Internship at CommSolid GmbH
Data Segmentation Tool

Mehmet Yaşar Alıcı

August, 30 2020

Abstract

A general-purpose pattern segmentation tool was developed within the internship. With the assumption of stillness between the repetitions of patterns, Euclidian distances between subsequent data points are utilized to extract movements. Interactive visualization tools in front-end are employed to guide users in the segmentation. A dataset adapter is proposed to relax assumptions with the tool. The efficiency of the tool was demonstrated with time complexity tests.

Contents

Chapter 1

Introduction

With the advance of machine learning optimized for devices with low computational power and resources, there has been a tremendous interest in the industry to implement machine learning directly on these devices, alleviating the need for an external computationally powerful server.

As observed from Figure 1.1, the cost sending data is much more than processing it. Running a neural network on device and sending the resulting information, the device will consume less power, hence, benefit from an increased battery life. On the other hand, since the information sent has less size than that of the raw data, less network bandwidth will be used. Thus, the network subscription fees drops as well.

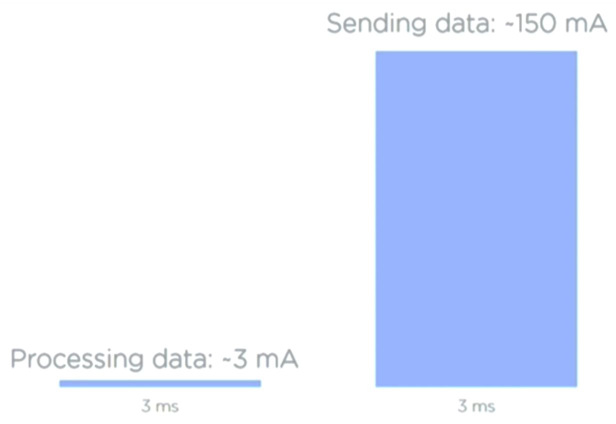


Figure 1.1: Comparison of current consumption between processing data and sending data [2]

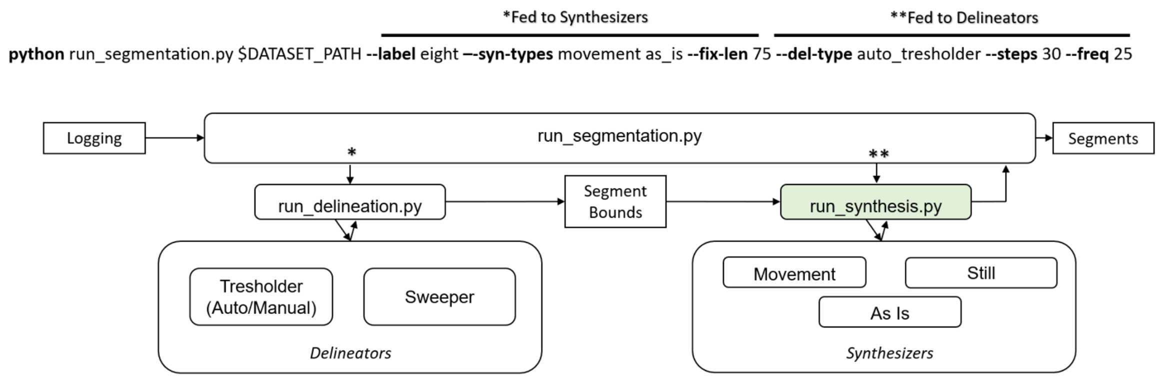
.

A common use case is the deployment of deep learning based pattern recognition algorithms. However, these algorithms require labeled patterns in huge amounts on training to perform well in inference. Therefore, the patterns from any logged data must be correctly extracted, prepared and labeled. On the other hand, the segmentation algorithms should be time efficient to cope with datasets huge in size.

To address these needs, a general-purpose pattern segmentation tool with a $\mathcal{O}(k * n)$time complexity was developed.

Chapter 2

Architecture



```
python run_segmentation.py $DATASET_PATH --label eight —-syn-types movement as_is --fix-len 75 --del-type auto_tresholder --steps 30 --freq 25
```

Figure 2.1: Architecture of Pattern Segmentation Tool

Refer to Figure 2.1. The tool runs from the command line with desired parameters. The segmentation and labelling is done in two major blocks in a sequential manner. First, Delineators obtain a dataset containing patterns to be extracted and mark bounds to the edges of the segments, then, Synthesizers labels and extracts these. Moreover, Synthesizers can also modify the segments before extraction, i.e. add a Gaussian noise or shift markings to create new and interesting segments. Delineators and Synthesizers will be explained in detail in the Chapters 3 and 4, respectively.

Chapter 3

Delineators

Delineators' duty is to describe the bounds of the segments in a dataset. The application contains two group of Delineators: i) Thresholders, that delineates by an help of a threshold and ii) Sweeper, that starts from start, marks bounds by fixed steps until the end. This chapter will explain the operation of these types.

3.1 Thresholders

The work in this section is inspired by [3], where the authors used an Euclidian Distance Vector to segment patterns. In a similar fashion, A one dimensional distance vector D out of the Euclidian distance between subsequent 3D accelerometer samples was created.

In other words, given a 3D accelerometer sample a_i,

$$a_i = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$$

define a distance vector D,

$$D = \{d_i, d_{i+1}, ...\},$$

where,

$$d_i = |a_{i+1} - a_i|$$

Figure 3.1 is obtained as a result of this operation. As observed from Figure 3.2, Thresholders segment patterns via setting an horizontal threshold τ to the distance vector. The segments are, then, marked with a comparison to that threshold.

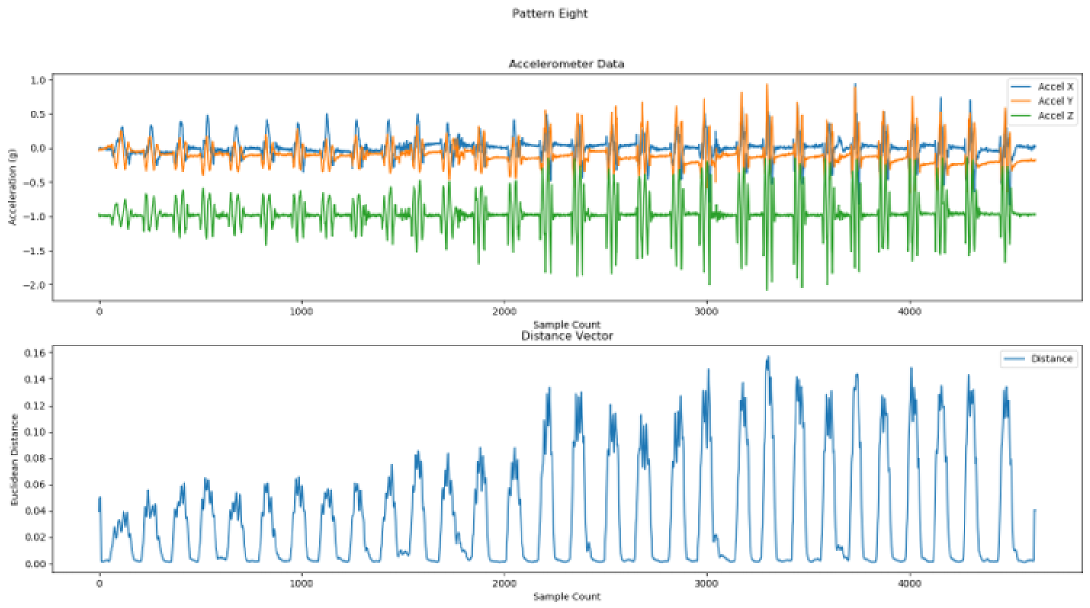


Figure 3.1: An accelerometer logging with its corresponding distance vector.

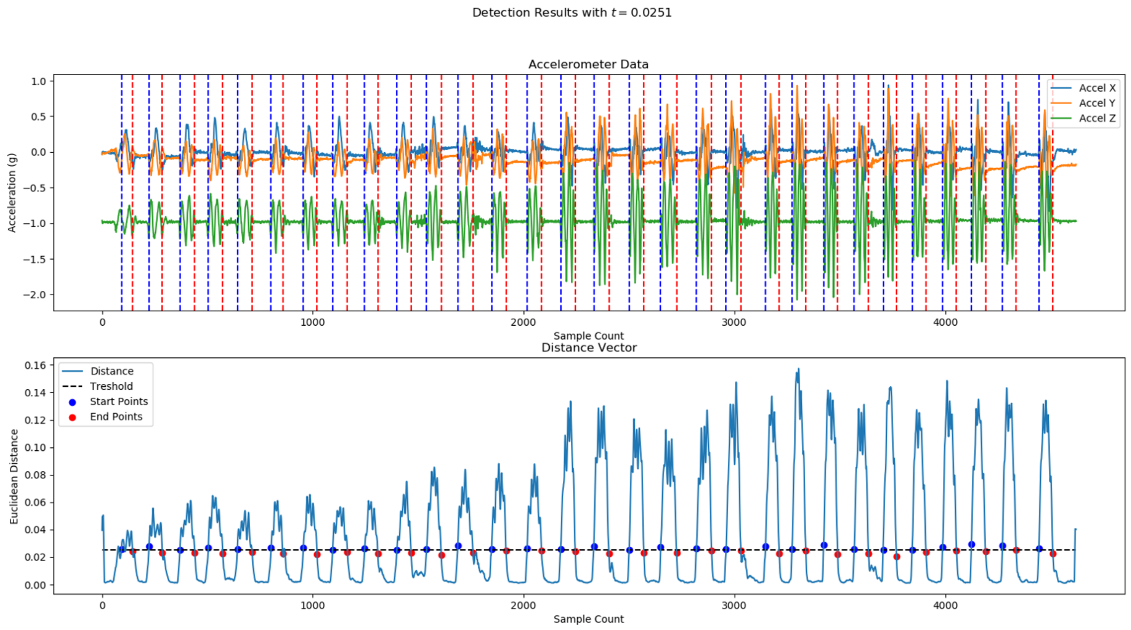


Figure 3.2: Horizontal threshold $t = 0.0251$ on distance vector and corresponding segments it marks.

3.1.1 Manual Thresholder

Refer to Figure 3.3. In this type of thresholder, a threshold τ_m is determined manually via clicking and dragging with mouse over the distance vector. Then, the values of the distance vector is compared with the selected threshold level to discriminate movements from stillness, as illustrated in Figure 3.2.

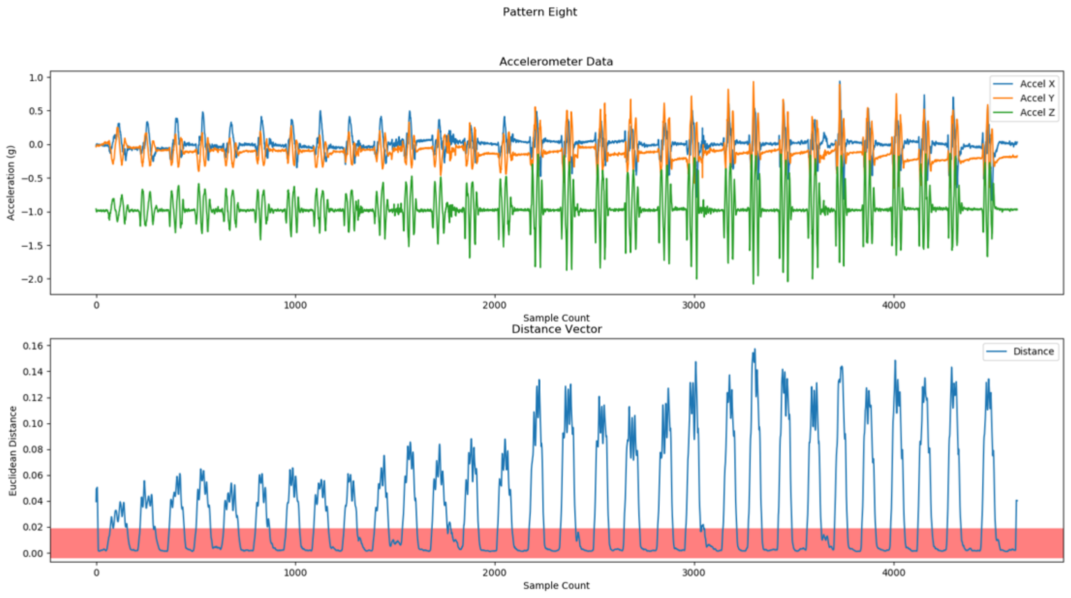


Figure 3.3: Selecting a threshold with mouse over the plot.

3.1.2 Auto Thresholder

Auto thresholder sweeps horizontal threshold candidates from top to bottom in distance vector with a fixed step size k. This value is provided by user in the command line. More step size gives more resolution, therefore, a better detection; but on the other hand, takes longer to finish.

For every threshold candidate τ, it collects tuples of locations of start and end points, (a_i, b_i), of segments and their lengths,

$$b = (a_1, b_1), ..., (a_n, b_n) \tag{3.1}$$
$$\ell_b = (l_1, ..., l_n) \tag{3.2}$$

where $l_i = (b_i - a_i)$, and n is the total number of segments delineated for that threshold.

Following the sweep, Auto Thresholder exploits the collected information over threshold candidates to determine the best threshold. For every candidate τ, it will consider,

i. number of segments delineated n,

ii. standard deviation of the segment lengths, $\sigma(\ell_b)$,

$n(\tau)$ is already known by Eq. 3.1. The standard deviation of a threshold candidate τ is calculated as,

$$\sigma(\ell_b(\tau)) = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}\left(l_i(\tau) - l(\bar{\tau})\right)^2} \tag{3.3}$$

where $l(\bar{\tau})$ is the mean of segment lengths for a threshold candidate τ.

Eq. 3.3 grants a measure of how much lengths of segments varies within a threshold candidate τ. As observed from upper plot of Figure 3.5, poor threshold values will include wrong segments, such as unexpectedly narrow and/or unexpectedly large ones, leading to a big σ.

On the other hand, in a successful delineation, all segments are supposed to be of similar lengths, leading to a relatively low σ. Also, number of segments delineated in this case will be relatively higher. Therefore, defining a score s of a threshold candidate τ as a ratio of segment count n and standard deviation σ,

$$s(\tau) = \frac{n(\tau)}{\sigma(\tau)}$$

Auto Thresholder optimizes the threshold selection by selecting the best threshold t^* that maximizes the score $s(\cdot)$,

$$t^* = \arg\max_t s(t)$$

The sweep results are illustrated in Figure 3.4. In the upper axes, $n(\tau)$ and $\sigma(\tau)$ are plotted in red and blue colors, respectively, with respect to threshold candidates τ. For bigger values of τ, number of delineated segments n shows a tendency to decrease, while σ tends to display gross fluctuations.

In the lower axes, the threshold that maximizes the score s, $t^* = 0.0251$ will be determined as the optimal threshold that gives the best delineation performance. Therefore, the Auto Thresholder will keep it to perform its final delineation, which is shown in the lower axes of Figure 3.5.

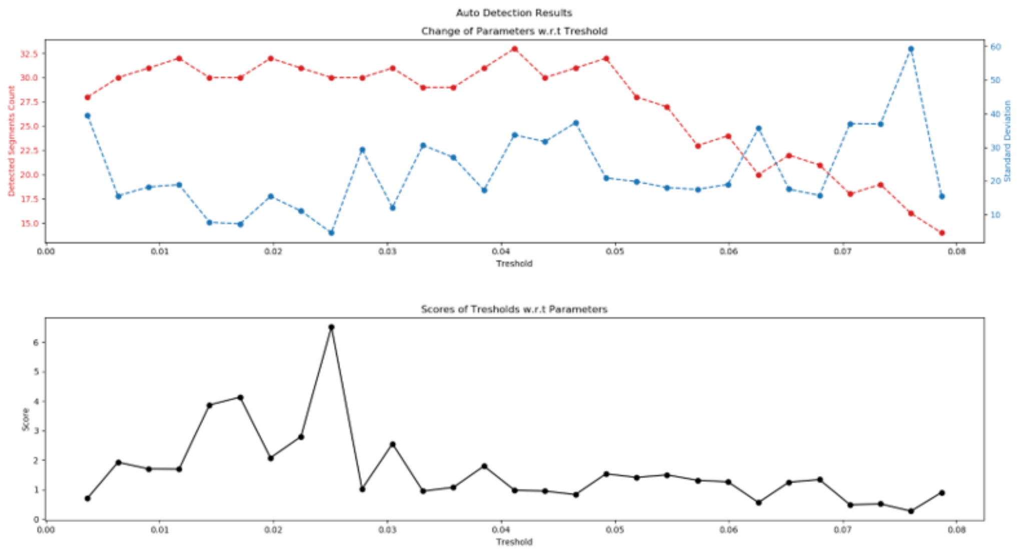
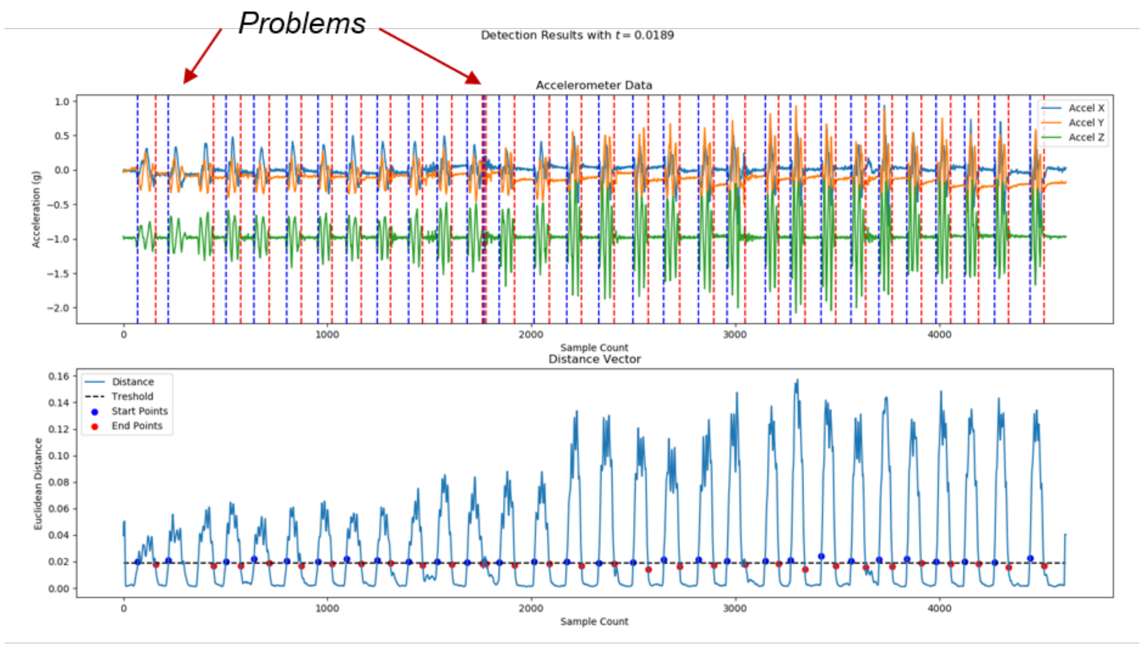


Figure 3.4: Sweep results of threshold candidates

Comparison of Auto Thresholder with Manual Thresholder

Manual Thresholder is problematic as it is prone to human error. A threshold selected with human eye τ_m might not capture all segments, and missing some segments might lead previous segment to be double in size. However, thanks to the statistics, Auto Thresholder is robust to those fluctuations, and performs better.

Figure 3.5 illustrates this phenomenon.



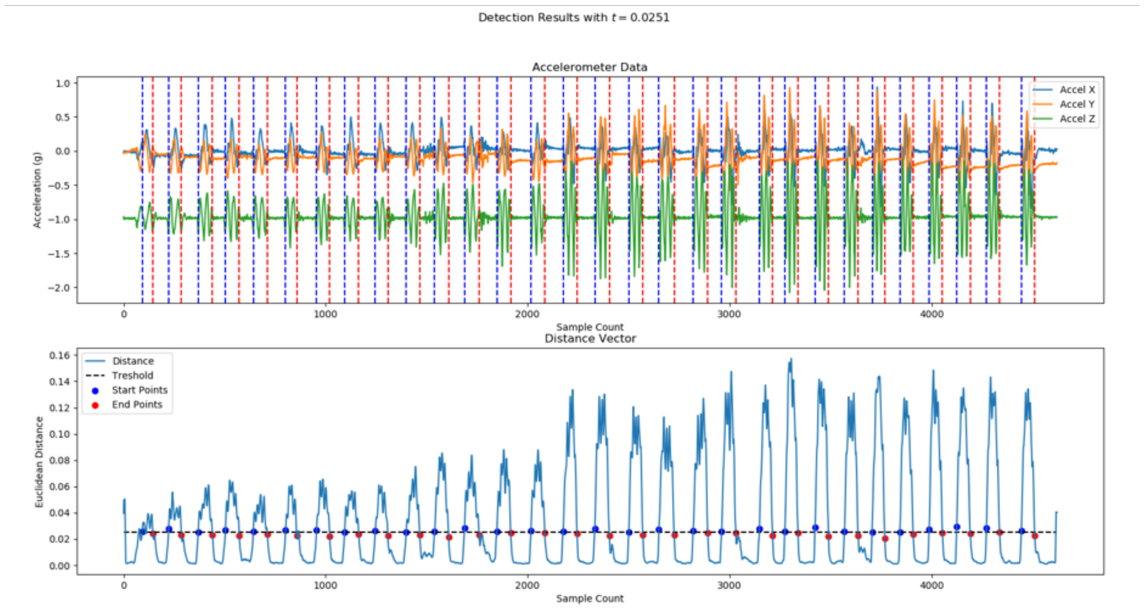


Figure 3.5: Comparison of Auto Thresholder (below) with Manual Thresholder (upper) on the same dataset. Note the problems with the Manual Thresholder.

3.1.3 Detection Algorithm for Thresholders

Refer to the Listing 3.1 which implements a comparison with a selected threshold. Essentially, the algorithm follows the following logic.

Given a status of either ”still” if a value in distance vector is below threshold, or ”movement” otherwise, the steps taken can be summarized as below:

1. check for a change in state.
2. If so, check samples in backwards and forwards if they remained in their state.
3. If so, determine if a pattern already started.
4. If so, mark the bound as finish b_i else a_i.

Listing 3.1: Thresholder's detection algorithm

```python
def _detect_bounds(self, horizon):
    """
    Mark the segment bounds by appending to self.seg_bounds
    :param horizon: An integer to designate how many samples in both directions the
        algorithm should consider during a state change in order to mark a bound.
    """

    # Initialize relevant variables
    self.seg_bounds = []
    outward_change = [True, False]
    inward_change = [False, True]
    gesture_started = False

    # A forward pass to distance vector. O(n)
    for first_idx in range(len(self.dist_df.values) - 1):
        # Set status, could be a change outwards, inwards or no change.
        sec_idx = first_idx + 1
        first, sec = [self.dist_df.values[first_idx], self.dist_df.values[sec_idx]]
        status = list(map(lambda i: i < self.treshold, [first, sec]))

        if status == outward_change and not gesture_started:
            # Outward change detected, check backwards and forwards for {horizon} number
                of samples. O(1)
            if self._is_all_upside(sec_idx, self.treshold, horizon) and \
                    self._is_all_downside(sec_idx, self.treshold, horizon, reverse=True):
                gesture_started = True
                self.seg_bounds.append([sec_idx])
        if status == inward_change and gesture_started:
            # Inward change detected, check backwards and forwards for {horizon} number of
                samples. O(1)
            if self._is_all_downside(sec_idx, self.treshold, horizon) and \
                    self._is_all_upside(sec_idx, self.treshold, horizon, reverse=True):
                gesture_started = False
                self.seg_bounds[-1].append(sec_idx)

    # A safe fix if the dataset is finished before the last segment.
    if len(self.seg_bounds) and len(self.seg_bounds[-1]) == 1:
        self.seg_bounds.pop(-1)
```

3.1.4 Time Complexity Analysis

Refer to Listing 3.1. Since there is only one forward pass to the distance vector D, and interally the *self._is_all_downside* and *-upside* functions loops over constant *horizon* times, we conclude that the complexity of delineation by one threshold τ is $\mathcal{O}(n)$.

Since manual threshold only uses one threshold -chosen by the user-, we conclude that the time complexity of Manual Thresholder is $\mathcal{O}(n)$.

In a similar fashion, Auto Thresholder runs τ by step size k prompted by user. Therefore, the complexity of Auto Thresholder is $\mathcal{O}(k * n)$.

Experimentation

Refer to Listing 3.2. To test time complexity, we double the size of a selected dataset at each iteration of five. The running time is timed and printed to *stdout* as observed from Listing 3.3. As a result, we obtain Table 3.1, which proves our analysis at Subsection 3.1.4.

Moreover, the row with asterisks (*) in Table 3.1 shows an estimation of dataset count segmented in one minute. This is to say that in one minute, the algorithm is able to delineate a dataset containing around 1.2 Million 3-axis data samples, which shows that it scales efficiently with the input size.

Listing 3.2: Test client to analyze time complexity.

```python
if __name__ == "__main__":
    # Test client
    dataset_path = os.environ.get("DATASET_PATH")
    df = pd.read_csv(dataset_path, index_col="index", sep=";")

    for i in range(5):
        df = pd.concat([df, df]) # Double the dataset at each iteration
        label = "eight"
        sampling_freq = 25
        steps = 4
        detect_manually = False
        run_tresholder(df, label, sampling_freq, detect_manually=detect_manually, steps=steps)
```

Listing 3.3: Information over one iteration. Note the printed running time information

```
    Detection Information

    With treshold = 0.0528:
    Count: 58
    Std: 19.98
    Mean: 42.79
    Histogram:
    6 - 12: 8
    12 - 18: 0
    18 - 23: 8
    23 - 29: 2
    29 - 35: 0
    35 - 41: 2
    41 - 47: 2
    47 - 52: 4
    52 - 58: 20
    58 - 64: 12
    Candidate's score: 2.90
    Running Time: 0.29s
```

Dataset Counts (per 4692 3-axis points)	Observed Running Time t (s)	$\frac{t_{i+1}}{t_i}$
1	0.29	-
2	0.61	2.17
4	1.18	1.94
8	2.23	1.89
16	3.82	1.71
256*	6 x 10*	15.7

Table 3.1: Observed running time as the dataset doubles. Note that $\frac{t_{i+1}}{t_i}$ approximates to 2 in iterations.

3.2 Sweeper

Refer to Figure 3.6. The Sweeper starts from the beginning of a dataset, repeatedly marks the segments *sweeping* a specified length until the end of the dataset. One obvious use case for Sweeper is that it can segment and label "Others" class, or negative class, for the training of Neural Networks.

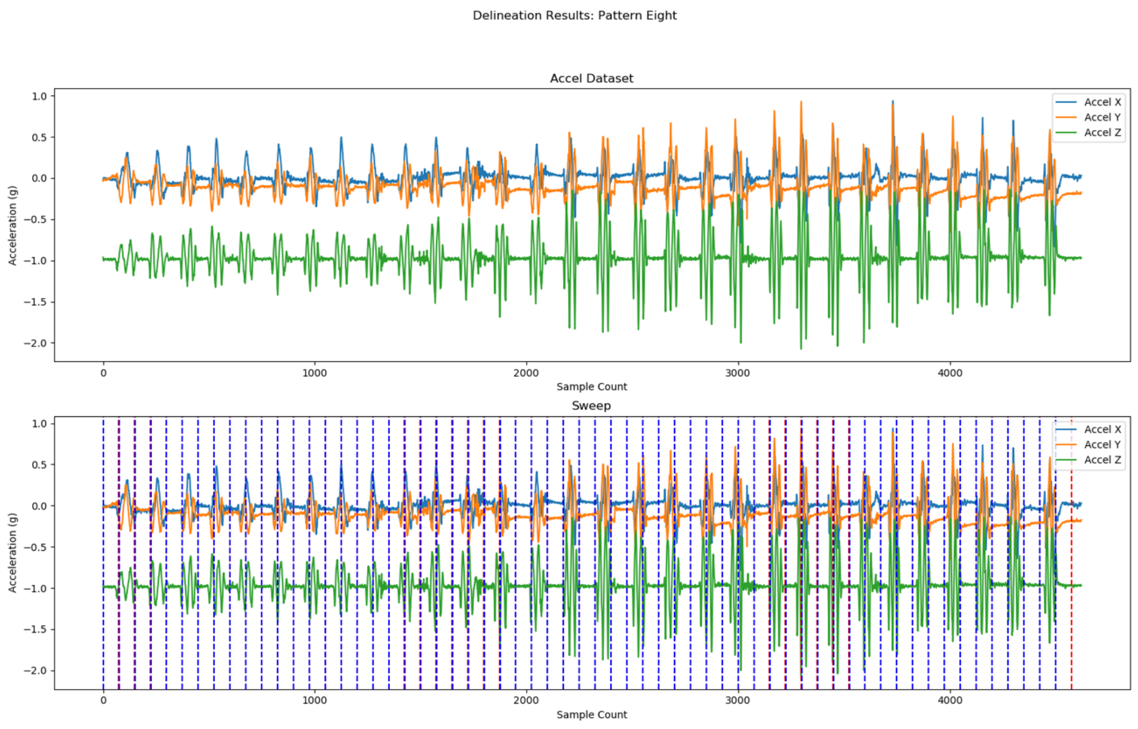


Figure 3.6: Sweeper as a method of delineation. Note the segments on the below plot.

Chapter 4

Synthesizers

Synthesizer family of objects extracts patterns harnessing the reference delineations fed by Delineators. There are three types of Synthesizers available within the tool, ”As Is”, ”Movement” and ”Still” and their operation will be explained in the following sections.

4.1 As Is Synthesizer

As Is Synthesizer, as its name suggests, does not play with the segment bounds obtained from Delineators. Upon receiving the segment bounds, it seperates the dataset exactly following these bounds. Then, it labels the segments and exports them to the filesystem.

Figure 4.1 illustrates its operation.

4.2 Movement Synthesizer

Movement Synthesizer supports only Thresholder family of Delineators. It obtains bound markings from a Thresholder and extracts them to segments. Then, it rearranges each segment to the specified fixed length and recombines all into a dataset. Finally, it recombines segments, add labels and exports them to filesystem.

Figure 4.2 illustrates the operation.

4.3 Still Synthesizer

Still Synthesizer obtains bound markings from Thresholders, as in the case of the Movement Synthesizer, and extracts the in-between sections of the segments, i.e so-called stillness. Then it rearranges each segment to the specified fixed length. If length is less than that of specified, it adds a Gaussian noise. Finally, it recombines all segments into a dataset, labels them and exports to filesystem.

Figure 4.3 illustrates the operation of Still Synthesizer.

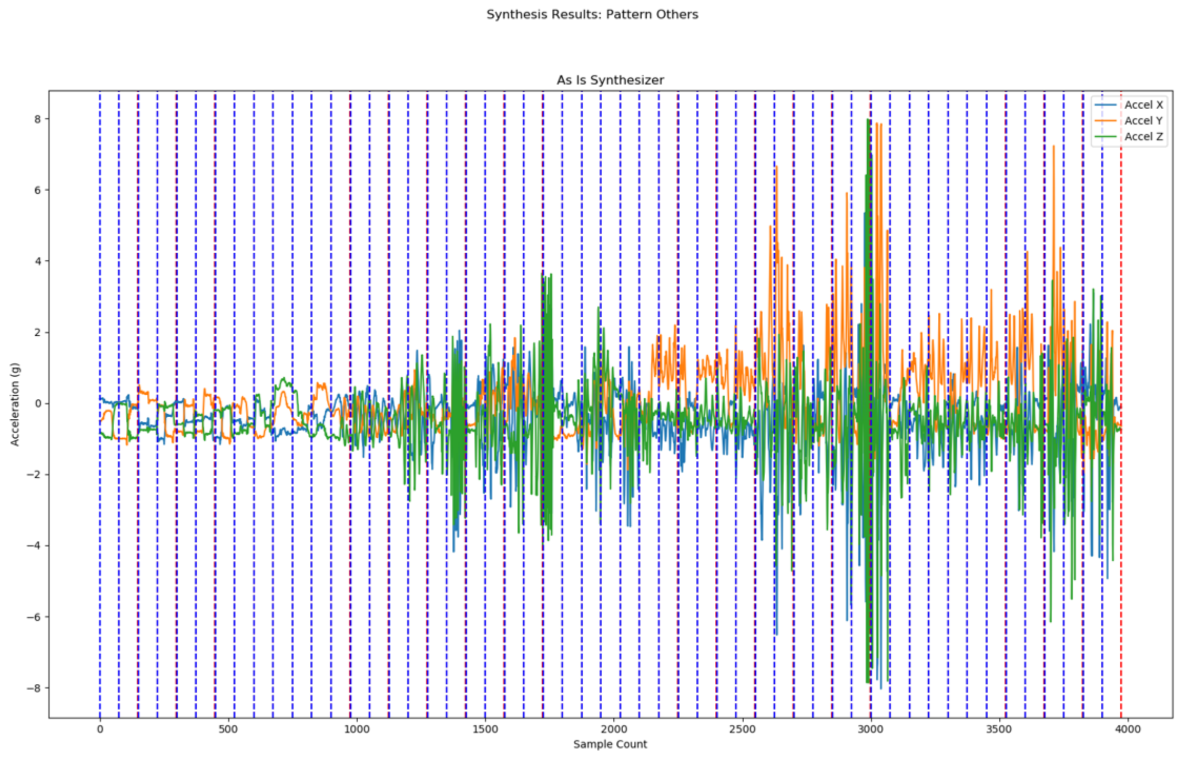


Figure 4.1: As Is Synthesizer selected to segment ”Others” class

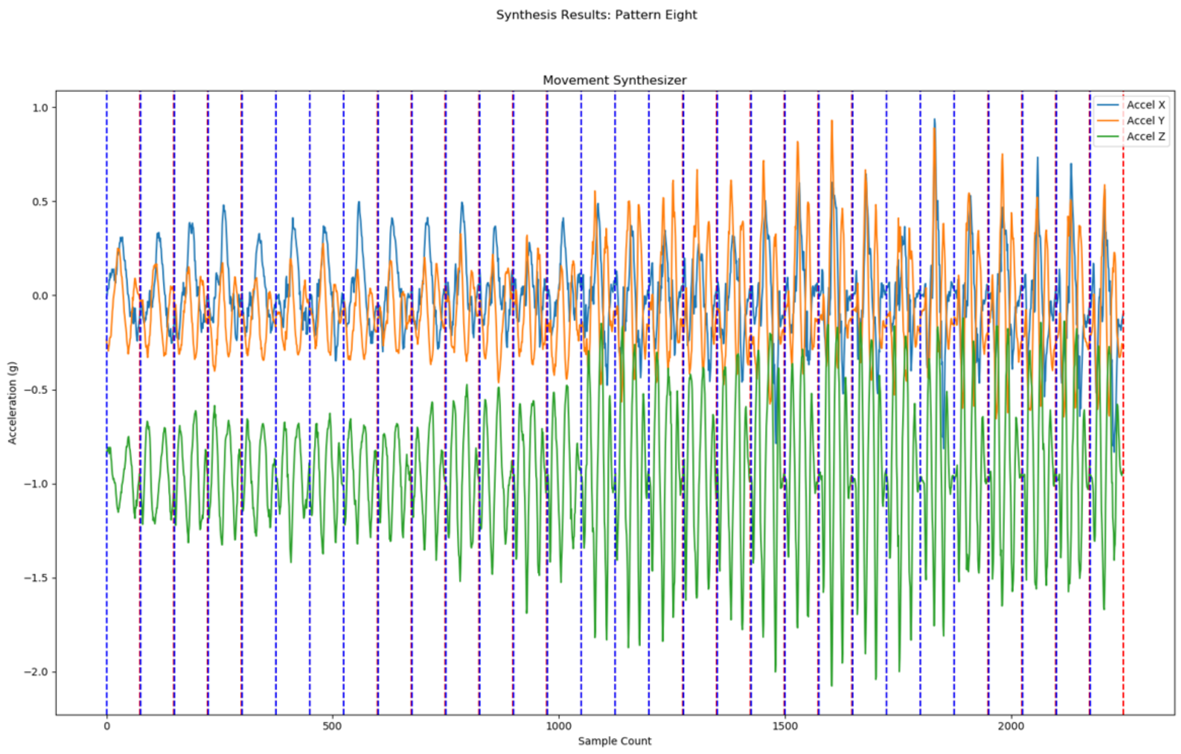


Figure 4.2: Movement synthesizer to segment ”Eight” patterns.

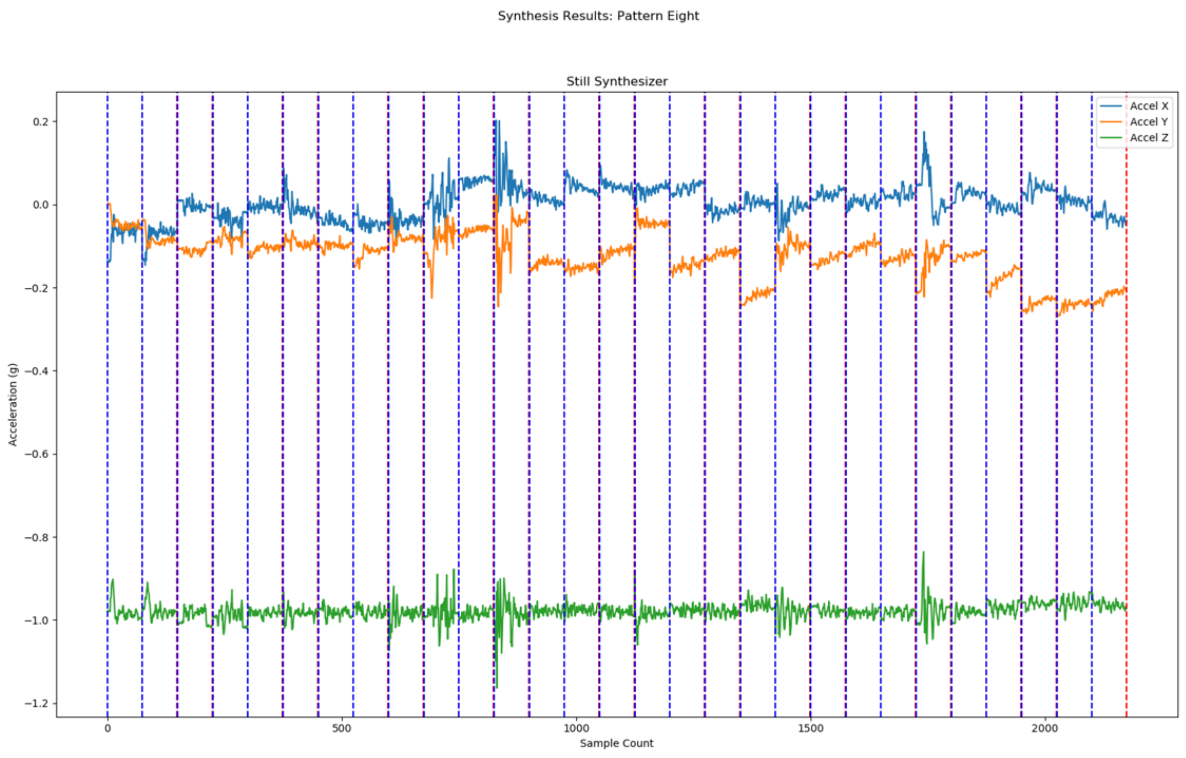


Figure 4.3: Stillness Synthesizer to segment in-between section between bounds taken from Thresholders.

Chapter 5

Assumptions and Relaxations

The Pattern Segmentation Tool only accepts datasets as input with certain criterias, namely,

1. Dataset must be ";" seperated and only contain 3-Axis accelerometer data columns.
2. Dataset must contain repetitions of only one pattern.
3. There must be at least around 2-second-long stillness between the repetitions of the pattern.

To relax these assumptions, a Dataset Adapter is developed. It supports,

- Resampling
- Reversal check
- Comparison with ground-truth data
- Discarding unwanted sections
- Discarding extra columns

Listing 5.1 shows a usage information.

Listing 5.1: Dataset Adapter Usage Information

```
usage: run_adapter.py [-h] [-x X_COL_IDXS [X_COL_IDXS ...]] [-t TIME_COL_IDX]
                      [-s SEP] [-d] [-e] [-u UNIT] [--truth-path TRUTH_PATH]
                      [-r] [--in-freq IN_FREQ] [--out-freq OUT_FREQ]
                      [--out-path OUT_PATH]
                      path

positional arguments:
  path                Path of the datasets to be adapted.

optional arguments:
  -h, --help          show this help message and exit
  -x X_COL_IDXS [X_COL_IDXS ...], --x-col-idxs X_COL_IDXS [X_COL_IDXS ...]
                      Accelerometer x, y, z column indexes. Defaults to
                      indexes 1, 2, 3.
  -t TIME_COL_IDX, --time-col-idx TIME_COL_IDX
                      Time column index. Must be in seconds. Defaults to 0.
                      Disregarded if '-in--freq specified.'
  -s SEP, --sep SEP   Separator of the datasets. Defaults to ','
```

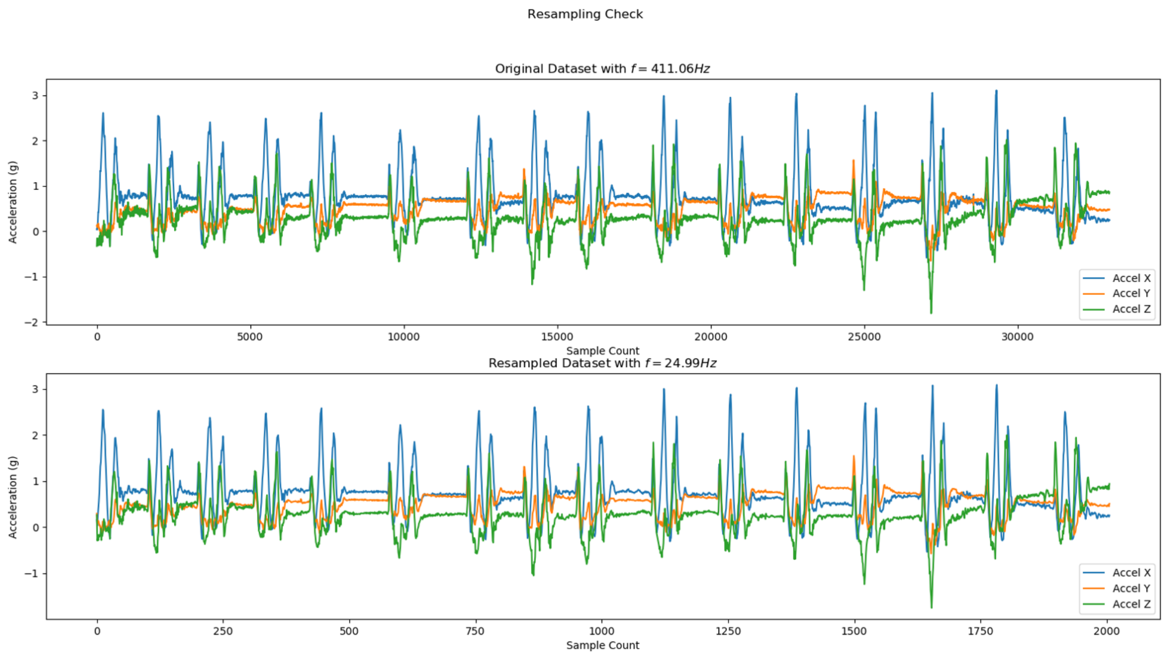


Figure 5.1: Resampling of a Dataset within Dataset Adapter

```
-d, --dis-unw       Specify to enable the selection of the desired segment
                    groups in the dataset
-e, --export        Set it to enable exportation. Resulting datasets is
                    exported under the same folder with the nameset in '--
                    out-name' argument. If it is not set, it defaults to
                    original+'_adapted_timestamp'
-u UNIT, --unit UNIT Unit of the measurement. Supported: 'm/s^2' or 'g'
--truth-path TRUTH_PATH
                    Compare the dataset with a ground-truth data.
-r, --resample      Set it to enable resampling. Note that the tool uses
                    scipy.signal.resample method in which a Fourier method
                    is used for resampling. Therefore the signal is
                    assumed to be periodic. For more information refer to
                    its documentation.
--in-freq IN_FREQ   Input sampling frequency of the datasets. If not
                    provided --time-col-idx must be specified.
--out-freq OUT_FREQ Desired output frequency for the datasets.
--out-path OUT_PATH Desired file path for the datasets. Supports relative
                    paths w.r.t the script's path. If not specified and
                    '-e' is set, outputs under the same directory with the
                    original dataset with a name{original}_adapted.csv
```

Figure 5.1 plots datasets before and after sampling operation. The original dataset is resampled using `scipy.signal.resample` method, which uses FFT transformations [1]. Due to that, the signal is assumed to be periodic and it can be very slow if the number of input and output samples is large and prime.

Chapter 6

Conclusion

Due to the need for high quality data to train various pattern recognition algorithms, there is a need to have a reliable and scalable pattern segmentation tool. The tool that is explained in this report meets these objectives. It builds upon ideas from [3] in terms of obtaining an Euclidian Distance Vector and introduces an Auto Thresholder mechanism which harnesses statistics to estimate parameters, which would be prone to error with human heuristics. Hyperparameters along with the dataset to be segmented are taken from command line, the data flows elegantly through Delineators followed by Synthesizers to become labeled segments. It is proved that the time complexity of Auto Thresholder, $\mathcal{O}(k * n)$, so that it scales well to huge datasets. Lastly, a Dataset Adapter is developed to make datasets compatible for the tool.

Bibliography

[1] *scipy.signal.resample*¶. URL: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.resample.html`.

[2] Nordic Semiconductor. *Low power cellular for the IoT - Peder Rand presentation at NDC Oslo.* 2020. URL: `https://www.youtube.com/watch?v=5Cx4xuHjHFM`.

[3] R. Xie and J. Cao. "Accelerometer-Based Hand Gesture Recognition by Neural Network and Similarity Matching". In: *IEEE Sensors Journal* 16.11 (2016), pp. 4537–4545.