



# National Technical University of Athens

## Data Science and Machine Learning

Pattern Recognition 2nd Lab  
Speech Recognition using HMMs and RNNs

Lymperopoulos Theodoros

03400140

Nanos Georgios

03400144

---

# Description

Our goal is the implementation of a speech recognition system, that recognizes isolated words. The first part involves the extraction of the appropriate acoustic features from our recordings and their further analysis. These features are the cepstral coefficients, that are computed using a filterbank (inspired by psychoacoustic methods). More specifically, the system will recognize isolated digits in English. Our dataset contains dictations of 9 digits from 15 different speakers in separate .wav files. In total, there are 133 files, since 2 dictations are missing. The name of each file (e.g. eight8.wav) declares both the dictated digit (e.g. eight) and the speaker (speakers are numbered from 1 to 15). The sampling rate is  $F_s=16k$  and the duration of each dictation differs.

## Step 1

We extracted the mean value of the pitch for the vowels "α" "ou" and "ɪ" by pressing F5, F1, F2 and F3 respectively

File	Vowel	Gender	Mean Pitch (Hz)	F1	F2	F3
onetwothree1.wav	α	male	133.744	766.294	1896.275	2931.994
onetwothree1.wav	α	female	178.123	716.779	1914.122	3078.091
onetwothree1.wav	ɪ	male	131.360	393.172	2055.706	2531.323
onetwothree8.wav	ɪ	female	173.168	337.877	1995.137	2685.517
onetwothree8.wav	ou	male	129.308	347.322	1780.061	2295.354
onetwothree8.wav	ou	female	180.818	333.502	1498.956	2618.900

We observe that the Mean pitch (Hz) differs a lot when the speaker is changed. In men it is close to 130Hz, while in women at 180Hz. On the other hand, formants (F1, F2, F3) are more actively involved in the differentiation of vowels, and therefore of digits. We also see that the formants differ from each other in different vowels and are not influenced a lot when the speaker is changed. So, these peaks (F1, F2, F3) can be used in speech recognition to distinguish vowels.

## Step 2

We created a function *data\_parser* that reads the sound files. It uses the `os.listdir` method to read the files. For every filename, by calling the `librosa.load` method we load it as an array, which is added to the wav list. In order to distinguish the speaker and the digit that are both into the name of the file, we uses the method *re.split()*. Let's take a look at the results.

```
# Print first two files
for i in range(2):
    print("Sample " + str(i))
    print("Waveform: " + str(wav[i][:]))
    print("Speaker: " + str(speaker[i]))
    print("Digit: " + str(digit[i]))
    print()

Sample 0
Waveform: [-0.0007019 -0.00088501 -0.00048828 ... 0.00088501 0.00112915
 0.00091553]
Speaker: 1
Digit: 8

Sample 1
Waveform: [ 9.1552734e-05 -3.0517578e-05 -2.7465820e-04 ... 6.4086914e-04
 1.1596680e-03 1.3427734e-03]
Speaker: 10
Digit: 8
```

Figure 1: The results of the *data\_parse* method.

## Step 3

We defined a method called *extract\_mel* to calculate the Mel-Frequency Cepstral Coefficients (MFCCs) from each sound file using librosa (13 features per file). We used 25 ms window size and 10 ms step size. Also, we computed the delta and the delta-deltas of the features. We will use these lists to create a supervector later on.

## Step 4

We displayed a histogram for the 1st and the 2nd MFCC of digits n1 and n2 for all recordings. In our case,  $n1 = 4$  and  $n2 = 9$ . The difference between the MFCC and MFSC lies in different logarithm and different nonlinear partial compression function. MFCC from the logarithm is homomorphic deconvolution, the compression algorithm is based on nonlinear MFSC strength, that is loudness perception transformation. MFCC combines voice frequency nonlinear perception and Mel domain bandpass filtering but the essence is based on the analysis cepstral homomorphic deconvolution, MFSC are based on the speech frequency and intensity of auditory perception feature representation.

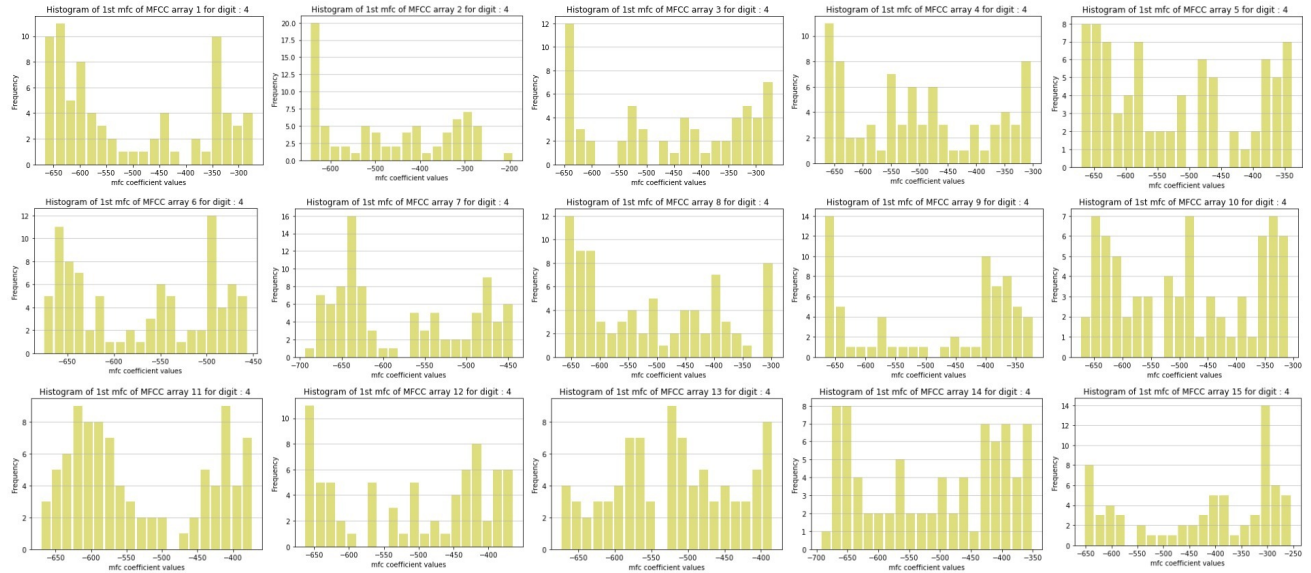


Figure 2: Histograms of the 1st mfcc for digit 4

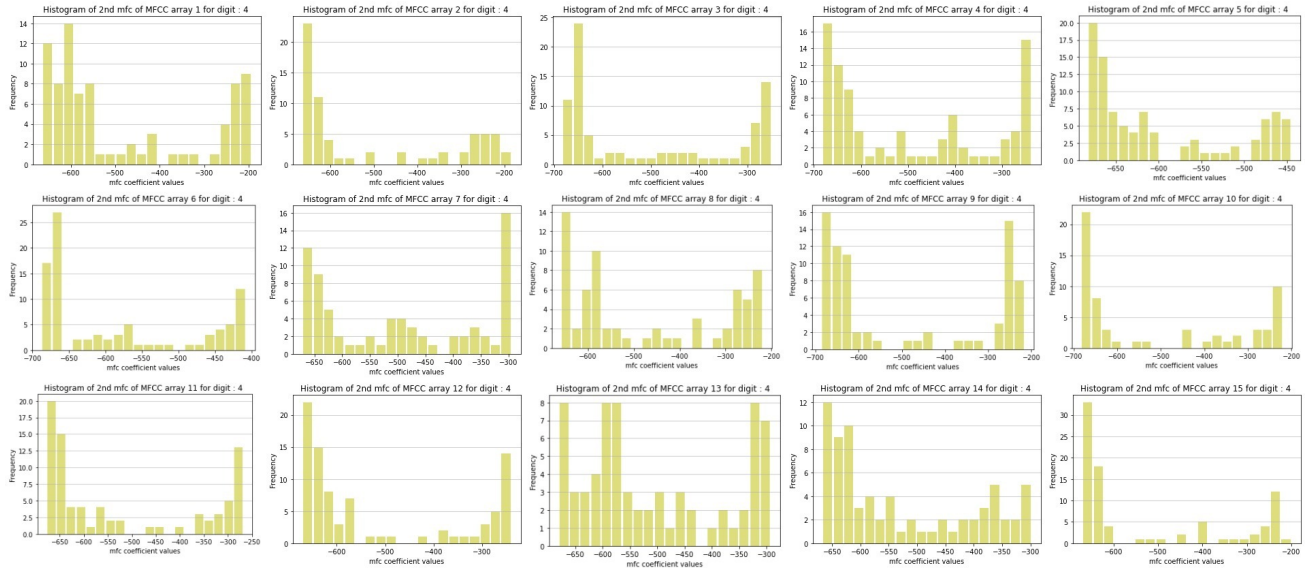


Figure 3: Histograms of the 2nd mfcc for digit 4

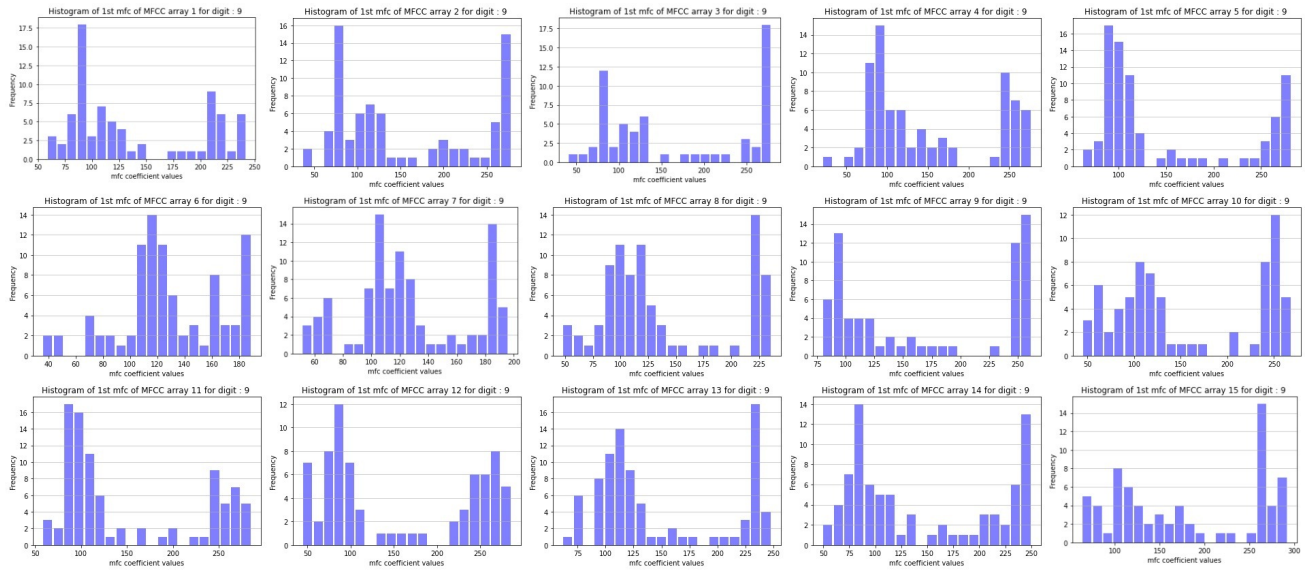


Figure 4: Histograms of the 1st mfcc for digit 9

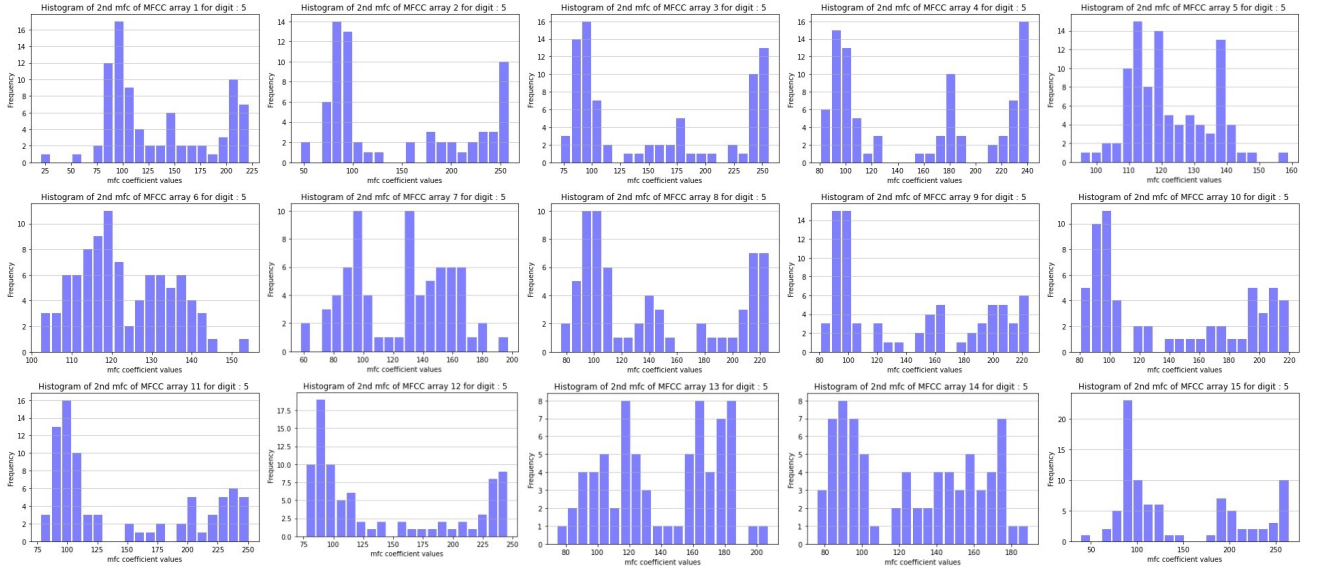


Figure 5: Histograms of the 2nd mfcc for digit 9

We then constructed the correlation plots.

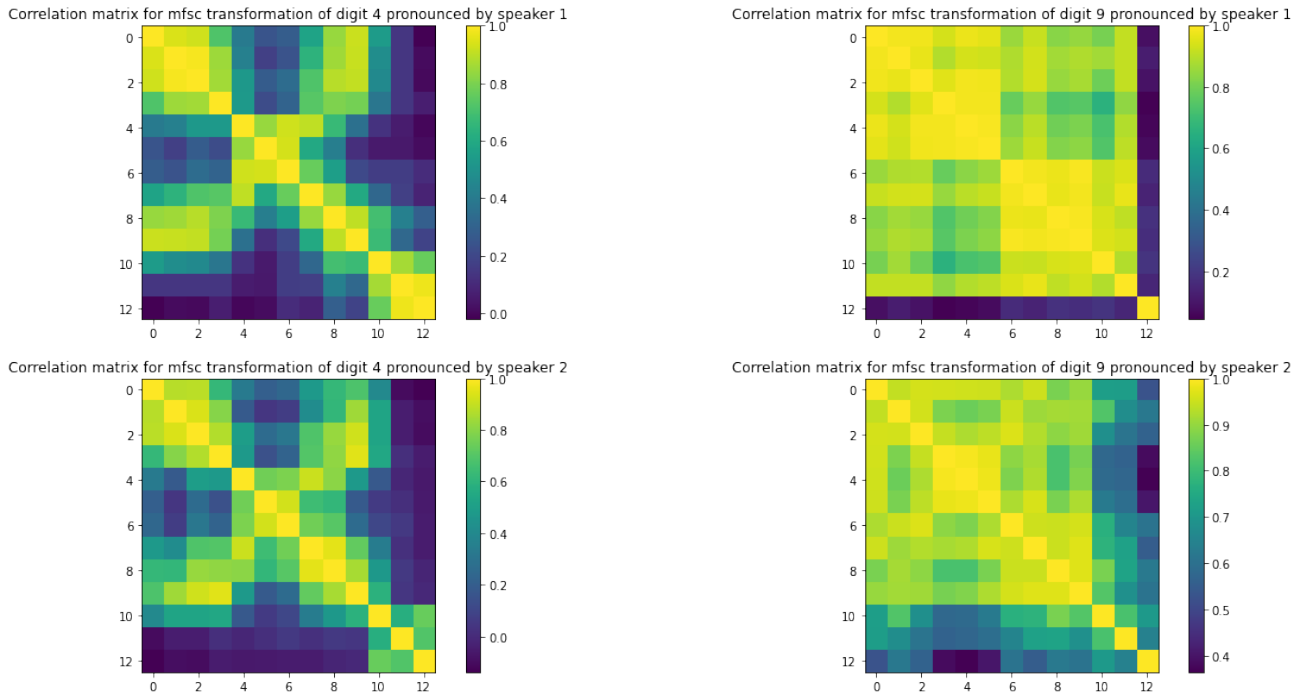


Figure 6: Correlation matrix for mfsc of digit 4 and 9 pronounced by 2 speakers

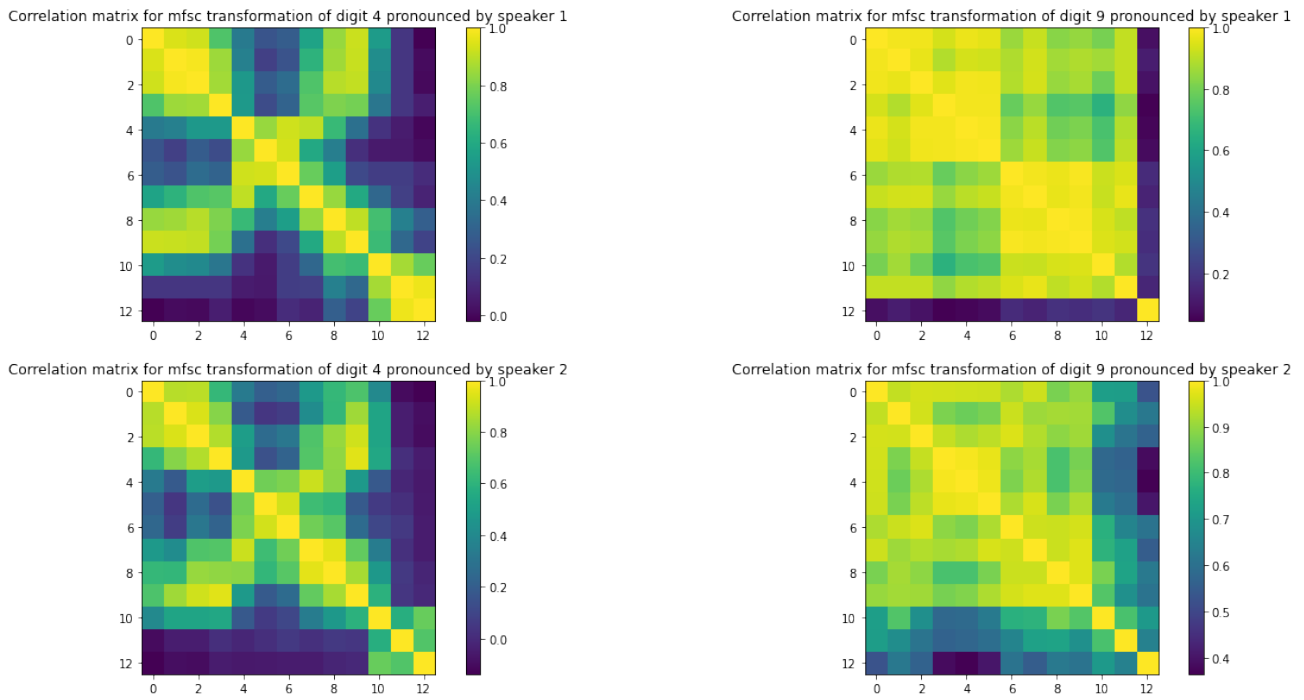


Figure 7: Correlation matrix for mfcc of digit 4 and 9 pronounced by 2 speakers

We observe that in the case of mfccs the values of the correlations are much closer to zero than those in the mfscs. The features in the mfsc case are more related. It is desired that the features are unrelated so that everyone can add new information that characterizes the sample and finally get as much and more condensed information as possible. This is a desirable attribute for learning features. That's why we prefer mfccs as descriptors for models.

## Step 5

We combine the mean value and the variance of the mfccs – deltas – delta-deltas for all the windows into one variable X. We can take a look at the characteristics of that variable:

```
print(len(X))
print(len(X[0]))
```

```
133
78
```

Figure 8: The characteristics of the X matrix.

Then, we scatter the first two dimensions of the resulting matrix. We achieve that with the use of *two\_dimensions\_plot* that classifies the rows of the matrix according to the digit they depict. We use different colours and shapes for data that depict different digits.

```
x, y = np.array([i[0] for i in vec]), np.array([i[1] for i in vec])
two_dimensions_plot(x, y, digit)
```

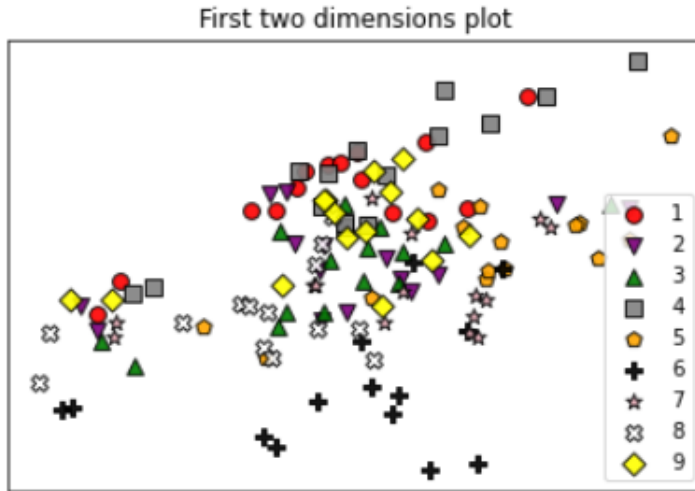


Figure 9: Scatter plot for the first two dimensions of the data.

As we can clearly see from the plot, digits cannot be distinguished based on only the first two vocal characteristics of our matrix. It is worth noting that even vocal words of the same digit are sparse and cannot be considered as dense clusters. Clusters are sparse and overlapping.

## Step 6

In order to achieve the PCA dimensionality reduction, we define a method, called *apply\_dim\_reduction* that receives the data and the number of dimensions to apply the PCA algorithm. It returns the data and, if asked explicitly, analyses the *explained\_variance\_ratio*.

Then, we apply the PCA algorithm for two and three dimensions. We also analyze the *explained\_variance\_ratio* for both dimensions and plot the results.

The analyzed ratio of two dimensions is calculated below.

```
pca = apply_dim_reduction(vec, 2, analyze_ratio=True)
[0.60235183 0.70894069]
```

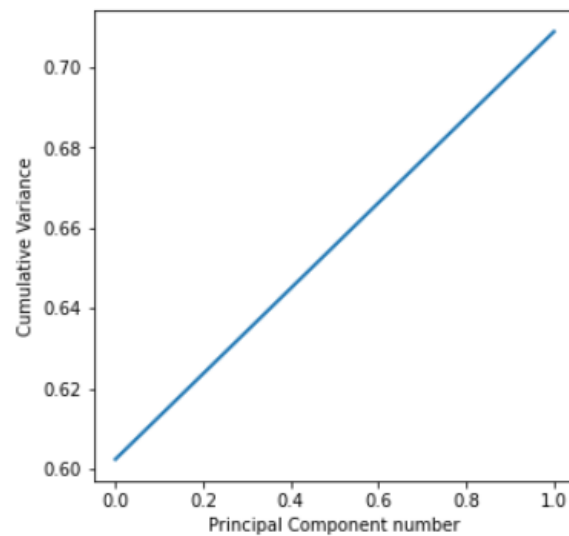


Figure 10: Plot of the explained variance ratio for two points.

It seems to be a linear function, although, only two points were used for that plot. We will have a better image, when we use three dimensions. We notice that even with only two dimensions, the variance explained of the model is high.

The resulting application of pca for two dimensions can be illustrated using the *two\_dimensions\_plot* method that we used before. We might be able to notice some regions more dense than others, the majority of the points contained, belonging to only one class.

```
two_dimensions_plot(pca, digit)
```

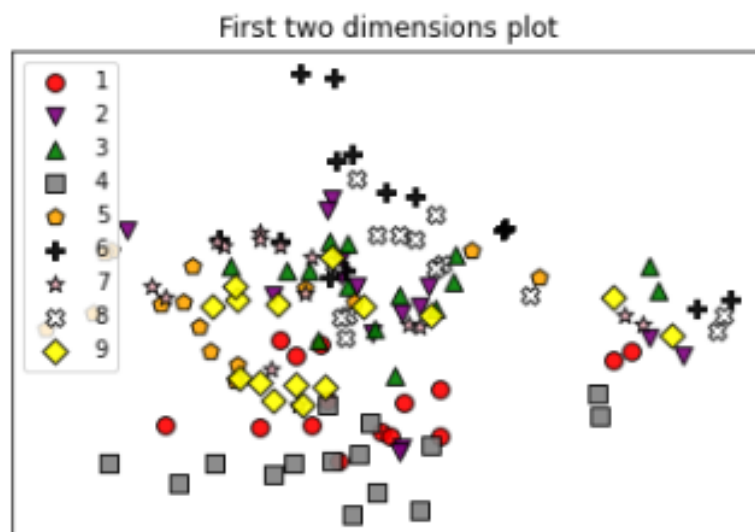


Figure 11: Two dimensions scatter plot, after the application of PCA algorithm.

As we can see, only a small part of the variance can be explained. Clusters are still sparse and overlapping.



Next we present the results for the application of PCA for three dimensions. The variance explained has increased almost linearly and some regions for each digit can now be discovered.

```
three_dimension_plot(vec, digit, analyze_ratio=True)
[0.60235183 0.70894069 0.81476645]
```

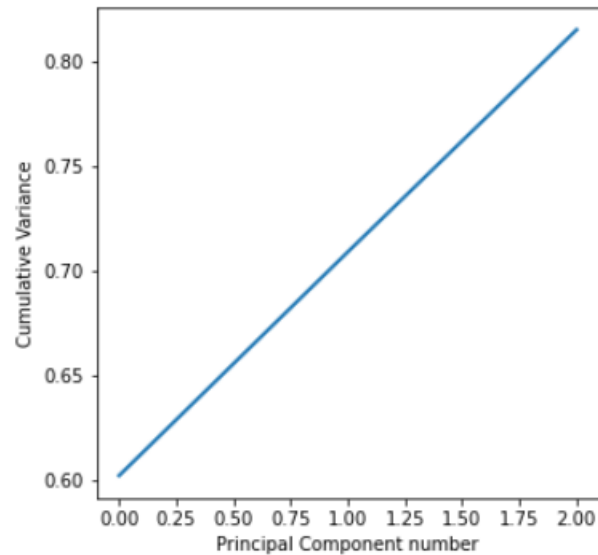


Figure 12: Plot representing the variance explained while adding dimensions until 3.

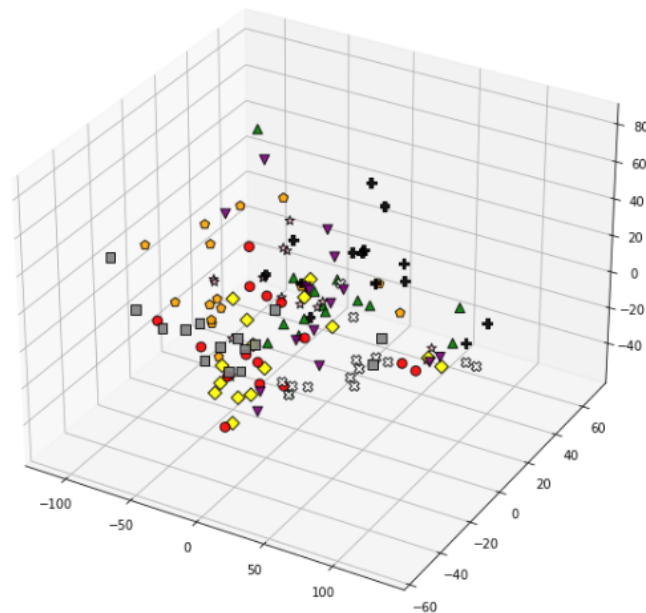


Figure 13: Three dimensions scatter plot, after the application of PCA algorithm.

```
three_dimension_plot(vec, digit, 1, 45, 60, analyze_ratio=False)
```

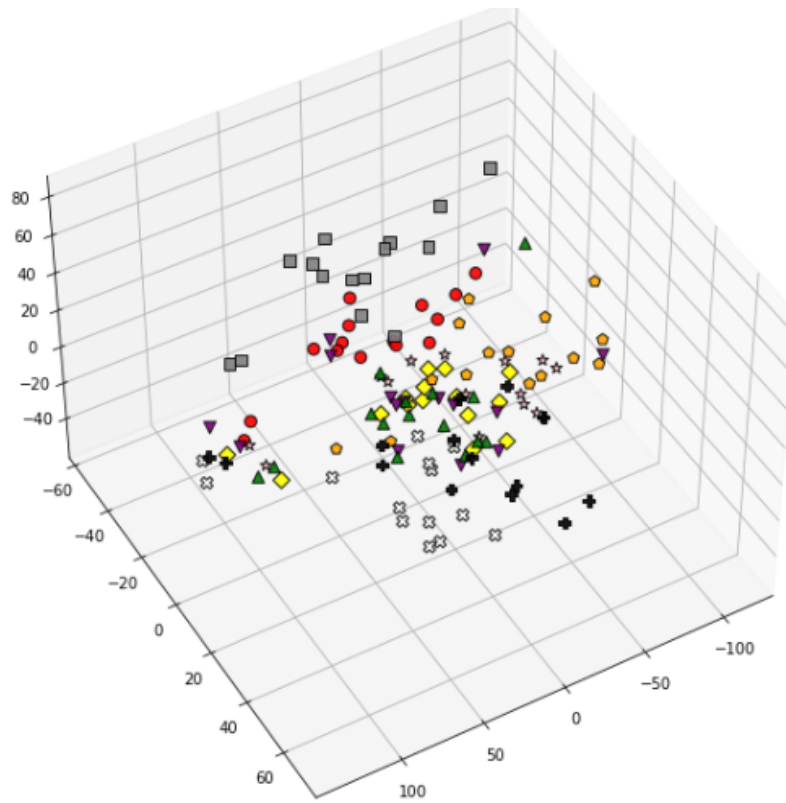


Figure 14: Three dimensions scatter plot, after the application of PCA algorithm. Different angles.

```
three_dimension_plot(vec, digit, 1, 0, 0, analyze_ratio=False)
```

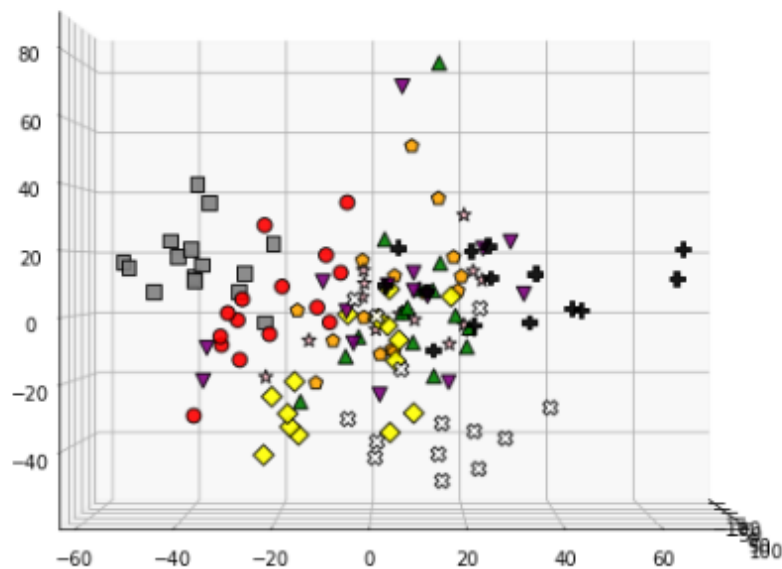


Figure 15: Three dimensions scatter plot, after the application of PCA algorithm. Different angles.

It is also interesting to plot the line of the explained variance for higher dimensions. We can get an intuition for our results after the use of two and three dimensions.

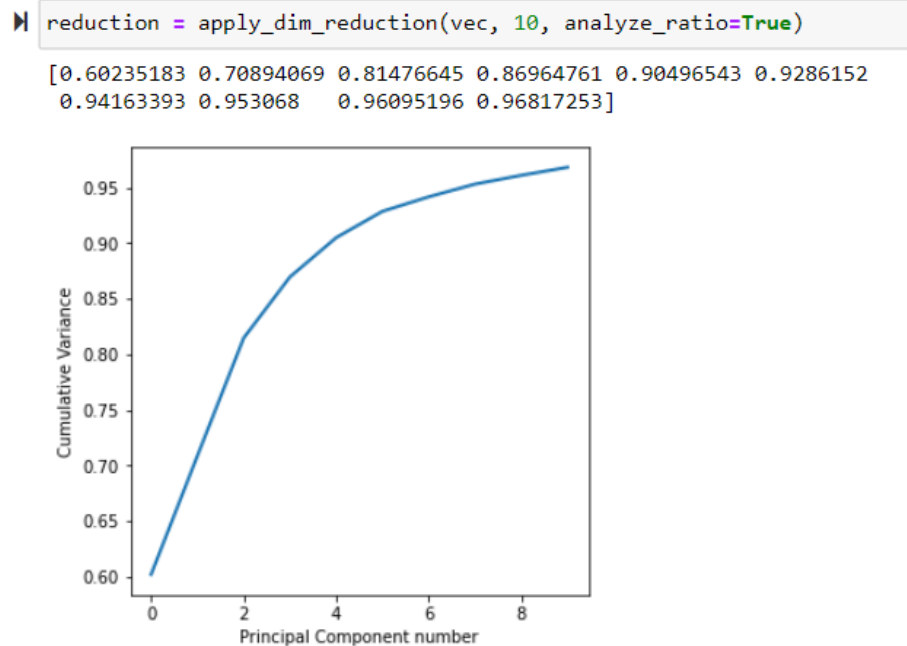


Figure 16: Explained ratio plot for 1 to 10 dimensions.

For  $n\_dimensions=6$ , variance explained surpasses 90%.

## Step 7

For the classification we firstly, split the data in train and test set in proportion 70%-30% and then performed a normalizing.

	Accuracy of initial data	Accuracy of normalized data
Custom GaussianNB	0.6 (smoothing = 1)	0.6 (smoothing = 1e-9)
Naive Bayes of sklearn	0.575	0.15
Nearest Neighbors classifier	0.575	0.425
SVM poly kernel	0.075	0.175
SVM linear kernel	0.75	0.175
SVM rbf kernel	0.075	0.175
Logistic Regression	0.65	0.175

## Step 8

We generated a 10-point sin and cosine waves with  $f = 40$  Hz and random amplitude.

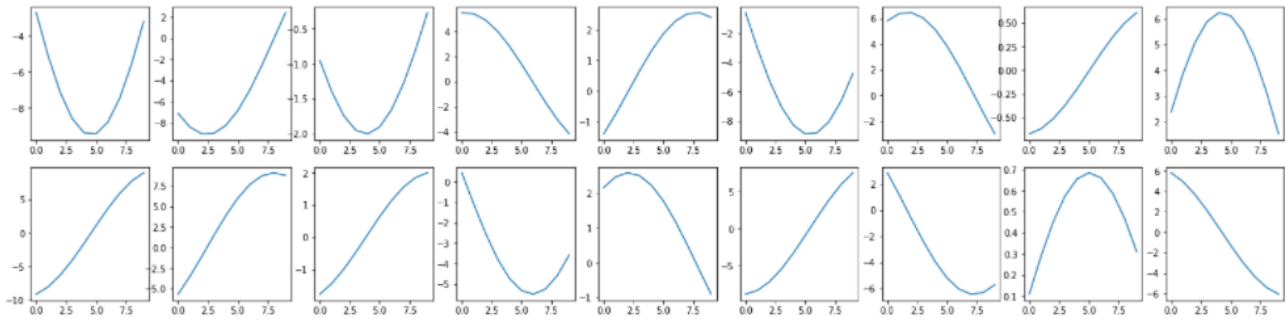


Figure 17:

## Step 9

We acquire and split the data to train and test set, using the predefined method *parser*. As we can see, this splits the data, conserving the percentage of each digit in the two datasets.

```
(unique, counts) = np.unique(y_train, return_counts=True)
frequencies = np.asarray((unique, counts)).T
print(frequencies)

[[ 0 240]
 [ 1 240]
 [ 2 240]
 [ 3 240]
 [ 4 240]
 [ 5 240]
 [ 6 240]
 [ 7 240]
 [ 8 240]
 [ 9 240]]
```

Figure 18: The resulting frequencies of each digit, after splitting the data.

We then apply a second split, in order to acquire a validation set, from the training set. The percentages remain.

```
(unique, counts) = np.unique(y_train, return_counts=True)
frequencies = np.asarray((unique, counts)).T
print(frequencies)

[[ 0 192]
 [ 1 192]
 [ 2 192]
 [ 3 192]
 [ 4 192]
 [ 5 192]
 [ 6 192]
 [ 7 192]
 [ 8 192]
 [ 9 192]]
```

Figure 19: The resulting frequencies of each digit, after the second split of the data.

## Step 10

We create a class, called *HMM\_model* that creates an HMM model, using Gaussian Mixture Models. All parameters for these models are already predefined (such as the start and end states) or they are defined by the *X\_train* dataset and the assumptions we make about our models.

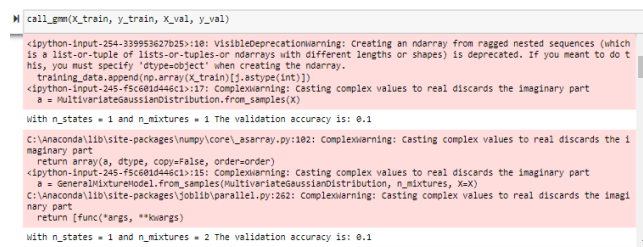
X is an np array containing the flattened elements of the X\_train dataset. Then, we calculate the GMMs and define the transition matrix, the start and end states and the data list. We continue by defining our HMM model, that receives all these parameters as inputs, plus the number of states and mixtures.

We also fit the model in the init method and define the return\_model to access it. Also, we define the predict method, that returns the prediction of the model for a particular sample.

Then, we define the *predictions\_HMM* method that, for a given sample, it calculates the predictions for all models passed in, and returns the model with the best performance. This method will be useful later on, when we will define different models (with different number of states and mixtures) and find the best parameters.

## Steps 11, 12

The *call\_hmm* method creates 10 HMM models (one for each digit) for different parameters (number of states from 1 to 4 and number of mixtures from 1 to 5). It then fits these models by using the *fit* method of the HMM\_model class and calculates the scores for the validation set. It prints the results and helps us choose the best parameters.



```

M call_hmm(X_train, y_train, X_val, y_val)

<ipython-input-254-339953627025>:10: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which
is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do t
his, you must specify 'dtype=object' when creating the ndarray.
  training_data.append(np.array(X_train[i].astype(int)))
<ipython-input-245-f5ce01d446c1>:17: ComplexWarning: Casting complex values to real discards the imaginary part
  a = MultivariateGaussianDistribution.from_samples(X)

With n_states = 1 and n_mixtures = 1 The validation accuracy is: 0.1

C:\Anaconda\lib\site-packages\numpy\core\asarray.py:102: ComplexWarning: Casting complex values to real discards the i
maginary part
  return array(a, dtype, copy=False, order=order)
<ipython-input-245-f5ce01d446c1>:15: ComplexWarning: Casting complex values to real discards the imaginary part
  a = GeneralMixtureModel.from_samples(MultivariateGaussianDistribution, n_mixtures, X=X)
C:\Anaconda\lib\site-packages\joblib\parallel.py:262: ComplexWarning: Casting complex values to real discards the imagi
nary part
  return [func(*args, **kwargs)
With n_states = 1 and n_mixtures = 2 The validation accuracy is: 0.1

```

Figure 20: The calculated scores for each pair of parameters.

Unfortunately, due to the many and complicated dependencies of the pomegranate package, we weren't able to run the models properly and get good results. We assume that the readers have installed properly the packages and run them for us. We expect much better results.

## Step 13

The confusion matrix is calculated for the best model. We found that this is when n\_states = 3 and n\_mixtures=5. Then, in one step, we calculate the confusion matrix and plot it, by calling the *plot\_confusion\_matrix* function. The results are biased due to the problem that we encountered, but they should look like the following plot., only with much higher scores.

```
plot_confusion_matrix(cm, range(10))
```

Confusion matrix, without normalization

```
[[ 0  0  5  0  0  0  0  0  0  55]
 [ 0  0  8  0  0  0  0  0  0  52]
 [ 0  0  2  0  0  0  0  0  0  58]
 [ 0  0  3  0  0  0  0  0  0  57]
 [ 0  0  5  0  0  0  0  0  0  55]
 [ 0  0  7  0  0  0  0  0  0  53]
 [ 0  0  4  0  0  0  0  0  0  56]
 [ 0  0  1  0  0  0  0  0  0  59]
 [ 0  0  4  0  0  0  0  0  0  56]
 [ 0  0  7  0  0  0  0  0  0  53]]
```

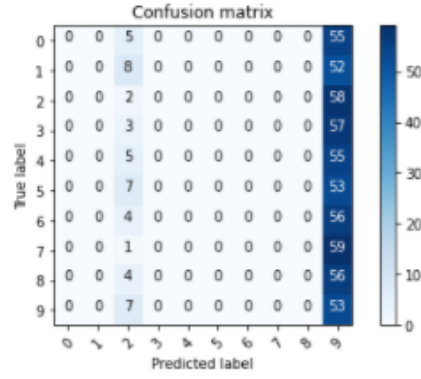


Figure 21: The calculated confusion matrix.

## Step 14

We defined an LSTM neural network

Table 3: The training loss for each epoch for 1 layer neural network

2.2409169112934784
1.987085857812096
1.7017268889090593
1.4841810254489674
1.3040842974887175
1.1343549209482529
1.0061117656090681
0.8953831318546744
0.7968143382493187
0.7322736329892102
0.6531374103882733
0.5952838562867221
0.5343314716044594
0.4850640673847759
0.4616855049834532
0.39586954520029183
0.3546861721312298
0.33129440160358653
0.30404119237380867
0.29502584215472727

0.3004431343253921
0.2650107513455784
0.28514480327858643
0.24455993929330042
0.2316960219074698
...
0.10767549265395193
0.1326697003096342
0.11224011304404806
0.14731035600690282
0.11651127226650715

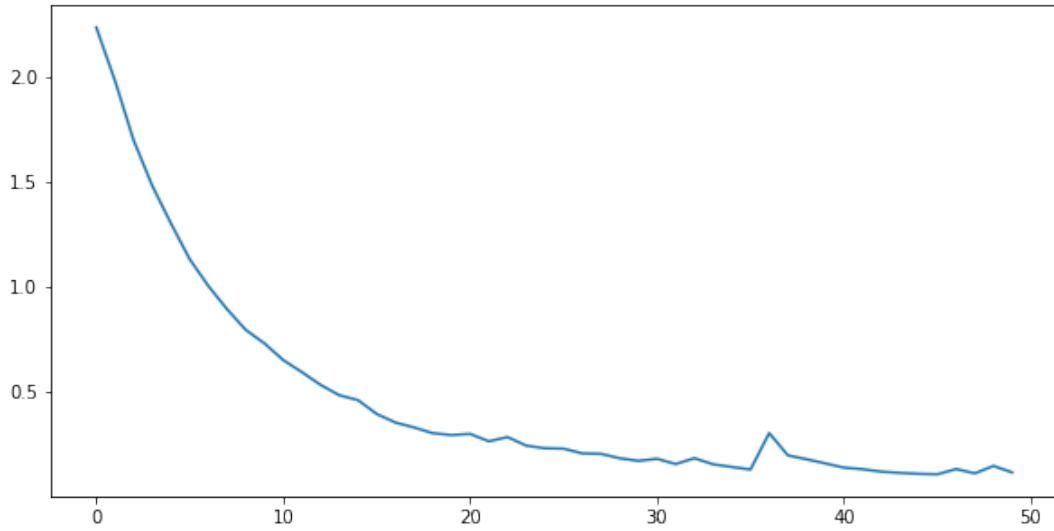


Figure 22: Training loss for every epoch for 1 layer neural network

We then trained the network, printing both the training and the validation loss in each epoch, resulting in the results below.

Table 4: The training and validation loss for each epoch for 1 layer neural network

Train:	2.252935
Validation:	2.254621
Train:	2.020030
Validation:	2.028776
Train:	1.668168
Validation:	1.690932
Train:	1.417545
Validation:	1.434813
Train:	1.224474
Validation:	1.251102

Train:	1.070784
Validation:	1.110195
Train:	0.923928
Validation:	0.952621
Train:	0.799437
Validation:	0.821259
Train:	0.704685
...	...
Validation:	0.252716
Train:	0.066726
Validation:	0.266055

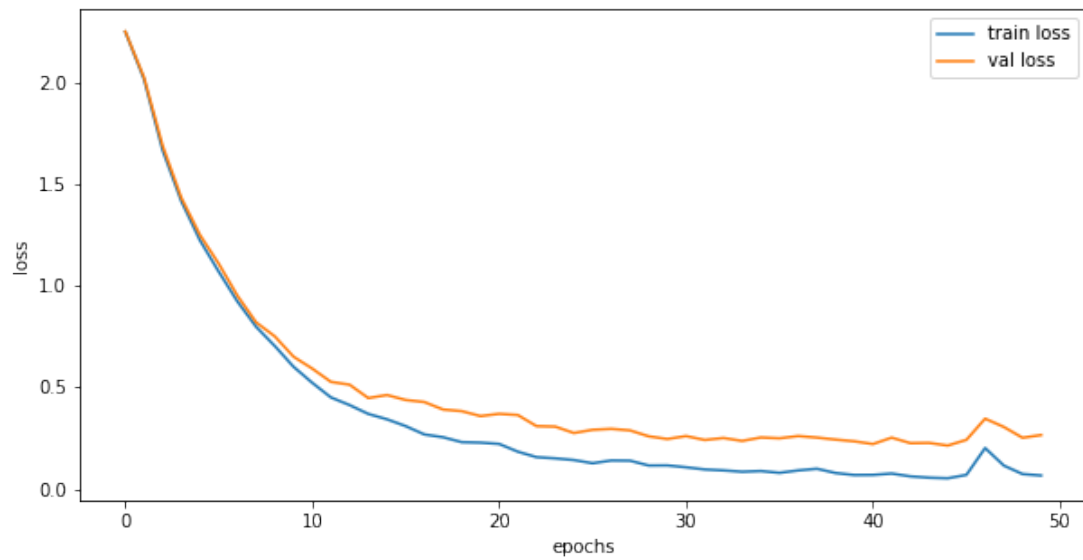


Figure 23: Training and validation loss for every epoch for 1 layer neural network

Loss:	0.280798
Accuracy:	0.924074

Table 6: The training and validation loss for each epoch for 2 layer neural network

Train:	2.263566
Validation:	2.263903
Train:	1.881201
Validation:	1.892562
Train:	1.454683
Validation:	1.483199
Train:	1.173927
Validation:	1.224016



Train:	1.047289
Validation:	1.111987
Train:	0.865622
Validation:	0.936778
Train:	0.794466
Validation:	0.834854
Train:	0.643456
Validation:	0.735317
Train:	0.553863
...	...
Validation:	0.180034
Train:	0.016223
Validation:	0.163059
Validation:	0.266055

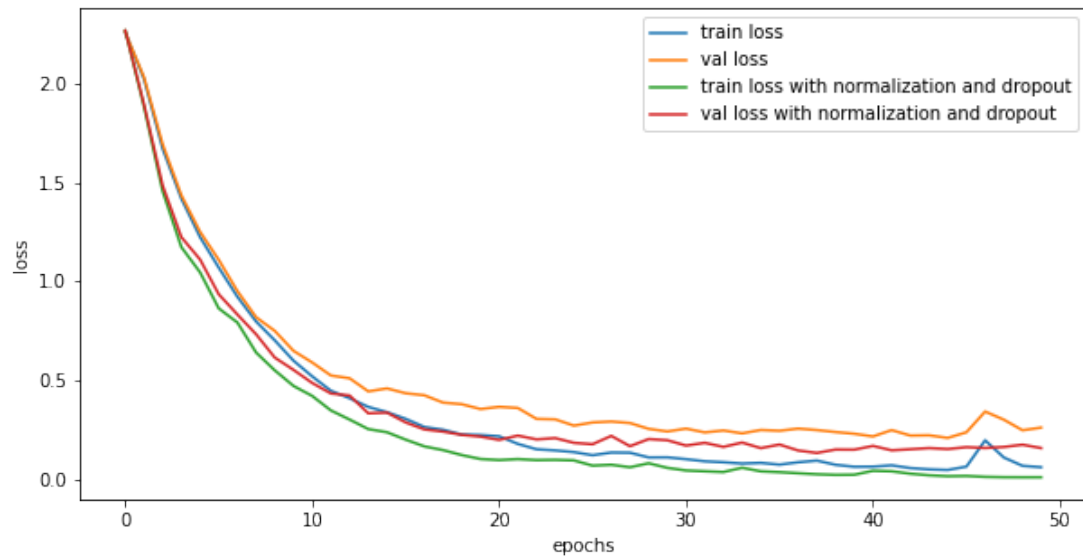


Figure 24: Training and validation loss for every epoch for 2 layer neural network

Loss:	0.147690
Accuracy:	0.961111

We then trained the 2 layer neural network using Early Stoppings and Checkpoints.

Table 8: The training and validation loss for each epoch for 2 layer neural network with early stopping and checkpoints

Train:	2.199549
Validation:	2.204038
Train:	1.729519

Validation:	1.755082
Train:	1.310663
Validation:	1.337638
Train:	0.996537
Validation:	1.022014
Train:	0.812357
Validation:	0.835686
Train:	0.705500
Validation:	0.749238
Train:	0.555586
Validation:	0.584471
Train:	0.477743
Validation:	0.508881
Train:	0.429077
...	...
Train:	0.030685
Validation:	0.127661
Early	stopping!

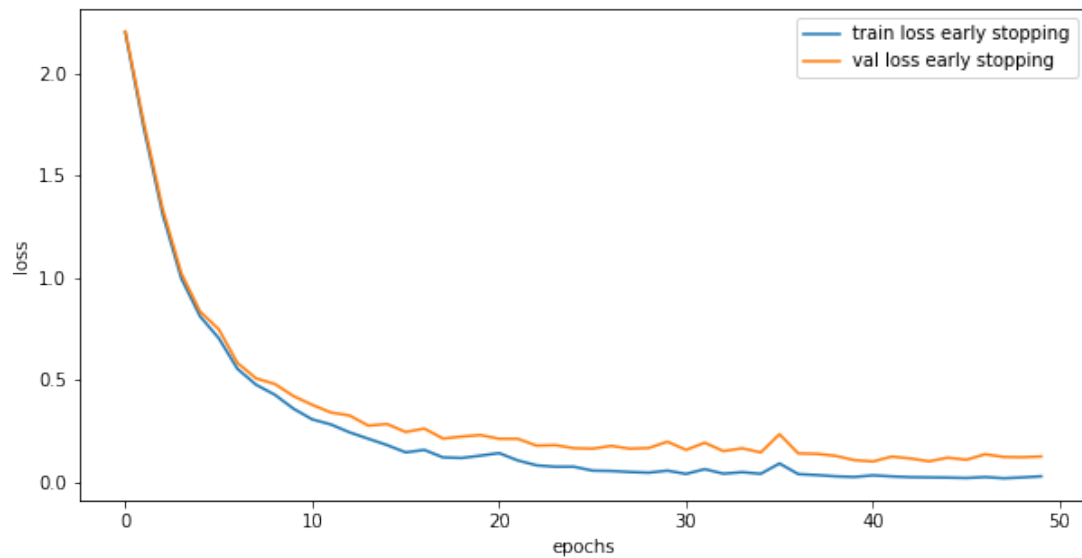


Figure 25: Training and validation loss for every epoch for 2 layer neural network using early stopping and checkpoints

Loss:	0.169462
Accuracy:	0.950000

We then trained a bidirectional LSTM with early stopping and checkpoints.

Table 10: The training and validation loss for each epoch for 2 layer bidirectional neural network with early stopping and checkpoints

Train:	2.099903
Validation:	2.101405
Train:	1.238116
Validation:	1.267836
Train:	0.715955
Validation:	0.744639
Train:	0.489546
Validation:	0.549359
Train:	0.347067
Validation:	0.415792
Train:	0.265868
Validation:	0.339673
Train:	0.218301
Validation:	0.286358
Train:	0.188495
Validation:	0.275127
Train:	0.143785
...	...
Train:	0.009660
Validation:	0.090201
Early	stopping!

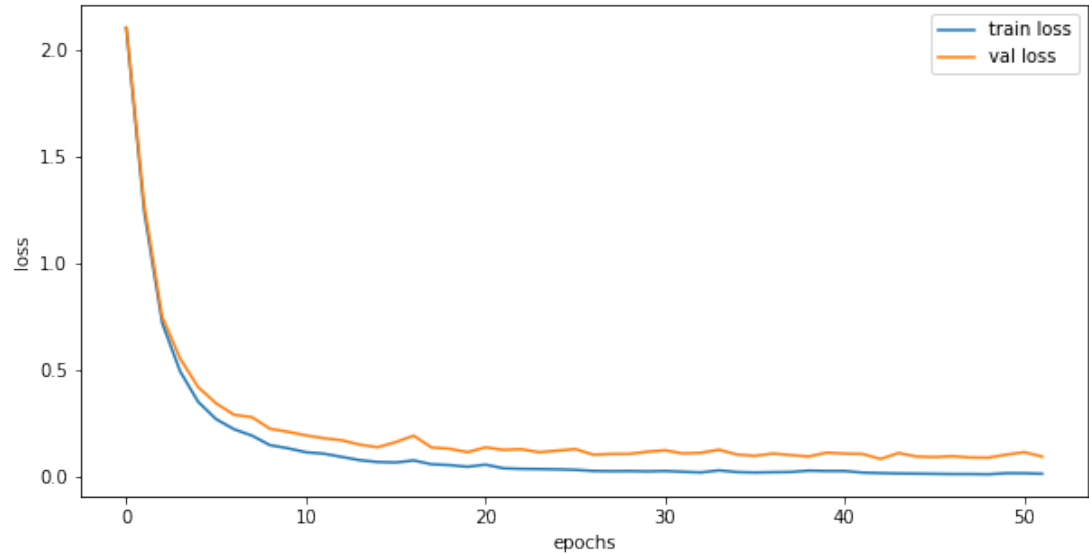


Figure 26: Training and validation loss for every epoch for 2 layer bidirectional neural network using early stopping and checkpoints

Loss:	0.084105
Accuracy:	0.972222

We then used pack padded sequence in the training of our model.

Table 12: The training and validation loss for each epoch for 2 layer neural network with early stopping and checkpoints, using pack padded sequence

Train:	2.131211
Validation:	2.129129
Train:	1.496319
Validation:	1.531903
Train:	0.967939
Validation:	0.984349
Train:	0.695209
Validation:	0.741859
Train:	0.523238
Validation:	0.569856
Train:	0.408503
Validation:	0.464315
Train:	0.316323
Validation:	0.374376
Train:	0.293108
Validation:	0.365988
Train:	0.225018
...	...
Train:	0.021956
Validation:	0.108345
Early	stopping!

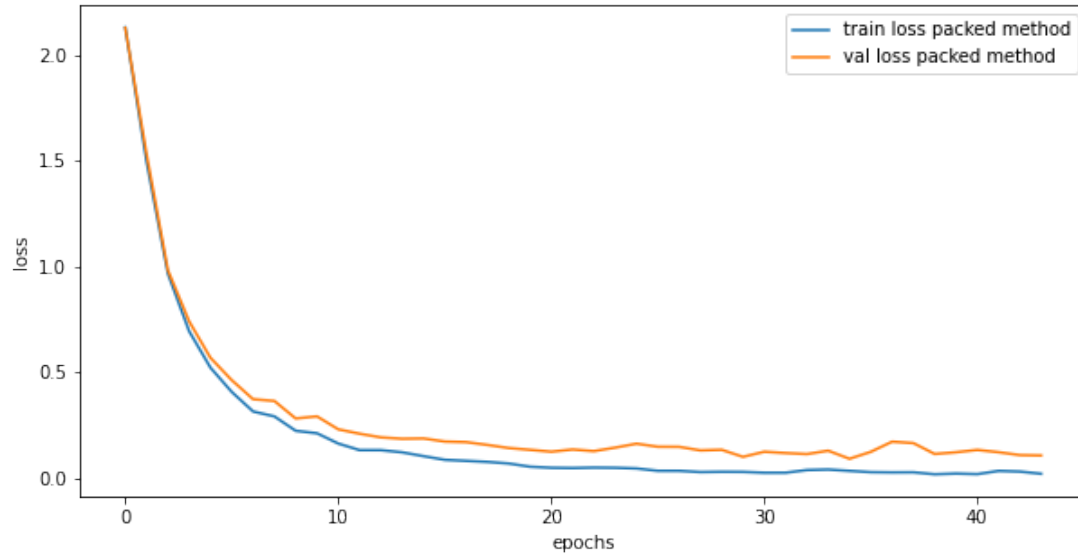


Figure 27: Training and validation loss for every epoch for 2 layer neural network with early stopping and checkpoints, using pack padded sequence

Loss: 0.114894  
Accuracy: 0.970370

For the best model which is the bidirectional packed with 2 layers the loss and the accuracy are:

Loss: 0.059567  
Accuracy: 0.986667

The confusion matrix is:

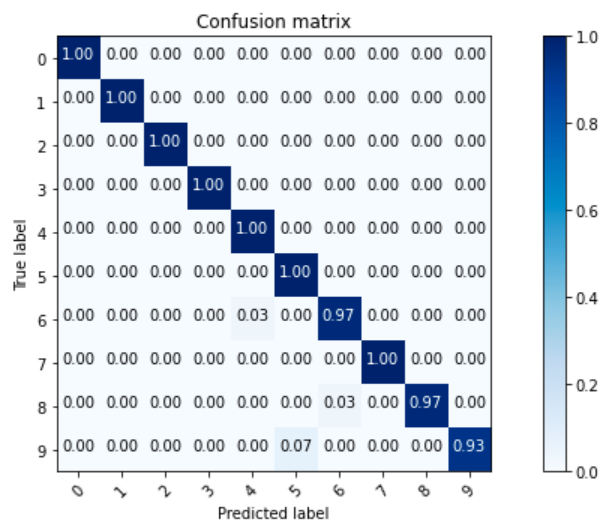


Figure 28: Confusion matrix for the best model

## References

- [1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [2] G. Karagiannis and G. Steinhauer. *Pattern Recognition and Machine Learning*. NTUA, 2001.
- [3] P.E. Hart R. O. Duda and D.G. Stork. *Pattern Classification*. Wiley, 2001.
- [4] S. Theodoridis and K. Koutroumbas. *Pattern Recognition*. Academic Press, 2009.