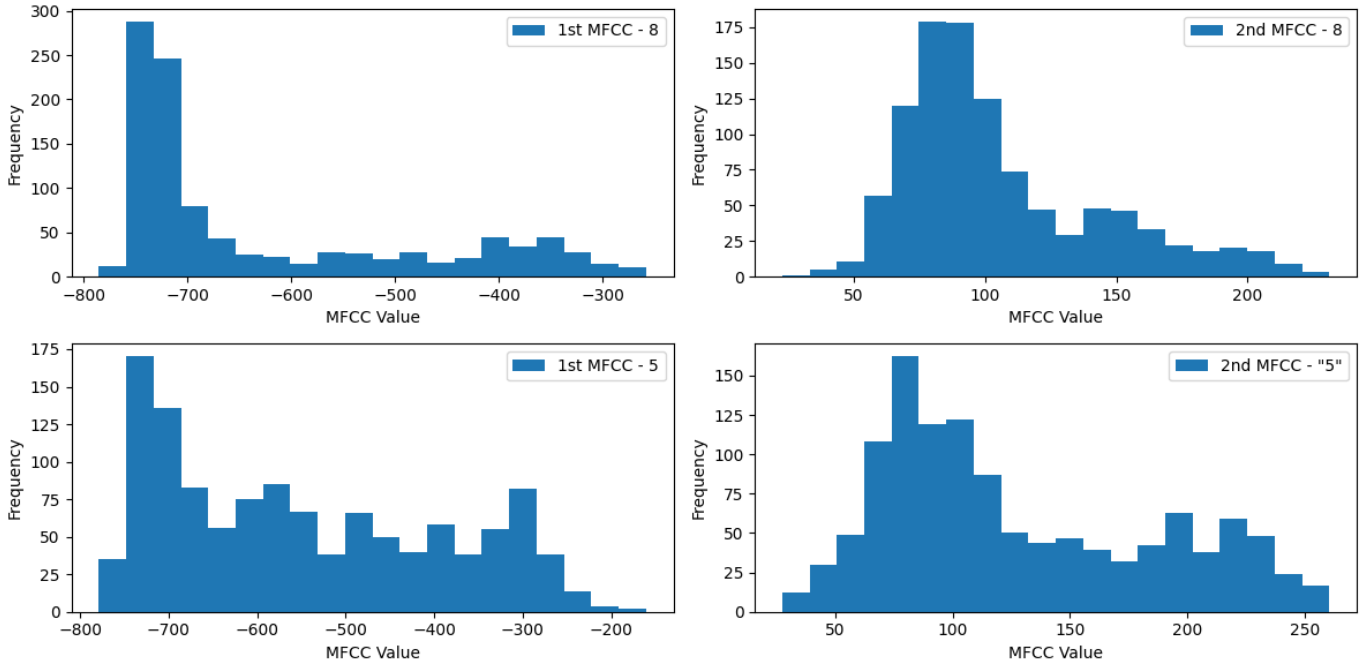
```
import os
import librosa
import re
from step2 import data_parser

def parse_audio_data_with_mfcc(data_dir = '/home/arismarkog/patrec1/pr_lab2_data/digits'):

    wav_data, speakers, digits = data_parser(data_dir)

    mfcc_features = []
    delta_features = []
    delta_delta_features = []


    for wav in wav_data:

        sr = 16000  #sample rate

        # Calculate MFCCs with 13 coefficients, 25ms window length, and 10ms hop length
        mfcc = librosa.feature.mfcc(y=wav, sr=sr, n_mfcc=13,
                                    n_fft=int(sr * 0.025), hop_length=int(sr * 0.01))
        mfcc_features.append(mfcc)

        # Calculate delta and delta-delta features
        delta = librosa.feature.delta(mfcc)
        delta_delta = librosa.feature.delta(mfcc, order=2)
        delta_features.append(delta)
        delta_delta_features.append(delta_delta)

    return wav_data, speakers, digits, mfcc_features, delta_features, delta_delta_features
```

Βήμα 4

Ο κώδικας που υλοποιεί τον data parser βρίσκεται στο αρχείο *scripts/step4.py*.

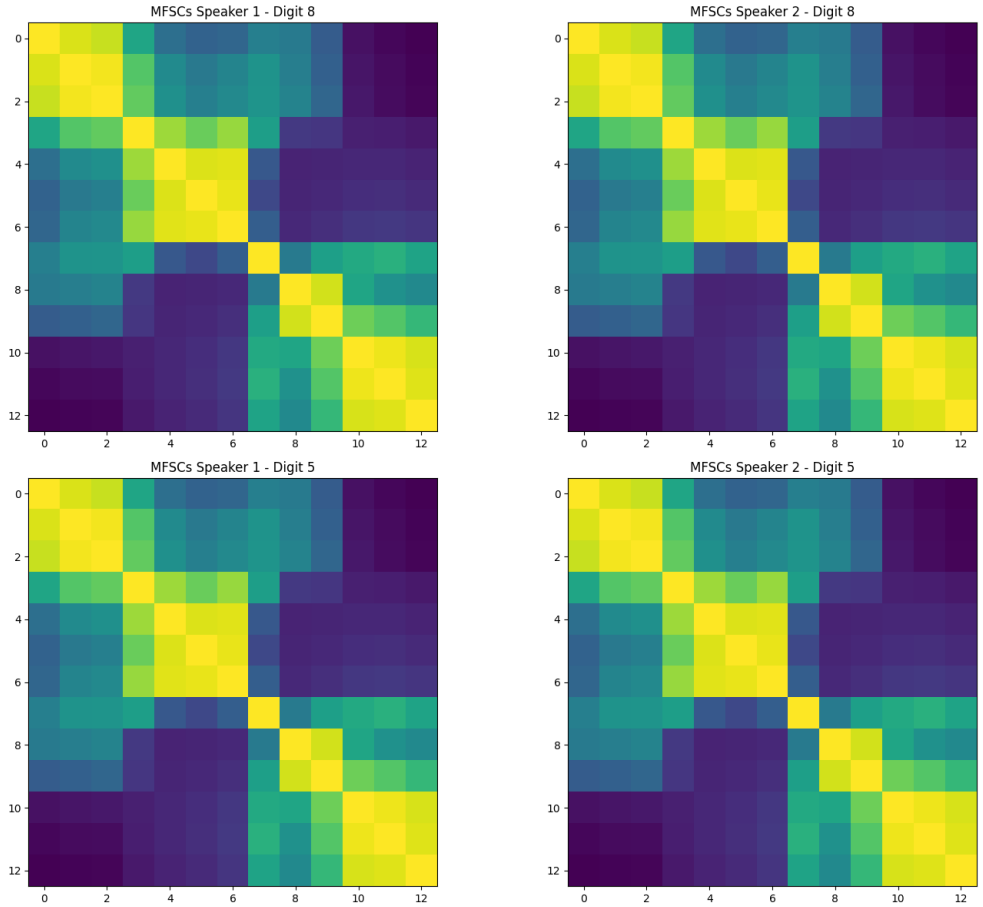
Έχουμε $n_1 = 8$ και $n_2 = 5$.

Παρακάτω φαίνονται τα ιστογράμματα του 1ου και του 2ου MFCC των ψηφίων n1='8' και n2='5'.

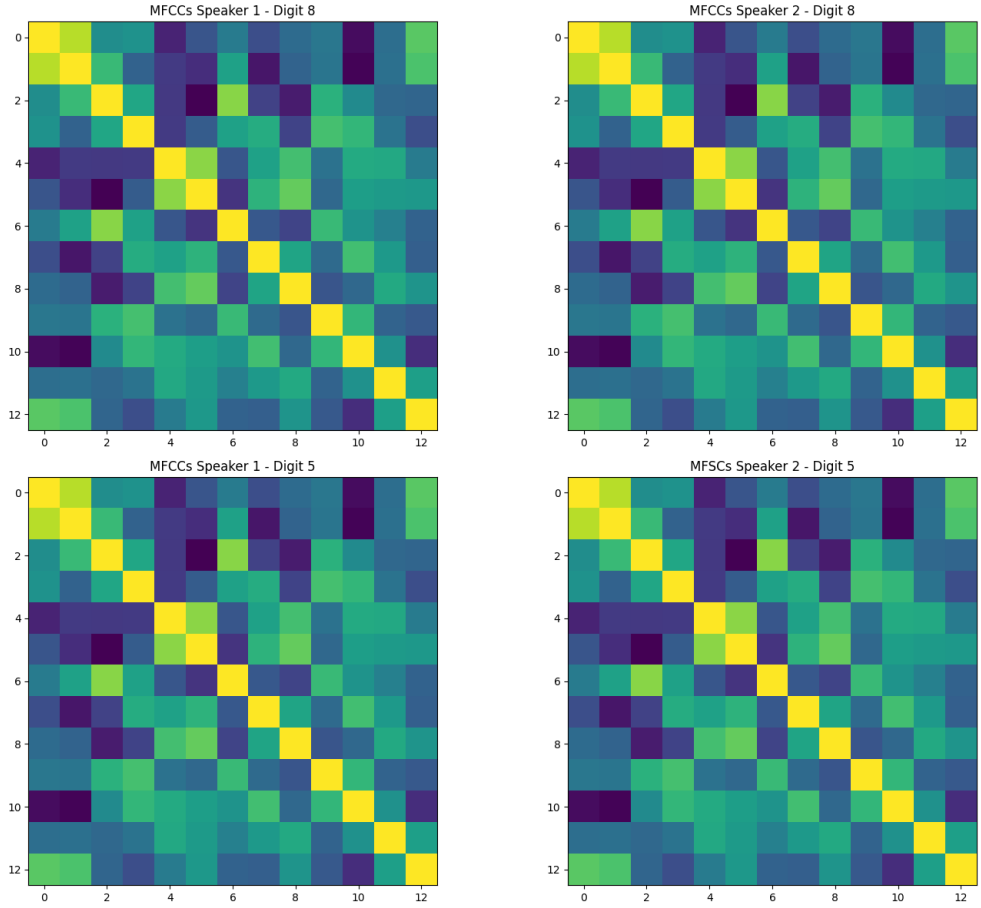


Παρατηρούμε ότι τα ιστογράμματα 1st MFCC - ”8” και 1st MFCC - ”5” μοιάζουν αρκετά μεταξύ τους όπως επίσης και τα ιστογράμματα 2nd MFCC - ”8” και 2nd MFCC - ”5”. Για αυτό δεν μπορούμε να αποφανθούμε για το ψηφίο που αναφέρει ο ομιλητής μόνο από τα ιστογράμματα των πρώτων και δεύτερων MFCCs.

Παρακάτω φαίνεται η γραφική αναπαράσταση της συσχέτισης των MFSCs για την κάθε εκφώνηση:



Παρακάτω φαίνεται η γραφική αναπαράσταση της συσχέτισης των MFCCs για την κάθε εκφώνηση:

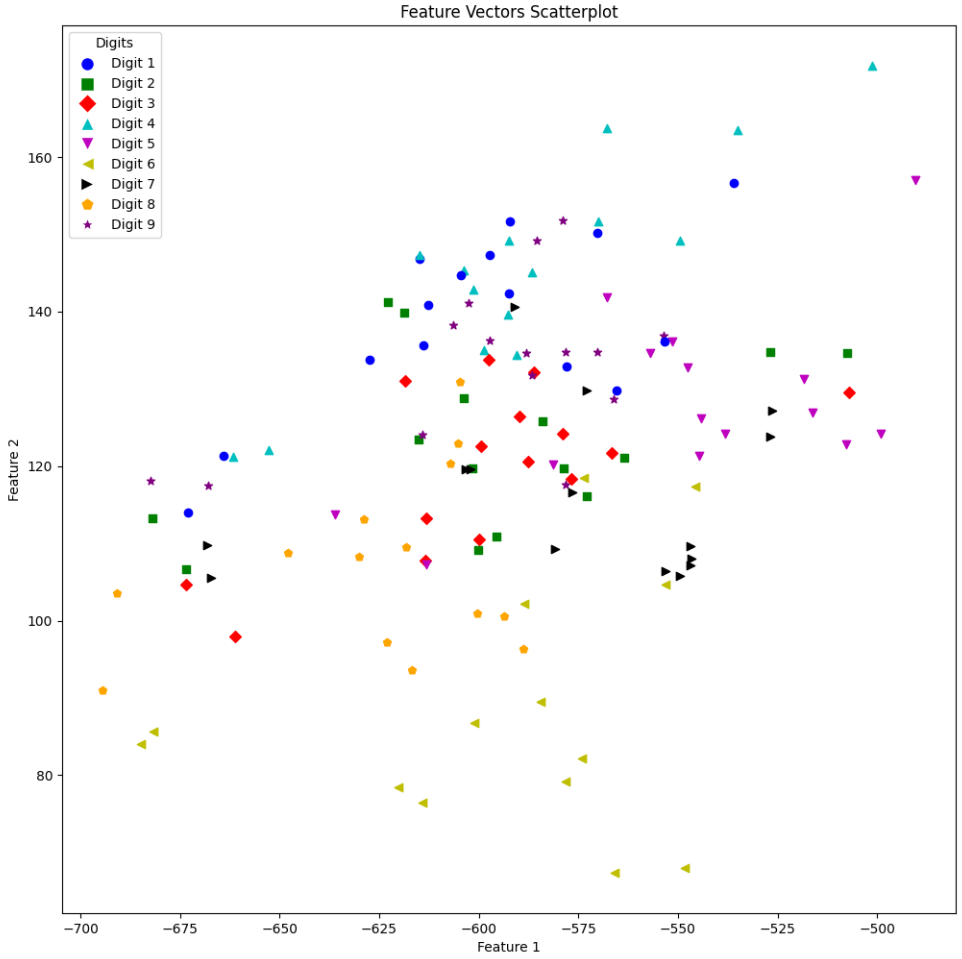


Στα MFCCs έχουμε κίτρινο χρώμα, δηλαδή συντελεστή συσχέτισης +1, μόνο στην κύρια διαγώνιο και σκούρο μπλε χρώμα, δηλαδή συντελεστή συσχέτισης -1, μόνο σε σχετικά λίγα μεμονωμένα κελία. Στα MFSCs όμως έχουμε κίτρινο και σκούρο μπλε χρώμα σε πολύ μεγάλο μέρος του πίνακα συσχέτισης και αυτό υποδηλώνει ότι τα MFSCs είναι υψηλά συσχετισμένα. Γενικά θέλουμε τα features που χρησιμοποιούμε στα μοντέλα μας να έχουν χαμηλή συσχέτιση καθώς με αυτό τον τρόπο παίρνουμε αρκετή καινούρια πληροφορία από κάθε feature που χρησιμοποιούμε. Για αυτό προτιμάμε τα MFCCs από τα MFSCs.

Βήμα 5

Ο κώδικας για αυτό το ερώτημα βρίσκεται στο αρχείο *scripts/step5.py*.

Ενώνουμε τα mfccs – deltas – delta-deltas και έπειτα για κάθε εκφώνηση δημιουργούμε ένα διάνυσμα παίρνοντας τη μέση τιμή και την τυπική απόκλιση κάθε χαρακτηριστικού για όλα τα παράθυρα της εκφώνησης. Αναπαρηστούμε με scatter plot τις 2 πρώτες διαστάσεις των διανυσμάτων αυτών, χρησιμοποιώντας διαφορετικό χρώμα και σύμβολο για κάθε ψηφίο. Το scatterplot φαίνεται παρακάτω:

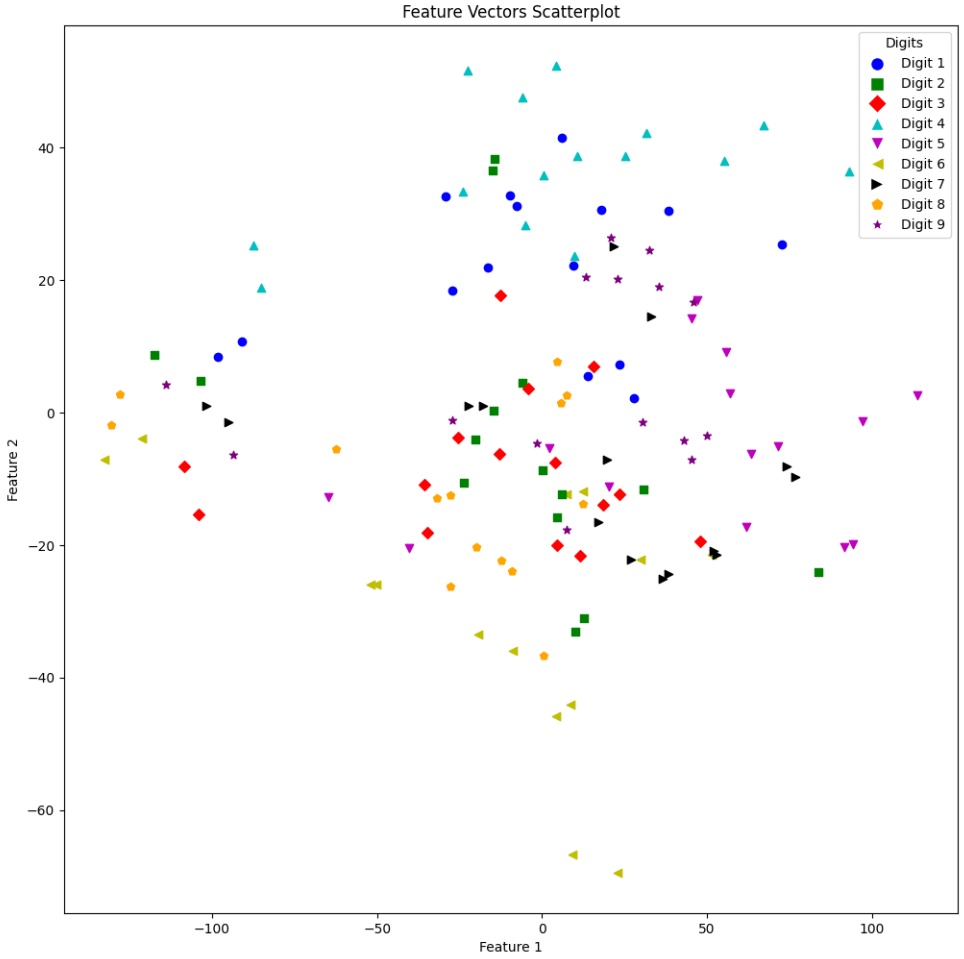


Παρατηρούμε ότι στο παραπάνω scatterplot τα data points για ένα δεδομένο ψηφίο δεν είναι συγκεντρωμένα σε μία περιοχή του επιπέδου και οι περιοχές για τα διαφορετικά ψηφία παρουσιάζουν σημαντική επικάλυψη, ειδικά στο κέντρο του scatterplot.

Βήμα 6

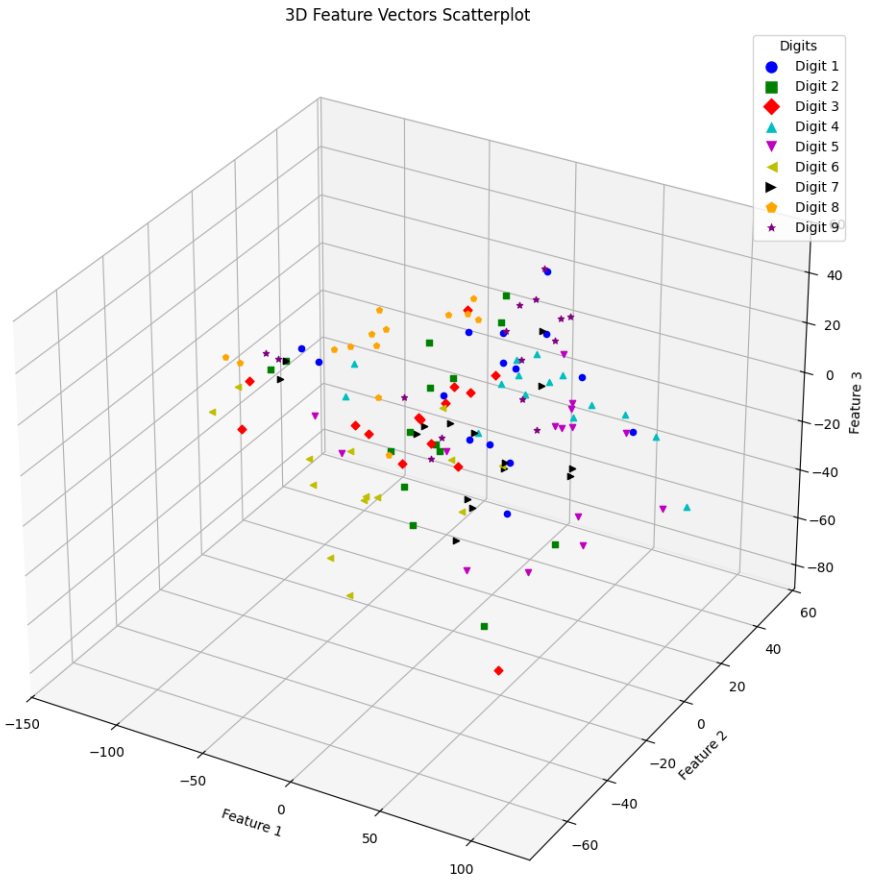
Ο κώδικας για αυτό το ερώτημα βρίσκεται στο αρχείο *scripts/step6.py*.

Το scatterplot αφότου εφαρμόσαμε PCA και μειώσαμε σε 2 τις διαστάσεις των διανυσμάτων:



Παρατηρούμε ότι σε σύκριση με το Βήμα 5, τα data points για κάθε ψηφίο είναι ομαδοποιημένα κάπως καλύτερα αφού οι ομάδες σημείων για κάθε ψηφίο είναι ελαφρώς πιο ευκρινώς διαχωρίσιμες από αυτές στο Βήμα 5.

Το scatterplot αφότου εφαρμόσαμε PCA και μειώσαμε σε 3 τις διαστάσεις των διανυσμάτων:



Παρατηρούμε ότι σε σύκριση με τα προηγούμενα scatterplots, τα data points για κάθε ψηφίο είναι ομαδοποιημένα καλύτερα και πιο ευκρινώς διαχωρίσιμα από τα προηγούμενα scatterplots.

Τα ποσοστά της αρχικής διασποράς για τις συνιστώσεις που προέκυψαν:

```
PCA with 2 components, explained_variance_ratio: [0.58794457 0.11858342]
PCA with 3 components, exaplined_variance_ratio: [0.58794457 0.11858342 0.10839216]
```

Κάθε τιμή στο explained_variance_ratio_ αντιστοιχεί στο ποσοστό της διασποράς των αρχικών δεδομένων που "περιέχεται" στην κάθε κύρια συνιστώσα. Το άθροισμα των τιμών για τις δύο ή τρεις πρώτες συνιστώσες δείχνει το ποσοστό της συνολικής διασποράς που διατηρείται στις μειωμένες διαστάσεις. Όπως βλέπουμε κρατώντας τις 2 πρώτες κύριες συνιστώσες διατηρούμε περίπου το 70.7% της αρχικής διασποράς ενώ κρατώντας τις 3 πρώτες κύριες διαστάσεις διατηρούμε περίπου το 81.5% της αρχικής διασποράς. Όσο μεγαλύτερη είναι η αθροιστική διασπορά των συνιστωσών τόση περισσότερη είναι η πληροφορία των αρχικών δεδομένων που διατηρείται μετά την μείωση διαστατικότητας. Επομένως βλέπουμε ότι τα ποσοστά που πήραμε είναι αρκετά ικανοποιητικά, ειδικά για τις 3 κύριες συνιστώσες.

Βήμα 7

Ο κώδικας για αυτό το ερώτημα βρίσκεται στο αρχείο *scripts/step7.py*.

Χωρίσαμε τα δεδομένα σε train-test με αναλογία 70%-30% χρησιμοποιώντας την συνάρτηση train_test_split του scikit learn και κανονικοποιήσαμε τα δεδομένα χρησιμοποιώντας τον MinMaxScaler() του scikit-learn. Τον προτιμάμε έναντι του StandardScaler() επειδή:

- Όπως είδαμε και από τα ιστογράμματα των MFCCs στα προηγούμενα ερωτήμαρα η κατανομή τους δεν μοιάζει με Γκαουσιανή.
- Παρακάτω θα χρησιμοποιήσουμε ταξινομητές που εξαρτώνται από την απόσταση SVM και kNN με τους οποίους ο MinMaxScaler() τα πηγαίνει καλύτερα.

Έπειτα θα εξετάσουμε τους ταξινομητές Naive Bayes(GaussianNB), kNN, SVM και MLP. Για τους kNN, SVM και MLP εκτελούμε grid search για να κάνουμε hyperparameter tuning. Συγκεκριμένα δοκιμάζουμε:

```
# SVM hyperparameter grid
param_grid_svm = {'C': [0.1, 1, 10], 'kernel' : ['poly', 'linear', 'rbf']}

# KNN hyperparameter grid
param_grid_knn = {'n_neighbors': [2, 3, 5], 'weights': ['uniform', 'distance']}

# MLP hyperparameter grid
param_grid_mlp = {'hidden_layer_sizes': [(100,), (50, 50), (100, 100)],
                  'batch_size': [8, 16,'auto']}
```

Τα αποτελέσματα που πήραμε:

Classifier	Best Parameters	Accuracy in Test Set
Naive Bayes	Default	0.4750
SVM	'C': 1, 'kernel': 'linear'	0.6000
kNN	'n_neighbors': 2, 'weights': 'distance'	0.5500
MLP	'batch_size': 8, 'hidden_layer_sizes': (50,50)	0.6750

(Bonus): Έπειτα προσθέτουμε στα feature_vectors τα zero-crossing-rate,spectral_centroid, spectral_rolloff chroma_stft και RMSE για κάθε αρχείο ήχου και παίρνουμε τα παρακάτω αποτελέσματα:

Classifier	Best Parameters	Accuracy in Test Set
Naive Bayes	Default	0.5750
SVM	'C': 10, 'kernel': 'rbf'	0.6750
kNN	'n_neighbors': 3, 'weights': 'distance'	0.5500
MLP	'batch_size': 16, 'hidden_layer_sizes': (100, 100)	0.7500

Βάσει των αποτελεσμάτων που προέκυψαν, παρατηρούμε ότι η απόδοση του Naive Bayes βελτιώθηκε από 0.475 σε 0.575 με την προσθήκη επιπλέον χαρακτηριστικών, υποδεικνύοντας ότι αυτά προσφέρουν επιπλέον διαχωριστική πληροφορία. Παρ' όλα αυτά, ο Naive Bayes παραμένει λιγότερο αποδοτικός από τους άλλους ταξινομητές, πιθανώς λόγω της υπόθεσης ανεξαρτησίας των χαρακτηριστικών που βασίζεται. Το SVM, με ρυθμίσεις 'C': 10 και 'kernel': 'rbf', αυξάνει την απόδοσή του από 0.600 σε 0.675, αναδεικνύοντας τη χρησιμότητα των νέων χαρακτηριστικών και πως επίσης αυτά δεν είναι γραμμικά διαχωρίσημα. Ο kNN δεν σημείωσε βελτίωση, μένοντας στο 0.550, πιθανόν λόγω της φύσης του αλγόριθμου που επηρεάζεται περισσότερο από τη δομή του χώρου των χαρακτηριστικών παρά από την πολυπλοκότητά τους. Ο MLP, από την άλλη, παρουσίασε σημαντική βελτίωση από 0.675 σε 0.750, επιβεβαιώνοντας την ικανότητά του να διαχειρίζεται πιο σύνθετα μοτίβα μέσω των προσθέτων χαρακτηριστικών και παρουσιάζει το καλύτερο accuracy μεταξύ των ταξινομητών τόσο πριν όσο και μετά από την προσθήκη των νέων χαρακτηριστικών. Συνολικά, η προσθήκη αυτών των χαρακτηριστικών ενίσχυσε τις αποδόσεις των περισσότερων ταξινομητών, καθιστώντας τους πιο ικανούς να διαχωρίσουν διαφορετικούς ήχους, γεγονός που υποδεικνύει ότι τα χαρακτηριστικά αυτά είναι καλοσχεδιασμένα για την ανάλυση ηχητικών δεδομένων.

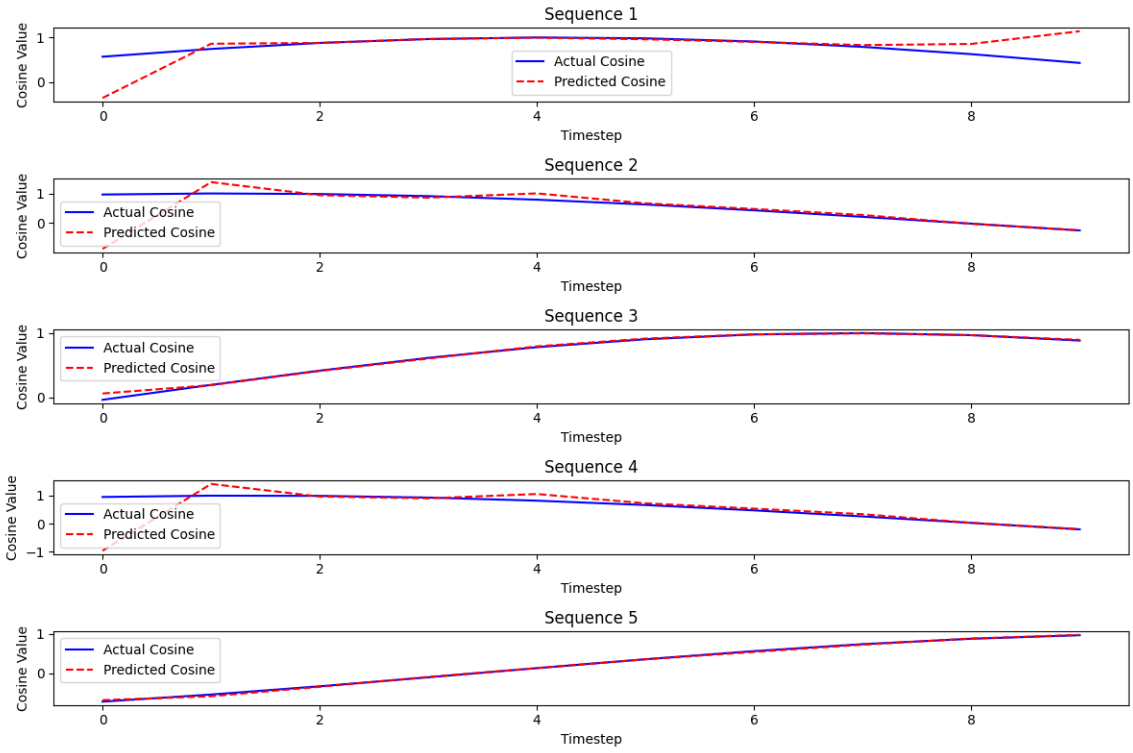
Βήμα 8

Ο κώδικας για αυτό το ερώτημα βρίσκεται στο αρχείο *scripts/step8.py*.

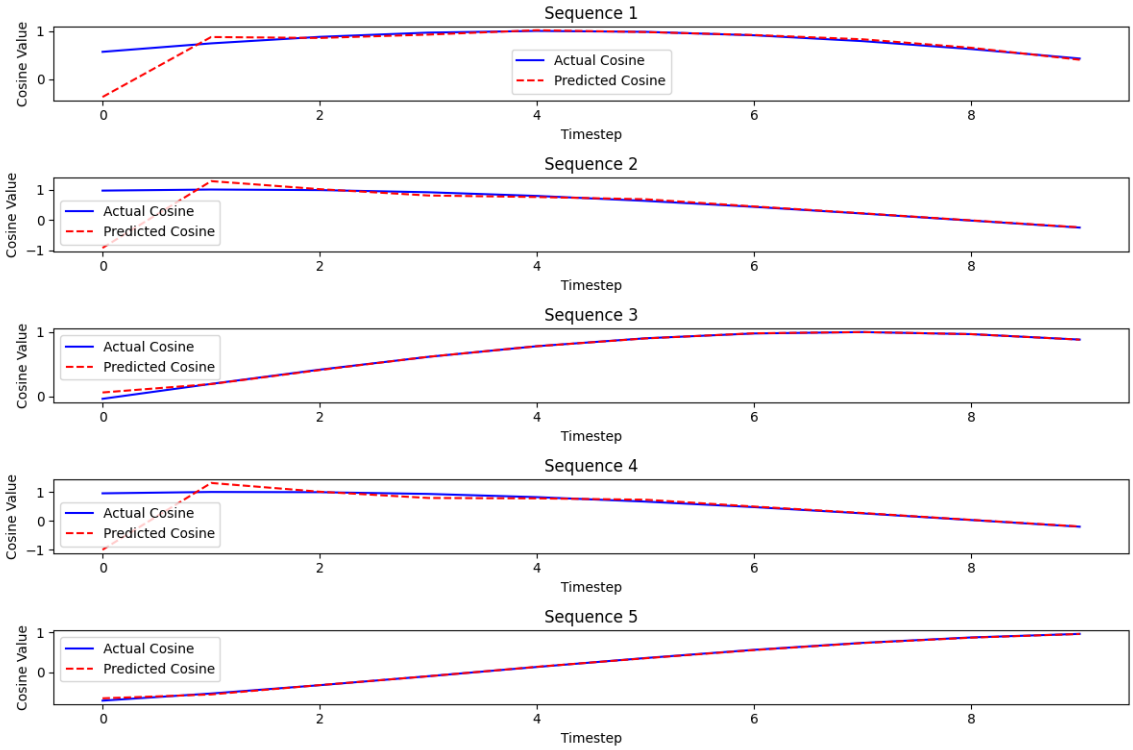
Η χρήση LSTM ή GRU αντί για ένα απλό RNN είναι ιδιαίτερα δημοφιλής λόγω της ικανότητάς τους να διατηρούν πληροφορίες για μεγαλύτερες ακολουθίες. Τα μοντέλα αυτά χρησιμοποιούν πύλες (gates) για να διαχειρίζονται τη ροή της πληροφορίας, επιτρέποντας έτσι τη διατήρηση ή την παράβλεψη παλαιότερων δεδομένων, προσπαθώντας να αντιμετωπίσουν το πρόβλημα των vanishing gradients που εμφανίζεται στα vanilla RNNs. Για αυτό τον λόγο καθίστανται ιδιαίτερα αποτελεσματικά στην πρόβλεψη ακολουθιών όπου η εξάρτηση από προηγούμενα δεδομένα είναι σημαντική.

Δημιουργήσαμε 20 ακολουθίες από 10 σημεία, ομοιόμορφα κατανεμημένα, μήκους $\frac{T}{3}$ με τυχαία αρχική φάση για να εκπαιδεύσουμε τα μοντέλα και 5 παρόμοιες ακολουθίες για να τα αξιολογήσουμε.

Παρακάτω φαίνονται τα αποτελέσματα χρησιμοποιώντας LSTM:



Παρακάτω φαίνονται τα αποτελέσματα χρησιμοποιώντας GRU:



Παρατηρούμε ότι και με τις δύο αρχιτεκτονικές (GRU και LSTM) οι προβλέψεις των μοντέλων είναι αρκετά ακριβείς. Το μόνο πρόβλημα που φαίνεται να παρουσιάζουν είναι στην πρόβλεψη του συνημιτόνου για την πρώτη χρονική στιγμή της κάθε ακολουθίας.

Κυρίως Μέρος

Βήματα 9-13

Ο κώδικας για αυτά τα ερωτήματα βρίσκεται στο αρχείο *scripts/hmm.py*.
Στο αρχείο hmm.py, στην συνάρτηση create_data, με την χρήση της συνάρτησης train_test_split του scikit-learn, διαχωρίζουμε με τέτοιο τρόπο τα δεδομένα ώστε να διατηρηθεί ίδιος ο αριθμός των διαφορετικών ψηφίων σε κάθε set, αξιοποιώντας την παράμετρο stratify. Συμπληρώνουμε τις κατάλληλες συναρτήσεις στο αρχείο hmm.py.

Στο αρχείο hmm.py, συμπληρώνουμε την συνάρτηση initialize_transition_matrix(), έτσι ώστε ο αρχικός πίνακας μετάβασης να μοιάζει:

$$A_{ij} = \begin{bmatrix} 0.5 & 0.5 & 0 & \cdots & 0 \\ 0 & 0.5 & 0.5 & \cdots & 0 \\ 0 & 0 & 0.5 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1.0 \end{bmatrix}$$

Συμπληρώνουμε την συνάρτηση initialize_starting_probabilities(), έτσι ώστε οι αρχικές πιθανότητες των καταστάσεων να είναι:

$$\pi_i = \begin{bmatrix} 1.0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

Συμπληρώνουμε την συνάρτηση initialize_end_probabilities(), έτσι ώστε οι τελικές πιθανότητες των καταστάσεων να είναι:

$$end_i = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1.0 \end{bmatrix}$$

Συμπληρώνουμε τις συναρτήσεις initialize_and_fit_normal_distributions() και initialize_and_fit_gmm_distributions() έτσι ώστε να αρχικοποιηθεί σωστά το emission probability distributions, μοντελοποιημένο ως GMM, για κάθε κατάσταση του HMM.

Έπειτα χρησιμοποιώντας τις συναρτήσεις train_single_hmm() και train_hmm() εκπαιδεύουμε το GMM-HMM μοντέλο και συμπληρώνουμε την συνάρτηση evaluate() έτσι ώστε το μοντέλο να επιλέγει το ψηφίο το οποίο εμφανίζει την μεγαλύτερη πιθανοφάνεια.

Έπειτα συνεχίζουμε στο hmm.py και πραγαμτοποιούμε ένα grid search έτσι ώστε να κάνουμε hyper-parameter tuning και βρούμε το καλύτερο δυνατό GMM-HMM μοντέλο. Δοκιμάζοντας διαφορετικές παραμέτρους στο validation set, μπορούμε να εντοπίσουμε τον καλύτερο συνδυασμό που αυξάνει την ακρίβεια αναγνώρισης. Αυτό βελτιώνει τη γενική απόδοση του μοντέλου και την ικανότητά του να αναγνωρίζει ψηφία με ακρίβεια. Αν προσαρμόζαμε κατευθείαν το μοντέλο στο test set, θα ρισκάραμε να «μάθει» τα χαρακτηριστικά μόνο αυτού του συνόλου, μειώνοντας τη δυνατότητά του να γενικεύει σε νέα δεδομένα.
Συγκεκριμένα δοκιμάζουμε:

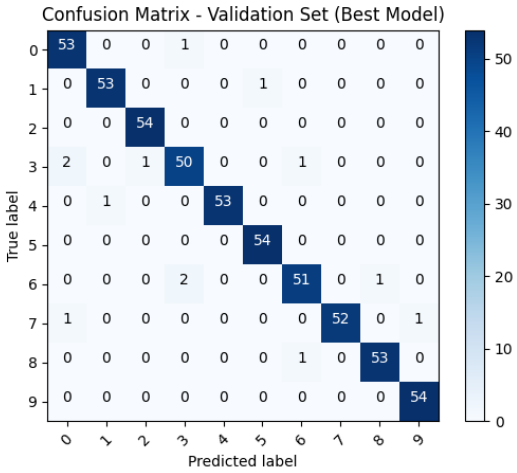
```
# Define the grid of parameters
n_states_list = [1, 2, 3, 4]
n_mixtures_list = [1, 2, 3, 4, 5]
```

Παρακάτω φαίνονται τα αποτελέσματα(accuracy) που πήραμε για το validation set κάνοτας το παραπάνω grid search:

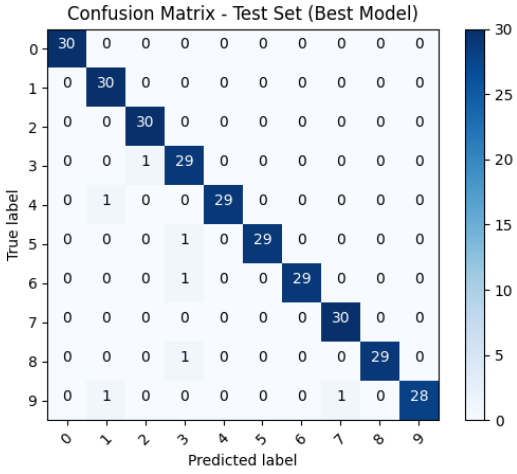
n_mixtures \ n_states	1	2	3	4
1	0.9241	0.9463	0.9704	0.9704
2	0.7796	0.7926	0.7944	0.7426
3	0.8481	0.8500	0.8463	0.9444
4	0.8870	0.9130	0.9519	0.9704
5	0.9167	0.9352	0.9704	0.9759

Παρατηρούμε ότι την καλύτερη επίδοση την έχουμε για n_states=4 και n_mixtures=5. Αξιοσημείωτο είναι επίσης ότι όταν n_mixtures=1 για n_states=1,2,3 έχουμε την καλύτερη επίδοση, δηλαδή χωρίς να χρησιμοποιήσουμε μίγμα γκαουσιανών. Παρατηρούμε όμως γενικά ότι αυξάνοντας τον αριθμό των states και των mixtures η επίδοση των μοντέλων αυξάνεται.

Best Parameters Validation Set Accuracy: 0.9759
Best Parameters Validation Set Confusion Matrix:



Best Parameters Test Set Accuracy: 0.9767
Best Parameters Test Set Confusion Matrix:



Παρατηρούμε οτι και στο test set το μοντέλο μας γενικεύει πάρα πολύ καλά.

Βήμα 14

Ο κώδικας για αυτό τα ερώτημα βρίσκεται στο αρχείο *scripts/lstm.py*.

1. Συμπληρώνουμε αρχικά τα σημεία του βοηθητικού κώδικα στο lstm.py που λείπουν.

2. Αρχικοποιούμε ένα απλό LSTM με rnn_size = 64, num_layers = 2, χωρίς dropout και early stopping.

3. Το εκπαιδεύουμε για 10 εποχές με batch_size = 32, learning_rate = 0,001 και weight_decay = 0. Σε κάθε εποχή τυπώνουμε το training loss. Τα αποτελέσματα φαίνονται παρακάτω:

```
Epoch 0: train loss = 2.0396179679562065,
Epoch 1: train loss = 0.9924112514537924,
Epoch 2: train loss = 0.5634900331497192,
Epoch 3: train loss = 0.3438186091096962,
Epoch 4: train loss = 0.2761176767594674,
Epoch 5: train loss = 0.23719831019201698,
Epoch 6: train loss = 0.1621253831202493,
Epoch 7: train loss = 0.13232789084534435,
Epoch 8: train loss = 0.11452448718688067,
Epoch 9: train loss = 0.09206494820468566,
Test loss: 0.09921362698078155, Test accuracy: 0.9566666666666667
```

Φαίνεται ότι πετυχαίνουμε πολύ καλό accuracy στο test set.

4. Τώρα σε κάθε εποχή τυπώνουμε το training loss, το validation loss και το validation accuracy. Τα αποτελέσματα φαίνονται παρακάτω:

```
Epoch 0: train loss = 2.0793, validation loss = 1.6454, validation accuracy 0.4426
Epoch 1: train loss = 1.1379, validation loss = 0.8040, validation accuracy 0.7778
Epoch 2: train loss = 0.6129, validation loss = 0.5617, validation accuracy 0.8296
Epoch 3: train loss = 0.4224, validation loss = 0.5861, validation accuracy 0.8074
Epoch 4: train loss = 0.4094, validation loss = 0.3065, validation accuracy 0.9056
Epoch 5: train loss = 0.2722, validation loss = 0.2748, validation accuracy 0.9074
Epoch 6: train loss = 0.2098, validation loss = 0.1900, validation accuracy 0.9426
Epoch 7: train loss = 0.1892, validation loss = 0.2426, validation accuracy 0.9241
Epoch 8: train loss = 0.1449, validation loss = 0.1708, validation accuracy 0.9500
Epoch 9: train loss = 0.1303, validation loss = 0.1778, validation accuracy 0.9574
Test loss: 0.15767625868320465, Test accuracy: 0.9533333333333334
```

Παρατηρούμε ότι το training loss είναι γνησίως φθίνουσα συνάρτηση του αριθμού των εποχών, το validation loss μειώνεται σε κάθε εποχή εκτός από την εποχή 6 και το validation accuracy αυξάνεται σε κάθε εποχή. Και για τα training loss, validation loss και validation accuracy ισχύει ότι παρουσιάζουν μεγαλύτερες μεταβολές στην τιμή τους στις πρώτες εποχές. Το test accuracy είναι και πάλι αρκετά ικανοποιητικό.

5. Τώρα προσθέτουμε στο μοντέλο μας Dropout και L2 Regularization.

Dropout: Κατά τη διάρκεια της εκπαίδευσης, το Dropout «απενεργοποιεί» τυχαία ένα ποσοστό νευρώνων σε κάθε επίπεδο, εμποδίζοντας το δίκτυο να βασίζεται υπερβολικά σε συγκεκριμένους νευρώνες. Έτσι, ενισχύει την ικανότητα γενίκευσης του μοντέλου.

L2 Regularization: Αποθαρρύνει τα βάρη των νευρώνων να παίρνουν πολύ μεγάλες τιμές. Προσθέτει ένα penalty στην συνάρτηση κόστους, που είναι ανάλογο με το τετράγωνο του μέτρου των βαρών. Αυτή η τεχνική βοηθά στη σταθερότητα του μοντέλου και στη μείωση του κινδύνου υπερεκπαίδευσης.

Εκπαιδεύμουμε το μοντέλο με dropout_probability=0.5 και για L2 Regulariaition λ=0.01. Τα αποτελέσματα φαίνονται παρακάτω:

```
Epoch 0: train loss = 2.2560, validation loss = 2.0237, validation accuracy 0.3704
Epoch 1: train loss = 1.6536, validation loss = 1.1863, validation accuracy 0.6444
Epoch 2: train loss = 1.1068, validation loss = 0.7977, validation accuracy 0.7370
Epoch 3: train loss = 0.7830, validation loss = 0.5324, validation accuracy 0.8630
Epoch 4: train loss = 0.6028, validation loss = 0.3882, validation accuracy 0.8907
Epoch 5: train loss = 0.4565, validation loss = 0.3049, validation accuracy 0.9148
Epoch 6: train loss = 0.4249, validation loss = 0.2216, validation accuracy 0.9500
Epoch 7: train loss = 0.3118, validation loss = 0.2580, validation accuracy 0.9278
Epoch 8: train loss = 0.3468, validation loss = 0.2022, validation accuracy 0.9333
Epoch 9: train loss = 0.2634, validation loss = 0.1302, validation accuracy 0.9611
Test loss: 0.10800996497273445, Test accuracy: 0.9633333333333334
```

Παρατηρούμε ότι σε σύγκριση με το προηγούμενο ερώτημα το traing loss και validation loss μειώνονται με πιο αργό ρυθμό και το validation accuracy αυξάνεται με πιο αργό ρυθμό. Παρατηρούμε ότι στο test set τελικά πετυχαίνουμε ελαφρώς καύτερο accuracy.

6. Τώρα προσθέτουμε στο μοντέλο μας Early Stopping και Checkpoints (δηλαδή αποθήκευση του καλύτερου μοντέλου).

Το Early Stopping διακόπτει την εκπαίδευση όταν το validation loss σταματά να μειώνεται, προστατεύοντας έτσι το μοντέλο από το φαινόμενο overfitting. Χρησιμοποιείται για να διασφαλιστεί ότι το μοντέλο δεν προσαρμόζεται υπερβολικά στα δεδομένα εκπαίδευσης και έτσι μπορεί να γενικεύσει καλύτερα.

Εκπαιδεύμουμε το μοντέλο για 30 εποχές αυτή την φορά, για να προλάβει να εφαρμοστεί και το early stopping, θέτοντας επιπλέον patience = 3. Τα αποτελέσματα φαίνονται παρακάτω:

```
Splitting in train test split using the default dataset split
Epoch 0: train loss = 2.1818, validation loss = 1.7821, validation accuracy 0.4222
Epoch 1: train loss = 1.4637, validation loss = 1.0128, validation accuracy 0.6648
Epoch 2: train loss = 0.9690, validation loss = 0.6849, validation accuracy 0.7722
Epoch 3: train loss = 0.7205, validation loss = 0.4596, validation accuracy 0.8796
Epoch 4: train loss = 0.5629, validation loss = 0.3781, validation accuracy 0.8926
Epoch 5: train loss = 0.5120, validation loss = 0.3569, validation accuracy 0.8963
Epoch 6: train loss = 0.4322, validation loss = 0.2908, validation accuracy 0.9130
Epoch 7: train loss = 0.3507, validation loss = 0.2436, validation accuracy 0.9315
Epoch 8: train loss = 0.2853, validation loss = 0.1996, validation accuracy 0.9463
Epoch 9: train loss = 0.2974, validation loss = 0.1964, validation accuracy 0.9463
Epoch 10: train loss = 0.2377, validation loss = 0.1769, validation accuracy 0.9481
Epoch 11: train loss = 0.3279, validation loss = 0.1713, validation accuracy 0.9574
Epoch 12: train loss = 0.2188, validation loss = 0.1841, validation accuracy 0.9574
Epoch 13: train loss = 0.2846, validation loss = 0.1433, validation accuracy 0.9611
Epoch 14: train loss = 0.1794, validation loss = 0.1331, validation accuracy 0.9630
Epoch 15: train loss = 0.1766, validation loss = 0.1178, validation accuracy 0.9593
Epoch 16: train loss = 0.1899, validation loss = 0.1320, validation accuracy 0.9593
Epoch 17: train loss = 0.1748, validation loss = 0.1615, validation accuracy 0.9611
Epoch 18: train loss = 0.1644, validation loss = 0.1624, validation accuracy 0.9556
early stopping...
Loading the best model from checkpoint...
Test loss: 0.10726522915065288, Test accuracy: 0.9666666666666667
```

Παρατηρούμε ότι πετυχαίνουμε αρκετά καλό accuracy στο test set και βελτιωμένο σε σχέση με τα προηγούμενα ερωτήματα. Επίσης εμφανίζεται το early stopping έπειτα από την εποχή με index 18.

7. Ένα Bidirectional LSTM είναι μια εκδοχή του LSTM όπου η πληροφορία ρέει τόσο προς τα εμπρός όσο και προς τα πίσω. Αυτό σημαίνει ότι το δίκτυο μπορεί να «θυμάται» όχι μόνο το παρελθόν, αλλά και να εξετάζει το μέλλον σε κάθε χρονική στιγμή. Αυτή η αμφίδρομη ροή πληροφορίας ενισχύει την ικανότητα του μοντέλου να αναγνωρίζει πρότυπα, ιδίως σε προβλήματα όπου το πλαίσιο πριν και μετά έχει σημασία, όπως σε επεξεργασία φωνής και φυσικής γλώσσας.

Θέτουμε τώρα την παράμετρο bidirectinal = True. Τα αποτελέσματα φαίνονται παρακάτω:

```
Epoch 0: train loss = 2.0091, validation loss = 1.2420, validation accuracy 0.6500
Epoch 1: train loss = 0.9187, validation loss = 0.5433, validation accuracy 0.8426
Epoch 2: train loss = 0.4300, validation loss = 0.2173, validation accuracy 0.9519
Epoch 3: train loss = 0.3124, validation loss = 0.1724, validation accuracy 0.9500
Epoch 4: train loss = 0.2554, validation loss = 0.1257, validation accuracy 0.9667
Epoch 5: train loss = 0.1608, validation loss = 0.1132, validation accuracy 0.9574
Epoch 6: train loss = 0.1403, validation loss = 0.0891, validation accuracy 0.9667
Epoch 7: train loss = 0.1190, validation loss = 0.0674, validation accuracy 0.9796
Epoch 8: train loss = 0.1139, validation loss = 0.1008, validation accuracy 0.9611
Epoch 9: train loss = 0.1103, validation loss = 0.0714, validation accuracy 0.9833
Epoch 10: train loss = 0.0828, validation loss = 0.0505, validation accuracy 0.9852
Epoch 11: train loss = 0.0795, validation loss = 0.0478, validation accuracy 0.9870
Epoch 12: train loss = 0.0804, validation loss = 0.0611, validation accuracy 0.9778
Epoch 13: train loss = 0.1022, validation loss = 0.0951, validation accuracy 0.9722
Epoch 14: train loss = 0.0767, validation loss = 0.0574, validation accuracy 0.9796
early stopping...
Loading the best model from checkpoint...
Test loss: 0.028032400365918874, Test accuracy: 0.9933333333333333
```

Παρατηρούμε ότι σε σχέση με τα προηγούμενα ερωτήματα τα training loss και validation loss μειώνονται πολύ πιο γρήγορα όπως επίσης και το test accuracy αυξάνεται πολύ πιο γρήγορα. Επίσης όπως θα ήταν λογικό σύμφωνα με τα παραπάνω, το μοντέλο σταματάει με το early stopping μετά την εποχή με index 14. Με την συγκεκριμένη αρχιτεκτονική πετυχαίνουμε το εντυπωσιακό Test accuracy: 0.9933333333333333.

8. (Bonus) Φτιάχνουμε την κλάση PackedLSTM η οποία υλοποιεί την αρχιτεκτονική που ζητείται. Εκπαιδεύουμε το μοντέλο και τα αποτελέσματα φαίνονται παρακάτω:

```
Epoch 0: train loss = 2.0494, validation loss = 1.5404, validation accuracy 0.4778
Epoch 1: train loss = 1.2500, validation loss = 0.8899, validation accuracy 0.7259
Epoch 2: train loss = 0.7599, validation loss = 0.5318, validation accuracy 0.8315
Epoch 3: train loss = 0.5351, validation loss = 0.3338, validation accuracy 0.9167
Epoch 4: train loss = 0.3497, validation loss = 0.2762, validation accuracy 0.9148
Epoch 5: train loss = 0.2837, validation loss = 0.2021, validation accuracy 0.9444
Epoch 6: train loss = 0.2352, validation loss = 0.2906, validation accuracy 0.8981
Epoch 7: train loss = 0.2403, validation loss = 0.1597, validation accuracy 0.9537
Epoch 8: train loss = 0.1841, validation loss = 0.1714, validation accuracy 0.9537
Epoch 9: train loss = 0.1350, validation loss = 0.1746, validation accuracy 0.9407
Epoch 10: train loss = 0.1516, validation loss = 0.1396, validation accuracy 0.9500
Epoch 11: train loss = 0.1389, validation loss = 0.1184, validation accuracy 0.9722
Epoch 12: train loss = 0.0938, validation loss = 0.0691, validation accuracy 0.9796
Epoch 13: train loss = 0.1035, validation loss = 0.0803, validation accuracy 0.9815
Epoch 14: train loss = 0.0786, validation loss = 0.0955, validation accuracy 0.9667
Epoch 15: train loss = 0.0803, validation loss = 0.0705, validation accuracy 0.9759
early stopping...
Loading the best model from checkpoint...
Test loss: 0.05857843463309109, Test accuracy: 0.9833333333333333
```

Παρατηρούμε ότι το μοντέλο είναι πιο αργό σε σχέση με τα προηγούμενα ερωτήμαρα. Αυτό οφείλεται μάλλον στο γεγονός ότι έχει αυξηθεί το μήκος των περισσότερων ακολουθιών κι έτσι το μοντέλο χρειάζεται περισσότερη ώρα για να τις επεξεργαστεί. Παρατηρούμε ότι και πάλι το μοντέλο πετυχαίνει πολύ καλό accuracy.

Για Bidirectional BasicLSTM κάνουμε grid search για τις παρακάτω παραμέτρους:

```
batch_sizes = [16, 32]
learning_rates = [1e-3, 1e-4]
dropouts = [0.3, 0.5]
weight_decays = [0.01, 0.001]
```

Τα αποτελέσματα φαίνονται παρακάτω:

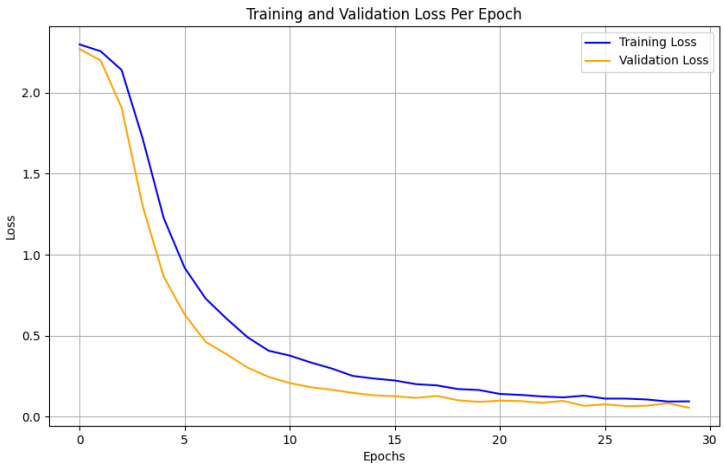
Table 3: Grid Search Accuracy for Different Hyperparameter Combinations

Batch Size	Learning Rate	Dropout	Weight Decay	Validation Accuracy
16	1e-3	0.3	0.01	0.9796
16	1e-3	0.3	0.001	0.9870
16	1e-3	0.5	0.01	0.9852
16	1e-3	0.5	0.001	0.9852
16	1e-4	0.3	0.01	0.9870
16	1e-4	0.3	0.001	0.9852
16	1e-4	0.5	0.01	0.9907
16	1e-4	0.5	0.001	0.9796
32	1e-3	0.3	0.01	0.9852
32	1e-3	0.3	0.001	0.9870
32	1e-3	0.5	0.01	0.9833
32	1e-3	0.5	0.001	0.9778
32	1e-4	0.3	0.01	0.9870
32	1e-4	0.3	0.001	0.9778
32	1e-4	0.5	0.01	0.9796
32	1e-4	0.5	0.001	0.9778

Ο καλύτερος συνδυασμός παραμέτρων με βάση τα παραπάνω είναι:

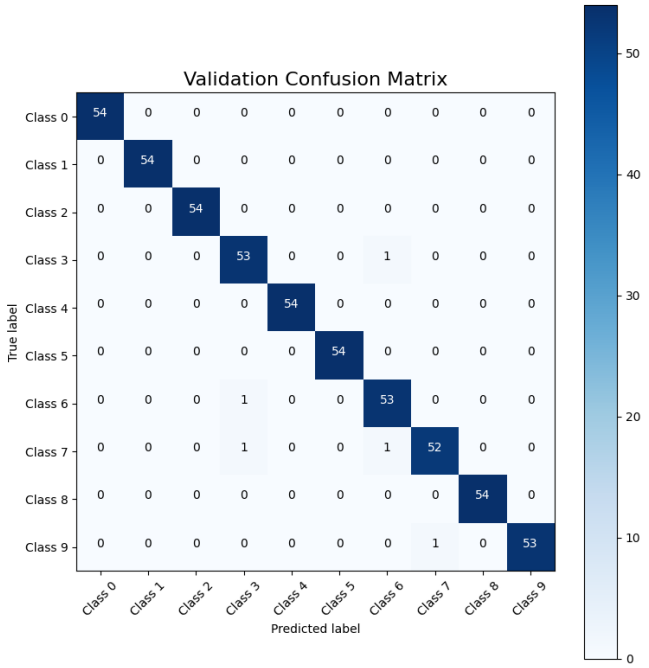
```
{'batch_size': 16, 'learning_rate': 0.0001, 'dropout': 0.5, 'weight_decay': 0.01}
```

Το διάγραμμα του loss στο validation set και στο test set:

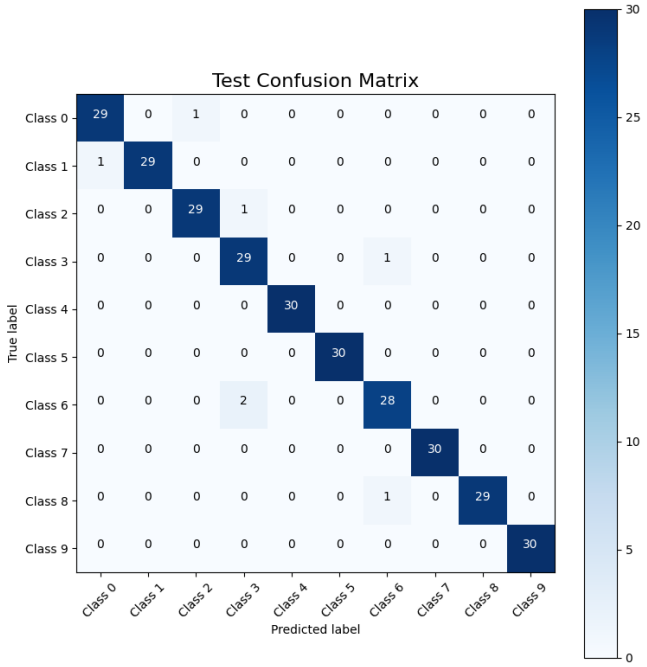


Παρατηρούμε ότι το loss στο validation και στο test set εχει αρκετά παρόμοια συμπεριφορά. Στις αρχικές εποχές το loss παρουσιάζει μεγάλες μεταβολές ενώ στην συνέχεια σταθεροποιείται σε χαμηλές τιμές.

Best Parameters Validation Set Accuracy: 0.9907
Best Parameters Validation Set Confusion Matrix:



Best Parameters Test Set Accuracy: 0.9767
Best Parameters Test Set Confusion Matrix:


Test Confusion Matrix
True label
Predicted label
Class 0
Class 1
Class 2
Class 3
Class 4
Class 5
Class 6
Class 7
Class 8
Class 9
29 0 1 0 0 0 0 0 0 0
1 29 0 0 0 0 0 0 0 0
0 0 29 1 0 0 0 0 0 0
0 0 0 29 0 0 1 0 0 0
0 0 0 0 30 0 0 0 0 0
0 0 0 0 0 30 0 0 0 0
0 0 0 2 0 0 28 0 0 0
0 0 0 0 0 0 0 30 0 0
0 0 0 0 0 0 1 0 29 0
0 0 0 0 0 0 0 0 0 30
30
25
20
15
10
5
0