POE Práctica 3

Sonia Castro Paniello
Eduard Ramon Aliaga

April 2023

1 Assignment tasks:

Modify the wer function to evaluate the speaker error rate (SER), i.e., the performance of the DTW algorithm for text-dependent speaker identification.

```
def ser(test_dataset,n_mfcc=12,cms=True, cmvn=True, lifter=0, ref_dataset=None,
    same_spk=False):
    # Compute mfcc
    test_mfcc = get_mfcc(test_dataset, n_mfcc, cms, cmvn, lifter)
    if ref_dataset is None:
        ref_dataset = test_dataset
        ref_mfcc = test_mfcc
    else:
        ref_mfcc = get_mfcc(ref_dataset,n_mfcc, cms, cmvn, lifter)

    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])[0]
                if distance < mincost:
                    mincost = distance
                    minref = ref
        if test['speaker'] != minref['speaker']:
            err += 1

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

Compare the WER with respect to the number of cepstral coefficients Change the default number of cepstral coefficients computed for each speech frame, and check how this number affects the WER of the speech recognizer. Use the obtained best value for the rest of the tasks.

	1	2	3	4	5	6	7	8	9	10
Same speaker	59.4	37.5	25.0	16.9	16.2	12.5	8.8	8.1	7.5	6.2
Only other speakers	76.7	50.0	41.7	30.2	27.5	27.6	27.8	26.3	24.3	26.0

Table 1: Wer value for different number of mfcc.

	11	12	13	14	15	16	17	18	19	20
Same speaker	5.6	6.9	7.5	6.9	6.2	6.2	6.2	6.9	7.5	6.9
Only other speakers	27.5	28.9	29.4	30.7	31.4	34.0	34.4	35.5	37.7	40.9

Table 2: Wer value for different number of mfcc.

As we can see, the WER is greater when using audio samples from different speakers compared to using samples from the same speaker, as expected. This is because each person has certain characteristic voice features. On the other hand, we have decided to perform the rest of the practice using 9 MFCC coefficients, as it gave us the minimum error in the case of different speakers and a value close to the minimum in the case of the same speaker.

Analyze the influence on the recognition accuracy of each of the cepstral normalization steps (mean and variance) with and without littering. Use the default littering parameter (sinus window size) of 22.

	Mean	Variance	Liftering(22)	Private	Public
Trial 1	true	true	0	0.83428	0.88333
Trial 2	true	false	0	0.78142	0.79666
Trial 3	false	false	0	0.67571	0.74333
Trial 4	true	true	22	0.73428	0.74666
Trial 5	true	false	22	0.84714	0.87
Trial 6	false	false	22	0.73571	0.79

Table 3: Wer value for different number of mfcc.

As we can observe from the results table, without using liftering, the best result is obtained using Cepstral mean and variance normalization (CMVN), which aims to reduce the variability in cepstral features within and across different speakers. However, when we use liftering, the best result is obtained when we only use Cepstral mean subtraction (CMS).

For the rest of the exercises, we have used the parameters with the best overall result, which are liftering (22) and Cepstral mean subtraction (CMS).

Extend the MFCC parameters with the first-order derivatives, and check that the WER is reduced.

```
def get_mfcc(dataset,n_mfcc=12, cms=True, cmvn=True,lifter=0, **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, cms, cmvn, lifter)
        # Move the temporal dimension to the first index
        M = M.T
        DM = librosa.feature.delta(M)
        DDM = librosa.feature.delta(M, order=2)
        M = np.hstack((M, DM))
        M = np.hstack((M, DDM))
        mfccs.append(M.astype(np.float32))
    return mfccs
```

We have tried to implement both using the first derivative and the second derivative to assess the improvements of the Word Error Rate (WER). "DM" refers to the first derivative, and "DDM" refers to the second derivative in the code.

Here is a table with WER values for different numbers of MFCCs using the first derivative:

	9	10	11	12	13	14
Same speaker	7.5	6.2	5.6	6.2	7.5	6.2
Only other speakers	16.6	16.3	18.3	18.1	18.8	21.0

Table 4: Wer value for different number of mfcc.

If we compare with the first two tables of WER values without using the first derivative, we can see that the values have decreased, especially in the case of different speakers.

	Liftering	Public	Private
DM	22	0.84285	0.86333
DDM	22	0.84571	0.87
DM + DDM	22	0.84	0.86333
DM	0	0.83571	0.87666
DDM	0	0.83571	0.88

Table 5: Accuracy using the first and second derivatives.

As observed, a higher public accuracy is achieved when using only the second derivative and a liftering of 22.

Complete the DTW algorithm to compute the alignment (backtracking).

```
def traceback(D):
    n = D.shape[0] - 1
    m =  D.shape[1] - 1
    P = [(n, m)]
    while n > 1 or m > 1:
        if n == 1:
            cell = (1, m - 1)
```

```
        elif m == 1:
            cell = (n - 1, 1)
        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)
```

We found an existing implementation of this algorithm and adapted it to our case, where the accumulated cost matrix has the first column and row filled with minus infinity values, except for the position [0,0].

The first step is to initialize the variables n and m with the size of the matrix. We also create an empty list called P, which will be used to store the traceback path. Moving on to the traceback process, we start by appending the current position (n, m) to the list P. We then enter a loop that continues until we reach the position (1, 1) in the matrix.

To determine the next cell in the traceback, we first check if either n or m is equal to 1. If either condition is true, it means we have reached the edge of the matrix. In this case, we set the next cell as (1, m-1) if n is equal to 1, indicating a horizontal movement. If m is equal to 1, we set the next cell as (n-1, 1) to indicate a vertical movement.

If neither n nor m is equal to 1, we compare the values of the three adjacent cells: (n-1, m-1), (n-1, m), and (n, m-1). Among these three cells, we find the minimum value. The cell with the minimum value becomes the next cell. . We append the determined next cell to the list P and update the (n, m) coordinates with the coordinates of the next cell. To obtain the correct order of positions, we reverse the order of elements in the list P. This ensures that the traceback path is in the correct sequence.

Optional: perform a comparative study of different variants of the DTW algorithm in terms of speed and accuracy.

First, we tried using a penalty to try to favor the diagonal. With values of 0, 0.5, and 1, the accuracy did not vary. So we decided to increase this value further. The following table summarizes the experiments carried out:

	Penalty	Public	Private
Trial1	0	0.84285	0.86333
Trial2	0.25	0.84285	0.86333
Trial3	0.5	0.84285	0.86333
Trial4	1	0.84285	0.86333
Trial5	3	0.84428	0.86333
Trial6	5	0.8714	0.86333
Trial7	10	0.84714	0.86666
Trial8	20	0.84714	0.86666
Trial9	30	0.84428	0.87

Table 6: Accuracy of dtw type2 with different penalties.

For every execution, the time was pretty much s=the same so it seems like the penalty does not have an important impact in the efficiency of the algortihm. All times were around 660s and 700s

```
def dtw_type2(x, y, metric='sqeuclidean'):
  r, c = len(x), len(y)
  penalty = 20

  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)
  #Initial=np.copy(D)


  for i in range(r):
    for j in range(c):
      min_prev = min(D[i, j], D[i+1, j] + penalty , D[i, j+1] + penalty)
      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
```

We implemented dtw type 1 but we only achieved an public and private accuracy of: 0.82285 0.85666 and an execution time of 736s. We can see that this type takes a little bit longer to finish and obtains worse results so in this case type 2 seems to work better than type 1.

```
def dtw_type1(x, y, metric='sqeuclidean'):
  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):
     if i == 0 and j == 0:
          min_prev = D[i, j]
     elif i == 0:
          min_prev = min(D[i, j], D[i, j-1])
     elif j == 0:
          min_prev = min(D[i, j], D[i-1, j])
     else:
        min_prev = min(D[i, j], D[i-1, j], D[i, j-1])

     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
```

After that, we wanted to check some more variants that overall came up worse for our case but still we found interesting to comment them:

1. weighted DTW: This is a penalty based DTW. As the names says, basically it adds different weights to the different points so they are treated in a different manner.

```
def dtw_weighted(x, y, metric='sqeuclidean'):
    """
    Computes Dynamic Time Warping (DTW) of two sequences with weights.
    :param array x: N1*M array
    :param array y: N2*M array
    :param func dist: distance used as cost measure
    :param array weights: N1*N2 array of weights
    """
    r, c = len(x), len(y)

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

    #initialize weight matrix
    weights = np.zeros((r, c))
    for i in range(r):
        for j in range(c):
            weights[i, j] = 1 / (abs(i - j) + 1) # Optimal weight
                initialization
```

```
    # Initialize the matrix with dist(x[i], y[j]) multiplied by weights[i,
        j]
    D[1:, 1:] = scipy.spatial.distance.cdist(x, y, metric) * weights
    # Initialize the matrix with dist(x[i], y[j]) multiplied by weights[i,
        j]
    if weights is None:
        D[1:, 1:] = scipy.spatial.distance.cdist(x, y, metric)
    else:
        D[1:, 1:] = scipy.spatial.distance.cdist(x, y, metric) * weights

    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] += min_prev

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

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

This is an example of the implementation. In this code we put the weight matrix initialization that gave the best results (despite of the one of all ones as it would make no difference). The accuracy obtained is not to be thrilled about as it was of 0.41285 and the execution time was 952 seconds, which is quite horrible. However, a proper initialization of the weight matrix may lead to better precision so we decided to mention it.

2. Sakoe Chiba.

```
def dtw_Sakoe_Chiba(x, y, metric='sqeuclidean'):
    """
    Computes Dynamic Time Warping (DTW) of two sequences with Sakoe-Chiba
        band constraint.
    :param array x: N1*M array
    :param array y: N2*M array
    :param str metric: distance metric used as cost measure
    :param int window: Sakoe-Chiba band width (optional)
    :return: DTW distance and optimal path
    """
    r, c = len(x), len(y)
    window = max(abs(r-c), max(r,c)//2)

    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):
            if abs(i - j) <= window:
                min_prev = min(D[i, j], D[i+1, j], D[i, j+1])
                D[i+1, j+1] += min_prev
            else:
                D[i+1, j+1] = np.inf

    print(D)
    if len(x) == 1:
        path = np.zeros(len(y)), range(len(y))
    elif len(y) == 1:
        path = range(len(x)), np.zeros(len(x))
    else:
        path = traceback(D)

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

In this implementation, we calculate the distance matrix, but we only compute the accumulated weights for the region within a window distance from the diagonal. The accuracy resulting is of 0.84751 private and 0.87 public. The execution time was of 650.0s.

3. Itakura parallelogram.

```
def _get_itakura_slopes(n_timestamps_1, n_timestamps_2, max_slope):
    min_slope = 1 / max_slope

    scale_max = (n_timestamps_2 - 1) / (n_timestamps_1 - 2)
    max_slope *= scale_max
    max_slope = max(1., max_slope)

    scale_min = (n_timestamps_2 - 2) / (n_timestamps_1 - 1)
    min_slope *= scale_min
    min_slope = min(1., min_slope)


    return min_slope, max_slope


def itakura_parallelogram(n_timestamps_1, n_timestamps_2=None,
    max_slope=2.):
    min_slope_, max_slope_ = _get_itakura_slopes(n_timestamps_1,
        n_timestamps_2, max_slope)

    # Now we create the piecewise linear functions defining the
        parallelogram
    # lower_bound[0] = min_slope * x
    # lower_bound[1] = max_slope * (x - n_timestamps_1) + n_timestamps_2

    centered_scale = np.arange(n_timestamps_1) - n_timestamps_1 + 1
    lower_bound = np.empty((2, n_timestamps_1))
    lower_bound[0] = min_slope_ * np.arange(n_timestamps_1)
    lower_bound[1] = max_slope_ * centered_scale + n_timestamps_2 - 1
```

```
    # take the max of the lower linear funcs
    lower_bound = np.round(lower_bound, 2)
    lower_bound = np.ceil(np.max(lower_bound, axis=0))

    # upper_bound[0] = max_slope * x
    # upper_bound[1] = min_slope * (x - n_timestamps_1) + n_timestamps_2

    upper_bound = np.empty((2, n_timestamps_1))
    upper_bound[0] = max_slope_ * np.arange(n_timestamps_1) + 1
    upper_bound[1] = min_slope_ * centered_scale + n_timestamps_2

    # take the min of the upper linear funcs
    upper_bound = np.round(upper_bound, 2)
    upper_bound = np.floor(np.min(upper_bound, axis=0))

    # Little fix for max_slope = 1
    if max_slope == 1:
        if n_timestamps_2 > n_timestamps_1:
            upper_bound[:-1] = lower_bound[1:]
        else:
            upper_bound = lower_bound + 1

    region = np.asarray([lower_bound, upper_bound]).astype('int64')
    return region

def dtw_itakura(x, y, metric='sqeuclidean'):
    r, c = len(x), len(y)
    window = max(abs(r-c), max(r,c)//2)

    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)

    Aux = np.full((r + 1, c + 1), np.inf)

    region = itakura_parallelogram(r+1,c+1,3.)
    for i in range(r+1):
        for j in prange(region[0, i], region[1, i]):
            Aux[i][j] = D[i][j]

    for i in range(r+1):
        for j in prange(region[0, i], region[1, i]):
          min_prev = min(Aux[i, j], Aux[i+1, j], Aux[i, j+1])
          # D[i+1, j+1] = dist(x[i], y[j]) + min_prev
          Aux[i+1][j+1] += min_prev

    if len(x) == 1:
        path = np.zeros(len(y)), range(len(y))
    elif len(y) == 1:
        path = range(len(x)), np.zeros(len(x))
    else:
        path = traceback(Aux)

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

Lastly, we attempted to implement the Itakura parallelogram by using the functions "*_get_itakura_slopes*" and "*itakura_parallelogram*" taken from https://github.com/johannfaouzi/pyts we were having trouble defining the region correctly. Finally, we created the "*dtw_itakura*" function to only calculate the accumulated cost within the positions of the interior of the region. However, we believe that there might be an error in the code since it gives us an accuracy of 0.32572 and 0.36333 with a runtime of 887.2s.