Assignment 4. Speech Recognition using Dynamic Time Warping

Alex Carrillo and Robert Tura
Spoken and Written Language Processing

April 29, 2020
Universitat Politècnica de Catalunya

1 Introduction

In this homework assignment we will work on a simple speech recognition task: digit recognition using *Dynamic Time Warping*. In essence, this algorithm is used to measure the similarity between two sequences which may vary in time or speed, *i.e.* digit audio recordings in our case. The objective is to get a distance metric between two input time series, converting the data into vectors and calculating some distance in this vector space.

Regarding the vector representation of these audio signals, we focus on *Mel-Frequency Cepstral Coefficients* (MFCCs) which are the most widely used features in speech recognition. We briefly describe how are they usually computed:

1. Separate the audio signal into small sections (windowed frames)
2. Apply the DFT to each section and obtain the spectral power of the signal
3. Map the above spectrum onto a mel-scale filter bank, using triangular overlapping windows and adding the energies in each of them
4. Take the logarithm of all energies of each mel frequency
5. Apply the DCT to these logarithms to obtain the MFCC coefficients (those are the amplitudes of the resulting spectrum)

2 Considerations

Before going into the details of our task, we want to note some aspects of the implementation which will further clarify the steps we have taken.

(i) The sliding windows used to split the signal into frames have been set to a duration of **25 ms** windowed frames taken **every 10 ms**. Moreover, in order to reduce the spectral distortion (shown as a lot of noise at high-frequency when suddenly cut the signal) at each frame, a Hamming window has been used.

(ii) As mentioned above, for humans, the perceived loudness changes according to frequency (in fact, the perceived frequency resolution decreases as frequency increases, *i.e.* we are less sensitive to higher frequencies), hence, the **mel scale** is used for mapping. We apply triangular band-pass filters to the frequency information to mimic what a human perceives.

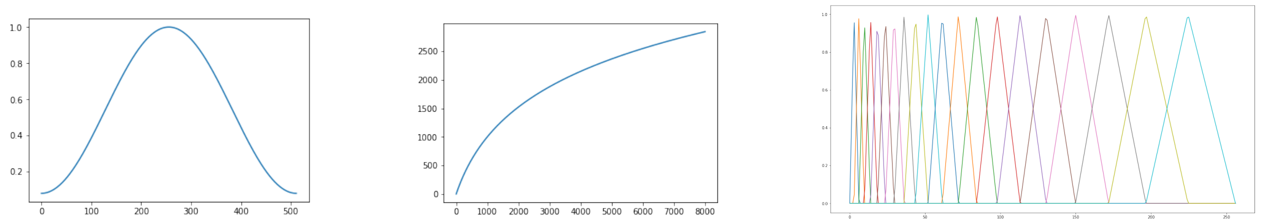

Figure 1: From left to right: Hamming window, mel scale frequency mapping and mel-scale filter bank.

3 The WER function

The *word error rate* (WER) is a common metric of the performance of a speech recognition or machine translation system. In the first task, we are asked to modify the WER function to evaluate the performance of the DWT algorithm for text-dependent speaker identification. To do so, we add a new output variable `swer` to it which simply computes the probability of detecting the correct speaker by counting the number of times we detect it and dividing by the number of samples. The resulting code chunk is as follows:

```
# Word Error Rate (Accuracy)
def wer(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
    s = 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
        if test['speaker'] == minref['speaker']:
            s = s + 1

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

In our case, the probability of detecting the correct speaker of the *free-spoken-digit-dataset* using the DTW distance to the rest of the recordings is $95,6\%$ and the word error rate gets a $6,9\%$ for the baseline configuration, *i.e.* maintaining the number of cepstrum coefficients we keep at `n_mfcc = 12`.

Now we compare the WER with respect to the number of cepstral coefficients. As a recap, keep in mind that the use of, say 12 MFCCs, is more than enough in most applications and in general we do not want too many coefficients because

(i) it is all about reducing the dimensionality of our feature space

(ii) DCT keeps most of the information in first few coefficients (is an orthogonal transformation and produces uncorrelated features)

	Free-Spoken-Digit-Dataset		***Google Speech Commands Dataset***
n_mfcc	WER	Speaker detection	WER
4	16,9%	81,9%	30,2%
5	16,3%	83,1%	27,5%
6	12,5%	86,9%	27,6%
7	8,8%	90,0%	27,8%
8	8,1%	90,0%	26,3%
9	7,5%	91,1%	24,3%
10	6,2%	93,1%	26,0%
11	5,6%	93,1%	27,5%
12	6,9%	95,6%	28,9%
13	7,5%	94,4%	29,4%
14	6,9%	95,6%	30,7%
15	6,2%	95,6%	31,4%
16	6,2%	95,0%	34,0%

Table 1: Comparison of WER and speaker detection for both datasets.

We tested a handful of values of cepstral coefficients, from 4 to 16, and there are two types of results: the first one is tested with the FSDD and has the same reference dataset, and the second result comes from testing with the 10x100 and has the reference dataset 10x10.

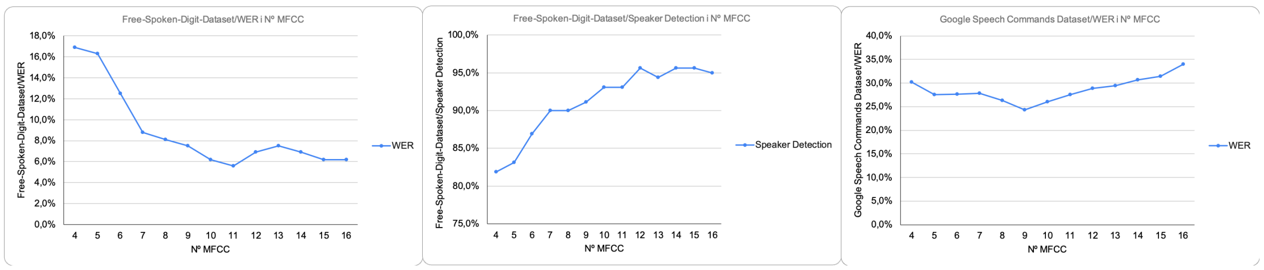

Figure 2: From left to right: Hamming window, mel scale frequency mapping and mel-scale filter bank.

As we can see, the results are not quite the same but they are close. The FSDD word error rate is closer to 0 with respect to the Google dataset, but that is because the difference of sizes between the datasets (number of samples) us huge. So, therefore, the conclusions that we could draw from the FSDD are important but not quite as much as the ones that can be drawn from the Google one. Be that as it may, and based in our results, we are deciding to keep using **9 ceptrum coefficients** for the rest of the lab.

4 Cepstral normalization and liftering

Moving on, we analyze the influence in the recognition accuracy of each of the cepstral normalization steps (mean and variance) with and without liftering. It should be noted that for each test the default liftering parameter (sinus window size) of 22 has been used. In order to study how influential those options are, we will make a test for each individual option and all the possible combinations, and then decide which of the layouts does better for the model.

Lifter	Mean	Variance	*FSDD* WER	*Google Dataset* WER
yes	yes	yes	7,5%	38,2%
yes	yes	-	5,6%	28,2%
yes	-	yes	6,2%	54,9%
yes	-	-	2,5%	41,4%
-	yes	yes	$\rightarrow$ 7,5%	$\rightarrow$ 24,3%
-	yes	-	9,4%	39,8%
-	-	yes	14,4%	57,6%
-	-	-	6,9%	54,6%

Table 2: Comparison of WER among different cepstral normalization and liftering configurations for both datasets.

After testing those configurations, analyzing the code and thinking about the algorithm overall, we intuited what can be achieved by these approaches. On the one hand, as we know, mean and variance normalization allows to adjust values in order to counteract the variants in each of the recordings we have (for voice recognition we are not interested in taking into account the effect of the channel, microphone or speaker). However, if the audio sample is short, this may not be reliable and we should instead compute the mean and variance values based on speakers, or even over the entire training dataset. Moreover, we also intuit that the normalization step may cancel or diminish the pre-emphasis done earlier in the processing.

On the other hand, due to the sensitivity of the first samples to the spectral slope and that the noise can affect the last samples, they are not used in the distance calculation. So, liftering means a windowing is applied because the cepstrum values in this central zone are more important and they are given more weight in the euclidean distance when comparing.

The best layout with respect to this parameter is the basic one: the baseline that is by default in the code. The experiment was nice because it was not much trouble and benefits that could draw may be notable.

5 First-order derivatives

As we saw in theory class, MFCC feature parameters can be extended to improve the model overall. For instance, there are usually 12 cepstrum coefficients (or 9 in our tests), plus the energy term which helps to identify phonemes. Then, we can have one more set of parameters corresponding to the first-order derivatives of the features (or even include the second-order derivatives). The main idea behind it is that characterizing feature changes over time provides the dynamic context information for pronunciation or for a specific phoneme. So, the derivative measures the changes in features from the previous frame of the audio signal to the next one, which is very enriching for the model.

In our particular case, we opted for two different approaches. The first one was using the `feature.delta` function of the `librosa` library [4] that was commented in the source code at the computation of the MFCC. This approach improved the accuracy of the model slightly, as the WER changed from 24,3% to 24% with almost no time penalty. The used code chunk was as follows:

```python
# Compute MFCC
import librosa
def get_mfcc(dataset, **kwargs):
    mfccs = []
    for sample in dataset:
        sfr, y = scipy.io.wavfile.read(sample['wav'])
        y = y/32768
        S = mfsc(y, sfr, **kwargs)
        M = mfsc2mfcc(S, n_mfcc=9).T
        DM = librosa.feature.delta(M)
        M = np.hstack((M, DM))
        mfccs.append(M.astype(np.float32))
    return mfccs
```

Next, we tried to make our own derivative function. In fact, we implemented the discrete approximations of the derivative based on the instructions of the class slides:

$$\Delta c[n] = \frac{c[n+1] - c[n-1]}{2} \tag{1}$$

With the above in mind, note we added a specific padding that allows for the first and last to have derivatives. The implementation of it was straightforward and our code chunks were as follows:

```python
def derivative(M):
    r, c = M.shape
    Q = np.empty((r,c))
    for i in range(r):
        Q[i,0] =  M[i,1] - M[i,0]
        Q[i,c-1] = M[i,c-1] - M[i,c-2]
        for j in range(1,c-1):
            Q[i,j] = (M[i,j+1] - M[i,j-1])/2
    return Q

# Compute MFCC
import librosa
def get_mfcc(dataset, **kwargs):
    mfccs = []
    for sample in dataset:
        sfr, y = scipy.io.wavfile.read(sample['wav'])
        y = y/32768
        S = mfsc(y, sfr, **kwargs)
        M = mfsc2mfcc(S, n_mfcc=9)
        DM = (derivative(M))
        M = np.hstack((M, DM)).T
        mfccs.append(M.astype(np.float32))
    return mfccs
```

However, after all, this change did not improved to our model accuracy, but taking down the WER from 24% to 23,7% actually. Furthermore, in opposition to us, we missed the submission of this model and we could not commit it to competition. In our view, this improvement could have taken us to a better place in the ranking.

6 DTW alignment

The basis of DTW algorithm is found on the computations of distance matrix between two time series. The idea is to compare arrays with different lengths, *i.e.* the audio recordings, and build one-to-many and many-to-one matches between each sample so that the total distance can be minimized between them. Now, the optimal match between the two given sequences is subject to certain restriction and rules (stated in the definition) and has the minimal cost. Informally, these restrictions are summarized to: *head and tail must be positionally matched*, *no cross-match* and *no left out*.

This time, we are asked to complete the DTW algorithm to compute the alignment (backtracking) of the optimal path, in order to graphically represent some results.

Starting on the base of the given `dtw` function, we modify it to return the full accumulated cost matrix D and not only the computed distance between the sequences. Note we can easily acces to the distance looking at the last cell of the matrix, *i.e.* `D[-1,-1]`. So, the function is as follows:

```python
from numba import jit
@jit
def dtw(x, y, dist='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

  D[1:, 1:] = scipy.spatial.distance.cdist(x, y, dist)

  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

  # we get rid of the INF padding
  return D[1:,1:]
```

From this point on, it is not too difficult to compute the *optimal warping path* using backtracking, given the accumulated cost matrix D. The following function simply traverses the matrix and keeps the trace of the cells which conform the path:

```python
def compute_optimal_warping_path(D):
    """
    Computes the warping path given an accumulated cost matrix.
    Args
        D: Accumulated cost matrix
    Returns
        P: Warping path (list of index pairs)
    """
    N = D.shape[0]
    M = D.shape[1]
    n = N - 1
    m = M - 1
    P = [(n, m)]
    while n > 0 or m > 0:
        if n == 0:
            cell = (0, m - 1)
        elif m == 0:
            cell = (n - 1, 0)
        else:
            val = min(D[n-1, m-1], D[n-1, m], D[n, m-1])
            if val == D[n-1, m-1]:
                cell = (n-1, m-1)
            elif val == D[n-1, m]:
                cell = (n-1, m)
            else:
                cell = (n, m-1)
        P.append(cell)
        (n, m) = cell
    P.reverse()
    return np.array(P)
```

As the function above returns an array of the pair indices of the optimal path, we can spread those points over the accumulated cost matrix D and color the matrix based on the distance between each sample of the sequences. We created the following function to plot this, in order to test several examples:

```
def plot_optimal_warping_path(D, P):
    P = np.array(P)
    plt.imshow(D, cmap='gray_r', origin='lower', aspect='auto')
    plt.plot(P[:, 1], P[:, 0], marker='o', color='r')
    plt.clim([0, np.max(D)])
    plt.colorbar()
    plt.title('Accumulated cost matrix $D$ with optimal warping path')
    plt.xlabel('Sequence Y')
    plt.ylabel('Sequence X')
    plt.tight_layout()
```

Finally, and after several test errors, we evaluate the functions above and see what happens on some recordings. On the one hand, we plot one example of the alignment with the closest reference when the test signal is different from the reference signal (different digit). Specifically, it refers to the sample {`'text': '0', 'speaker': 'jackson'`} with the sample {`'text': '1', 'speaker': 'theo'`}. On the other hand, we plot another example when the test signal is equal from the reference signal (same digit). In particular, it refers to the comparison between sample {`'text': '0', 'speaker': 'jackson'`} and sample {`'text': '0', 'nicolas': 'theo'`}.

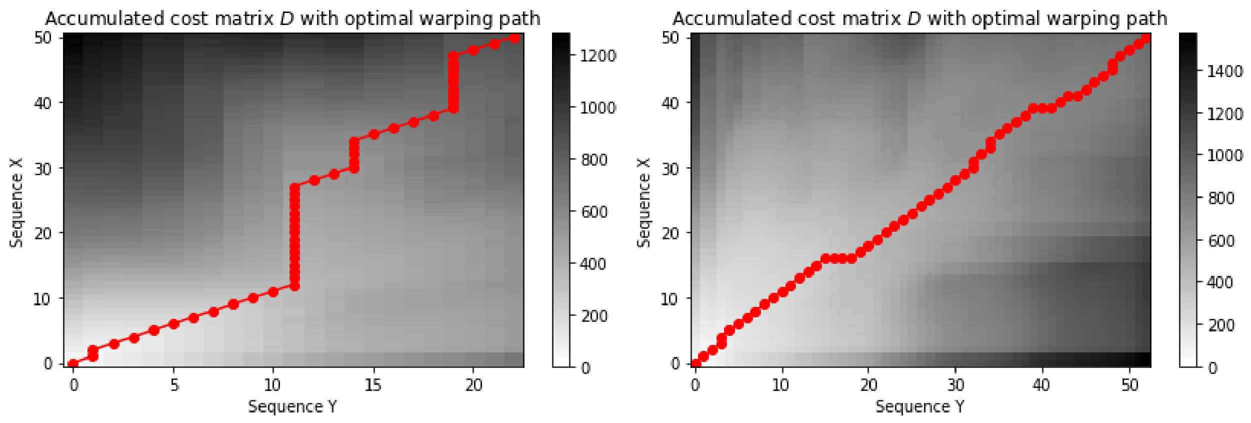

Figure 3: Warping paths for not-aligned sequences and aligned ones, respectively.

As it can be seen clearly, is obvious which plot refers to sequences of the same digit and which not, with a naked eye. We would simply like to comment that it is amazing how the plot on the right (same digit) is practically aligned with a straight line, even when the audio recording are from different speakers. As a conclusion, we think it widely represents the power and accuracy of this relative simple algorithm.

7 Bonus track: variants of DTW

As a further study, we tried different things in order to improve or somehow change the DTW algorithm. First of all, we tried changing the type of the dynamic time warping to the first type and adding weights but that did not make and improvement on accuracy, and time was roughly 10% slower.

Also, we tried to change the distance from `sqeuclidean` to `cosinus`, which had a worse accuracy than the previous approach, but it was 25% faster. Moreover, we tried another distance more, the `hamming` distance and it had an awful accuracy and no improvement time-wise. Maybe, for future research related to these nuances, other approaches must be considered and deeply studied.

References

[1] Shyalika Chathurangi. *Dynamic Time Warping (DTW)*. 2019. URL: https://medium.com/datadriveninvestor/dynamic-time-warping-dtw-d51d1a1e4afc.

[2] Antoine Choné. *Computing MFCCs voice recognition features on ARM systems*. 2018. URL: https://medium.com/linagoralabs/computing-mfccs-voice-recognition-features-on-arm-systems-dae45f016eb6.

[3] Jonathan Hui. *Speech Recognition — Feature Extraction MFCC PLP*. 2019. URL: https://medium.com/@jonathan_hui/speech-recognition-feature-extraction-mfcc-plp-5455f5a69dd9.

[4] *librosa.feature.delta*. 2019. URL: https://librosa.github.io/librosa/generated/librosa.feature.delta.html.

[5] Kyaagba Shachia. *Dynamic Time Warping with Time Series*. 2018. URL: https://medium.com/@shachiakyaagba_41915/dynamic-time-warping-with-time-series-1f5c05fb8950.

[6] *Word error rate*. 2020. URL: https://en.wikipedia.org/wiki/Word_error_rate.

[7] Jeremy Zhang. *Dynamic Time Warping*. 2020. URL: https://towardsdatascience.com/dynamic-time-warping-3933f25fcdd.