LAB 4

SPEECH RECOGNITION USING DYNAMIC TIME WARPING

POE

DATA SCIENCE AND ENGINEERING
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Adrián Cerezuela Hernández - 48222010A

Ramon Ventura Navarro - 21785256R

May 2024

Contents

1 Introduction

In this lab assignment we will work on a simple speech recognition task: digit recognition using *Dynamic Time Warping*.

Dynamic Time Warping (DTW) is a method used to measure the similarity between two sequences that may vary in time or speed. It works by finding an optimal alignment between the two sequences by warping them in the time dimension. It achieves this by iteratively stretching or compressing the sequences to minimize the overall distance between corresponding points. This allows DTW to capture similarities even when sequences have different lengths or exhibit temporal distortions.

Taking that into account, and given an initial notebook with the algorithm already implemented, we will attempt to make various modifications in order to enhance its performance, thus yielding better results.

2 Evaluate the Speak Error Rate (SER)

The DWT baseline notebook includes a function called *wer()* which computes the Word Error Rate (WER), indicating the likelihood of errors in word classification tasks. By default, it compares the 'text' of the correct response with the 'text' of the labeled recording that closely matches each of the given test wav files.

The initial objective is to adapt it in order to assess the Speaker Error Rate (SER), a metric used for evaluating the effectiveness of a speaker identification system.

Therefore, our task involves identifying the line responsible for error counting and replacing 'text' with 'speaker' to increment error count if the speaker of the closest recording differs from the speaker of the test signal.

Additionally, we are tasked with determining the likelihood of correctly identifying the speaker in the free-spoken-digit-dataset by using DTW distance in comparison to the remaining recordings.

With the given *wer()* function the obtained results are:

- WER including reference recordings from the same speaker: 6.9%
- WER using only reference recordings from other speakers: 28.9%

The implementation of the *ser()* function is the following:

```python
def ser(test_dataset, ref_dataset=None, same_spk=False):
    # Compute mfcc
    test_mfcc = get_mfcc(test_dataset)
    if ref_dataset is None:
        ref_dataset = test_dataset
        ref_mfcc = test_mfcc
    else:
        ref_mfcc = get_mfcc(ref_dataset)

    err = 0
    for i, test in enumerate(test_dataset):
        mincost = np.inf
        minref = None
        for j, ref in enumerate(ref_dataset):
            if not same_spk and test['speaker'] == ref['speaker']:
                # Do not compare with refrence recordings of the same speaker
                continue
            if test['wav'] != ref['wav']:
                distance = dtw(test_mfcc[i], ref_mfcc[j])
                if distance < mincost:
                    mincost = distance
                    minref = ref
        if test['speaker'] != minref['speaker']:
            err += 1

    ser = 100*err/len(test_dataset)
    return ser
```

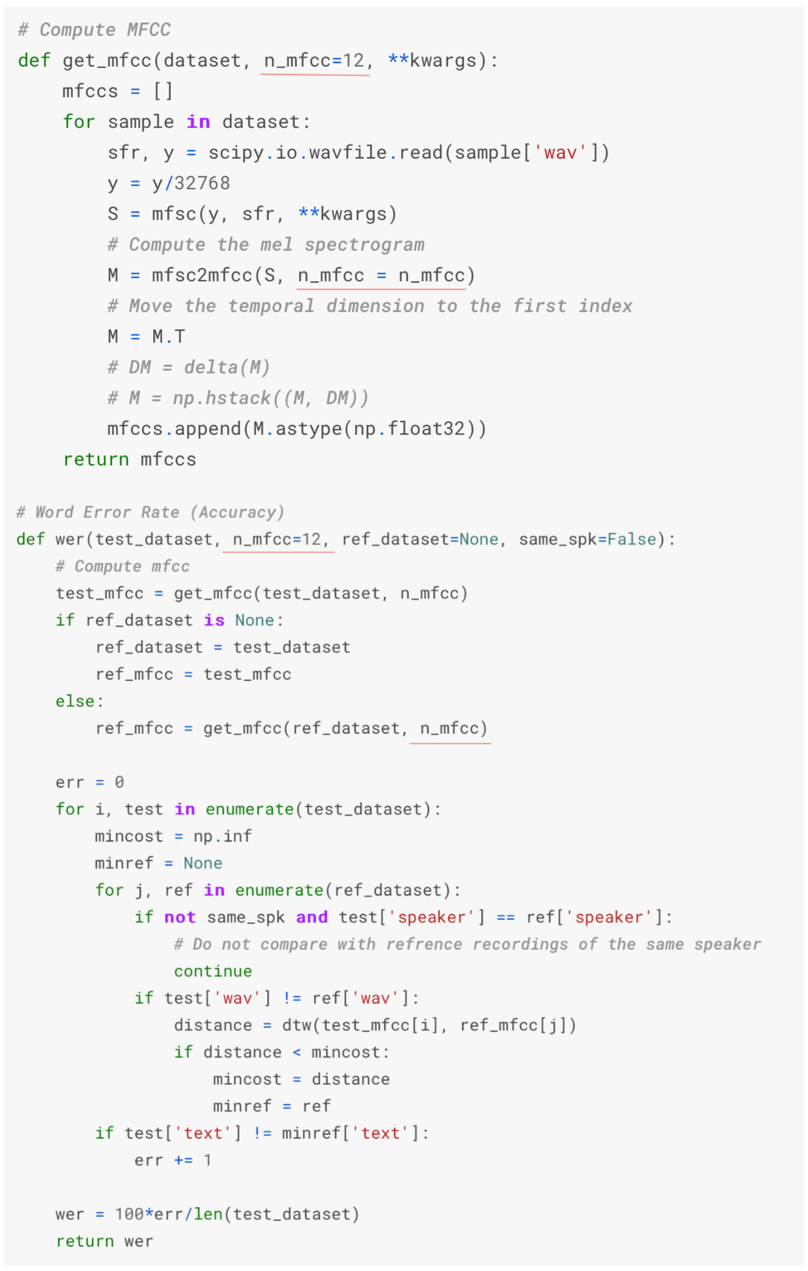
The metrics obtained, along with those obtained using the *wer()* function, are as follows:

	WER	SER
Including reference recordings from the same speaker	6.9%	4.4%
Using only reference recordings from other speaker	28.9%	100.0%

As can be seen in the table, the SER (Speaker Error Rate) including reference recordings from the same speaker is 4.4%, which is lower than the previous one with the *wer()* function. However, the SER using only reference recordings from other speakers is 100.0%, as expected, since it is impossible to guess a speaker if we do not have them in the training dataset. We also observe that it is easier for it to guess the speaker rather than the words being spoken.

3 Cepstral Coefficients Comparative Analysis

Following the first task, in this second one we are asked to perform a comparative analysis between performances of our model changing the number of cepstral coefficients computed for each speech frame and how does it affects the WER. With that goal, we will check values from 0 to 12 and see which one obtains better results. If we observe a decreasing tendency in our metrics, then more numbers will be tried. Howerver, we expect an optimal value within that range. This has been implemented as follows:



```python
# Compute MFCC
def get_mfcc(dataset, n_mfcc=12, **kwargs):
    mfccs = []
    for sample in dataset:
        sfr, y = scipy.io.wavfile.read(sample['wav'])
        y = y/32768
        S = mfsc(y, sfr, **kwargs)
        # Compute the mel spectrogram
        M = mfsc2mfcc(S, n_mfcc = n_mfcc)
        # Move the temporal dimension to the first index
        M = M.T
        # DM = delta(M)
        # M = np.hstack((M, DM))
        mfccs.append(M.astype(np.float32))
    return mfccs

# Word Error Rate (Accuracy)
def wer(test_dataset, n_mfcc=12, ref_dataset=None, same_spk=False):
    # Compute mfcc
    test_mfcc = get_mfcc(test_dataset, n_mfcc)
    if ref_dataset is None:
        ref_dataset = test_dataset
        ref_mfcc = test_mfcc
    else:
        ref_mfcc = get_mfcc(ref_dataset, n_mfcc)

    err = 0
    for i, test in enumerate(test_dataset):
        mincost = np.inf
        minref = None
        for j, ref in enumerate(ref_dataset):
            if not same_spk and test['speaker'] == ref['speaker']:
                # Do not compare with refrence recordings of the same speaker
                continue
            if test['wav'] != ref['wav']:
                distance = dtw(test_mfcc[i], ref_mfcc[j])
                if distance < mincost:
                    mincost = distance
                    minref = ref
        if test['text'] != minref['text']:
            err += 1

    wer = 100*err/len(test_dataset)
    return wer
```

```
for n in range(13):
    print("--------------------------")
    print("Number of cepstral coefficients -> n_mfcc: " + str(n))

    # Free Spoken Digit Dataset
    print(f'WER including reference recordings from the same speaker: {wer(free10x4x4, n_mfcc
= n, same_spk=True):.1f}%')
    # Google Speech Commands Dataset (small digit subset)
    print(f'WER using only reference recordings from other speakers: {wer(commands10x100, n_m
fcc = n, ref_dataset = commands10x10):.1f}%')
```

The results for each of the tested values are shown below:

Number of cepstral coefficients (n_mfcc)	0	1	2	3	4	5
WER including reference recordings from the same speaker	90.0%	59.4%	37.5%	25.0%	16.9%	16.2%
WER using only reference recordings from other speaker	90.0%	76.7%	50.0%	41.7%	30.2%	27.5%

Number of cepstral coefficients (n_mfcc)	6	7	8	9	10	11	12
WER including reference recordings from the same speaker	12.5%	8.8%	8.1%	7.5%	6.2%	5.6%	6.9%
WER using only reference recordings from other speaker	27.6%	27.8%	26.3%	24.3%	26.0%	27.5%	28.9%

Judging by the table and the values obtained, a clear decreasing trend can be observed in both cases for the WER. However, we notice how upon reaching *n_mfcc = 10*, the WER using only reference recordings from other speakers starts to increase again. Additionally, starting from 12 cepstral coefficients, the WER including reference recordings from the same speaker also increases. Therefore, considering that we are aiming for a system that recognizes the speech of a new speaker, we will choose 9 cepstral coefficients, the last value before the increase in values. From this point onwards, that value for the variable *n_mfcc* will be used.

4 Cepstral normalization and liftering analysis

Next, the system's behavior will be analyzed regarding mean and variance normalization and the presence or absence of liftering. A liftering value of 22, corresponding to the sinus window size, will be used. All combinations of normalized variables and liftering values, along with the corresponding metrics from their executions, are shown below:

Liftering	cms	cmvn	WER including same speaker	WER using other speakers
22	True	True	7.5%	38.2%
22	False	True	7.5%	38.2%
22	True	False	5.6%	28.2%
22	False	False	2.5%	41.4%
0	True	True	7.5%	24.3%
0	False	False	6.9%	54.6%
0	True	False	9.4%	39.8%
0	False	True	7.5%	24.3%

From the table displayed, we can observe that the use of liftering does not improve the previously obtained results in any case. Even when normalizing the mean and not the variance, it only slightly approaches, but the WER using other speakers remains higher, so it is not considered an improvement. If we look at those systems without liftering, we can see that we obtain the same result whether we maintain both normalization processes or only normalize the variance, which are the ones already presented in the previous section, with WER of 7.5% and 24.3% respectively in each case.

Therefore, we are not interested in modifying anything from the previous code, and we conclude that the system improves with mean and variance normalization and without the use of liftering. From here onwards, we will continue with this system configuration.

5 Extension of the MFCC parameters with the first-order derivatives

In this new section, we have now enhanced the MFCC (Mel-Frequency Cepstral Coefficients) parameters by incorporating the first-order derivatives, which are also known as delta coefficients. The first-order derivatives provide additional temporal information by capturing the rate of change of the cepstral features over time. As we know, this temporal information is crucial for improving the performance of models related to speech recognition.

The specific code utilized for this section is as follows:

```
# Compute the first-order derivatives
def delta(features):
    delta = np.zeros_like(features)
    num_frames, num_features = features.shape
    for i in range(1,num_features-1):
        delta[:,i] = (features[:,i+1] - features[:,i-1]) / 2

    return delta
```

```
# Compute MFCC
def get_mfcc(dataset, n_mfcc=9, **kwargs):
    mfccs = []
    for sample in dataset:
        sfr, y = scipy.io.wavfile.read(sample['wav'])
        y = y/32768
        S = mfsc(y, sfr, **kwargs)
        # Compute the mel spectrogram
        M = mfsc2mfcc(S, n_mfcc = n_mfcc, cms=True, cmvn=True)
        # Move the temporal dimension to the first index
        M = M.T
        DM = delta(M)
        M = np.hstack((M, DM))
        mfccs.append(M.astype(np.float32))
    return mfccs
```

	WER
Including reference recordings from the same speaker	6.2%
Using only reference recordings from other speaker	24.5%

By incorporating these first-order derivatives into our feature set, we have observed notable improvements in the Word Error Rate (WER) of our speech recognition model. Specifically, the inclusion of reference recordings from the same speaker resulted in a WER of 6.2%, while the use of reference recordings from different speakers yielded a WER of 24.5%. These results are significant because they highlight the model's enhanced ability to generalize and accurately transcribe speech, even when the reference recordings are from different speakers.

This improvement was anticipated, as the first-order derivatives provide the model with dynamic information about the features. The additional temporal context allows the model to better understand the variations and transitions in the speech signal, leading to more accurate recognition performance.

Furthermore, the improvement in WER demonstrates the effectiveness of combining static and dynamic features in speech recognition tasks. The static MFCC features capture the essential spectral properties of the speech signal, while the first-order derivatives complement this by adding information about the temporal dynamics. This combination provides a more comprehensive representation of the speech signal.

6 Alignment (backtracking)

Finally, we have implemented a backtracking algorithm to determine the minimum alignment path between two sequences. The distance metric used in our implementation is the Euclidean distance, which measures the straight-line distance between points in multi-dimensional space. This metric is particularly suitable for continuous data where the magnitude of differences is significant.

Here is the specific code utilized for implementing the backtracking algorithm:

```python
from numba import jit
@jit
def dtw(x, y, metric='sqeuclidean'):
    """
      Computes Dynamic Time Warping (DTW) of two sequences.
      :param array x: N1*M array
      :param array y: N2*M array
      :param func dist: distance used as cost measure
    """
    r, c = len(x), len(y)

    D = np.zeros((r + 1, c + 1))
    D[0, 1:] = np.inf
    D[1:, 0] = np.inf

      # Initialize the matrix with dist(x[i], y[j])
    D[1:, 1:] = scipy.spatial.distance.cdist(x, y, metric)

    for i in range(r):
        for j in range(c):
            min_prev = min(D[i, j], D[i+1, j], D[i, j+1])
            D[i+1, j+1] = dist(x[i], y[j]) + min_prev
            #D[i+1, j+1] += min_prev

    if len(x) == 1:
        path = zeros(len(y)), range(len(y))
    elif len(y) == 1:
        path = range(len(x)), zeros(len(x))
    else:
        path = _traceback(D)

    return D[-1, -1], path
```

```python
def _traceback(D):
    """
    Performs backtracking throught the D matrix to find the optimal path
    """
    n, m = np.array(D.shape) - 2
    path = [(n, m)]
    while (n, m) != (0, 0):
        val = np.argmin((D[n, m], D[n, m + 1], D[n + 1, m]))
        if val == 0:
            n -= 1
            m -= 1
        elif val == 1:
            n -= 1
        else:  # (tb == 2)
            m -= 1
        path.append((n, m))
    path.reverse()
    return path
```

```python
def dist(x,y):
    """
    Returns the Euclidean distance between two sequences
    """
    return np.sum((x-y)**2)
```

Through backtracking, we can determine the minimum alignment path between two sequences, using Euclidean distance. This process provides detailed temporal relationships between frames, essential for accurate sequence analysis. By preserving temporal correspondence, backtracking ensures efficient and precise alignment, enhancing the performance the model.

For example, in the following case we obtain the following alignment path:

```
a = np.array([1,4,4,5,9,3,1,8,8], dtype=np.float32)[:,np.newaxis]
b = np.array([1,2,3,8,8,3,1,8], dtype=np.float32)[:,np.newaxis]
dtw(a, b, metric='sqeuclidean')
```

```
(9.0,
 [(0, 0),
  (0, 1),
  (1, 2),
  (2, 2),
  (3, 2),
  (4, 3),
  (4, 4),
  (5, 5),
  (6, 6),
  (7, 7),
  (8, 7)])
```

Which means that

- Frame at index 0 of sequence A aligns with the frame at index 0 of sequence B.
- Frame at index 1 of sequence A aligns with the frame at index 1 of sequence B.
- Frame at index 1 of sequence A aligns with the frame at index 2 of sequence B.
- And so on.

We see that the algorithm identifies the optimal way to align the frames, even when the sequences have different lengths or varying temporal dynamics. In conclusion, implementing backtracking for sequence alignment with Euclidean distance allows us to achieve a highly accurate and detailed alignment path between two sequences.