Assignment 4

Speech Recognition using Dynamic Time Warping

Course: Spoken and Written Language Processing

(POE-GCED)

Jordi Aguilar

Miquel Escobar

April 29, 2020

Contents

1 Introduction

The goal of this homework assignment is to extract the spoken words from a set of short audios corresponding to different speakers (also known as Speech Recognition). In order to do so we have implemented and tested several modifications of the baseline model which are explained in detail in section **2. Tasks** and compared in section 3. Summary Table.

To test the variations of the model, we have modified the given baseline **Kaggle** notebook with the corresponding contributions, and we have obtained the accuracy of the word extractions.

The methodology consists on converting the audio file into an array of cepstrum coefficients, then computing the distance between this array and all the arrays corresponding to audios of a labeled dataset, and classify the input audio as the label of the most similar audio file from the labeled dataset.

2 Tasks

2.1 Modified `wer` function

The goal of this task is to modify the **`wer`** function provided by the baseline notebook, in a manner that it returns the *Word Error Rate* with respect to the **predicted speaker, not the spoken text**. In order to do so, the code has been modified as shown in Source Code 1.

```
def wer(test_dataset, ref_dataset=None, same_spk=False, field='text'):
    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[field] != minref[field]:
            err += 1
    wer = 100*err/len(test_dataset)
    return wer
```

Source Code 1: Modified `wer` function

The changes (highlighted in cyan) consist of the addition of a new parameter `field`, which is a string referencing the field we want to measure the accuracy by. For instance, when we set `field='text'`, we are comparing the predicted text with respect to the real text, and when we set `field='speaker'` we are comparing the predicted speaker with respect to the

real one. The latter was the purpose of the task and the modification.

2.2 Optimal number of cepstral coefficients

The number of cepstral coefficients is an important feature since it has an important role on the similarity between two cepstral vectors. Few coefficients may not differentiate appropriately two vectors, but adding a lot of coefficients may lead to higher differences between two samples of the same kind.

These have been the results:

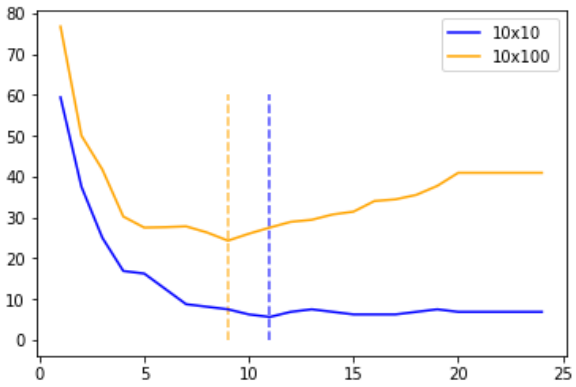

Figure 1: Error rates for the two datasets.

From now on, we will stick with 11 cepstral coefficients.

2.3 Influence of cepstral normalization steps

The next modification to our code is the steps of normalization (subtract the mean and divide by the variance). We will also take into account liftering. We have tried the following combinations of normalizations:

- Mean and variance.
- Mean and liftering.
- Liftering.

2.4 Extending `MFCC` paramaters with first-order derivatives

Another variant of this model is to extend the MFCC parameters with the first-order derivatives. To calculate the derivative of the MFCC at the point i, j we have simply subtracted

the ith coefficient of the vector $j+1$ minus the ith coefficient of the vecctor j.

The code is the following:

```
#Derivative of M
M_d = M[:, 1:] - M[:, :-1]
```

Source Code 2: Derivative in the temporal domain

Once we have calculated the derivative we have used it in two differed ways:

- Concatenate the MFCC matrix with the derivative: This option constructs a larger sample where we also have to align the derivative.

```
def mfsc2mfcc(S, n_mfcc=11, dct_type=2, norm='ortho', lifter=22, cms=True, cmvn=True):
    # Discrete Cosine Transform
    M = scipy.fftpack.dct(S, axis=0, type=dct_type, norm=norm)[:n_mfcc]

    M_d = M[:, 1:] - M[:, :-1]
    #Concatenate M with M_d
    M = np.concatenate((M, M_d), axis = 1)

    # Ceptral mean subtraction (CMS)
    if cms or cmvn:
        M -= M.mean(axis=1, keepdims=True)

    # Ceptral mean and variance normalization (CMVN)
    if cmvn:
        M /= M.std(axis=1, keepdims=True)

    # Liftering
    elif lifter > 0:
        print("liftering...")
        lifter_window = 1 + (lifter / 2) * np.sin(np.pi * np.arange(1, 1 + n_mfcc,
        ↪  dtype=M.dtype) / lifter)[:, np.newaxis]
        M *= lifter_window

    return M
```

Source Code 3: Concatenate matrices

- Perform separate alignment of the MFCC and its derivative, then add the distances obtained: Although this option may seem we can highlight a couple of things. First of all we are forcing that the alignment of the MFCC ends at the last vector of both samples, as well as the alignment of the derivative starts at the first vector of both samples. Secondly, we compute the distance between samples less times than concatenating both matrices.

```
def mfsc2mfcc(S, n_mfcc=11, dct_type=2, norm='ortho', lifter=22, cms=True, cmvn=True):
    # Discrete Cosine Transform
    M = scipy.fftpack.dct(S, axis=0, type=dct_type, norm=norm)[:n_mfcc]

    M_d = M[:, 1:] - M[:, :-1]

    # Ceptral mean subtraction (CMS)
    if cms or cmvn:
        M -= M.mean(axis=1, keepdims=True)

    # Ceptral mean and variance normalization (CMVN)
    if cmvn:
        M /= M.std(axis=1, keepdims=True)

    # Liftering
    elif lifter > 0:
        print("liftering...")
        lifter_window = 1 + (lifter / 2) * np.sin(np.pi * np.arange(1, 1 + n_mfcc,
        ↪  dtype=M.dtype) / lifter)[:, np.newaxis]
        M *= lifter_window

    return M, M_d

def distance(test_mfcc, ref_mfcc)
    distance = dtw(test_mfcc.M, ref_mfcc.M) +
               dtw(test_mfcc.M_d, ref_mfcc.M_d)

    return distance
```

Source Code 4: Add alignments of MFCC and its derivative matrices

2.5 Backtracking of the DTW algorithm

To obtain the followed path by the algorithm, we have implemented the `_traceback` function (see Source Code 6), as well as modified the `dtw` function (see Source Code 5) in order to keep returning the distance but also the algorithm's followed path. We have also added the option to plot the path, which is controlled by the `plot` parameter.

```
def dtw(x, y, dist='sqeuclidean', plot=False):
  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] += 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)
  if plot:
    plot_path(D[1:,1:], path)
  return D[-1, -1], path
```

Source Code 5: Modified `wer` function with backtracking added.

```
def _traceback(D):
    D = D[1:,1:]
    r, c = D.shape
    x_path, y_path = [], []
    i, j = 0, 0
    print(D)
    while i < r and j < c:
        x_path.append(i)
        y_path.append(j)
        if i == r-1:
```

```
            j += 1
        elif j == c-1:
            i += 1
        else:
            arg_min_next = np.argmin([D[i+1, j+1], D[i+1, j], D[i, j+1]])
            i += 1 * (arg_min_next != 2)
            j += 1 * (arg_min_next != 1)
    return x_path, y_path
```

Source Code 6: `_traceback` function.

We have plotted the algorithm results for a case in which the classification was correct (in Figure 2.

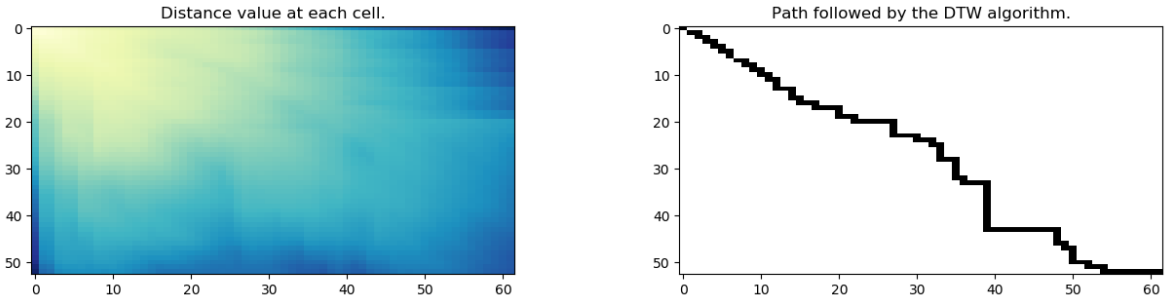

Figure 2: Correct classification DTW plots.

And also for a case in which it was incorrect (see Figure 3). We also made the plots for multiple other cases, and we observed two interesting trends: firstly, as more "diagonal" is the path the results seem to be better and secondly, whenever the audio we are trying to classify has a similar length to the one labeled with the minimum distance, results also seem to be better (and the other way around).

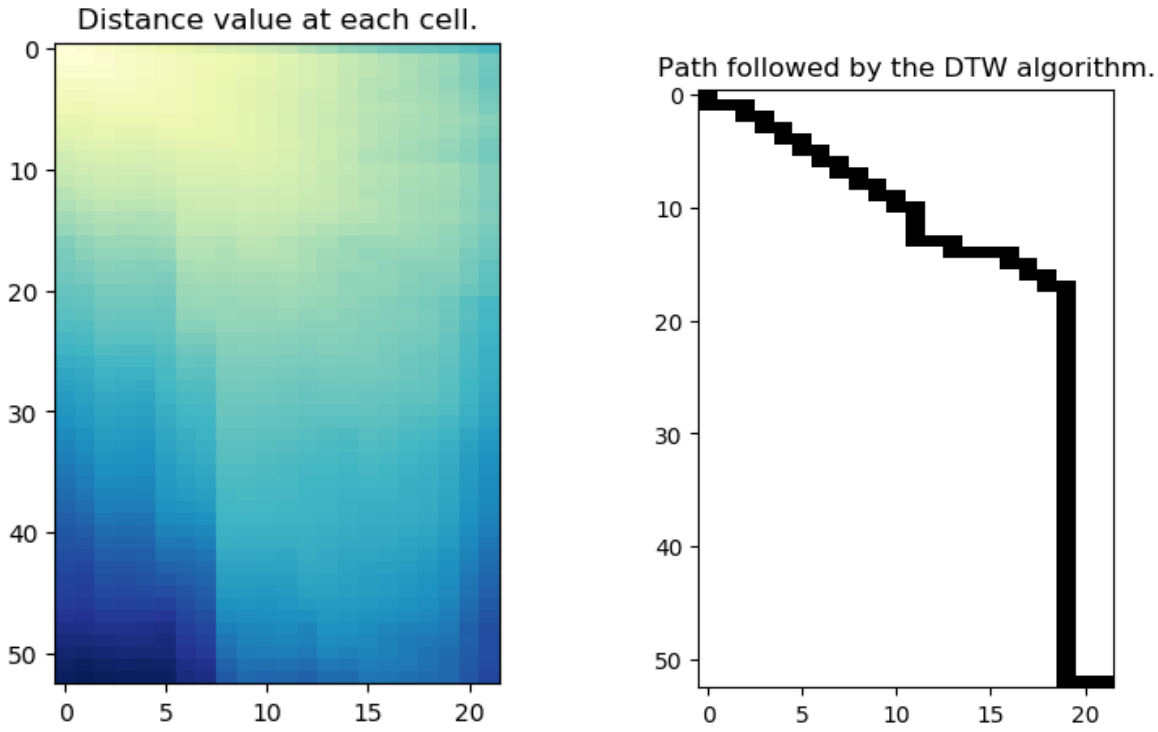

Figure 3: Incorrect classification DTW plots.

2.6 Comparative study of different variants of the DTW algorithm

According to the lecture notes, there exist two main variants of the DTW algorithm.

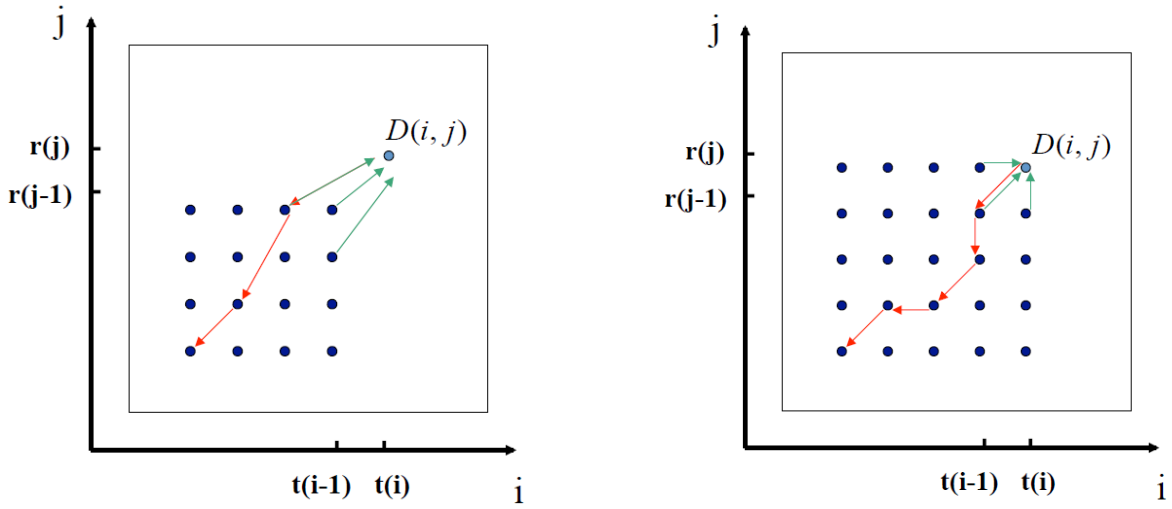

Figure 4: DTW algorithm of *Type I* (left) and *Type II* (right).

The DTW algorithm of *Type II* is the one already implemented into the baseline notebook (at function `dtw`). In order to be able to use the *Type I* algorithm, we implemented the modifications present in Source Code 7 to the function (highlighted in cyan).

```
1  def dtw(x, y, dist='sqeuclidean'):
2    r, c = len(x), len(y)
3    D = np.zeros((r + 1, c + 1))
4    D[0, 1:] = np.inf
5    D[1:, 0] = np.inf
6    D[1:, 1:] = scipy.spatial.distance.cdist(x, y, dist)
7    for i in range(r):
8      for j in range(c):
9        min_prev = min(D[i, j], D[i, j-1], D[i-1, j])
10       D[i+1, j+1] += min_prev
11   return D[-1,-1]
```

Source Code 7: Modified `wer` function

Basically, the value assigned to the current cell is its corresponding distance plus the minimum value out of the three cells corresponding to the green lines in Figure 1.

Following the recursion it is trivial to deduce that the value residing in `D[-1,-1]` after the

updates on all cells is the one corresponding to the accumulated distance in the minimum path.

3 Summary table

After testing the different options of the method, we have summarized the models into the table below. This helps to identify the best performing models taking into account all available information.

Model name	Test Acc.
baseline	0.79
Concat derivative	0.75
Add derivative	0.79
Mean and variance	0.79
Mean and Liftering	0.84
Liftering	0.74
DTW type 1	0.82

Table 1: Private test accuracy of the models

From this table we can highlight that the best model is the one that uses the liftering normalization.

It is also interesting to see that concatenating the derivative doesn't work as well as comparing separately the MFCC and its derivative.

Finally we can remark that `DTW type 1` performs a little better. The reason behind this could be that as it takes longer steps in the alignment, it is more likely that it doesn't "overfit" the comparison between samples.

4 Conclusions

To wrap up this report, we can say that we now understand the basics of speech processing and identification. It has been an assignment really helpful to internalize all the steps required to perform a proper MFCC.

This system combines a lot of methodologies that we have learnt during the degree (DFT, DCT, Mel frequencies) and peaks with the DTW algorithm, a simple yet powerful way to align two samples and determine its global distance.

Although not having a direct relation with this course, we would also want to highlight the use of the library numba, specially the constructor `@jit`, that has enormously reduced the computing time by compiling the code.

In general, it has been a great assignment to get in touch with audio classification, an area that we find of high interest.