ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ & ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΑΝΑΓΝΩΡΙΣΗ ΠΡΟΤΥΠΩΝ
Χειμερινό Εξάμηνο Ε.ΔΕ.Μ.Μ. 2022-23

Προπαρασκευή 2ης Εργαστηριακής Άσκησης:

Αναγνώριση φωνής με Κρυφά Μαρκοβιανά Μοντέλα και Αναδρομικά Νευρωνικά Δίκτυα

Κοτσιφάκου Κοντιλένια Μαρία 03400174

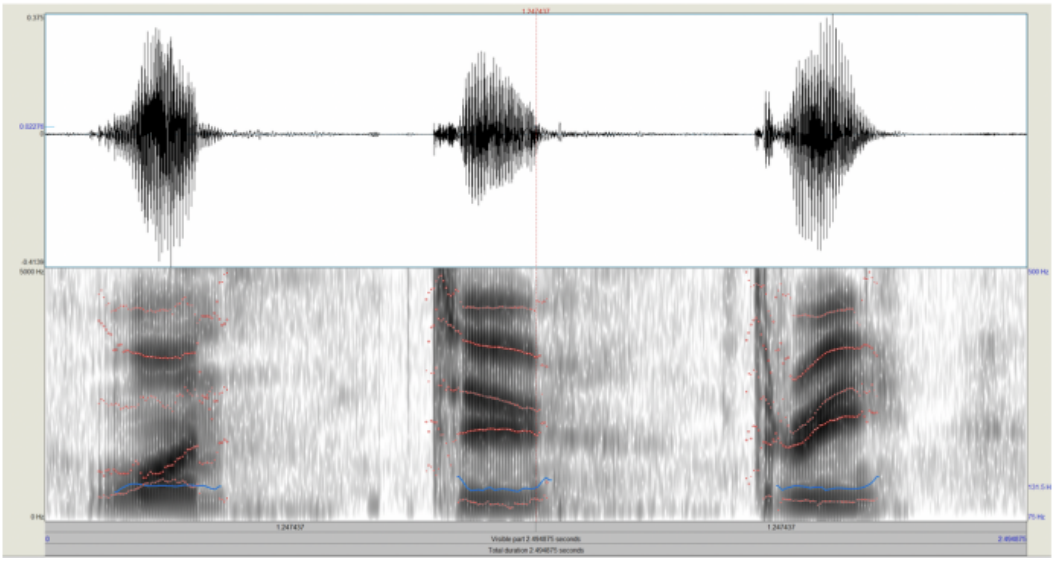
Παπακωνσταντίνου Άννα 03400187

Βήμα 1

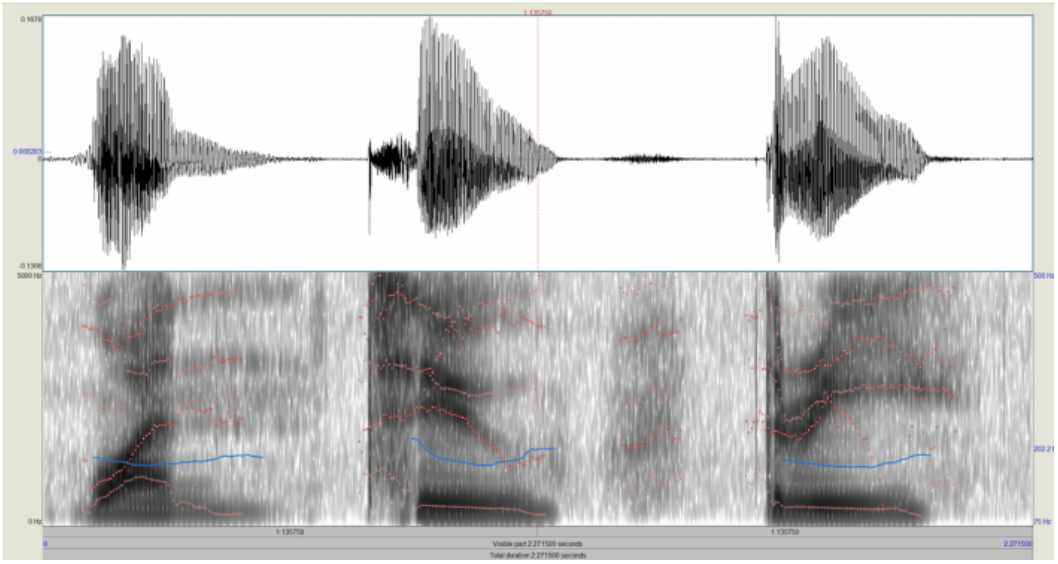
Στα πλαίσια της δεύτερης εργαστηριακής άσκησης αντικείμενο ενασχόλησης είναι η αναγνώριση ακουστικών σημάτων. Συγκεκριμένα, στην διάθεσή μας έχουμε εκφωνήσεις ψηφίων από το 0 έως και το 9 και σκοπός μας είναι να αναπτύξουμε ένα σύστημα ικανό να αναγνωρίσει και στην συνέχεια να ταξινομήσει τα ακουστικά αυτά ψηφία στην σωστή κατηγορία.

Αρχικά πραγματοποιούμε επεξεργασία των αρχείων onetwothree1.wav και onetwothree8.wav με χρήση του προγράμματος Praat. Τα αρχεία αυτά περιλαμβάνουν την ηχητική προσφώνηση της πρότασης “one two three” από έναν άνδρα και από μια γυναίκα ομιλητή αντίστοιχα. Μετά την εισαγωγή των ηχητικών αρχείων στο σύστημα Praat παρατηρούμε την κυματομορφή και το spectrogram κάθε ηχητικού σήματος:

- Για τον άνδρα ομιλητή (onetwothree1.wav)



- Για την γυναίκα ομιλητή (onetwothree8.wav)



Παρατηρούμε ότι το σήμα από την γυναίκα ομιλητή έχει μεγαλύτερο πλάτος και πιο μεγάλες συχνότητες. Στην συνέχεια εξάγουμε τη μέση τιμή του pitch και τη μέση συχνότητα πρώτου (F1), δεύτερου(F2) και τρίτου(F3) Formant στα φωνήεντα “α”, “ου”, “ι” για τα 3 ψηφία και για κάθε ομιλητή. Τα φωνήματα αυτά είναι περιοδικά και επομένως μπορούμε να τα βρούμε είτε ακούγοντας το ηχητικό αρχείο είτε βλέποντας που γίνεται περιοδική η κυματομορφή είτε συνδυάζοντας και τις δύο μεθόδους. ΄Εχουμε τα εξής αποτελέσματα:

Ομιλητής	**Φωνήεν**	**Χρονική Διάρκεια (sec)**	**Μέση Τιμή Pitch (Hz)**	**Formant 1(Hz)**	**Formant 2(Hz)**	**Formant 3(Hz)**
άντρας	α	0.299 - 0388	134.675	757.513	1249.326	2413.343
γυναίκα	α	0.167 - 0.285	177.849	880.211	1571.909	3004.935
άντρας	ου	1.056 - 1.273	130.710	361.707	1785.622	2388.273
γυναίκα	ου	0.861 - 1.184	188.658	347.835	1722.251	2686.739
άντρας	ι	1.929 - 2.126	132.670	399.743	1984.117	2423.754
γυναίκα	ι	1.741 - 2.034	179.282	396.339	2178.810	1858.908

Από τα παραπάνω μπορούμε να δούμε πως το pitch εξαρτάται από τον ομιλητή. Είναι δηλαδή πολύ κοντά σε τιμές για το κάθε φωνήεν. ΄Εχουμε, λοιπόν:

- Για τον άντρα ομιλητή: pitch mean ≈ 132Hz
- Για την γυναίκα ομιλητή: pitch mean ≈ 181Hz

Γενικά παρατηρούμε ότι τα formants επηρεάζονται μεν από τον ομιλητή (κυρίως από το αν είναι άντρας ή γυναίκα αλλά όχι μόνο), εξαρτώνται όμως κυρίως από το φωνήεν (παρατηρούμε ότι τα formants για το ίδιο φωνήεν είναι παρόμοια και για τους δύο ομιλητές, αλλά τα formants για διαφορετικό φωνήεν διαφέρουν σημαντικά, ιδιαίτερα αυτά για το φωνήεν “α”). Παρατηρούμε επίσης ότι τα formants του φωνήεντος “α” διαφέρουν αρκετά από αυτά των “ι” και “ου”, ενώ τα formants των δύο τελευταίων φωνηέντων είναι πιο κοντά. Αυτό, όπως διαπιστώνουμε από σχετικές μελέτες των formants στην βιβλιογραφία ήταν αναμενόμενο. Τέλος, αξίζει να σημειώσουμε πως, καθώς η επιλογή των τμημάτων του ηχητικού αρχείου που αποτελούν το κάθε φωνήεν σε μια λέξη έγινε χειροκίνητα, τα μετρούμενα pitch και formants μπορεί να προκύπτουν ελαφρώς διαφορετικά σε διαφορετικές μετρήσεις.

Για την εκτέλεση της άσκησης χρησιμοποιήθηκε το αρχείο parser.py που είναι ανεβασμένο στο github του μαθήματος. Στην συνέχεια έχουμε διαφοροποιήσει τις υπάρχουσες συναρτήσεις και έχουμε εμπλουτίσει περαιτέρω τον κώδικα.

Βήμα 2

Για την εισαγωγή όλων των αρχείων ήχου που δίνονται μέσα στον φάκελο digits, αρχικά καλείται η συνάρτηση *parser* όπου με την σειρά της καλεί την συνάρτηση *parse_free_digits* που καλύπτει τα πλαίσια αυτού του βήματος.

```
# Step 2 and 3
frames, wavs, speakers, ids, all_digits, Fs = parser("digits")
```

Εικόνα 2.1. *Κλήση συνάρτησης*

```
def parser(directory, n_mfcc=13):
    wavs, Fs, ids, y, speakers = parse_free_digits(directory)
    frames, _, _ = extract_features(wavs, n_mfcc=n_mfcc, Fs=Fs)
    return frames, wavs, speakers, ids, y, Fs
```

Εικόνα 2.2. *Συνάρτηση διαχείρισης αρχείων ήχου και παραγωγής χαρακτηριστικών*

Η συνάρτηση *parse_free_digits* επιστρέφει πέντε δομές, λίστες και πίνακα.
Η μεταβλητή *wavs,* με βάση την Εικόνα 2.2, αντικατοπτρίζει μία λίστα με αριθμό θέσεων ίσο με το συνολικό αριθμό διαφορετικών αρχείων ήχου ψηφίων, όπου κάθε στοιχείο της λίστας είναι ένας μονοδιάστατος πίνακας που περιέχει audio time series έτσι όπως προέκυψε με τη χρήση της συνάρτησης *librosa.core.load()* . Μέσω της μεταβλητής *Fs* επιστρέφεται η συχνότητα δείγματος του πρώτου αρχείου η οποία υποθέτουμε πως είναι ίδια και για τα υπόλοιπα. Ακόμα μέσω της speakers, επιστρέφεται μία λίστα με τον αριθμό του εκφωνητή που προφέρει το κάθε ψηφίο. Οι εκφωνητές είναι δεκαπέντε συνεπώς είναι λογικό ένας εκφωνητής να αντιστοιχίζεται σε περισσότερα ψηφία. Τέλος επιστρέφονται και τα *ids* των speakers που πρακτικά πρόκειται για έναν αύξοντα αριθμό από 0 εώς 132.
Αναλυτικότερα, το όρισμα της συνάρτησης *parse_free_digits* είναι το relative path του φακέλου που περιέχει τα δεδομένα- αρχεία ήχου, τα οποία έχουν περάσει ως όρισμα της συνάρτησης *parser ("digits").* Έπειτα διαχωρίζονται τα ονόματα όλων των αρχείων που περιέχονται στο φάκελο σε εκφωνηθεμένο ψηφίο (με γράμματα π.χ. eight) και εκφωνητή (σε ακέραιο αριθμό).

```
def parse_free_digits(directory):
    # Parse relevant dataset info
    files = glob(os.path.join(directory, "*.wav"))
    f_names2 = [[f.split("\\")[1].split(".")[0]] for f in files]
    all_splits = [split_name_number(i[0]) for i in f_names2]
    digits, speakers = list(map(list, zip(*all_splits)))
    digits = [w2n.word_to_num(i) for i in digits]
    ids = [i for i in range(len(speakers))]
    _, Fs = librosa.core.load(files[0], sr=None)

    def read_wav(f):
        wav, _ = librosa.core.load(f, sr=None)
        return wav

    # Read all wavs
    wavs = [read_wav(f) for f in files]

    # Print dataset info
    print("Total wavs: {}. Fs = {} Hz".format(len(wavs), Fs))

    return wavs, Fs, ids, digits, speakers
```

Εικόνα 2.3. *Συνάρτηση διαχείρισης αρχείων ήχου*

Για τον διαχωρισμό των ψηφίων από τους ομιλητές δημιουργήθηκε η συνάρτηση *split_name_number*. Η συγκεκριμένη συνάρτηση παίρνει σαν όρισμα ένα string και το χωρίζει σε 2 μέρη head και tail. Στο μεν head αποθηκεύεται το string εκτός των χαρακτήρων των ψηφίων 0-9, δηλαδή πρακτικά η λέξη του ψηφίου (π.χ. eight15 -> head=eight). Τέλος στο tail αποθηκεύεται καθαρά ο ακέραιος αριθμός του εκφωνητή.

```
def split_name_number(s):
    head = s.rstrip('0123456789')  # at the end cut the numbers
    tail = int(s[len(head):])  # whatever remains
    return head, tail
```

Εικόνα 2.4. *Διαχωρισμός λέξης και ψηφίου*

Βήμα 3

Ζητούμενο:

Εξάγετε με το librosa τα Mel-Frequency Cepstral Coefficients (MFCCs) για κάθε αρχείο ήχου. Εξάγετε 13 χαρακτηριστικά ανά αρχείο. Χρησιμοποιήστε μήκος παραθύρου 25 ms και βήμα 10 ms. Επίσης, υπολογίστε και την πρώτη και δεύτερη τοπική παράγωγο των χαρακτηριστικών, τις λεγόμενες deltas και delta-deltas (hint: υπάρχει έτοιμη υλοποίηση στο librosa).

Η συνάρτηση *parser* καλεί στην συνέχεια την συνάρτηση extract_features με όρισμα τα ηχητικά δεδομένα wavs, τον αριθμό των επιθυμητών εξαγόμενων χαρακτηριστικών και την συχνότητα δειγματοληψίας. Με τα κατάλληλα ορίσματα, καλούνται οι έτοιμες συναρτήσεις από την librosa library, mfcc και delta χρησιμοποιώντας μήκος παραθύρου 25 ms και βήμα 10 ms, ώστε να παραχθούν και να επιστραφούν τα *MFCCs, deltas* και *delta-deltas*.

Το γεγονός ότι η απόσταση των κέντρων των παραθύρων είναι μικρότερη του μήκους αυτών συνεπάγεται την ύπαρξη επικαλύψεων του αρχικού σήματος από γειτονικά παράθυρα. Έχοντας χωρίσει το σήμα σε παράθυρα εφαρμόζεται σε αυτά Discrete Fourier Transformation (DFT) προκειμένου να μεταφερθούμε στο πεδίο της συχνότητας, ώστε τελικά να λάβουμε τον λογάριθμο του φάσματος του ηχητικού σήματος. Στην συνέχεια, μετασχηματίζεται ο λογάριθμος τους φάσματος κάθε παραθύρου με βάση την κλίμακα MEL , κλίμακα ευαισθησίας του ανθρώπινου αυτιού στις διαφορετικές συχνότητες. Τέλος, στα μετασχηματισμένα λογαριθμικά φάσματα εφαρμόζεται αντίστροφος DCT προκειμένου να επιστρέψουμε στον πεδίο του χρόνου και να καταλήξουμε με τα λεγόμενα MFCCs -Mel Filter Cepstral Coefficients του σήματος.

```python
def extract_features(wavs, n_mfcc=13, Fs=16000):
    # Extract MFCCs for all wavs
    window = Fs * 25 // 1000
    step = Fs * 10 // 1000
    mfcc_frames = [
        librosa.feature.mfcc(
            y=wav, sr=Fs, n_fft=window, hop_length=window - step, n_mfcc=n_mfcc
        )

        for wav in tqdm(wavs, desc="Extracting mfcc features...")
    ]
    deltas_frames = [
        librosa.feature.delta(mfcc_f)

        for mfcc_f in tqdm(mfcc_frames, desc="Extracting delta features...")
    ]
    delta_deltas_frames = [
        librosa.feature.delta(mfcc_f, order=2)

        for mfcc_f in tqdm(mfcc_frames, desc="Extracting delta deltas features...")
    ]
    print("Feature extraction completed with {} mfccs per frame".format(n_mfcc))
    return mfcc_frames, deltas_frames, delta_deltas_frames
```

Εικόνα 3.1. *Εξαγωγή MFCCs, deltas, delta-deltas*

Βήμα 4

Στην συνέχεια, συλλέγονται τα MFCCs των αριθμών 4 και 7. Αυτό γίνεται σε πρώτο στάδιο με την βοήθεια της συνάρτησης *find_digits()* και μετέπειτα με την χρήση της συνάρτησης *plot_hist()* αποτυπώνονται τα ιστογράμματα.

```python
# step 4
mfcs = []  # list for Mel Filter-bank Spectral Coefficients
mfcc = []  # list for Mel-Frequency Cepstral Coefficients
n1 = 4  # define n1
n2 = 7  # define n2
n1_list, n2_list = find_digits(all_digits, n1, n2)  # func to find the indexes of all n1 and n2
plot_hist(mfccs_all, n1_list, n2_list)
```

Εικόνα 4.1. *Ιστογράμματα του 1ου και του 2ου MFCC των ψηφίων 4 και 7*

Η find_digits() χρησιμοποιείται για την αναζήτηση των συγκεκριμένων ψηφίων, 4 και 7. Παίρνει ως όρισμα το σύνολο όλων των διαθέσιμων ψηφίων βάσει των αρχείων ήχου που διαβάστηκαν, καθώς επίσης και τον αριθμό των δύο ψηφίων που αναζητούμε (n1 και n2).

```python
def find_digits(all_digits, n1, n2):
    n1_indexes = []
    n2_indexes = []
    n1_indexes = [i for i in range(len(all_digits)) if all_digits[i] == n1]
    n2_indexes = [i for i in range(len(all_digits)) if all_digits[i] == n2]
    return n1_indexes, n2_indexes
```

Εικόνα 4.2. *Αναζήτηση indexes των ψηφίων 4 και 7*

Τελικά η *find_digits()* επιστρέφει δύο λίστες *n1_indexes* και *n2_indexes* με όλα τα indexes για κάθε ένα ψηφίο αντίστοιχα.
Με βάση τα indexes αυτά, χρησιμοποιείται η συνάρτηση *plot_hist()*. Η *plot_hist()* παίρνει ως όρισμα εκτός των indexes την λίστα με τα MFCCs έτσι όπως επιστρέφεται από την *extract_features()*, δηλαδή μία λίστα όπου κάθε στοιχείο της είναι ένας πίνακας για κάθε ένα ψηφίο ο οποίος περιέχει τα 13 MFCCs για κάθε window. Βάσει των indexes η συνάρτηση απομονώνει απο την λίστα των MFCCs εκείνα που αντιστοιχίζονται στα ψηφία 4 και 7. Έπειτα συνενώνει για κάθε ένα από τα δύο ψηφία 4 και 7 τα πρώτα δύο MFCCs από τα συνολικά δεκατρία σε ένα πίνακα τον οποίο και χρησιμοποιεί τελικά για την απεικόνιση του ιστογράμματος.

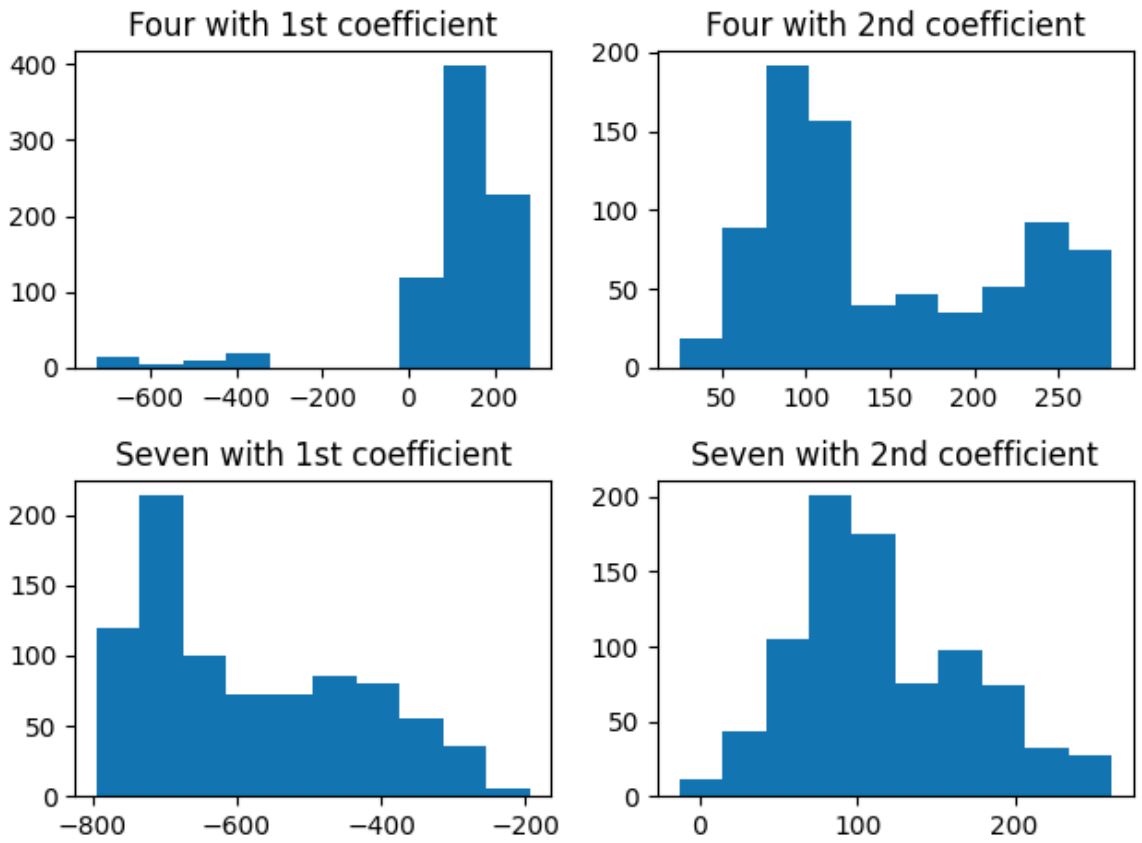
```
def plot_hist(mfcc, n1_list, n2_list):
    n1_0 = mfcc[n1_list[0]][:][0]  # from each n1 takes the first coefficient
    n1_1 = mfcc[n1_list[0]][:][1]  # from each n1 takes the second coefficient
    n2_0 = mfcc[n2_list[0]][:][0]  # from each n2 takes the first coefficient
    n2_1 = mfcc[n2_list[0]][:][1]  # from each n2 takes the second coefficient
    for i in range(1, len(n1_list)):
        n1_0 = np.concatenate((n1_0, mfcc[n1_list[i]][:][0]),
                              axis=0)  # make a union array with all features of the first coefficients of n1
        n1_1 = np.concatenate((n1_1, mfcc[n1_list[i]][:][1]),
                              axis=0)  # make a union array with all features of the second coefficients of n1

    for i in range(1, len(n2_list)):
        n2_0 = np.concatenate((n2_0, mfcc[n2_list[i]][:][0]),
                              axis=0)  # make a union array with all features of the first coefficients of n2
        n2_1 = np.concatenate((n2_1, mfcc[n2_list[i]][:][1]),
                              axis=0)  # make a union array with all features of the second coefficients of n2
```

Εικόνα 4.3(α). *Κώδικας εκτύπωσης ιστογραμμάτων*

```
fig, ((ax0, ax1), (ax2, ax3)) = plt.subplots(nrows=2, ncols=2)
ax0.hist(n1_0.T, histtype='bar')
ax0.set_title('Four with 1st coefficient')
ax1.hist(n1_1.T, histtype='bar')
ax1.set_title('Four with 2nd coefficient')
ax2.hist(n2_0.T, histtype='bar')
ax2.set_title('Seven with 1st coefficient')
ax3.hist(n2_1.T, histtype='bar')
ax3.set_title('Seven with 2nd coefficient')
fig.tight_layout()
plt.show()
```

Εικόνα 4.3(β). *Κώδικας εκτύπωσης ιστογραμμάτων*

Συγκρίνοντας τα ιστογράμματα της πρώτης coefficient των MFCCs με της δεύτερης coefficient (Εικόνα 4.4) βλέπουμε ότι παρουσιάζουν διαφορές αρχικά ως προς το σχήμα του ιστογράμματος αλλά και ως προς τις τιμές καθώς η πρώτη MFCC κυμαίνεται σε αρνητικές τιμές και συγκεκριμένα μεταξύ [-800,-200] ενώ η δεύτερη σε θετικές τιμές και συγκεκριμένα μεταξύ [0, 300]. Πιο συγκεκριμενα παρατηρείται πως για τη πρώτη συνιστώσα, τα ψηφία 4 και 7 έχουν αρκετά διαφορετική δομή με το μεν 4 να έχει συγκέντρωση πιο πυκνή στο εύρος [0,200] και το μεν 7 στο διάστημα [-800,-600]. Στη περίπτωση της δεύτερης συνιστώσας ωστόσο υπάρχουν περισσότερες ομοιότητες. Εν γένει δε φαίνεται να μπορούμε να αποφανθούμε κάτι από τα συγκεκριμένα διαγράμματα ωστόσο το ότι τα 2 ψηφία δεν έχουν πολλές ομοιότητες μεταξύ τους θα βοηθήσει στην μετέπειτα ανάλυση και εκπαίδευση του μοντέλου.



Εικόνα 4.4. *Εκτύπωση ιστογραμμάτων*

Στην συνέχεια χρησιμοποιείται η συνάρτηση *digits_and_speakers()* με ορίσματα τα indexes των ψηφίων 4 και 7. Η *digits_and_speakers()* επιλέγει τυχαία δύο διαφορετικούς εκφωνητές για δύο εκφωνήσεις των ψηφίων (Εικόνα 4.5). Με χρήση των indexes των εκφωνητών, θα εξαχθούν δεκατρία Mel Filterbank Spectral Coefficients (MFSCs), δηλαδή χαρακτηριστικά που εξάγονται αφού εφαρμοστεί η συστοιχία φίλτρων της κλίμακας Mel πάνω στο φάσμα του σήματος φωνής αλλά χωρίς να εφαρμοστεί στο τέλος ο μετασχηματισμός DCT. Για τον υπολογισμό αυτών των χαρακτηριστικών χρησιμοποιείται η συνάρτηση melspectrogram() η οποία είναι βασισμένη στην βιβλιοθήκη librosa (Εικόνα 4.6).

```
def digits_and_speakers(n1_list, n2_list):

    random.seed(10)
    speaker1_n1, speaker2_n1 = random.sample(n1_list, 2)
    speaker1_n2, speaker2_n2 = random.sample(n2_list, 2)
    return speaker1_n1, speaker1_n2, speaker2_n1, speaker1_n2
```

Εικόνα 4.5. *Κώδικας επιλογής εκφωνητών*

```
def melspecrogram(wavs, Fs, window, step):
    melspec = librosa.feature.melspectrogram(y=wavs, sr=Fs, n_fft=window, hop_length=window - step, n_mels=13)
    return melspec
```

Εικόνα 4.6. *Κώδικας εξαγωγής MFSCs*

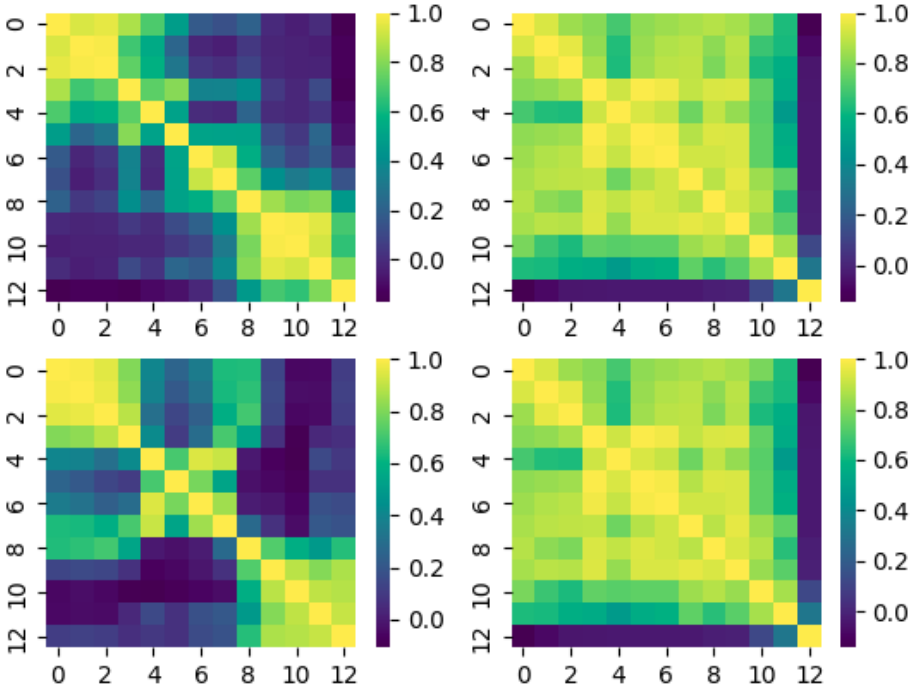
Συνεπώς για κάθε ένα εκφωνητή και εκφώνημα καλείται η *melspectrogram()* και επιστρέφει τον πίνακα των χαρακτηριστικών μεγέθους (windows x 13) ο οποίος καταχωρείται σε μία λίστα *mfsc*. Τέλος με χρήση της συνάρτησης *corrcoef()* της βιβλιοθήκης numpy και παράμετρο εισόδου κάθε ένα στοιχείο

της λίστας *mfsc*, επιστρέφονται οι συσχετίσεις μεταξύ των χαρακτηριστικών οι οποίες και χρησιμοποιούνται για την αποτύπωση των heatmaps. Ο κώδικας περιγράφεται στην Εικόνα 4.7.

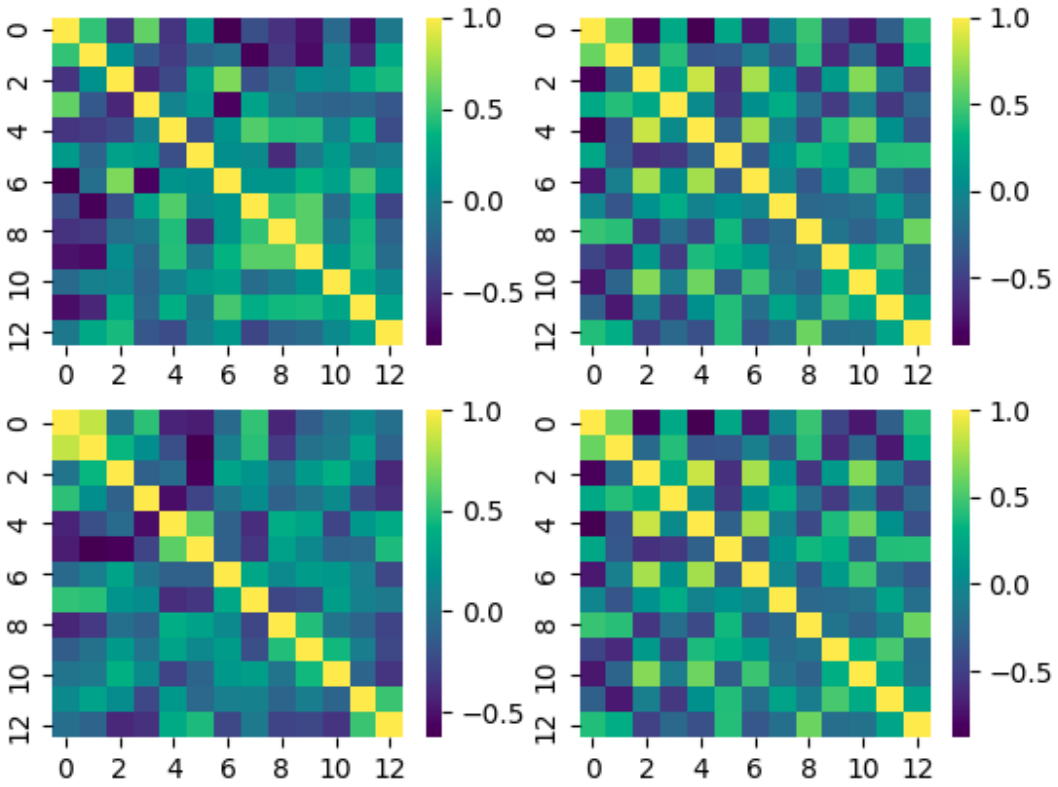
```
# speaker1_n1, speaker1_n2, speaker2_n1, speaker2_n2 = digits_and_speakers(n1_list, n2_list, speakers)
speaker1_n1, speaker1_n2, speaker2_n1, speaker2_n2 = digits_and_speakers(n1_list, n2_list)
speak = [speaker1_n1, speaker1_n2, speaker2_n1, speaker2_n2]
# MFSC calculations and heatmaps
window = Fs * 25 // 1000
step = Fs * 10 // 1000
for i in speak:
    mfcs.append(melspecrogram(wavs[i], Fs, window, step))
    mfcc.append(mfccs_all[i])
fig, ((ax0, ax1), (ax2, ax3)) = plt.subplots(nrows=2, ncols=2)
ax1 = [ax0, ax1, ax2, ax3]
for i in range(len(ax1)):
    sn.heatmap(np.corrcoef(mfcs[i]), cmap='viridis', ax=ax1[i])
fig, ((ax0, ax1), (ax2, ax3)) = plt.subplots(nrows=2, ncols=2)
ax = [ax0, ax1, ax2, ax3]
for i in range(len(ax)):
    sn.heatmap(np.corrcoef(mfcc[i]), cmap='viridis', ax=ax[i])
plt.show()
```

Εικόνα 4.7. *Κώδικας δημιουργίας διαγραμμάτων συσχέτισης*

Να σημειωθεί πως τα heatmaps αποτυπώνονται τόσο για τα MFSC όσο και για τα MFCC για τη μεταξύ τους σύγκριση.



Εικόνα 4.8. *Διαγράμματα συσχετίσεων MFSCs*



Εικόνα 4.9. *Διαγράμματα συσχετίσεων MFCCs*

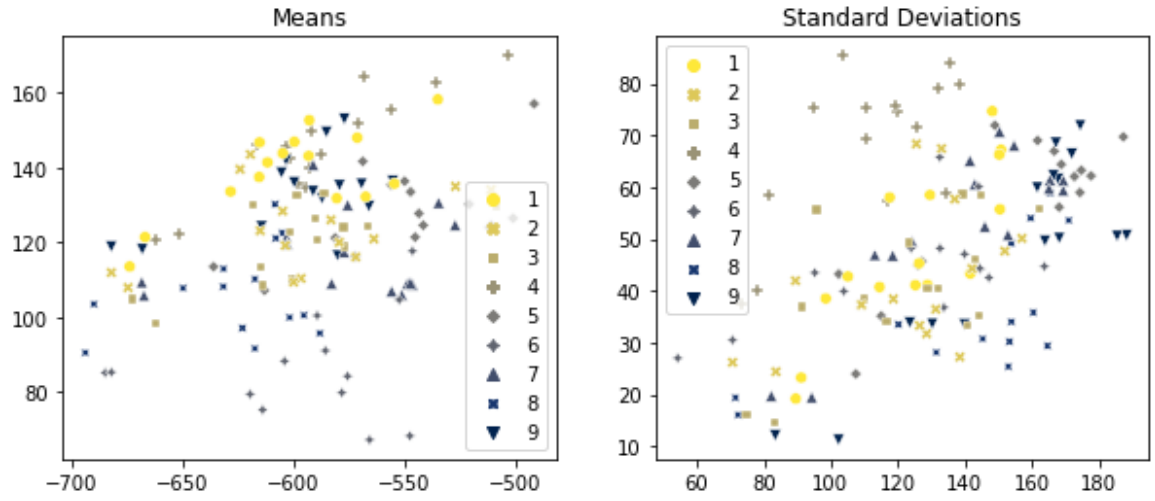
Όσον αφορά τα MFSC / MFCC, Παρατηρούμε πως τα MFSCs έχουν πολύ μεγαλύτερη συσχέτιση ενώ η εφαρμογή του DCT παράγει ασυσχέτιστα μεταξύ τους χαρακτηριστικά και εξαλείφει αυτό το πρόβλημα. Τα ασυσχέτιστα χαρακτηριστικά είναι προτιμότερα για τα μοντέλα HMM. Για deep learning μπορούμε να χρησιμοποιήσουμε και τα MFCCs και μάλιστα να έχουμε καλύτερα αποτελέσματα γιατί δεν χάνεται πληροφορία από το DCT.

Βήμα 5

Για την αναγνώριση των ψηφίων απαραίτητη είναι η εξαγωγή ενός μοναδικού διανύσματος χαρακτηριστικών για κάθε εκφώνηση. Για το σκοπό αυτό χρησιμοποιείται η συνάρτηση *analysis()* η οποία συγκεκριμένα δημιουργείται για κάθε εκφώνημα, ένα διάνυσμα με 78 στήλες, 39 (3x13) στήλες με τις μέσες τιμές και άλλες 39 με την τυπική απόκλιση για όλα τα windows των mfccs – deltas – delta-deltas με τη χρήση της εντολής concatenate.
Στη συνέχεια επιλέγονται οι πρώτες δύο διαστάσεις του διανυσματος που αντιστοιχούν στις πρώτες δύο μέσες τιμές και καθώς επίσης και οι δύο πρώτες τυπικές αποκλίσεις για κάθε ψηφίο, και αποτυπώνεται το διάγραμμα διασποράς τους.

```python
def analysis(mfccs, deltas, deltas2, digits):

    features = np.zeros((len(wavs), 78))
    # for i in range(133):
    for i in range(len(wavs)):  # more general code
        features[i, :] = np.concatenate(
            (mfccs[i].mean(axis=1), deltas[i].mean(axis=1), deltas2[i].mean(axis=1), mfccs[i].std(axis=1),
             deltas[i].std(axis=1), deltas2[i].std(axis=1)), axis=0)
        # πίνακας 133 σειρών όπου κάθε σειρά αντιπροσωπεύει μία εκφώνηση
        # και κάθε σειρά αποτελείται από 2x39 = 78 στήλες: μέση τιμή των 13 mfcc + deltas +deltas2 + αντίστοιχες stds
    fig, axs = plt.subplots(ncols=2, figsize=(10, 4))
    scatter(features[:, 0], features[:, 1], digits, 'Means', ax=axs[0])
    scatter(features[:, 39], features[:, 40], digits, 'Standard Deviations', ax=axs[1])
    plt.show()
    return features
```

Εικόνα 5.1 *Κώδικας δημιουργίας μοναδικού διανύσματος χαρακτηριστικών κάθε εκφωνήματος*



Εικόνα 5.2 *Διαγράμματα διασποράς των 2 πρώτων διαστάσεων των διανυσμάτων χαρακτηριστικών*

Παρατηρείται πως σε όλες τις περιπτώσεις υπάρχει εξαιρετικά μεγάλη ανάμιξη των δειγμάτων με βάση τα πρώτα δύο features που έχουν επιλεγεί, συνεπώς δεν υπάρχει καλός διαχωρισμός ανάμεσα στις διάφορες συστάδες ψηφίων.

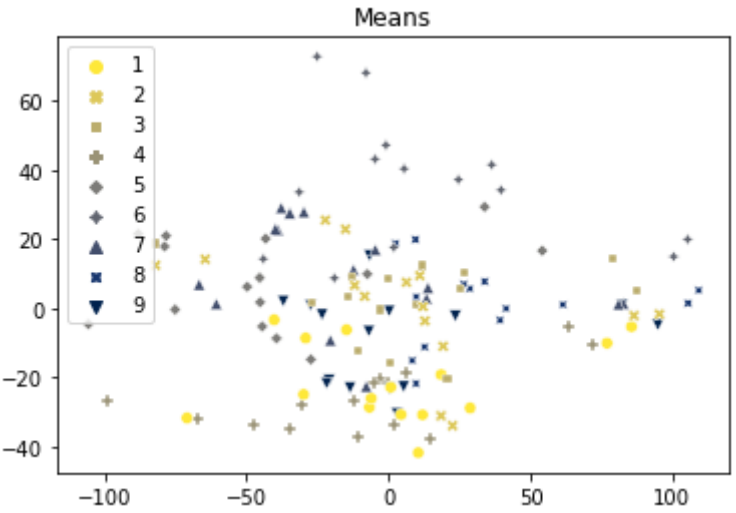
Βήμα 6

Μια καλή τακτική για απεικόνιση πολυδιάστατων διανυσμάτων είναι η μείωση των διαστάσεων τους με Principal Component Analysis (PCA) με σκοπό τη διαστατική μείωση του διανύσματος χαρακτηριστικών σε 2 και την πλήρη αξιοποίηση της χρήσιμης πληροφορίας.
Η μέθοδος (PCA) υπολογίζει μία νέα ορθοκανονική βάση για τα δεδομένα, ώστε οι διαστάσεις των δεδομένων σε αυτή να είναι γραμμικά ασυσχέτιστες. Αυτό επιτυγχάνεται με τον υπολογισμό του πρώτου διανύσματος ως αυτό που μπορεί να προσεγγίσει καλύτερα τα δεδομένα, στην συνέχεια του δεύτερου καλύτερου για τον σκοπό αυτό, το οποίο όμως είναι ορθογώνιο με το πρώτο, του τρίτου το οποίο είναι ορθογώνιο με τα άλλα δύο κ.ο.κ. ΄Ετσι υπολογίζονται τελικά τα διανύσματα τα οποία ονομάζονται principal components και τα οποία είναι γραμμικοί συνδυασμοί των αρχικών μεταβλητών. Το χαρακτηριστικό των διανυσμάτων αυτών είναι πως η μέγιστη ποσότητα πληροφορίας βρίσκεται στα λίγα πρώτα διανύσματα. ΄Ετσι, επιτυχώς μειώνεται η διαστατικότητα κρατώντας τις δύο ή τρεις πρώτες συνιστώσες με όσο το δυνατόν λιγότερη απώλεια πληροφορίας.

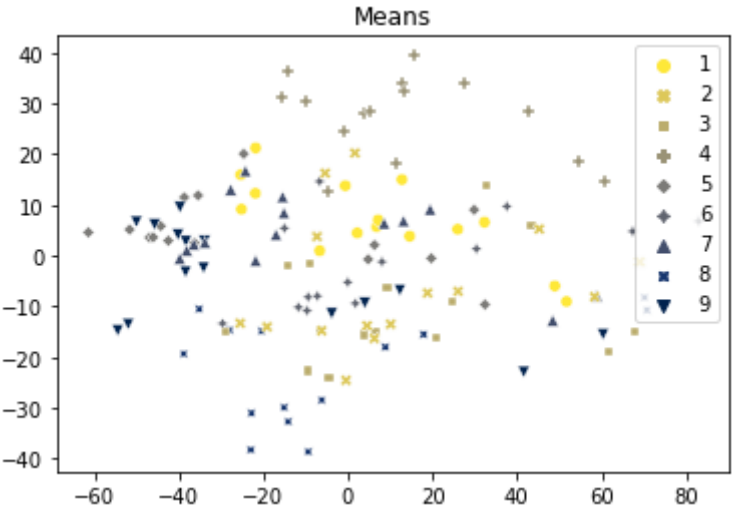
Για το σκοπό αυτό στην επικείμενη περίπτωση χρησιμοποιήθηκε η συνάρτηση *pca_analysis()* (Εικόνα 6.1) μέσω της οποίας μειώνονται σε 2 οι διαστάσεις των διανυσμάτων του προηγούμενου βήματος με και δημιουργούνται εκ νέου τα scatter plots όπως αποτυπώνονται στις εικόνες 6.2(α) και (β).

```
def pca_analysis(meancon, digits):
    pca = PCA(n_components=2)
    new_mean = pca.fit_transform(meancon)
    fig, axs = plt.subplots(ncols=1, figsize=(6, 4))
    scatter(np.asarray(new_mean)[:, 0], np.asarray(new_mean)[:, 1], digits, 'Means', ax=axs)
    plt.show()
    print("The original variance retained using 2-component PCA is {:.2%}".format(
        pca.explained_variance_ratio_.cumsum()[-1]))
    return new_mean
```

Εικόνα 6.1 *Κώδικας για τα διαγράμματα διασποράς των 2 πρώτων διαστάσεων των διανυσμάτων χαρακτηριστικών με PCA*



Εικόνα 6.2 (α) *Διάγραμμα διασποράς της μέσης τιμής των χαρακτηριστικών μετά την εφαρμογή PCA*

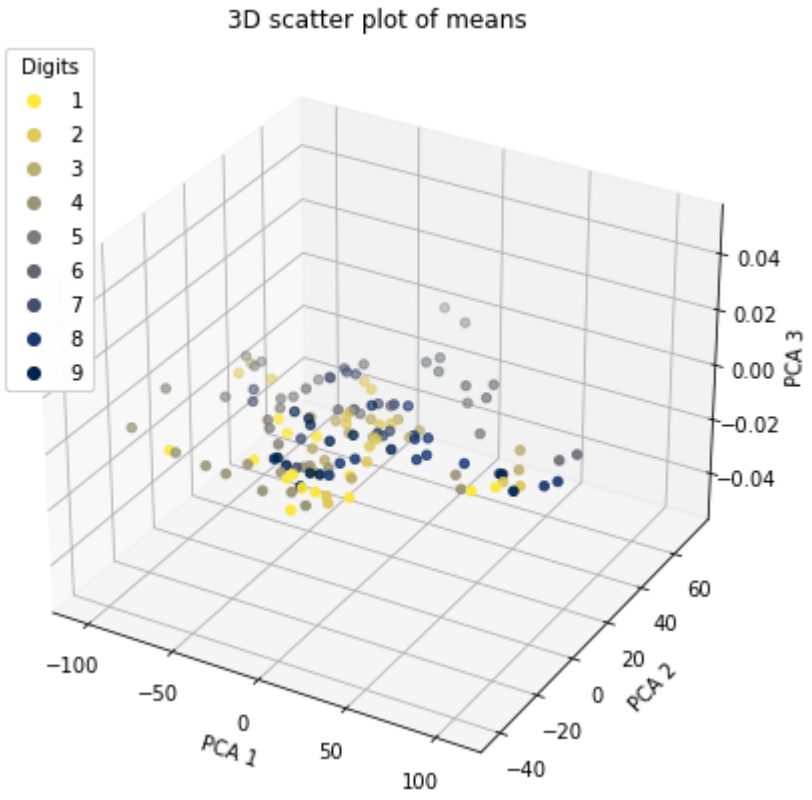


Εικόνα 6.2 (β) *Διάγραμμα διασποράς της τυπικής απόκλισης των χαρακτηριστικών μετά την εφαρμογή PCA*

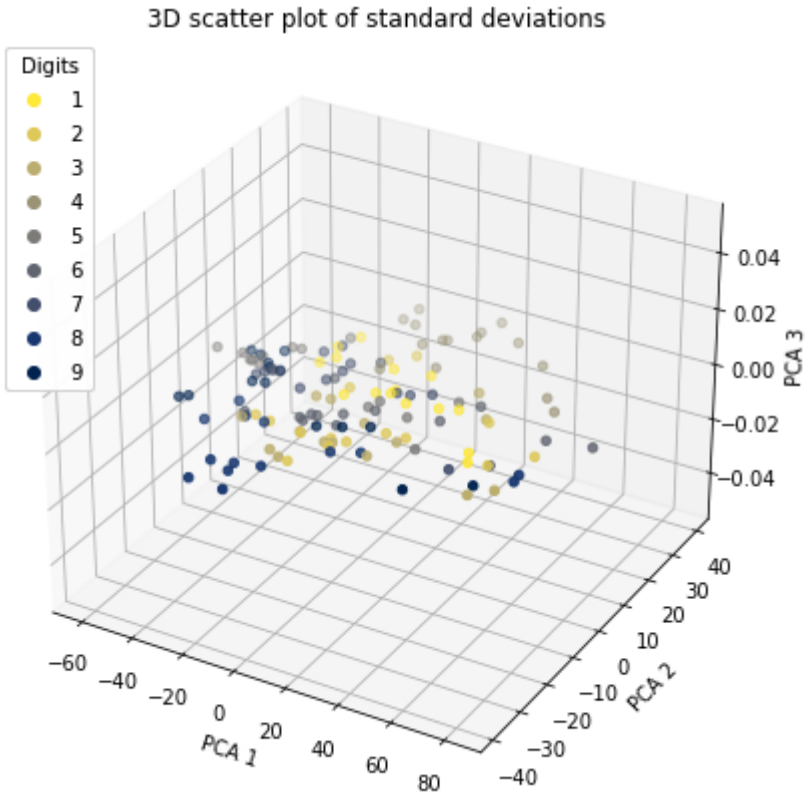
Και μετά την εφαρμογή της PCA όπως φαίνεται και από τα νέα διαγράμματα διασποράς, εξακολουθεί να υπάρχει σε κάποιο βαθμό επικάλυψη των στοιχείων που ανήκουν σε ίδιες ή διαφορετικές συστάδες, δηλαδή ψηφία, και συνεπώς υπάρχει αδυναμία να οριστούν με σαφήνεια οι περιοχές απόφασης.
Η διαδικασία επαναλαμβάνεται και στις 3 διαστάσεις μέσω της συνάρτησης *pca_aalysis_3d()* (Εικόνα 6.3).

```
def pca_analysis_3d(meancon, digits, text):
    pca = PCA(n_components=3)
    new_mean = pca.fit_transform(meancon)
    # Creating figure
    fig = plt.figure(figsize=(10, 7))
    ax = plt.axes(projection="3d")
    # Creating plot
    scatter = ax.scatter3D(np.asarray(new_mean)[:, 0], np.asarray(new_mean)[:, 1], c=digits, cmap='cividis_r')
    plt.title("3D scatter plot of " + text)
    legend_m = ax.legend(*scatter.legend_elements(), loc="upper left", title="Digits")
    ax.add_artist(legend_m)
    ax.set_xlabel('PCA 1')
    ax.set_ylabel('PCA 2')
    ax.set_zlabel('PCA 3')
    # show plot
    plt.show()
    print("The original variance retained using 3-component PCA is {:.2%}".format(
        pca.explained_variance_ratio_.cumsum()[-1]))
    return new_mean
```

Εικόνα 6.3 *Κώδικας για τα διαγράμματα διασποράς των 2 πρώτων διαστάσεων των διανυσμάτων χαρακτηριστικών με PCA στο 3D*



Εικόνα 6.4 (α) *Διάγραμμα διασποράς της μέσης τιμής των χαρακτηριστικών μετά την εφαρμογή PCA στο 3D*



Εικόνα 6.4 (β) *Διάγραμμα διασποράς της τυπικής απόκλισης των χαρακτηριστικών μετά την εφαρμογή PCA στο 3D*

Η παράμετρος *Explained variance* είναι ένα στατιστικό μέτρο του πόση διακύμανση σε ένα σύνολο δεδομένων μπορεί να αποδοθεί σε καθένα από τα κύρια συστατικά, principal components, (ιδιοδιανύσματα) που δημιουργούνται από τη μέθοδο ανάλυσης κύριας συνιστώσας (PCA). Συνοπτικά, αναφέρεται στο μέγεθος της μεταβλητότητας σε ένα σύνολο δεδομένων που μπορεί να αποδοθεί σε κάθε μεμονωμένο κύριο συστατικό. Δηλαδή επεξηγεί τι ποσοστό της συνολικής διακύμανσης «εξηγείται» από κάθε συνιστώσα. Αυτό είναι σημαντικό γιατί επιτρέπει να ταξινομούνται τα στοιχεία με σειρά σπουδαιότητας και να απομονώνονται τα πιο σημαντικά κατά την ερμηνεία των αποτελεσμάτων της ανάλυσή. Μια υψηλότερη επεξηγημένη διακύμανση θα ήταν καλύτερη γιατί θα σήμαινε ότι το μοντέλο κάνει καλύτερη δουλειά στην πρόβλεψη.

```
The original variance retained using 2-component PCA is 70.45%
The original variance retained using 3-component PCA is 81.26%
```

Εικονα 6.5. *Explained variance σε 2D και 3D*

Το ποσοστό της διασποράς των αρχικών δεδομένων που εκφράζεται με την χρήση 2 συνιστωσών ισούται με 70.45% ενώ στην περίπτωση των 3 κύριων συνιστωσών ισούται με 81.26%, γεγονός που φανερώνει πως η διαστατική μείωση σε τόσο λίγες διαστάσεις οδηγεί σε σημαντική απώλεια πληροφορίας. Μάλιστα παρατηρώντας το τρισδιάστατο scatter plot για τα τρία πρώτα PCA

components είναι φανερό πως τα clusters των διαφόρων ψηφίων έχουν αρκετά παρόμοια μορφή με αυτή στο 2D plot, το οποίο είναι λογικό, αφού το τρίτο PCA component περιλαμβάνει ένα 10% επιπλέον της αρχικής διασποράς σε σχέση με το ποσοστό της αρχικής διασποράς που περιλαμβάνουν τα δύο πρώτα components.

Βήμα 7

Στην συνέχεια χωρίζονται τα δεδομένα σε train και test set, με χρήση της συνάρτησης *sklearn.train_test_split()* με την επιπρόσθετη παράμετρο stratify. Στην συνέχεια τα δεδομένα κανονικοποιούνται έτσι ώστε να έχουμε μηδενική μέση τιμή και διασπορά ίση με 1,το οποίο επιτυγχάνεται με χρήση της συνάρτησης *normalize_data()* της *sklearn.preprocessing.normalize* όπου και εκπαιδεύουμε (fit) πάνω στο σύνολο εκπαίδευσης και στην συνέχεια εφαρμόζουμε την κανονικοποίηση στο σύνολο εκπαίδευσης και στο σύνολο ελέγχου.

```
X_train, X_test, y_train, y_test = train_test_split(np.array(features),
    np.array(all_digits), test_size=0.3, train_size=0.7,random_state=2000) # scikit learns works better with np
X_train = normalize_data(X_train)
X_test = normalize_data(X_test)
```

Εικόνα 7.1 *Διαχωρισμός των data σε train και test sets*

```
def normalize_data(X):
    return normalize(X,axis=0, norm='max')
```

Εικόνα 7.2 *Συνάρτηση normalize_data()*

Επακόλουθα έγινε ταξινόμηση των ψηφίων με χρήση του customized Bayesian ταξινομητή της πρώτης εργαστηριακής άσκησης, καθώς και του Naive Bayes του scikit-learn. Επιλέχθηκαν επίσης άλλοι 4 classifiers, SVM με 3 διαφορετικούς πυρήνες (linerar, rbf και sigmoid) και ο Knn (k nearest neighbors). Τα αποτελέσματα αποτυπώνονται στην Εικόνα 7.3:

```
Success Performance of our custom Naive Bayes classifier is: 62.50%
Success of each scikit-learn classifier follows below in sorted list:
Model: Naive Bayes has performance: 62.50%
Model: SVM linear kernel has performance: 60.00%
Model: KNeighborsClassifier has performance: 50.00%
Model: SVM RBF kernel has performance: 20.00%
Model: SVM sigmoid kernel has performance: 20.00%
```

Εικόνα 7.3 *Αποτελέσματα ταξινόμησης*

Αφού πρώτα ο ταξινομητής Customized Naive Bayes αναπροσαρμόστηκε ώστε να μπορεί να ταξινομεί μόνο 9 ψηφία, εκπαιδεύτηκε στο σύνολο των δεδομένων εκπαίδευσης και αξιολογήθηκε στα test δεδομένα με ακρίβεια 62.5%. Την ίδια ακριβώς επιτυχία είχε και ο ταξινομητής GaussianNB της sklearn βιβλιοθήκης. Αυτοί οι ταξινομητές παρουσιάζουν και το μέγιστο score σε σύγκριση με το σύνολο. Ακολουθεί ο SVM με linear kernel σε ποσοστό 60% έπειτα ο Knn με 8 neighbors σε ποσοστό 50% και τέλος οι SVM με rbf και sigmoid kernel στο 20%.

Bonus:

Για να διαπιστώσουμε αν θα αυξηθεί το ποσοστό επιτυχίας, προσθέτουμε επιπλέον ηχητικά χαρακτηριστικά στο διάνυσμα, με την τεχνική **zero-crossing rate**. Ο ρυθμός μηδενικής διέλευσης zero crossing rate (ZCR) ενός πλαισίου ήχου είναι ο ρυθμός μεταβολών του σήματος κατά τη διάρκεια του καρέ. Με άλλα λόγια, είναι ο αριθμός των φορών που το σήμα αλλάζει την τιμή, από θετικό σε αρνητικό και αντίστροφα, διαιρούμενο με το μήκος του πλαισίου. Το πλήθος μηδενισμών του σήματος της φωνής ανά ένα παράθυρο.
Με χρήση της συνάρτησης *librosa.feature.zero_crossing_rate()* και ορίσματα το window μήκος παραθύρου 0.25 ms και το hop_leght βήμα 10ms για κάθε ένα αρχείου ήχου προκύπτουν διαφορετικού μήκους zero crossing rate, συνεπώς χρησιμοποιείται το ελάχιστο μήκος το οποίο είναι ίσο με 53 χαρακτηριστικά. Τα χαρακτηριστικά αυτά εισάγονται σε μία λίστα *zero_crossing_rate.* Έπειτα συνενώνεται το διάνυσμα χαρακτηριστικών του κάθε αρχείου ήχου όπως προέκυψε σε προηγούμενο ερώτημα με τα επιπλέον 53 χαρακτηριστικά που προέκυψαν από το zero_crossing_rate(Εικόνα 7.4).

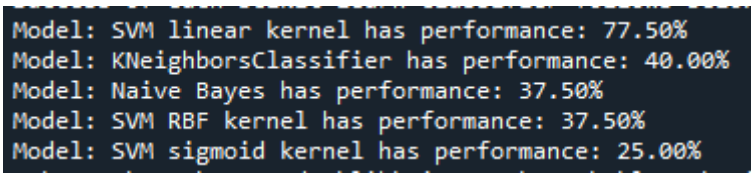
```
# bonus: extract zero-crossing rate
zero_crossing_rate = []
for i in range(len(wavs)):
    zc = librosa.feature.zero_crossing_rate(wavs[i], frame_length=int(Fs*window), hop_length=step)[:, :53]
    zero_crossing_rate.append(zc)

feature_vector_zc = np.zeros((len(wavs), 6*13+53))
for record in range(len(wavs)):
    feature_vector_zc[record,   :78] = features[record]
    feature_vector_zc[record, 78:  ] = zero_crossing_rate[record][0]
```

Εικόνα 7.4 *Κώδικας zero-crossing rate*

Σημαντικό ρόλο στην βελτίωση της απόδοσης θα μπορούσε να παίξει και η ενέργεια- **Energy** του σήματος.Η βραχυχρόνια ενέργεια των σημάτων ομιλίας αντανακλά τη διακύμανση του πλάτους. Σε ένα τυπικό σήμα ομιλίας μπορούμε να δούμε ότι ορισμένες ιδιότητες του αλλάζουν σημαντικά με το χρόνο. Για παράδειγμα, μπορούμε να παρατηρήσουμε μια σημαντική διακύμανση στο πλάτος κορυφής του σήματος και μια σημαντική διακύμανση της θεμελιώδους συχνότητας εντός των περιοχών φωνής σε ένα σήμα ομιλίας. Αυτά τα γεγονότα υποδηλώνουν ότι απλές τεχνικές επεξεργασίας πεδίου χρόνου θα πρέπει να είναι ικανές να παρέχουν χρήσιμες πληροφορίες για τα χαρακτηριστικά του σήματος, όπως η ένταση, ο τρόπος διέγερσης, το ύψος, και πιθανώς ακόμη και οι παράμετροι φωνητικών οδών, όπως οι συχνότητες σχηματισμού. Χρησιμοποιείται η συνάρτηση *librosa.feature.rms()*.
Εισάγουμε ακόμα ένα main audio feature, το **Spectral roll off** που δηλώνει την συχνότητα κάτω από την οποία βρίσκεται το 85% της ενέργειας του πλάτους του φάσματος του σήματος για κάθε παράθυρο. Και πάλι χρησιμοποιείται η συνάρτηση *librosa.feature.spectral_rolloff()*, μήκος παραθύρου 25 ms και βήμα 10 ms.
Τέλος αξιολογούνται εκ νέου οι διάφοροι ταξινομητές με τα νέα Train και test sets τα οποία πρώτα έχουν κανονικοποιηθεί. Ακολουθούν τα αποτελέσματα:

```
Model: Naive Bayes has performance: 62.50%
Model: SVM linear kernel has performance: 60.00%
Model: KNeighborsClassifier has performance: 50.00%
Model: SVM RBF kernel has performance: 20.00%
Model: SVM sigmoid kernel has performance: 20.00%
```

Εικόνα 7.5 *Αποδόσεις πριν την προσθήκη επιπλέον ηχητικών χαρακτηριστικών*



Εικόνα 7.6 *Αποδόσεις μετά την προσθήκη επιπλέον ηχητικών χαρακτηριστικών*

Από τη σύγκριση των εικόνων 7.5 και 7.6 παρατηρείται πως η προσθήκη των επιπλέον χαρακτηριστικών βελτίωσε σημαντικά τις επιδόσεις όλων των ταξινομητών. Σχετικά με την σύγκριση των επιδόσεων των διαφόρων ταξινομητών μεταξύ τους, παρατηρούμε ότι αυτή δεν άλλαξε σε σχέση με πριν, δηλαδή εξακολουθεί ο SVM linear να είναι ο βέλτιστος για το παρόν πρόβλημα, με τους kNN, SVM rbf, και Naive Bayes και SVM sigmoid να ακολουθούν, σε αυτή την σειρά.

Βήμα 8

Σε αυτό το βήμα ο κώδικας βρίσκεται στο αρχείο sin_to_cos.py. Δημιουργείται το dataset όπου αποτελείται από 1000 ακολουθίες 10 συνεχόμενων σημείων ενός ημιτόνου με τιμή εισόδου στο διάστημα [0,1000/40].

```
f = 40.
w = 2. * np.pi
time_interval = np.arange(0, 1000 / f, 0.002)
sin_all = np.sin(w * time_interval * f)
cos_all = np.cos(w * time_interval * f)
a = np.random.randint(0, len(sin_all) - 10, 1000, dtype=int)
X = np.zeros((1000, 10))
y = np.zeros((1000, 10))
for i in range(len(a)):
    X[i, :] = sin_all[a[i]:a[i] + 10]
    y[i, :] = cos_all[a[i]:a[i] + 10]
```

Εικόνα 8.1 *Δημιουργία dataset*

Έπειτα, δημιουργείται μια κλάση *SinDataset* η οποία ενώνει τα δεδομένα σε μια ενιαία δομή και τα μετατρεπει σε torch tensors ώστε να μπορούν στην συνέχεια να χωριστούν σε batches, όπως φαίνεται στην Εικόνα 8.2.

```python
class SinDataset(Dataset):
    def __init__(self, X, y):
        # all the available data are stored in a list
        self.data = np.zeros((X.shape[0], 2 * X.shape[1]))
        self.data[:, 0:X.shape[1]] = X
        self.data[:, X.shape[1]:] = y
        self.data = torch.tensor(self.data).float()

    def __len__(self):
        return self.data.shape[0]

    def __getitem__(self, idx):
        return self.data[idx]


train_dataset = SinDataset(X_train, y_train)
test_dataset = SinDataset(X_test, y_test)

torch.manual_seed(101)
batch_size = 4

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Εικόνα 8.2 *Επεξεργασία dataset και προετοιμασία για εκπαίδευση*

Αφού δημιουργήσουμε δύο νευρωνικά (ένα LSTM και ένα RNN) και τις αντίστοιχες κλάσεις (Εικόνες 8.3 και 8.4), δίνουμε τιμές στις παραμέτρους και αρχικοποιούμε το επιλεγμένο μοντέλο. Ένα LSTM δίκτυο έχει πλεονέκτημα σε σχέση με ένα RNN μοντέλο γιατί δεν μηδενίζονται οι παράγωγοι του μοντέλου.

```python
class RNNModel(nn.Module):
    def __init__(self, input_dim,seq_len, hidden_dim, num_lay, out_dim):
        super(RNNModel, self).__init__()

        # Defining the number of layers and the nodes in each layer
        self.hidden_dim = hidden_dim
        self.num_layer = num_lay
        self.seq_len = seq_len

        # RNN layers
        self.rnn = nn.RNN(
            input_dim, hidden_dim, num_lay,batch_first=True)
        # Fully connected layer
        self.fc = nn.Linear(hidden_dim , out_dim*seq_len)

    def forward(self, x):
        # Add one more dimension
        x = x.unsqueeze(-1)

        # Initializing hidden state for first input with zeros
        h0 = torch.zeros(self.num_layer, x.size(0), self.hidden_dim).requires_grad_()

        # Forward propagation by passing in the input and hidden state into the model
        out, h0 = self.rnn(x, h0.detach())

        # Reshaping the outputs in the shape of (batch_size, seq_length * hidden_dim)
        # out = out.contiguous().view(batch_size, -1)
        out = out[:, -1, :]

        # Convert the final state to our desired output shape (batch_size, output_dim)
        out = self.fc(out)
        return out
```

Εικόνα 8.3 *Κλάση RNN*

```python
class LSTMModel(nn.Module):
    def __init__(self, input_dim, seq_len, hidden_dim,cell_dim, layer_dim, out_dim):
        super(LSTMModel, self).__init__()
        # Hidden dimensions and hidden cells
        self.hidden_dim = hidden_dim
        self.cell_dim = cell_dim

        # Number of hidden layers
        self.layer_dim = layer_dim

        # Number of sequence length
        self.seq_len = seq_len

        # batch_first=True causes input/output tensors to be of shape
        # (batch_dim, seq_dim, feature_dim)
        self.lstm = nn.LSTM(input_dim, hidden_dim, layer_dim, batch_first=True)

        # Readout layer
        self.fc = nn.Linear(hidden_dim, out_dim*seq_len)

    def forward(self, x):
        # Add one more dimension
        x = x.unsqueeze(-1)

        # Initialize hidden state with zeros
        h0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim).requires_grad_()

        # Initialize cell state
        c0 = torch.zeros(self.layer_dim, x.size(0), self.cell_dim).requires_grad_()

        out, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))

        # Index hidden state of last time step
        out = out[:, -1, :]
        out = self.fc(out)
        return out
```

Εικόνα 8.3 *Κλάση LSTM*

Το επιλεγμένο μοντέλο εκπαιδεύεται με ανανέωση των βαρών με backpropagation με βάση το σφάλμα.

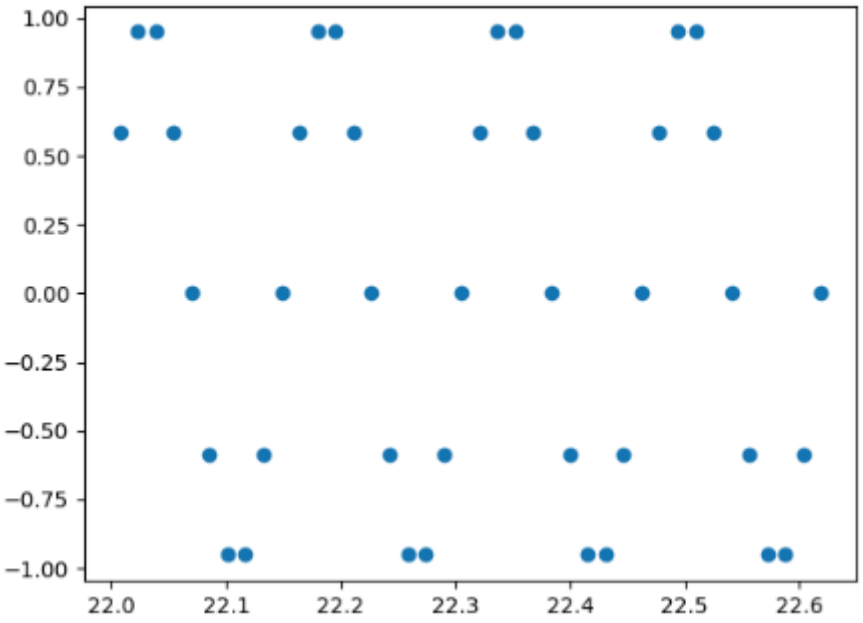
```
for epoch in range(num_epochs):
    for i in train_loader:
        sin_var = i[:,0:10]
        cos_val = i[:, 10:]
        train = sin_var
        true_val = cos_val

        # Clear gradients
        optimizer.zero_grad()

        # Forward propagation
        outputs = model(train)

        # Calculate softmax and ross entropy loss
        loss = error(outputs, true_val)

        # Calculating gradients
        loss.backward()

        # Update parameters
        optimizer.step()

        count += 1
```

Εικόνα 8.3 *Εκπαίδευση μοντέλου*

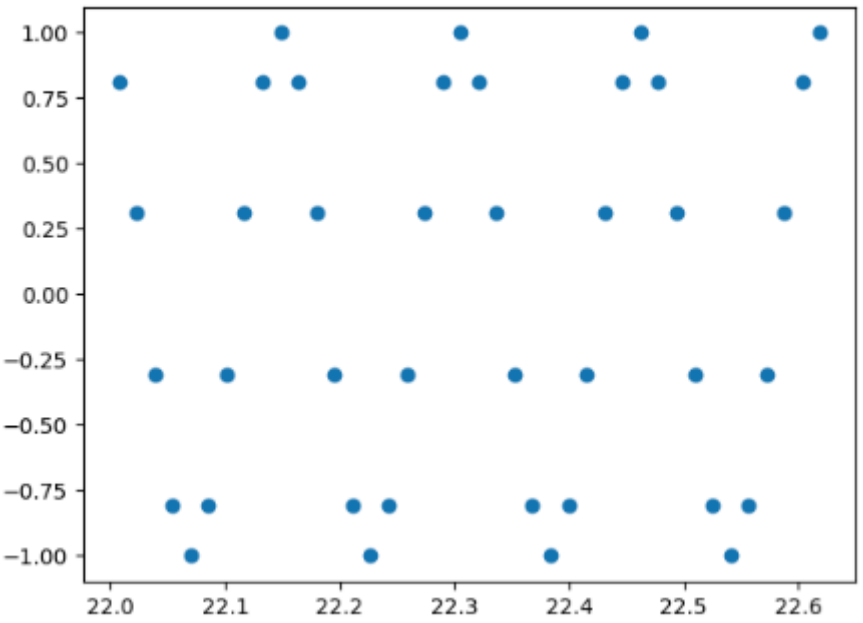
Για να ελεγχθεί το μοντέλο, δημιουργήσαμε ένα οπτικό testing. Ουσιαστικά, γίνεται αναπαράσταση της εισόδου του μοντέλου (Εικόνα 8.4), της επιθυμητής εξόδου (Εικόνα 8.5) και της πρόβλεψης (Εικόνα 8.6).

```
time_interval = np.arange(100/f + 1.0025, 104/f +1, 0.0025)
sin_all_test = np.sin(w * time_interval * f)
plt.scatter(w*time_interval,sin_all_test)
plt.show()

cos_all_test = np.cos(w * time_interval * f)
plt.scatter(w*time_interval,cos_all_test)
plt.show()

sin_all_test = sin_all_test.reshape((4, 10))
cos_all_test = cos_all_test.reshape((4,10))

test_dataset = SinDataset(sin_all_test,cos_all_test)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

outputs = []
for j in test_loader:
    val_sin_var = j[:, 0:10]
    val_cos_val = j[:, 10:]
    val_true_val = val_cos_val
    model.eval()
    # Forward propagation
    outputs = model(val_sin_var)

a = outputs.detach().numpy()
b = w*time_interval
plt.scatter(b,a)
plt.show()
```

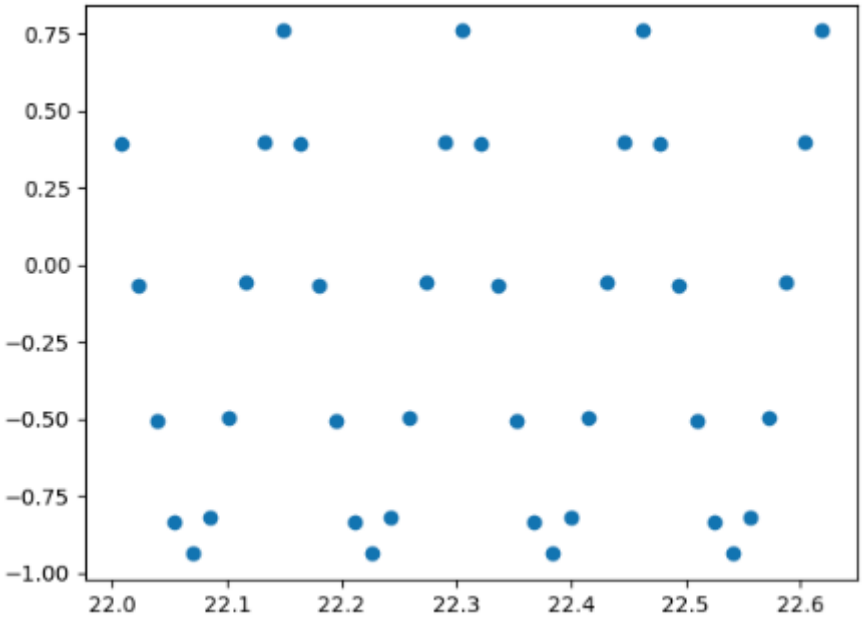
Εικόνα 8.3 *Δημιουργία οπτικού test set*



Εικόνα 8.4 *Είσοδος στο μοντέλο*



Εικόνα 8.5 *Επιθυμητή έξοδος του μοντέλο*



Εικόνα 8.6 *Πρόβλεψη του μοντέλου*

Φαίνεται πως το μοντέλο αρχίζει να μαθαίνει το σχήμα του συνημιτόνου δοσμένης μιας ακολουθίας 10 σημείων του ημιτόνου.

Βήμα 9

Στο συγκεκριμένο βήμα, έγινε λήψη του δεύτερου αρχείου που περιέχει ένα μεγάλο dataset, το Free Spoken Digit Dataset (FSDD). Το συγκεκριμένο dataset περιέχει 3000 δείγματα δηλαδή αρχεία ήχου. Πιο συγκεκριμένα υπάρχουν 6 ομιλητές κάθε ένας από τους οποίου εκφωνεί 50 φορές κάθε ένα από τα ψηφία 0-9.
Η διαδικασία που ακολουθήθηκε είναι παρόμοια με αυτή του βήματος 2. Διαβάστηκαν από τον κατάλληλο φάκελο ‘recordings’ όλα τα αρχεία ήχου και έπειτα με χρήση της εντολής *split()* εκμαιεύονται όλες οι χρήσιμες πληροφορίες σχετικά με το εκάστοτε ψηφίο του αρχείου, τον εκφωνητή αλλά και το id του κάθε αρχείου (οι τιμές του ποικίλουν από 0 εώς και 50 για κάθε γράμμα και ομιλητή).
Σε αρχικό βήμα, δημιουργήθηκαν τα training και test sets. Ο διαχωρισμός αυτός έγινε με τη χρήση της συνάρτησης parser η οποία υπήρχε ήδη στο parser.py αρχείο. Να σημειωθεί πως η τεχνική για τον διαχωρισμό σε train και test set στηρίχθηκε στην συνάρτηση *split_free_digits()*. Πρακτικά για κάθε ψηφίο και για κάθε εκφωνητή στο test set εκχωρήθηκαν τα 5 πρώτα εκφωνήματα ενώ τα υπόλοιπα συμπεριλήφθηκαν στο train set όπως φαίνεται και στην Εικόνα 9.1.

```python
def split_free_digits(frames, ids, speakers, labels):
    print("Splitting in train test split using the default dataset split")
    # Split to train-test
    X_train, y_train, spk_train = [], [], []
    X_test, y_test, spk_test = [], [], []
    test_indices = ["0", "1", "2", "3", "4"]

    for idx, frame, label, spk in zip(ids, frames, labels, speakers):
        if str(idx) in test_indices:
            X_test.append(frame)
            y_test.append(label)
            spk_test.append(spk)
        else:
            X_train.append(frame)
            y_train.append(label)
            spk_train.append(spk)

    return X_train, X_test, y_train, y_test, spk_train, spk_test
```

Εικόνα 9.1 *Διαχωρισμός δεδομένων σε train και test set*

Έπειτα χωρίστηκαν τα δεδομένα του training set σε επιμέρους training και validation δεδομένα με ποσοστό 80%-20%. Μάλιστα για να εξασφαλιστεί ότι ο διαχωρισμός έγινε με τέτοιο τρόπο ώστε να διατηρηθεί ίδιος ο αριθμός των διαφορετικών ψηφίων σε κάθε set χρησιμοποιήθηκε η παράμετρος *stratify = y_train* .

```python
#Step 9
# parse data
X_train, X_test, y_train, y_test, spk_train, spk_test = parser("recordings")
print(len(X_train)+len(X_test))
#we have about 3000 samples (50*10*6)
# split data
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, stratify=y_train)
# next, we normalize
scale_fn = make_scale_fn(X_train)
X_train = scale_fn(X_train)
X_val = scale_fn(X_val)
X_test = scale_fn(X_test)
```

Εικόνα 9.2. *Εντολές βήματος 9*

Βήμα 10

Γι την αναγνώριση των ψηφίων με τη χρήση GMM-HMM χρησιμοποιείται η βιβλιοθήκη hmmlearn. Αρχικά προσπαθήσαμε να χρησιμοποιήσουμε την βιβλιοθήκη pomegranate ωστόσο υπήρχαν αρκετά θέματα με τις εκδόσεις των επιμέρους βιβλιοθηκών οπότε και καταφύγαμε στην hmmlearn βιβλιοθήκη. Αρχικά αρχικοποιείται ένα GMM-HMM μοντέλο για κάθε ψηφίο. Το GMM χρησιμοποιείται για την μοντελοποίηση της κατανομής των διανυσμάτων χαρακτηριστικών με πολυδιάστατες γκαουσιανές. Μέσω του αλγορίθμου Expectation Maximization υπολογίζεται το διάνυσμα μέσων τιμών και ο πίνακας συνδιακύμανσης για κάθε μία από τις γκαουσιανές. Το HMM είναι μία μαρκοβιανή αλυσίδα με κρυφές καταστάσεις στην οποία οι μεταβάσεις μεταξύ καταστάσεων μοντελοποιούν την αλλαγή φωνήματος. Το κάθε μοντέλο είναι left-right. Αυτό σημαίνει ότι χρησιμοποιούμε ένα άνω τριγωνικό και στοχαστικό πίνακας μετάβασης. Αυτός με βάση τα δεδομένα της εκφώνησης θα έχει την εξής μορφή:

$$trans\_mat = \begin{bmatrix} 0.5 & 0.5 & 0 & \dots & 0 \\ 0 & 0.5 & 0.5 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

Επιπλέον, εκτός από τον πίνακα μετάβασης κάθε μοντέλο GMM-HMM έχει τις εξής μεταβλητές:

1. *X*: διάνυσμα χαρακτηριστικών από ένα ψηφίο
2. *n states*: ο αριθμός των HMM καταστάσεων
3. *n mixtures*: ο αριθμός των Gaussians. ΄Οταν το n mixtures είναι μεγαλύτερο του 1 χρησιμοποιείται GMM διαφορετικά 1D Gaussian.
4. *starts*: αρχικός πίνακας πιθανοτήτων με $\pi_i = 0$ για $i \neq 1$ και $\pi_i = 1$ για $i = 1$
5. *model*: το μοντέλο HMM για το κάθε ψηφίο που θα χρησιμοποιηθεί.

Αρχικά ορίζεται μία λίστα X στην οποία κάθε στοιχείο της αντιστοιχεί σε ένα από τα 9 ψηφία (1-9). Κάθε στοιχείο της λίστας είναι πρακτικά μία άλλη λίστα που αποτελείται από πολλούς δισδιάστατους πίνακες (n_frames x 6) οι οποίοι αντιστοιχούν στα εξαγόμενα features. Για να γίνει καλύτερα αντιληπτό, για παράδειγμα το X[0] αντιστοιχεί σε μία λίστα μήκους ίσου με τον αριθμό των δειγμάτων που αντιστοιχούν στους άσσους και κάθε στοιχείο αυτής της λίστας αποτελείται από τα mfccs για κάθε ένα δείγμα.
Ορίζονται ως *n_states =4* το άθροισμα των πιθανών σεναρίων των καταστάσεων, επομένως πρόκειται για μία ανάλυση όπου θα μελετηθούν τα HMM για n_states=1,2,3 και 4. Με το ίδιο σκεπτικό ορίζεται και η μεταβλητή *n_mixtures=5* που δηλώνει τον αριθμό των επιμέρους γκαουσιανών που απαρτίζουν το GMM. Άρα μελετούνται οι περιπτώσεις όπου *n_mixtures = 1,2,3,4 και 5 .*
Σε επόμενο βήμα ορίζεται η λίστα *digits_models.* Κάθε στοιχείο της λίστας είναι ένα dictionary με key το συνδυασμό του συνόλου των καταστάσεων και value το εκάστοτε GMMHMM model που εκπαιδεύεται κατάλληλα. Η περιγραφή που προηγήθηκε αντιστοιχεί στην Εικόνα 10.1(α).

```
from hmmlearn import hmm
X = [] # data from a single digit (can be a numpy array)
n_states = 4 # the number of HMM states
n_mixtures = 5 # the number of Gaussians
n_iter=5
X_train_np = np.array(X_train)
y_train_np = np.array(y_train)
for i in range(10):
    X.append(X_train_np[y_train_np==i])
#πρακτικά κάθε στοιχείο της λίστας περιέχει μία άλλη λίστα, όπου
#κάθε στοιχείο της δεύτερης είναι πίνακας n_frames x 6 (<-mfcc)
#για το κομματι του gmm και από σχόλιο στον github τα πετάω μέσα σα κουβά, δε με ενδιαφέρουν οι διαστάσεις
digits_models: list[dict[tuple[int, int], hmm.GMMHMM]] = [{} for _ in range(10)]
```

Εικόνα 10.1(α) *Κώδικας GMM-HMM models*

Στην Εικόνα 10.1(β) ακολουθούν οι υπόλοιπες εντολές για την δημιουργία των GMM-HMM. Πιο συγκεκριμένα για κάθε ένα από τα ψηφία και για κάθε περίπτωση αριθμού καταστάσεων και γκαουσιανών υπολογίζεται το transition matrix ώστε να προκύπτουν Left to right μοντέλα και αρχικοποιείται ο πίνακας πιθανοτήτων ακολουθώντας την δομή που περιγράφηκε στην αρχή αυτού του βήματος. Έπειτα ορίζεται για κάθε έναν από τους συνδυασμούς αυτούς ένα μοντέλο GMM-HMM με παραμέτρους το σύνολο των καταστάσεων, το σύνολο των επιμέρους γκαουσιανών, τα iterations για τον αλγόριθμο Expectation Maximization και τα επιμέρους parameters: ‘s’: starting probabilities, ‘t’: transition matrix, ‘m’:means , ‘c’: covariance και ‘w’: τα βάρη των mixing GMM.

```
for d in range(10):
    for n_states in range(1, 5):
        for n_mix in range(1, 6):
            trans_mat = np.diag(np.ones(n_states)) + np.diag(np.ones(n_states - 1), 1)
            trans_mat = (trans_mat.T / np.sum(trans_mat, axis=1)).T
            starts = np.zeros(n_states)
            starts[0] = 1

            model = hmm.GMMHMM(
                n_components=n_states,
                n_mix=n_mix,
                init_params="mcw",
                params="tmcw",
                n_iter=n_iter
            )
            model.transmat_ = trans_mat
            model.startprob_ = starts
            digits_models[d][(n_states, n_mix)] = model
```

Εικόνα 10.1(β) *Κώδικας GMM-HMM models*

Βήμα 11

Το βήμα αυτό έγκειται στην εκπαίδευση των 10 μοντέλων με χρήση του αλγορίθμου Expectation Maximization όπως αναφέρθηκε και το βήμα 10. Ο αλγόριθμος εφαρμόζεται για καθορισμένο πλήθος επαναλήψεωνN ή εως να υπάρξει σύγκλιση. Η σύγκλιση $iter$ ελέγχεται μέσω της μεταβολής του λογαρίθμου της πιθανοφάνειας (Log Likelihood, πιθανότητα των δεδομένων με γνωστό μοντέλο). Για την εκπαίδευση κάθε μοντέλου (που αντιστοιχεί σε κάποιο ψηφίο) χρησιμοποιούνται όλα τα διαθέσιμα δεδομένα για το ψηφίο αυτό. Οι εντολές αποτυπώνονται στην Εικόνα 11.1.

```
for d in range(10):
    for n_states in range(1, 5):
        for n_mix in range(1, 6):
            X_digit = X[d]
            lengths = [i.shape[0] for i in X_digit]
            digits_models[d][(n_states, n_mix)].fit(np.concatenate(X_digit), lengths)
```

Εικόνα 11.1 *Κώδικας εκπαίδευσης των GMM-HMM models*

Βήμα 12

Στο βήμα αυτό γίνεται η αναγνώριση μεμονωμένων ψηφίων. Ολοκληρώνοντας την διαδικασία της εκπαίδευσης έχουμε καταλήξει στην εκτίμηση των παραμέτρων των μοντέλων. Στην συνέχεια υπολογίζεται μέσω της predict() ο λογάριθμος της πιθανοφάνειας (log likelihood) για κάθε εκφώνηση η οποία ανήκει στο σύνολο των δεδομένων για αναγνώριση. Το μοντέλο το οποίο δίνει τη μέγιστη πιθανοφάνεια είναι και το αποτέλεσμα της αναγνώρισης για τη συγκεκριμένη εκφώνηση.
Τέλος, για κάθε μοντέλο υπολογίζεται η ακρίβεια ταξινόμησης μέσω της συνάρτησης accuracy(). Η διαδικασία αυτή αρχικά πραγματοποιείται μόνο στο validation set.
Είναι γνωστό πως για την εύρεση του βέλτιστου συνδυασμού των υπερ παραμέτρων οποιουδήποτε μοντέλου (διαδικασία γνωστή και ως fine tuning), χρειάζεται να εκτελεστούν πολλαπλές εκπαιδεύσεις και αξιολογήσεις του μοντέλου μέχρι τελικά να βρεθεί το μοντέλο με την βέλτιστη απόδοση ως προς την μετρική που μελετάται. Ωστόσο, καθ' όλη αυτή την επαναληπτική διαδικασία δοκιμών και αξιολογήσεων, πρέπει να διατηρηθεί το test set άγνωστο στο μοντέλο , προκειμένου υπάρχει πλήρης αμεροληψία στην επιλογή των τελικών υπερ παραμέτρων του. Συνεπώς για να ικανοποιείται αυτή η

απαίτηση πραγματοποιούνται αξιολογήσεις του μοντέλου όπως προκύπτει για κάθε συνδυασμό τιμών των υπερ παραμέτρων του, πάνω στα δεδομένα του Validation set.
Στη περίπτωση που οι διαδοχικές αξιολογήσεις του μοντέλου γίνονταν στο test set υπήρχε ο κίνδυνος το τελικό μοντέλο να αποδίδει εξαιρετικά πάνω στα test δεδομένα, αλλά να αδυνατεί να διατηρεί το ίδιο καλές επιδόσεις σε άλλα δεδομένα χάνοντας έτσι την ιδιότητα της γενίκευσης.

```
def predict(models, X):
        likelihoods = [m.score(X) for m in models]
        return np.argmax(likelihoods)

def accuracy(models, X, y):
    predictions = [predict(models, x) for x in X]
    assert len(predictions) == len(y)
    return sum(pred == true for pred, true in zip(predictions, y)) / len(y)
```

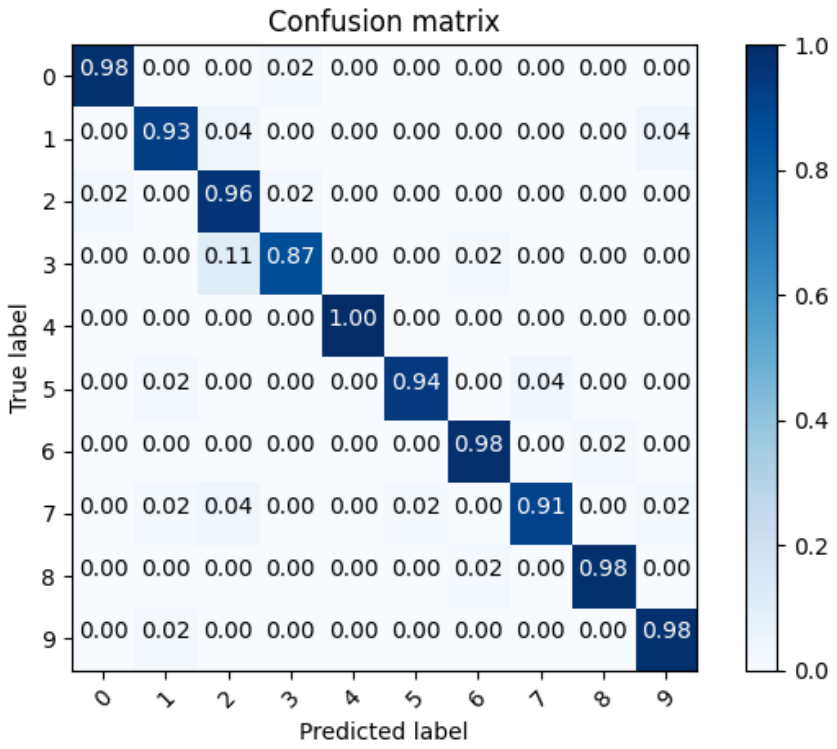
Εικόνα 12.1(α) *Κώδικας πρόβλεψης και αξιολόγησης των μοντέλων*

```
accuracies = np.zeros((4,5))
for n_states in range(1, 5):
    for n_mix in range(1, 6):
        print(n_states, n_mix)
        accuracies[n_states-1,n_mix-1]=accuracy([d[(n_states, n_mix)] for d in digits_models], X_val, y_val)
        print(accuracy([d[(n_states, n_mix)] for d in digits_models], X_val, y_val))
result = np.where(accuracies == np.amax(accuracies))
print("Best model is:{}".format(result[0]+1))
```

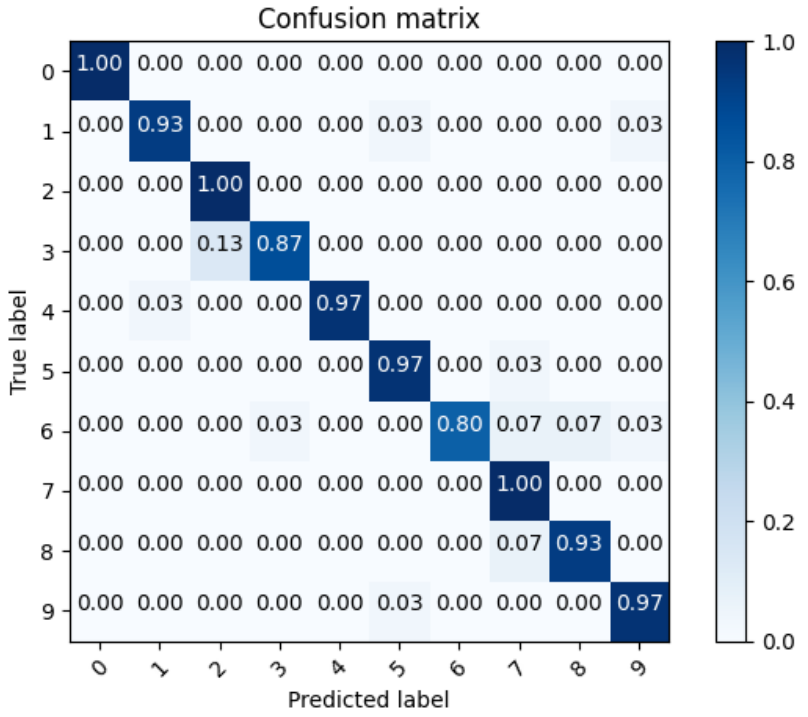
Εικόνα 12.1(β) *Κώδικας πρόβλεψης και αξιολόγησης των μοντέλων*

Βήμα 13

Ο πίνακας σύγχυσης (Confusion Matrix), είναι ένας τετραγωνικός δισδιάστατος πίνακας μεγέθους ίσου με το πλήθος των κλάσεων που συμμετέχουν στο πρόβλημα της αναγνώρισης. Στη συγκεκριμένη περίπτωση πρόκειται για ένα πίνακα διαστάσεων 10x10 και υπολογίζεται βάσει της συνάρτησης confusion_matrix του πακέτου sklear.metrics.
Ο συγκεκριμένος πίνακας, στην κάθε του διάσταση περιέχει διατεταγμένες σε αύξουσα σειρά τις κλάσεις που εμπλέκονται στο πρόβλημα , με την οριζόντια διάστασή του να αντιστοιχεί στις προβλέψεις του μοντέλου και την κατακόρυφη διάσταση στις πραγματικές ετικέτες των δειγμάτων. Επομένως, ο πίνακας αυτός στο κελί (i,j) περιέχει το πλήθος των δειγμάτων που ανήκουν στην κατηγορία i , αλλά το μοντέλα τα κατέταξε στην κατηγορία j. Ωστόσο να σημειωθεί πως τα δεδομένα είναι κανονικοποιημένα.



Εικόνα 13.1 *Confusion matrix για τα validation δεδομένα*



Εικόνα 13.2 *Confusion matrix για τα test δεδομένα*

Παρατηρείται πως τόσο στο validation όσο και στο test set πετυχαίνουμε πολύ ικανοποιητικά σκορ και άρα το κριτήριο της γενίκευσης πληρείται στον κώδικα μας.

Βήμα 14

Η κλάση LSTMModel περικλείει το ζητούμενο LSTM μοντέλο αλλά χωρις την χρήση του βοηθητικού κώδικα. Επιτρέπει την δημιουργία bidirectional ή μη, LSTM μοντέλου δουλεύοντας με την συνάρτηση pack_padded_sequence.

```python
class LSTMModel(nn.Module):
    def __init__(self, input_dim, hidden_dim,cell_dim, layer_dim, output_dim, dropout_prob, bidirectional=False):
        super(LSTMModel, self).__init__()

        # 1 for not bidirectional and 2 for bidirectional
        self.D = 2 if bidirectional else 1
        # Number of hidden layers
        self.layer_dim = layer_dim
        # Hidden dimensions and hidden cells
        self.hidden_dim = hidden_dim
        self.cell_dim = cell_dim
        # LSTM layers
        self.lstm = nn.LSTM(input_dim, hidden_dim, layer_dim, batch_first=True, dropout=dropout_prob,
                            bidirectional=bidirectional)
        # Linear layer
        self.fc = nn.Linear(int(self.D * hidden_dim), output_dim)

    def forward(self, x):

        # Read the maximum sequence length
        unpacked_x, unpacked_lengths = pad_packed_sequence(x, batch_first=True)
        # Hidden and cell state initializations
        h0 = torch.zeros(int(self.D * self.layer_dim), len(unpacked_lengths), self.hidden_dim).requires_grad_()
        c0 = torch.zeros(int(self.D * self.layer_dim), len(unpacked_lengths), self.cell_dim).requires_grad_()
        # Forward propagation
        output, (hn, cn) = self.lstm(x, (h0.detach(), c0.detach()))
        # Unpacking output
        unpacked, unpacked_len = pad_packed_sequence(output, batch_first=True)
        output = unpacked
        # Index hidden state of last time step
        output = output[:, -1, :]
        # Output shape (batch_size, output_dim)
        output = self.fc(output)
        return output
```

Εικόνα 14.1 *Κλάση LSTMModel*

Επίσης δημιουργήθηκε η κλάση FrameLevelDataset ώστε να μετασχηματίζει κατάλληλα τα δεδομένα.

```python
class FrameLevelDataset(Dataset):
    def __init__(self, feats, labels):
        """
            feats: Python list of numpy arrays that contain the sequence features.
                   Each element of this list is a numpy array of shape seq_length x feature_dimension
            labels: Python list that contains the label for each sequence (each label must be an integer)
        """
        # self.lengths = len(feats[0]) # Find the lengths
        self.lengths = [i.shape[0] for i in feats]

        self.feats = self.zero_pad_and_stack(feats)
        if isinstance(labels, (list, tuple)):
            self.labels = np.array(labels).astype('int64')

    def zero_pad_and_stack(self, x):
        """
            This function performs zero padding on a list of features and forms them into a numpy 3D array
            returns
                padded: a 3D numpy array of shape num_sequences x max_sequence_length x feature_dimension
        """
        padded = []
        # --------------- Insert your code here ---------------- #
        # padded = np.zeros((num_sequences, max_sequence_length, feature_dimension))
        max_sequence_length = max(self.lengths)
        padded = np.zeros((len(x),max_sequence_length,x[0].shape[1]))
        for idx,elem in enumerate(x):
            padded[idx,0:len(elem),:] = elem
        return padded

    def __getitem__(self, item):
        return self.feats[item], self.labels[item], self.lengths[item]

    def __len__(self):
        return len(self.feats)
```

Εικόνα 14.2 *Κλάση FrameLevelDataset*

Στην συνέχεια, τα ήδη χωρισμένα δεδομένα μετατρέπονται σε tensorDataset για να χρησιμοποιηθούν στην εκπαίδευση.

```python
batch_size = 64

# step 14
train_features, train_targets = torch.Tensor(X_train), torch.Tensor(y_train)
test_features, test_targets = torch.Tensor(X_test), torch.Tensor(y_test)
val_features, val_targets = torch.Tensor(X_val), torch.Tensor(y_val)

train = TensorDataset(train_features, train_targets)
test = TensorDataset(test_features, test_targets)
val = TensorDataset(val_features, val_targets)
```

Εικόνα 14.3 *Preprocessing του dataset*

Βιβλιογραφία

[1] https://vitalflux.com/pca-explained-variance-concept-python-example/

[2] https://wiki.aalto.fi/display/ITSP/Zero-crossing+rate