Avik Bhuiyan

CMPSC 463 (section 1)

Professor Yang

Time-Series Clustering and Segment Analysis Using Divide-and-Conquer Algorithms

Description of Project

This project focuses on unsupervised clustering of physiological time-series segments (ABP, ECG, PPG) extracted from PulseDB, using divide-and-conquer algorithms. The aim is to group similar signal segments and analyze their internal structure using:

- Divide-and-Conquer Clustering: recursive splitting based on similarity metrics.
- Closest-Pair Analysis: identify the most similar pair within each cluster to validate cohesion.
- Maximum Subarray Analysis (Kadane's Algorithm): highlight the most active or anomalous intervals in a segment.

The project emphasizes algorithmic design rather than machine learning heuristics to generate meaningful segmentation and insight from biomedical signals.

Installation and Usage

To run this project on your own machine, follow these instructions:

1. Clone the GitHub repository (https://github.com/Avik812/PulseDBClustering)
2. The project incorporates many libraries such as numpy, scipy, matplotlib. These can be installed using "*pip install numpy scipy h5py matplotlib fastdtw pandas*"
3. PulseDB_Vital data must also be downloaded (https://rutgers.app.box.com/s/ezmiotu66dtnahxoirnukn25nrcylybc)
4. Place the dataset in the data/Segments folder.
5. Run main.py to execute clustering and analysis.

Structure of Code

```
PulseDBClustering/              # project folder
├── data/
│   └── Segments/               # dataset
│       ├── p0000001.mat
│       ├── p0000002.mat
│       └── ...
├── src/                        # main algorithms
│   ├── clustering.py
│   ├── closest_pair.py
│   ├── max_subarray.py
│   └── utils.py
└── main.py                     # main script integrating all components
```

Description of Algorithms

1. Divide-and-Conquer Clustering of Time Series Segments

```python
def divide_and_conquer_clustering(segments, threshold=10):
    """
    Recursively cluster segments using a divide-and-conquer approach based on DTW.
    """
    if len(segments) <= 1:
        return [segments]

    # Compute pairwise DTW distances
    n = len(segments)
    dist_matrix = np.zeros((n, n))
    for i in range(n):
        for j in range(i+1, n):
            a = np.array(segments[i]).flatten()
            b = np.array(segments[j]).flatten()
            dist, _ = fastdtw(a, b, dist=euclidean)
            dist_matrix[i, j] = dist
            dist_matrix[j, i] = dist

    median_dist = np.median(dist_matrix)

    # If segments are very dissimilar, split into halves
    if median_dist > threshold and n > 1:
        cluster1 = segments[:n//2]
        cluster2 = segments[n//2:]
        return divide_and_conquer_clustering(cluster1, threshold) + \
               divide_and_conquer_clustering(cluster2, threshold)
    else:
        return [segments]
```

Purpose: Identify the most similar pair of time-series segments within a cluster.

Method:

- For every pair of segments in a cluster, compute the DTW distance.
- Track the pair with the smallest distance.

Validates cluster cohesion and selects representative segments for analysis.

2. Closest Pair of Time Series Within Clusters

```python
from fastdtw import fastdtw
from scipy.spatial.distance import euclidean
import numpy as np

def closest_pair(cluster):
    """
    Find the closest pair of segments in a cluster using DTW.
    """
    min_dist = float('inf')
    pair = (None, None)
    n = len(cluster)

    for i in range(n):
        for j in range(i+1, n):
            seg_i = np.array(cluster[i]).flatten()
            seg_j = np.array(cluster[j]).flatten()
            dist, _ = fastdtw(seg_i, seg_j, dist=euclidean)
            if dist < min_dist:
                min_dist = dist
                pair = (i, j)
    return pair, min_dist
```

Purpose: Recursively group similar time-series segments based on Dynamic Time Warping (DTW) distances.

Method:

- Compute pairwise DTW distances between all segments.
- Calculate the median distance.
- If the median exceeds a threshold, split segments into two halves and recursively cluster each half.
- Otherwise, return the segments as a cluster.

Key Feature: Automatically forms clusters of similar segments without predefined cluster counts.

3. Maximum Subarray Analysis (Kadane's Algorithm)

```python
def kadane(arr):
    """
    Kadane's algorithm to find maximum subarray sum.
    Returns (max_sum, start_index, end_index)
    """
    if len(arr) == 0:
        return 0, None, None

    max_sum = current_sum = arr[0]
    start = end = s = 0

    for i in range(1, len(arr)):
        if current_sum < 0:
            current_sum = arr[i]
            s = i
        else:
            current_sum += arr[i]

        if current_sum > max_sum:
            max_sum = current_sum
            start = s
            end = i

    return max_sum, start, end
```

Purpose: Find the contiguous subarray with the maximum sum in a time series.

Method:

- Iterate through the array, keeping track of the current subarray sum.
- Reset the sum if it becomes negative.
- Update the maximum sum and indices when a new maximum is found.

Highlights of intervals of peak activity or anomalies in physiological signals.

Verification of Functionality (toy example)

The project's intended functionality is to cluster time-series segments, find the closest pairs, and detect maximal subarrays using Kadane's algorithm. The toy example approach uses simple 1-D arrays to test the algorithm before scaling to real data.

Problem encountered:

Despite using toy arrays like [1,2,3,2,1] and [2,3,4,3,2], running the divide_and_conquer_clustering or closest_pair functions produces errors such as: ValueError: Input vector should be 1-D from fastdtw ValueError: setting an array element with a sequence when constructing distance matrices IndexError: index 0 is out of bounds from Kadane's function when segments are empty or improperly formatted

Why this happens:

Input shape issues: fastdtw expects a strict 1-D numeric array. Even small inconsistencies like lists inside a NumPy array or nested arrays of differing lengths trigger errors. Inhomogeneous arrays: When attempting to build np.array(segments), if the segments are of different lengths (common in physiological signals), NumPy creates an object array, which cannot be processed directly with DTW or distance metrics. Residual .mat or .npy file complexities: Real PulseDB segments may contain metadata, missing values, or string entries (like '00b'), which causes conversion errors when trying to cast to floats.

Attempted solutions:

Using np.ravel() or flatten() to force 1-D arrays. Switching to small toy examples with hardcoded floats. Isolating code paths to test fastdtw and kadane individually.

Result: Even with toy data, unless all segments are explicitly 1-D NumPy arrays of consistent numeric type, fastdtw throws ValueError. This prevents a full verification run on even the simplest datasets.

Execution Results with 1000 time series

I could not execute the project on 1000 time series due to the persistent input errors.

Attempting to scale up:

- Real PulseDB data segments contain non-numeric entries or inconsistent lengths.
- fastdtw and the clustering function cannot handle empty or object arrays.

- Recursive divide-and-conquer clustering fails immediately if any distance computation encounters invalid input.

Implication:

Without preprocessing to guarantee 1-D numeric segments of equal length (or padding sequences properly), the algorithm cannot run on large datasets.

Hypothetical results if executed correctly:

- Clusters would form based on median DTW distance thresholds.
- Closest pairs would identify the most similar segments within each cluster.
- Kadane's algorithm would detect the maximal subarrays indicating peak signal activity.

Because the input pipeline was never fully functional, these results are theoretical.

Discussion on Results

The main issue preventing successful execution was that the input time-series segments were not in the proper format for the algorithms. The DTW-based clustering and closest-pair analysis require strictly 1-D numeric arrays, but the PulseDB segments often contain nested structures, inconsistent lengths, or non-numeric entries. Even with small toy examples, the algorithms failed unless all sequences were explicitly formatted as flat numeric arrays.

This highlights an important practical lesson: even well-designed algorithms can fail on real-world biomedical data if the input is not properly preprocessed. While the divide-and-conquer clustering, closest-pair detection, and Kadane's maximum subarray analysis are conceptually sound, their success depends entirely on clean, consistent input data.

Conclusions

The project demonstrates the design of algorithms for clustering and analyzing physiological time-series data. Although execution on the full dataset was not possible due to input formatting issues, the methods themselves, recursive clustering with DTW, closest-pair analysis, and Kadane's algorithm are valid and would provide meaningful insights if the data were properly prepared.

This experience emphasizes the importance of data preprocessing in time-series analysis. In future work, ensuring that all segments are numeric, 1-D, and of compatible length would allow these algorithms to function correctly and reveal patterns in the physiological signals.