Scientific Computing - Comparison of various methods for 3D time series classification

Morteza Ezzabady, Théo Dubus

November 2020

Contents

1 Introduction

1.1 Objective

This report is the result of a practical work for the subject 'Scientific Computing'. The objective of this report is to present different classification algorithms including Dynamic Time Warping, K Nearest Neighbors and derivatives of these algorithms. We will then conclude this report with a comparison of the performance between these classification methods.

1.2 The dataset

The dataset used to test and benchmarks these classification methods is provided by our teacher and has not been modified. The raw data that we are trying to classify comes for the output of a 3-axis gyroscopic sensor, usually embedded in smartphones, car and most wearable objects. The data is sampled at a rate of 50Hz. The dataset is organised as a grouping of samples, each sample is then identified with it's class and a sample number (i.e. an identifier corresponding to a test user). The 6 classes, each one corresponding to a human physical activity, provided in this dataset are the following :

['Downstairs', 'Jogging', 'Sitting', 'Standing', 'Upstairs', 'Walking']

You can find the meaning of each line including examples below:

user-id	activity	timestamp	x-axis accel.	y-axis accel.	z-axis accel.
33	Upstairs	4995080235300	-3.1463003	8.430995	2.2201166
17	Walking	57919062303000	4.440233	16.3989	4.862464

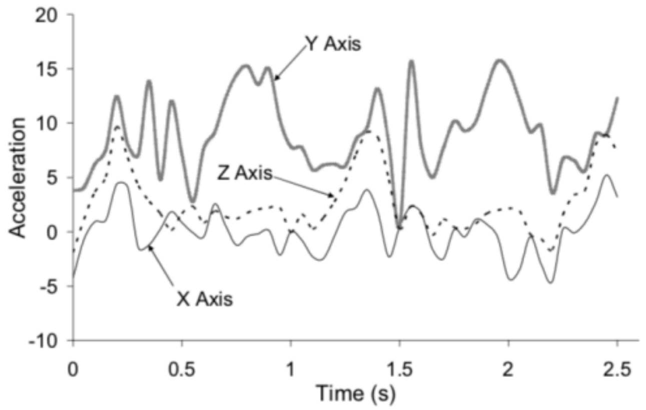

Figure 1: Data sample from the class 'Upstairs'.

2 Dynamic Time Warping

2.1 Principle an applications

Dynamic Time Warping allows us to measure the similarity between 2 time series (1) . This method is useful for applications where time stretching and other temporal perturbations occurs. DTW is a good choice for speech recognition applications as the speaking speeds can vary between subjects and/or accents. DTW is also a good candidate for our application, it can cope with the different walking speeds and sudden acceleration/deceleration of our subjects.

Our implementation of DTW consists of two steps :

1. Computation of the optimal warp path between time series A and B, the output is an array of pair[idxA, idxB], note that some index of a time series can match with multiple index of the other.

2. Computation of the total cost for those 2 times series : the cost is calculated as the sum of absolute differences between the two times series for each matched pair. This methods then outputs a score, a low score indicates a high similarity and vise versa.

2.2 Implementation

Here is an example of DTW algorithm, this version directly outputs the cost but the path can also be added as the output (c.f. commented code in our notebook):

```python
def dtw(v, u, metric='euclidean'):
    v, u = np.atleast_2d(v, u)
    dist = scipy.spatial.distance.cdist(v.T, u.T, metric=metric)
    n, m = dist.shape
    cost = np.zeros((n, m))
    cost[0, 0] = dist[0, 0]
    for i in range(1, n):
        cost[i, 0] = cost[i - 1, 0] + dist[i, 0]
    for j in range(1, m):
        cost[0, j] = cost[0, j - 1] + dist[0, j]
    for i in range(1, n):
        for j in range(1, m):
            cost[i, j] = dist[i, j]
            + min(cost[i - 1, j], cost[i, j - 1], cost[i - 1, j - 1])
    n -= 1
    m -= 1
    return cost[-1, -1]/(n + m)
```

2.3 Our results

We ran the DTW algorithm over the dataset to measure it's accuracy, this step is performed by `predict_with_dtw`.
This method takes two list of randomly selected times series:

- `X_test` : The time series whose Class we want to predict.
- `X_train` : Training time series that will be used to find the the closest (i.e. the lowest DTW cost) matching times series, these time series are labeled.

For each `X_test` we will return the label of the closest match found, this is the prediction. We compare the predicted class with the actual class of our data to calculate the accuracy score of and build a confusion matrix to help us visualize what has been accurately labeled or not. The resulting matrices are available below. Note that we are currently generating our training data over 70% of the entire dataset (15371 time series of size 50 or 25622 of size 30). This way of generating the training data can seem unfair but we estimated that in a real world application we want to utilize the dataset as much as possible. The accuracy can be affected by a low number of training time series or by very short time series that doesn't contain enough data to give accurate results. The key to maximize accuracy is to find the sweet spot between both.
We didn't use our own DTW method but an optimized cython version (2) to get a fair comparison with the other optimized cython methods in this report.
The titles of each matrix are formatted this way :
`algorithm_accuracy_segment-size_test-size_compute-time(sec)`

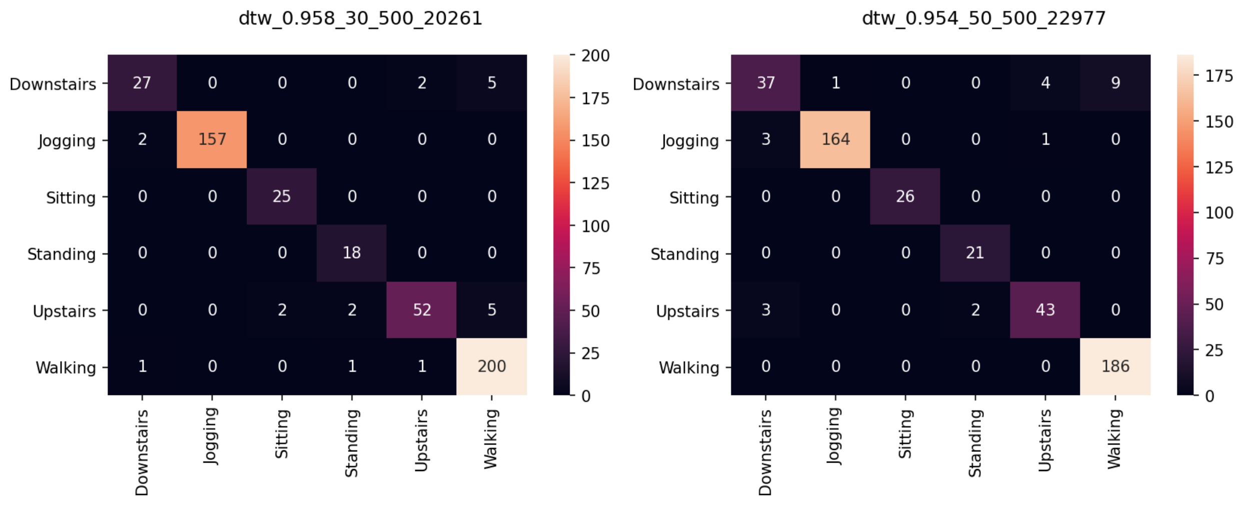

Figure 2: Our DTW results. Higher segment size is slower and does not improves the overall accuracy score.

3 K Nearest Neighbors

3.1 Principle an applications

The K Nearest Neighbors algorithm can be used either for regression analysis or classification, in our case we want to classify time series. (3) The principle of k-NN is relatively simple. It outputs the k nearest times series (or neighbors) to our test series and the label of the test series is then chosen as the most frequent label among it's neighbors.
In this project we first apply dimensionality reduction with PCA to reduce the number of dimensions from `segment_size` to 3. This will ensure that we only compute the k-NN for the attributes that account for the most variance in our data, called principal components. k-NN is also known to be less efficient for high dimensionality problems.
k-NN is applied to this data called `PCA_train` and `PCA_test` in our code. We use the k-NN method implemented in Scikit-Learn `KNeighborsClassifier`.

3.2 Implementation

Here is an example of a k-NN classifier implemented in python (4) :

```python
# Calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
        distance += (row1[i] - row2[i])**2
    return sqrt(distance)

# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
    distances = list()
    for train_row in train:
        dist = euclidean_distance(test_row, train_row)
        distances.append((train_row, dist))
    distances.sort(key=lambda tup: tup[1])
    neighbors = list()
    for i in range(num_neighbors):
        neighbors.append(distances[i][0])
    return neighbors

# Make a prediction with neighbors
def predict_classification(train, test_row, num_neighbors):
    neighbors = get_neighbors(train, test_row, num_neighbors)
    output_values = [row[-1] for row in neighbors]
    prediction = max(set(output_values), key=output_values.count)
    return prediction
```

3.3 Our results

We ran the k-NN algorithm over the dataset to measure it's accuracy, this step is performed by `predict_with_knn`. We use the same parameters as DTW except for the number of selected neighbors in k-NN called `K`. The very fast compute times allows us to increase the training data from 70% of the entire dataset to the maximum possible (size of dataset minus size of test data).
The titles of each matrix are formatted this way :
`algorithm_K_accuracy_segment-size_test-size_compute-time(sec)`

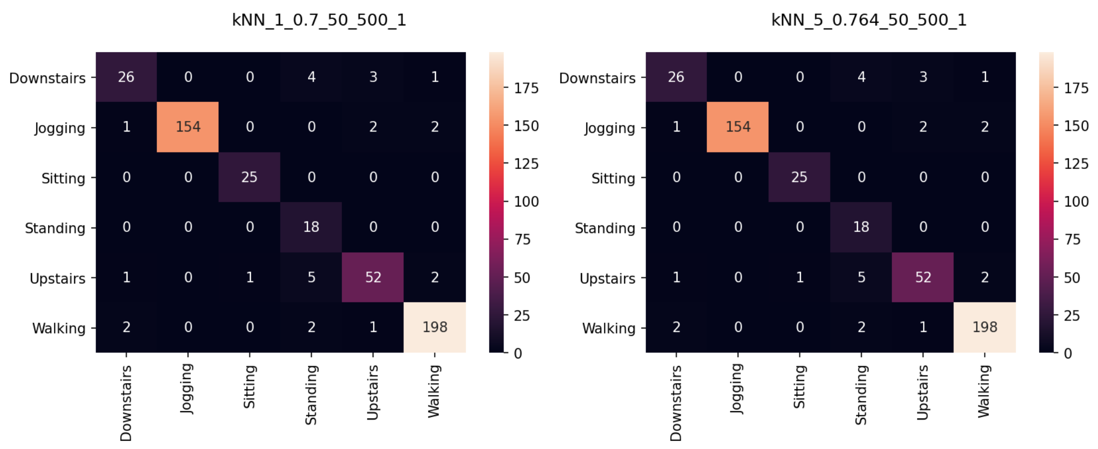

Figure 3: k-NN is extremely fast (rounded to 1 second) but the accuracy is low

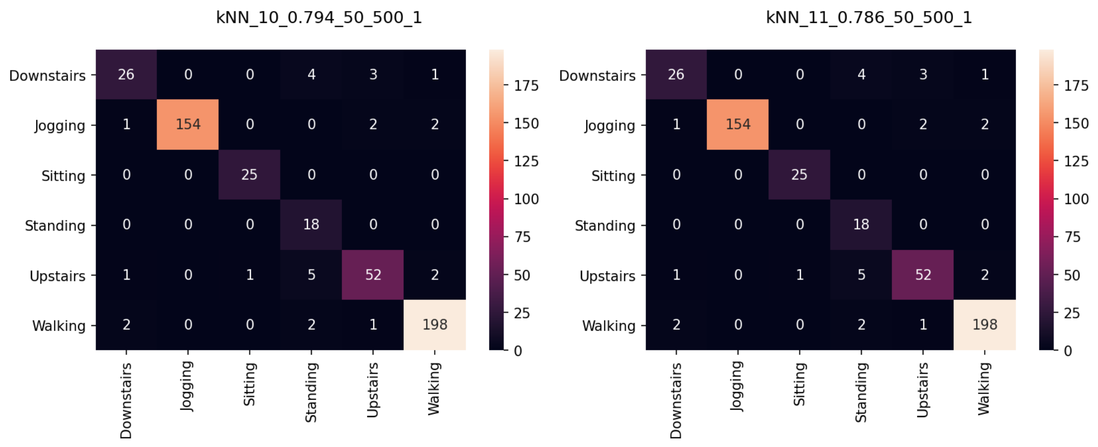

Figure 4: The accuracy peaks at K = 10 then decreases for higher K values

4 FastDTW

4.1 Differences with DTW

FastDTW is a faster alternative to DTW (5) : Due to the principle of DTW itself, it has a quadratic time and space complexity that makes it really slow and memory consuming for large time series dataset. FastDTW is an optimal to sub-optimal approximation of DTW and is able to compute any dataset with a linear time and space complexity(O(n)). According to it's paper, FastDTW uses a multilevel approach that recursively projects a solution from a coarse resolution and refines the projected solution.

4.2 Implementation

In this implementation of FastDTW , (6) we can see the recursive approach that it's creators Mr.Salvador and Mr.Chan took. reduce_by_half and expand_window have also been added for context.

```
def fastdtw(x, y, radius, dist):
    min_time_size = radius + 2
    # run the full DTW for small time series
    if len(x) < min_time_size or len(y) < min_time_size:
        return dtw(x, y, dist=dist)

    # lower the time series resolution by half
    x_shrinked = reduce_by_half(x)
    y_shrinked = reduce_by_half(y)
    # recursive call to fastdtw with the shrinked time series
    distance, path =
        fastdtw(x_shrinked, y_shrinked, radius=radius, dist=dist)
    # define the projected warp path,'radius' specify the additional
    # number of cells to be added on each side of the projected path.
    window = expand_window(path, len(x), len(y), radius)
    # Run DTW with the constraint that only cells in the projected
    # warp path are evaluated (note that this function is the original
    # dtw with an extra constraint), this is where compute time savings
    # occurs.
    return dtw(x, y, window, dist=dist)
```

```
def reduce_by_half(x):
    return [(x[i] + x[1+i]) / 2 for i in range(0, len(x) - len(x) \% 2, 2)]

def expand_window(path, len_x, len_y, radius):
    path_ = set(path)
    for i, j in path:
        for a, b in ((i + a, j + b)
                     for a in range(-radius, radius+1)
                     for b in range(-radius, radius+1)):
            path_.add((a, b))
    window_ = set()
    for i, j in path_:
        for a, b in ((i * 2, j * 2), (i * 2, j * 2 + 1),
                     (i * 2 + 1, j * 2), (i * 2 + 1, j * 2 + 1)):
            window_.add((a, b))
    window = []
    start_j = 0
    for i in range(0, len_x):
        new_start_j = None
        for j in range(start_j, len_y):
            if (i, j) in window_:
                window.append((i, j))
                if new_start_j is None:
                    new_start_j = j
            elif new_start_j is not None:
                break
        start_j = new_start_j
    return window
```

4.3 Our results

As expected, this method is way faster than the regular DTW, especially in our case since we manipulate long time series. The accuracy delta Δ between DTW and FastDTW results is reasonably low.
- $\Delta < 0.012$ with `segment-size=30`
- $\Delta < 0.032$ with `segment-size=50`

Just like regular DTW examples, we are generating our training data over 70% of the entire dataset.
The titles of each matrix are formatted this way :
`algorithm_accuracy_segment-size_test-size_compute-time(sec)`

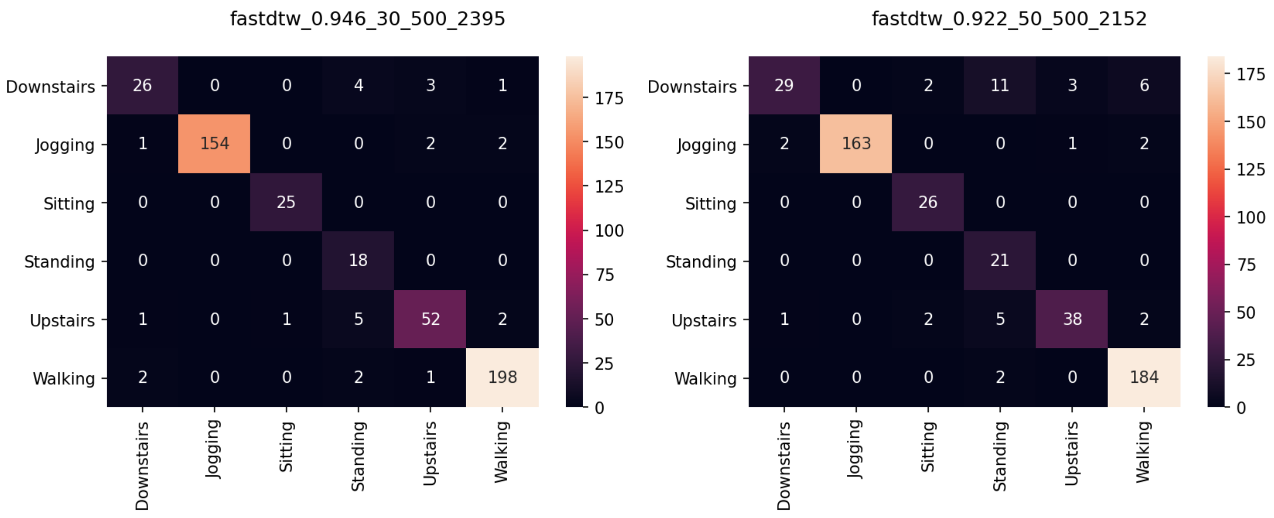

Figure 5: Our FastDTW results. Notice that we divided the compute time by 9 compared to regular DTW.

5 Conclusion

5.1 Who's the winner?

It depends on the accuracy requirements and the compute time you want to dedicate to predict a time series. If computing power isn't limited, regular DTW is the most precise option, but in a real world application of this specific problem, FastDTW provides a faster alternative with minimal precision loss and would therefore be the preferred method.
k-NN's accuracy might be low compared to DTW methods, but it might be suitable for real time and/or embedded applications, especially if we want to differentiate obvious Classes like Walking and Sitting.

5.2 Conclusion

The source code of the project was mainly written by Morteza Ezzabady while Théo Dubus focused on this report. Every result shown in this report has been calculated within Google Colab. No major struggles have been encountered during this project but understanding the role of the ACP in the k-NN method was not obvious.
This study focused on the comparison between DTW and FastDTW, to prove how and by how much time and space can be saved by using optimizations.

References

[1] DTW definition,
https://www.wikiwand.com/en/Dynamic_time_warping

[2] The cython version of DTW, courtesy of T. Giorgino.
T. Giorgino Computing and Visualizing Dynamic Time Warping Alignments in R: The dtw Package.
J. Stat. Soft., doi:10.18637/jss.v031.i07.
https://github.com/DynamicTimeWarping/dtw-python

[3] k-NN definition,
https://www.wikiwand.com/en/K-nearest_neighbors_algorithm

[4] k-NN implementation in python,
https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/

[5] FastDTW paper,
Salvador, S. and P. Chan. "FastDTW: Toward Accurate Dynamic Time Warping in Linear Time and Space." (2004).
https://www.semanticscholar.org/paper/FastDTW

[6] FastDTW implementation by slaypni,
https://github.com/slaypni/fastdtw