Bachelor thesis

Artificial Intelligence

Radboud University

Trajectory segmentation using greedy best first change point detection based on trajectory similarity

Author:
T.H. van Buuren
s1000876

First supervisor:
E.A. van Dam
Artificial Intelligence,
Artificial Cognitive
Systems group
e.vandam@donders.ru.nl

Second supervisor:
prof. dr. M.A.J. van Gerven
Artificial Intelligence,
Artificial Cognitive
Systems group
m.vangerven@donders.ru.nl

June 25, 2021

Abstract

In day to day life finding structure in information is easy for humans, but this is not so easy for smart machine learning methods. They require structured data in order to be smart and become smarter, the more data, the better. There is a immense amount of data in existence, but most of this data is unstructured. Especially trajectory data, that consists of a series of x and y coordinates, is difficult to structure because of the large variety of information it can represent.
Many methods exist that can find structure in trajectory data by splitting it into many small segments, creating a trajectory segmentation. Many of these methods lack the ability to fit to patterns of different length and thus struggle to find segments of different lengths. Many methods also use complex approaches that are difficult to reason about and thus to generalize to other data, where a simple rule-based strategy could yield similar results. Most notably trajectory similarity methods are simple methods to compare trajectories. They are an ideal candidate to be used in the context of trajectory segmentation, but are scarcely being used.
We identify a new solution that uses a trajectory similarity-based cost function and multiple temporal time scales to identify the points of largest pattern change, by fitting windows to patterns in the data. This has the simple intuition that, if any two patterns are very different, they are likely to be different processes and should thus be separate segments. In a comprehensive test using numerous different parameter configuration on a dataset consisting of 5 trajectories of mice, we compare our method to two of the best change point detection methods: Binary segmentation and Pelt. The results indicate that our method achieves the best performance when the window size can be optimized for the data. When our method is not optimized for the data, Pelt generally has the better performance.

Contents

1 Introduction **3**
1.1 The intuition of our method 5
1.2 How it solves the problem 5
1.3 Research question 6
1.3.1 Contributions 6

2 Preliminaries **8**

3 Related Work **10**
3.1 Change point detection 10
3.2 Trajectory segmentation 11
3.3 Trajectory similarity 13
3.4 Deep Learning in Change Point Detection and Trajectory Segmentation 14

4 Methods **16**
4.1 Step 1: Scoring Pattern Change 17
4.1.1 Trajectory Similarity 20
4.2 Step 2: Greedy Change Point Selection 21
4.3 Stop Criterion 24

5 Experiments **25**
5.1 Dataset 25
5.2 Feature sets 26
5.2.1 Feature extraction 26
5.3 Metrics 27
5.3.1 Covering metric 27
5.3.2 F1 metric 28
5.4 Model Parameters 28
5.4.1 Experiment 1: Comparison to GSBS 28
5.4.2 Experiment 2: Effect of Maximum Window Size . . . 29
5.4.3 Experiment 3: Comparison to Binseg and Pelt 29

6 Results **31**
6.1 Comparison to GSBS 31
6.2 Effect of the Maximum Window Size 36
6.3 Comparison to Binseg and Pelt 37

7 Discussion **40**
7.1 Comparison to GSBS 40
7.2 Effect of Maximum Window Size 41
7.2.1 Metric Scores 41
7.2.2 Runtime 43
7.3 Best Methods Comparison 43
7.3.1 Main trial partitions 43
7.3.2 Full dataset 44

8 Future work **46**
8.1 Possible improvements 46
8.2 Trajectory testing with trajectory segmentation methods . . . 47
8.3 Possible extensions 47
8.3.1 Monte Carlo Tree Search 47
8.3.2 Hierarchy detection 47

9 Conclusions **49**

A Appendix **54**
A.1 Effect of Maximum Window Size 54
A.2 Table of All Results on Six Partitons of Mice_trajectories_1 . 56

Chapter 1

Introduction

In day to day life humans and other animals deal with a constant input stream of data, information rarely exists 'on its own', without a connection to a previous event. As such, segmenting events based on their similarity to each other or their relevance to the current situation is paramount to the thriving of our species. In most situations, we as humans can detect when one event goes over into the other [20]. The transition from eating to doing the dishes for example is a transition that you don't even think about, but is important to distinguish where the first ends and the second one starts: You wouldn't want to pour soap on your plate and start polishing it while you haven't finished your meal! Furthermore it requires no effort to determine that having dinner consists of many smaller actions, such as scooping food on your plate, pouring a drink in your glass and eating. Then you could subdivide eating in picking up your knife and fork, cutting a piece with your knife and picking it up with your fork, each of these actions can be subdivided into even smaller actions.
Finding structure in our actions may seem trivial to us, but is not trivial in machine learning. Machine learning methods do not know when having dinner stops and doing the dishes starts unless this structure is provided to them. Furthermore, for training machine learning models a large amount of labelled or structured data is needed. There exists a lot of data in the world that machines can potentially learn from, but most of this data is unlabelled or unstructured.
Creating labels and thus structure in data is expensive in both time and money, it is infeasible to let humans create labels in all existing data. A substantial amount of data thus goes unused or is less useful for machine learning methods. Particularly time-series trajectory data is difficult to structure, because of the large abstraction from reality and the high variety of information that this data can represent. Trajectory data consists of a series of x,y coordinates through time. These coordinates can represent any form: It could represent your stroll through the park, the movements of a

mouse in a cage, the movements of your cursor on a website due to moving a different kind of mouse or even the movements of a hurricane through the ocean and across the coast. The bottom-line is that, any information that can be represented as a sequence of coordinates representing movement, is considered trajectory data. The large movement complexities of your stroll through the park are thus condensed into a series of coordinates only. Multiple elements of a moving object can be tracked to create a trajectory, e.g. the nose, tail and paws of a mouse all create a different trajectory. The trajectories of all these elements together form what is known as a multi-trajectory.
Why not just use other forms of data if information is so difficult to extract from trajectories? For one, a substantial amount of data only consists of trajectories such as GPS data, and for other forms of data creating structure is also not so easy [34][5][14]. Moreover, creating structure in trajectories could generalize to find structure in other types of data. Techniques are in development that can extract trajectories from different and more complex forms of information such as video [24], where problems of detecting structure are also still ongoing. Creating structure in trajectory data could thus generalize to create structure in many different domains.

Besides the high cost, human provided structure is inconsistent and subjective [22][29], whereas creating structure using an algorithm uses consistent, objective rules. From these rules we can reason about why it chose to place the change points a certain way, that strategy is the same throughout the data and between datasets.
There already exist many methods that can create structure in trajectory data, notably change point detection and trajectory segmentation methods [17][16][21][10][7][8][27][15][19][1]. These method identify where one pattern (e.g. eating) changes into another (e.g. doing the dishes), hereby creating segments of consisting of single patterns. The segments form a structure that is also known as a trajectory segmentation. Taking the example of your stroll through the park, one pattern could be you running, another could be you walking or sitting. The current methods that find segmentations often have one or more of a number of problems however, we identify the four main problems here:

1. Change point detection methods use cost functions which are too unspecific to be applied to trajectory data. They are unable to capture the complexities of different patterns in trajectories.

2. A set of great tools for comparing trajectories - trajectory similarity methods - have scarcely been used in trajectory segmentation.

3. Many methods use fixed-size windows of sub-trajectories in their cost functions, assuming that patterns within the trajectory are all of a

similar size. However, patterns can be of different lengths and a single window size will then never be able to capture them all properly.

4. There is no one definitive best solution in the literature that solves the general problem of trajectory segmentation.

We identify a gap for a simple and intuitive trajectory segmentation algorithm based on change point detection with trajectory similarity, which uses multiple window sizes to compare sub-trajectories.

1.1 The intuition of our method

Our method finds the timepoints of largest pattern change. This makes use of the idea that the more alike two patterns are, the more likely they are to be from the same process. Conversely, the more different two patterns are, the more likely they are to be from a different process. You can imagine that the pattern of movements you perform while running at different times are very similar, while the pattern of movements when you run compared to when you brush your teeth at any time are very different. If you were running for a while first and then suddenly stopped and started brushing your teeth, then the point where "running" stopped and "brushing teeth" started is the point of largest pattern change. This method identifies the timepoints where the largest pattern changes take place. It then takes the first number of timepoints with the largest pattern change as change points.

1.2 How it solves the problem

We want to find structure in trajectory data by finding a segmentation of the trajectory. We find a segmentation by finding the points of largest pattern change, and setting these timepoints as the change points. In order to find the points of largest pattern change, the method calculates the amount of pattern change in each timepoint.
To determine the amount of pattern change in a timepoint, the method looks at a pattern $before$ the timepoint and at a pattern $after$ the timepoint. The $before$ pattern is used to identify how unique the pattern is that occurred before the timepoint. The $after$ pattern is used to check for a change of pattern. The more unique the $before$ pattern and the more different the $after$ pattern, the larger the pattern change. The computation is performed using the trajectory similarity method Edit Distance With Real Penalty (ERP) [3] (section 4.1.1). When the pattern change score is calculated for each timepoint, the highest scoring timepoints are chosen as change points by the greedy best-first change point selection (section 4.2).

1.3 Research question

Having identified that generic trajectory segmentation needs improvement, we identify the following research question:

Can a trajectory segmentation be found using greedy best first change point detection based on trajectory similarity?

1.3.1 Contributions

The contributions of this paper are as follows:

1. Introduce a simple and intuitive general trajectory segmentation algorithm based on trajectory similarity (section 4.1), which follows a clear rule based decision strategy, designed for both single- and multi-trajectory data.

2. Introduce a method to fit to patterns in the data by using multiple window sizes, in an efficient way using a greedy best first approach (section 4.2), with suggestions for determining the best window size (section 6.2).

3. An extensive performance comparison of our method with two of the best change point detection methods: Pruned Exact Linear time (Pelt) and Binary segmentation (Binseg) (section 6).

The rest of this project is structured as follows: In the preliminaries (secton 2) some concepts are explained that are helpful for understanding our method and trajectory segmentation in general. In the related work (section 3) prominent methods are discussed in the field of change point detection, trajectory segmentation, trajectory similarity. A brief overview of deep learning in the context of change point detection and trajectory segmentation is also provided. In the methods (section 4) it is explained how our method determines the pattern change scores and selects change points using multiple window sizes. Information on an optional stop criterion is also provided. In the experiments (section 5) The characteristics of the different trials of the dataset and the features used are explained, and the metrics used are introduced. Furthermore, the setup of three experiments are described: Firstly, a change point placement comparison to GSBS [10], secondly an experiment to test the effect of different maximum window sizes, and lastly an extensive comparison of metric scores between our method, Pruned Exact Linear Time (Pelt) [15] and Binary Segmentation (Binseg) [27], on five trials of a mouse dataset. The results of these experiments are shown in section 6 and their implications are discussed in section 7. A suggestion for future improvements is provided in section 8. We then conclude in section 9 that

our method has the potential to be a high performance method when window size optimization is possible.

Chapter 2

Preliminaries

In this chapter, some concepts are briefly defined that frequently appear in this research project. This chapter serves both as a reference for readers experienced in trajectory segmentation as an introduction to some concepts. It must be noted that the purpose of this chapter is to make the rest of this research understandable for the reader, and not to give strict airtight mathematical definition that is correct in any context. The concepts are approached from a trajectory data perspective. A summation of different concepts is from here on provided.

A **trajectory** is a type of time-series data that consists of a series of x and y coordinates, ordered through time. This most commonly represents some sort of movement, where the x and y coordinates are different locations that the object was in during the movement.
A **timepoint** in the context of trajectories is one combination of an x and a y coordinate, that indicated a position at a specific time. If for example a trajectory consists of the coordinates (0,0), (1,1) and (2,2), then (0,0), (1,1) and (2,2) are all timepoints of the trajectory.
A **Multi-trajectory** is a trajectory that has more than one coordinate at each timepoint. These could represent movements of multiple objects moving together through time. An example could be the coordinates of the nose and the tail of a mouse walking through the garden.
A **pattern** in the context of trajectories is some sequence of x and y coordinates that forms a distinctly different shape or has some other different characteristic than the multiple surrounding x and y coordinates. Patterns can come in any shape and can be more or less different than other patterns. Any sequence of x and y coordinates can form a pattern.
A **change point** is some timepoint in a trajectory where a particularly distinct pattern stopped and another pattern started. This is thus the timepoint where a change in patterns takes place. In this research the change point is seen as the first point of the newly started pattern.

A **segmentation** is the set of patterns that is left when a trajectory is cut at the positions of the change points. The **segments** are separated by change points. A segmentation is thus a trajectory that is divided into multiple distinct patterns, then referred to as segments.

Chapter 3

Related Work

This chapter gives a brief overview of related and relevant research on change point detection (3.1), trajectory segmentation (3.2) and trajectory similarity (3.3). A brief overview of deep learning related to these fields is also provided (3.4).

3.1 Change point detection

Change point detection methods create a segmentation of time series data by finding the points of change, which are thus referred to as change points. There are many different ways that a change can be detected, here the most relevant methods to trajectory segmentation are discussed.
Binary segmentation (Binseg) [27] uses a greedy best first search strategy. In each iteration it calculates the value of each potential change point using the provided cost function. The best scoring timepoint is chosen as the first change point, which influences the score of the other timepoints in the next iteration. This two step process each iteration is repeated until a desired number of change points is reached. This method generally has high speed due to the fact that it approximates the best solution, but is slowed by the fact that scores have to be recomputed for every iteration. This is mostly a search strategy and can therefore use many different cost functions, resulting in high versatility. It is one of the best performing methods for unsupervised change point detection, as identified by the extensive change point detection comparison in [2].
Pruned Exact Linear Time (Pelt) [15] is a method that uses a linear penalty and a linear search strategy resulting in generally low computational time. The linear penalty function is used along with a pruning rule to remove timepoints form a list of potential change points. It passes through the timepoints in order and for each following timepoint it considers whether it should be the last change point or not. This method determines the number of change points based on the penalty provided and cannot find a

pre-determined number of change points and is thus most suitable for cases where the number of change points is unknown. Statistical measures can be used to determine the threshold and thereby the number of change points [2]. It is one of the best performing methods for unsupervised change point detection, as identified by the extensive change point detection comparison in [2]. Just like Binseg this method is more of a search strategy and can be used with many different cost functions. This method finds an exact best solution given the cost function, whereas our method and Binary segmentation find only an approximation. However, the exact solution is not always better as can be seen in the results section of this paper.
Window sliding segmentation (WSS) [2] is another method that approximates the best change points by computing the difference between two windows before and after a potential change point. This is different from our method because of the fact that it generates an error signal to determine the best change points instead of using a greedy best first strategy. It also has a fixed window size and no inter-comparison on the left window. This method has a low computational cost: it is linear in the number of input samples.
Greedy State Based Search's (GSBS) [10] uses a greedy best first strategy just like Binseg, with a correlation based cost function. It maximizes the directional similarity of the values in the same segment, where a segment is defined as all timepoints from one change point up to the next. It does this by selecting a change point in each iteration, that causes the largest increase of correlation for all timepoints with the mean of segment they are in. This method can be used to find both a pre-determined number of change points as well as determine the number of change points itself. It achieved state of the art performance in the domain of fMRI brain voxel change point detection [10]. The method is well-suited for change point or segmentation problems where each timepoint within a segment is highly similar, due to the combination of the greedy search strategy and mean-correlation based cost function. Due to the same reason, this method is less well suited for cases where within-segment timepoints are very different. In this second case the automatic change point detection is also less accurate. This method was the inspiration and starting point for the development of our method.

3.2 Trajectory segmentation

Most research in trajectory segmentation seems to be focused on GPS trajectories. [8][33][36][31][18] GPS trajectory problems can often be slightly simplified versions of a more general trajectory segmentation problem. If the goal is to find a segmentation based on the means of transport, then the property of speed can already be enough information to a create segmentation [36]. Clustering based methods are often suited for finding seg-

mentations where the similarity within segments is high (such as the speed in the previous example) [33], as these look for common properties between timepoints. Another type of approach are change point based methods, that create a segmentation by finding the change points. These have the possibility of creating a segmentation based on the differences between consecutive segments [7][21]. A downside is that these do not necessarily create clusters of segments that are similar.

When facing a segmentation problem where specific properties of sequences are apparent (such as the speed), it can be better to use a specialised approach for that problem [10][19][33]. General trajectory segmentation approaches are necessary however, in cases where specific properties are unknown or where it is undesired to spend too much time trying to understand the data. Such a case could be when segments are created as pre-processing for labelling data or when segments are used to train models directly, when manually annotated data is unavailable.

Sliding Window Segmentation (SWS) [7] (not to be confused with the change point detection method Window Sliding Segmentation (WSS)) is one such general trajectory segmentation algorithm, based on change point detection. Intuitively this method defines change points as the points that are least expected, given some window of timepoints before the point. This method uses a window before each timepoint t to predict the location of t. The predicted location is then compared to the actual location, creating an error score for t. The timepoints with an error score above a certain threshold are taken as the change points, thereby creating a segmentation. The predictions use a kernel interpolation system, where different kernels can be used depending on the domain of the trajectory. In their paper the method obtained the best results when compared to three other trajectory segmentation methods on three trajectory datasets of different domains.

Moving Pattern Change Detection (MPCD) [8] is a trajectory segmentation algorithm specifically designed for GPS trajectories. Intuitively this method creates a segmentation by finding points where the direction of the data changes. For each timepoint it creates a movement pattern consisting of speed and time interval features. For each movement pattern a Singular Value Decomposition (SVD) is performed and a number of components are identified as the subspace for the pattern. For each timepoint, the distance between the current subspace pattern and the previous subspace pattern is computed, which represents the degree of pattern variation at that timepoint. The best change points are selected by fitting a curve over the degree of pattern variation values and selecting the peaks of the curve. This method then automatically creates an annotation of the data using domain knowledge. In their paper the method obtained the best results when compared to four other trajectory segmentation methods on three trajectory GPS datasets.

3.3 Trajectory similarity

Trajectory similarity or trajectory distance methods are methods that can score the level of similarity of two trajectories. In trajectory segmentation they can be used as (part of) a cost function, such as ERP [3] is in our method. From a general segmentation perspective trajectory distance measures are seemingly very specific, but within the perspective of trajectory segmentation they are actually very broad: Many cost functions assume that similarity between timepoints and segments can be expressed in variations of certain parameters, such as the correlation in the cost function of GSBS. However, without some knowledge of the data you cannot assume these parameters to be the best representatives of change. This is where trajectory distance measures come in: The best measures can find similarity based on the patterns in the timepoints of each trajectory. Generally, there are two different properties that trajectory similarity methods make use of to determine similarity: The order of values and the shape of the trajectory [30]. If only the first property is used on two trajectories, $T1$ and $T2$ that are of the same shape, but of different length with length of $T1$ 10x that of $T2$, then they will be very dissimilar. When only using the second property, $T1$ and $T2$ will be very similar, but this doesn't take into account time based element of the trajectory. Depending on the problem it can be better to exploit one property over the other or use a combination of both.
An important property of trajectory similarity methods for our purpose is the fact whether the method is metric or non-metric. A method that is metric has as most important property that the distance scores of different methods can be compared to each other, where if $dist(T1, T2) > dist(T2, T3)$, then $T1$ and $T2$ are more similar to each other than $T2$ and $T3$. This is important to make sure in our method that a larger pattern change score actually means a larger change occurred. For a mathematical definition see [30].
Dynamic Time Warping (DTW) is a time-series distance measure of origin [9]. This is one of the most popular trajectory segmentation methods and also one of the oldest [9][25][30]. This computes the distance between trajectories $T1$ and $T2$ by matching the points of $T1$ to the points closest in value in $T2$. It does this by going through the trajectories in order. A timepoint t of $T1$ cannot match to a timepoint earlier in $T2$ than the the match of timepoint $t-1$. Multiple timepoints of $T1$ can match to the same timepoint in $T2$. This allows DTW to detect similarities in shape between trajectories even if they are on different timescales. This also allows it to be robust against trajectory resizing as identified by [30] in their comprehensive trajectory distance measure comparison. The total distance of DTW is the sum of the absolute distance between the values of all matched timepoints.
Spatiotemporal Linear Combine Distance (STLC) [28] computes the difference between two trajectories by combining the spatial distance and the

temporal distance between the trajectories. Given two trajectories $T1$ and $T2$, The spatial distance of timepoint t with value x_t of $T1$ is the L2-norm of the timepoint in $T2$ that is closest to x_t. The temporal distance of timepoint t is the distance to the closest time position in $T2$, meaning that distance is only above 0 if $T1$ and $T2$ are of different length. The total distance is the sum of these two computations for all timepoints.
Because the computational result is different depending on if $T1$ or $T2$ is taken as the first trajectory, the spatial and temporal distance are computed twice for all timepoints, once with $T1$ first and once with $T2$ first. The result is combined into a single distance score, with a weight determining the importance of the spatial distance. STLC was also found to be robust against trajectory resizing and to noise [30].
Both DTW and STLC are non-metric, which could have a negative effect on their use inside a cost-function. However, there is no proof that they do not work and in a short, informal test with DTW in our method it achieved performance close to our similarity metric of choice, ERP.

3.4 Deep Learning in Change Point Detection and Trajectory Segmentation

Deep learning has been applied in situations related to trajectory segmentation, most often in the context of trajectory classification. Commonly, deep learning is only a part of the algorithm, such as a classification step [4] [12] [6] or feature extraction step [12] [6] [35]. The segmentation itself always seems to be handled by non-deep learning methods in the limited research that we came across while researching trajectory segmentation.
The method created by [35] is a segmentation based GPS-trajectory anomaly detection method, that uses a three step process to detect an anomaly. In the first step the trajectory is segmented into many small segments of similar direction. Four features (velocity, acceleration, distance change and angle change) are extracted from each segment. In the second step an autoencoder is used to create a single feature vector for each segment from these four features. Segments are then clustered using DBSCAN and the most different segments are detected as anomalies.
Dabiri [4] uses a CNN to classify segments of a trajectory after training on examples of segments. The trajectory is first transformed into features (speed, acceleration, jerk, and bearing rate) and then fed into the CNN, which is trained using back-propagation and gradient descent.
Both Hatami [12] and Endo [6] create images of time-series data to feed into deep neural networks. Here [6] is specific for trajectories. The deep architecture is used in both to automatically extract features from the time-series and is then used to classify segments after a training phase where labelled segments are provided.

In all feature extraction with deep architectures described above, it was found that the deep features improved classification accuracy over traditional hand-crafted features.

Chapter 4

Methods

In this chapter The details of the inner workings of our algorithm are described. Section 4.1 describes how the amount of pattern change is calculated with the cost function. Section 4.1.1 describes the trajectory similarity method used in the cost function. Section 4.2 describes how change points are selected using the Greedy best-first strategy and Section 4.3 describes how the optional stop criterion of the method works. A short overview is provided first.

Our method identifies the timepoints of largest pattern change. The intuition behind this is that any two patterns that are very similar are likely to come from the same process and any two patterns that are very different are likely to be from different processes. The timepoints where the largest pattern changes occur are thus most likely to be change points. Remember the example from the introduction: Imagine that you are in a hurry to get to work and run to your sink to brush your teeth, after which you run to your bike. The points where you stopped running and started brushing your teeth and where you stopped brushing and started running represent the points of largest pattern changes, because the pattern of movements performed while running is very different from the pattern of brushing your teeth. This is congruent with the desired result as running and brushing your teeth are indeed very different processes. To find the right patterns to compare, the amount of pattern change is computed for multiple window sizes. The idea is that the window best fitting a pattern yields the highest pattern change score, allowing the best pattern to be found.
The method operates in two steps: First the pattern change score is calculated for each timepoint and each window size 4.1, then the best change points are selected based on the pattern change scores 4.2. Both of these steps operate slightly differently for single- and mult-trajectory data. A stopping critertion 4.3 can optionally be used to automatically determine the number of change points based on a threshold parameter.

4.1 Step 1: Scoring Pattern Change

In this step the amount of pattern change for each timepoint and window size is computed. The pseudo-code is shown in *Algorithm* 1. This Algorithm calculates the pattern change for a single trajectory. To calculate the pattern change of multi-trajectory data, *Algorithm* 2 is used. For both algorithms, first the pseudo-code is displayed, followed by a brief explanation of the algorithm.

Algorithm 1 Calculate Pattern Change Score

```
1: function patternChangeScore(n_timepoints, max_win_size, T):
2:     min_fit = array[n_timepoints × max_win_size]
3:     max_fit = array[n_timepoints × max_win_size]
4:     fit_values = array[n_timepoints × max_win_size]
5:
6:     for each timepoint t in Trajectory T do:
7:         for window_size ws in (2 to max_win_size) do:
8:             before_window = T[t − ws...t]
9:             left_before = T[t − ws...t − (ws/2)]
10:            right_before = T[t − (ws/2)...t]
11:            after_window = T[t...t + ws]
12:
13:            min_fit[t, ws] = trajectoryDistance(left_before, right_before)
14:            max_fit[t, ws] = trajectoryDistance(before_window, after_window)
15:        end for
16:    end for
17:
18:    min_fit = −min_fit + max(min_fit)
19:    fit_values = min_fit + max_fit
20:
21:    return fit_values
22: end function
```

The amount of pattern change in any timepoint t is determined by the combined value of two trajectory distance computations, calculated for each window size:

1. A window of timepoints (sub-trajectory) before t ($before_window$) is compared to a window of timepoints after t ($after_window$). The more different these two windows are, the better the score at t. Meaning the larger the trajectory distance between the two windows, the more likely it is that t is a change point.

2. The left half of the $before_window$ ($left_window$) is compared to the right half ($right_window$). The more similar these windows are, the

better the score at t. Meaning the smaller the distance between the two windows, the more likely it is that t is a change point.

The trajectory distance computation is explained below in 4.1.1.
A schematic example of the windows is provided in figure 4.1.

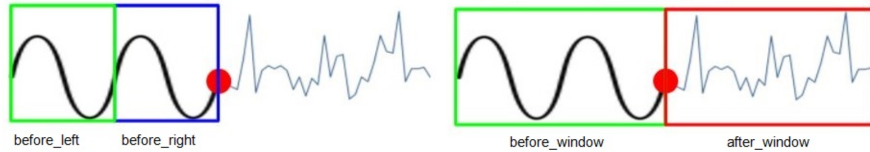

Figure 4.1: Given a trajectory with two different patterns, the change point (in red) is identified by computing the pattern change score: Comparing the $left_before$ and the $right_before$ windows to each other (shown on the left). and the $before_window$ and the $after_window$ to each other (shown on the right).

For each timepoint, the computations 1 and 2 are performed for each window size up to max_win_size. The idea is that the window size that best represents a pattern will yield the best pattern change score. Computing multiple window sizes will then allow patterns of different size to be found. Figure 4.2 shows a schematic overview of how the window size might change to eventually fit to a pattern.

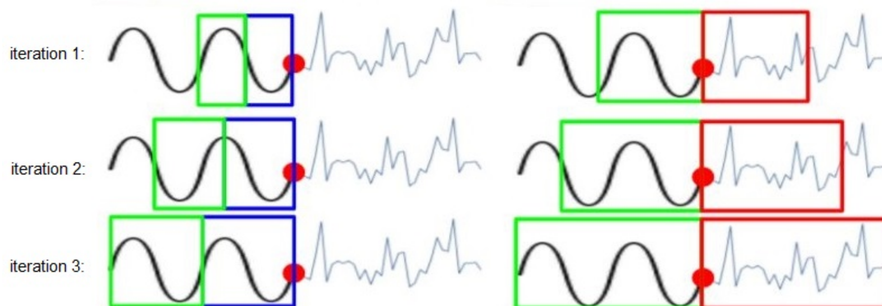

Figure 4.2: Given a trajectory with two different patterns, the change point(in red) is identified by computing the pattern change score for multiple window sizes. For each window size the same steps are performed: Comparing the $left_before$ and the $right_before$ windows to each other (shown on the left), and the $before_window$ and the $after_window$ to each other (shown on the right). In this example the window size in iteration 3 best fits the pattern and yields the best overall pattern score, leading to the change point being identified.

To perform a trajectory distance computation, the windows representing the sub-trajectory must have at least two timepoints, this means that

both the $before_window$ and the $after_window$ need to be at least size 2. The maximum size of these windows is either max_win_size or the maximum distance to either the start ($before_window$) or the end ($after_window$) of the trajectory, whichever is smaller.
To store the values of the trajectory distance computations, two matrices, min_fit and max_fit are used. Max_fit stores the result of computation 1 and min_fit stores the result of computation 2. For both matrices, each row represents a timepoint at that index, and each column represents a window size of that timepoint.
When for all timepoints, all window sizes are computed, min_fit is transformed to reverse the order of values, such that the previous smallest value is now the largest. Afterwards, min_fit and max_fit are combined into fit_values, which represents the pattern change score of each timepoint and window size. Because of the transformation of min_fit, the scores of fit_values from largest to smallest represent the most pattern change to the least pattern change. The fit_values matrix is used to select the change points in the Greedy Change Point Selection step.

To calculate pattern change scores in multi-trajectory data, *Algorithm* 2 is used. The pseudo code is shown below.

Algorithm 2 Calculate Pattern Change Score Multi-Trajectory

```
function PATTERNCHANGESCOREMULTI(n_timepoints, max_win_size, all_T):
    if all_T is uneven then:
        zero_pad = zeros_array[n_timepoints]
        all_T.concatenate(zero_pad)
    end if

    all_fit_values = array[all_T/2 ×n_timepoints × max_win_size]

    for i in (0 to all_T.size) with stepSize=2 do:
        trajectory = all_T[i...i + 1, :]
        all_fit_values[i/2] = patternChangeScore(n_timepoints, max_win_size, trajectory)
    end for

    return all_fit_values
end function
```

This algorithm calculates the fit_values for each trajectory in the multi-trajectory data separately and stores each in the all_fit_values matrix. While this algorithm is designed for trajectory data, it can accept any time series data consisting of an even number of dimensions in the feature dimension, since trajectory distance measures can only compute with 2D data. To ensure that an even number of dimensions is provided, any data with an uneven number of features receives an additional dimension of zeros.

4.1.1 Trajectory Similarity

Trajectory similarity methods, also known as trajectory distance methods, take any two trajectories and return a single score, representing the distance between the two trajectories. The closer to 0 the result is, the more similar the two trajectories are, the farther from 0, the less similar. There exist many trajectory similarity metrics, many of which are extensively discussed in [23][30]. This method operates with any trajectory similarity metric, but for the purposes of this paper we decided on Edit distance with Real Penalty ERP [3]. This method was chosen because it is found to be robust against noise and trajectory stretching or suppressing [30], which is important to make sure that trajectories with similar shapes but with slight permutations are indeed detected as similar.
ERP is also metric, which has the important property that if trajectories a and b have a larger distance score than a and c, then a and c are more similar than a and b.
ERP was implemented using the implementation by Guillouet [11].

ERP uses the edit distance to find the timepoints in two trajectories that are closest to each other (match points) and then computes the L1-norm distance for all match points. Each timepoint can only have a single match point. In cases where $before_window$ is uneven, it cannot be split evenly and two computations are performed. One where $before_left$ is longer and one where $before_right$ is longer. For example if $before_window$ was five timepoints, then in the first computation $before_left$ is 3 timepoints and $before_right$ is 2 timepoints and in the second computation this $before_left$ is 2 and $before_right$ is 3. The trajectory distance is then set as the mean of these two computations. Such an uneven computation means one timepoint has no match point, in which case it is compared to the mean timepoint value of the two trajectories. The total distance between the trajectories is the total L1-norm, where a larger distance means the trajectories are less similar.
The total L1-norm being the distance between trajectories causes the slight problem of longer trajectories having a larger distance score, as these have more timepoints for which the L1 can be computed. Conversely, shorter trajectories generally have a lower distance score. The effects of either of these properties are minimized in our cost function, where the pattern change score is a combination of two different computations: One where minimizing the distance (computation 2) is better and one where maximizing the distance (computation 1) is better. In general our method tends to have higher scores for larger windows, but this is not necessarily a problem as will be discussed in the next section.

4.2 Step 2: Greedy Change Point Selection

In this step the change points are selected based on the pattern change scores in the *fit_values* matrix. How the change points are selected differs slightly for single- and multi-trajectories. The single trajectory solution is shown in *Algorithm* 3 and the multi-trajectory solution is shown in *Algorithm* 4.

Algorithm 3 Calculate Pattern Change Score Single-Trajectory

```
function greedyChangePointSelection(fit_values):
    n_timepoints = fit_values.length
    n_cp = 1
    best_cp = array[n_timepoints]
    best_scores = array[n_timepoints]

    while n_cp < max_cp do:
        // Store best change point score and position
        cp = argmax(fit_values)[0]
        best_cp[cp]= n_cp
        best_scores[cp] = max(fit_values)

        // Prune overlapping windows
        fit_values[cp, :] = -inf

        n_points_right = min(max_win_size - cp, max_win_size)
        for i in (0 to n_points_right) do:
            fit_values[cp + i, i:] = -inf
            n_points_left = min(cp, max_win_size)
        end for
        for i in (0 to n_points_left) do:
            fit_values[cp - i, i:] = -inf
        end for

        n_cp += 1
    end while

    return best_cp, best_scores
end function
```

In each iteration, the greedy change point selection chooses the timepoint with the largest value in *fit_values* as the next change point. Timepoints that are close to each other have similar windows and tend to have similar scores. To make sure the algorithm doesn't choose timepoints right next to each other on each iteration, the window sizes that overlap with the chosen timepoint are no longer allowed as an option. Unless there is a really strong

change within a close timepoint of the change point, it makes more sense to have the change point represent the change in that small area, since the true location of the change point is not known. A schematic overview of the windows being pruned from the fit_values matrix when a change point is selected is shown in figure 4.3. The small bias for larger windows in combination with decreasing window sizes in each iteration ensures that larger patterns are more likely to be detected in earlier iterations and smaller patterns are more likely to be detected in later iterations.
The process of selecting change points and pruning neighbouring windows is repeated until a pre-defined number of change points is reached or a stopping criterion (section 4.3) is reached.

1	w1	w2	w3	w4
2	w1	w2	w3	w4
3	w1	w2	w3	w4
4	w1	w2	w3	w4
5	w1	w2	w3	w4
6	w1	w2	w3	w4
7	w1	w2	w3	w4

Figure 4.3: The rows are the timepoints. The left most column indicates the number of the timepoint, the other columns are the window sizes for the timepoint in that row (These are the columns of the fit_values matrix). A change point is placed on timepoint 3 (green colour). The red colours in the left most column indicate for which timepoint the window sizes will be adjusted. The red colours in the other columns indicate which window sizes are pruned for each timepoint. The white window sizes are unaffected.

To determine the change points in multi-trajectory data, *Algorithm* 4 is used. The pseudo code is shown below.

Algorithm 4 Calculate Pattern Change Score Multi-Trajectory

```
function greedyChangePointSelectionMulti(all_fit_values):
    all_fit_values = normSumTo1(all_fit_values)
    n_timepoints = all_fit_values[1].length
    n_cp = 1
    best_cp = array[n_timepoints]
    best_scores = array[n_timepoints]

    while n_cp < max_cp do:
        // Store best change point score and position
        cp = bestWindowEachFit(all_fit_values)
        best_cp[cp]= n_cp
        best_scores[cp] = max(fit_values)

        // Prune overlapping windows
        all_fit_values[:, cp, :] = -inf

        n_points_right = min(max_win_size - cp, max_win_size)
        for i in (0 to n_points_right) do:
            all_fit_values[:, cp + i, i:] = -inf
            n_points_left = min(cp, max_win_size)
        end for
        for i in (0 to n_points_left) do:
            all_fit_values[:, cp - i, i:] = -inf
        end for

        n_cp += 1
    end while

    return best_cp, best_scores
end function
```

The intuition of the multi-trajectory change point selection is to select the best window per trajectory for each timepoint. In each iteration, a timepoints' score is set as the sum of the highest window scores of each trajectory at that timepoint. Since multiple trajectories may have slightly different lengths for a pattern (you can imagine the tail of a mouse that moves slightly out of sync with its nose), there is likely to be no one window size that can perfectly capture a pattern between multiple trajectories. Therefore selecting the highest score window of each trajectory allows the best fitting window to be selected for each. To ensure equal weight between the trajectories, each trajectories' fit_values matrix is set to sum to one. The pattern change score of each timepoint and window is then a percentage

of the total score found.
The rest of the process is equal to the single-trajectory algorithm.

4.3 Stop Criterion

Besides choosing a number of change points to be found the method can find a number of change points based on a threshold. When this is used, all change points are selected in the change point selection step until a maximum number is found, after which the best number is selected using the threshold. The threshold is set as a value between 0 and 1, where 1 represents the total amount of pattern change that was found by the method in selecting all change points. The total amount of pattern change here is thus the sum of values of all change points. The values of the change points are normalized to sum to 1. The threshold value is thus a percentage of the total amount of pattern change. When the threshold is set, change points are selected until the desired percentage of pattern change is reached.
For this method the maximum number of change points (or a pre-defined maximum) always needs to be found to determine the best number of change points. The computational time of the selection is negligible however compared to computational time required for the pattern change score computation.

Chapter 5

Experiments

In this chapter the setup and parameters for three different experiments are described. The first section is a description of the dataset 5.1, features 5.2 and metrics 5.3 used. The second section describes the setup of the experiments 5.4. The first experiment compares the choices of change points of our method to the method that inspired our method (GSBS) [10] 5.4.1. The second experiment looks at the effect of maximum window size on change point placement of our method 5.4.2. The third experiment is an extensive comparison between our method, Binseg [27] and Pelt [15]. Here many different configurations are tested on 6 partitions of the dataset of a mouse in a cage, after which the best configurations of each method are tested further on five trials of a mice dataset 5.4.3.

5.1 Dataset

In these experiments a dataset consisting of five trials of trajectories of mice are used. Each of the trials consists of the trajectories of three keypoints of a mouse, extracted from a top-down view video of the mouse's behaviour in a cage. The keypoints were found using automatic tracking of the Nose, Tail-base and Center of Gravity of the mouse using EthoVision® XT from Noldus [26]. For each keypoint there are approximately 15000 (slightly different per trial) x,y-coordinates, sampled with a frequency of 25 frames per second, making the entire trial 600 seconds long. The data was gathered with different mice but always in the same cage. Each of the experiments described below uses these trials or a subset of Mice_trajectories_8, which is the main trial. Further elaboration is provided in the corresponding sections. The dataset was provided and annotated by Noldus [26]. The annotations are based on all three keypoints. Their characteristics are presented in table 5.1.

Trial	**Number of timepoints**	**Nr of datapoints**	**Number of change points**	**Average sequence size**	**Number of different behaviours**	**Number of null datapoints**
Mice_trajectories_8	15000	90000	320	46.875	17	0
Mice_trajectories_1	14976	89856	580	25.821	17	15624
Mice_trajectories_2	15296	91776	388	39.423	17	49662
Mice_trajectories_3	1500	90000	164	91.46	18	1548
Mice_trajectories_4	15000	90000	51	294.118	8	0

Figure 5.1: Characteristics of the trials in the dataset used in the experiments.

The null entries of trials 1,2 and 3 are replaced by the mean of their none-null values.

5.2 Feature sets

In the experiments below three different feature sets are used. A feature set can be seen as a pre-processing of the data to change its form. In this experiment features are extracted for each keypoint and put together as a single vector, in the order that the keypoints are in in the data. Every row is a feature and every column is a timepoint.
The form of the input can have a big influence on the segmentation performance of a method. The most basic form is simply the x and y coordinates of the data, and this is also the first feature set (abbreviation of coordinates is coords). A problem with pure coordinates is that this takes into account the position of the moving object, while the different processes may not be location dependent. Take for example the mice in the dataset described above, they can perform any type of behaviour in any location in their cage.
The most common feature set in the literature is the combination of speed, acceleration, jerk and turn angle [4][12]. Because of its frequent use and proven success, this is the second feature set. Definitions are provided below.
The third feature set is known as forward-sideward. This is consists of two variables per timepoint: the first expresses how much the object moved forward, the second how much the object moved sideward. This can be seen as a form of the coordinates, without the locational information and it thus alleviates the problem of positioning playing too large of a role in determining the change points.

5.2.1 Feature extraction

Here follows a description of each feature and the formal definition.
Take two coordinates ($x1$,$y1$) and ($x2$, $y2$), where ($x1$,$y1$) comes directly before ($x2$, $y2$):

1. *Speed* (abbreviation *sp*) $\rightarrow$ This is the absolute value of the rate of change of location. It consists of an xSpeed and a ySpeed. xSpeed is the absolute value of ($x2$ - $x1$). *ySpeed* is the absolute value of ($y2$ - $y1$).

2. *Acceleration* (abbreviation *acc*) → This is the rate of change of *speed*. It consists of an xAcceleration and a yAcceleration. xAcceleration is the speed of $x2$ - the speed of $x1$. yAcceleration is the speed of $y2$ - the *speed* of $y1$.

3. *Jerk* → This is the rate of change of acceleration. It consists of an xJerk and a yJerk. xJerk is the speed of $x2$ - the speed of $x1$. yJerk is the speed of $y2$ - the speed of $y1$.

4. *turnAngle* (abbreviation *turn_a*) → This is how much a timepoints' angle has changed compared to the previous timepoint. This is defined as the difference between the direction of the velocity vectors of two timepoints.
   The *velocity* consists of xVelocity and yVelocity. xVelocity is $x2$ - $x1$. yVelocity is $y2$ - $y1$. The *velocityangle* is $arctan2(yVelocity, xVelocity)$. The turn angle is the next velocity angle - the current velocity angle. For every two dimensions in the data (every set of x and y coordinates) this results in a single dimension.

5. *forward-sideward* (abbreviation fw_sw) → This is how much the movement of a timepoint went forward and how much it went sideward compared to the previous timepoint.
   $forward$ is how much the point went forward compared to the previous point. This is the dot_displacement / norm_speed. norm_speed is matrixNorm(speed($x1$, $y1$)). The dot_placement consists of the speed of $x2$ * the speed of $x1$ + the speed of $y2$ * the speed of $y1$.
   sideward is how much the point went sideward compared to the previous point. This is the matrixNorm(speed($x2$, $y2$) - v_fw). v_fw consists of $forward$ * speed($x1$, $y1$) / norm_speed.

5.3 Metrics

We report two metrics, The covering metric and f1 metric. These are used to compare the performance of our method to Binary Segmentation and Pelt. These metrics are commonly used to compare change point detection methods, notably [2] used them in their change point detection survey. These metrics take as input the predicted change points as provided by an algorithm and the true change points, and as output they provide a score between 0 and 1. 1 Indicates that the algorithm perfectly identifies the change points, 0 indicates the change points are not identified at all.

5.3.1 Covering metric

The covering metric calculates its score based on the segments in between the change points. For all true sequences x the subsequence y for which

the Jaccard score of x and y is the highest is used. The Jaccard score is multiplied by the length of x. Given all these scores, the covering metric is their sum divided by the number of change points. The Jaccard score is computed as the intersection of x and y divided by the union of x and y. The implementation is based on the one in [2].

5.3.2 F1 metric

The f1 metric calculates its score based on the location of the change points. This is a combination of both precision (the fraction of correctly identified change points divided by the number of detected change points) and recall (the fraction of correctly identified change points divided by the true number of change points).
Change point predictions that are close to true change points but not exactly in the same position are seen as false in this setup, but a close prediction shouldn't be incorrect to the same degree that a far of prediction is. The true change points can also be different depending on who annotated the dataset. To accommodate the uncertainty of the true change point, a margin of error is thus introduced. The margin indicates how far off a prediction is allowed to be, to be seen as correct. In this paper we use two versions of the f1 metric, one with a margin of error of 5 and one with a margin of error of 1. The implementation was taken from [2], which provides further elaboration on the methods.

5.4 Model Parameters

5.4.1 Experiment 1: Comparison to GSBS

As a demonstration of the performance of our method compared to GSBS for trajectory data, the change point placement is compared on a small segment of Mice_trajectories_1. This is a segment of 200 timepoints, with 13 true change points. It is free of null values. For the purpose of clearly showing the decision differences between the methods, only a single keypoint (Center of Gravity) is used. This can cause some change points to be harder to detect, but choices regarding change point placement are easier to reason about. As feature set speed, acceleration, jerk and turn angle are used, as this set was experimentally found to give the best results for GSBS in a pre-test. Both methods gave two predictions with a different number of change points: One equal to the true number of change points, one equal to twice the true number of change points. The second prediction is intended to see if the methods keep making similar decisions and whether they come closer to the true change points or not when they continue making predictions. The results are shown in section 6.1.

5.4.2 Experiment 2: Effect of Maximum Window Size

In this experiment the effect of different maximum window sizes are compared on: the mean of the covering metric and f1 metric (margin=5) (further referred to as meanCovF1) and on the runtime. The effect on the covering metric, the f1 metric (margin=5) and the f1 metric (margin=1) is shown in appendix A.1. 10 different window sizes [4, 8, 12, 20, 30, 40, 50, 60, 80] are used on six different forms of the data (Nose as single keypoint with three feature sets and All three keypoints with three feature sets): [nose+fw_sw, nose+sp,acc,jerk,turn_a, nose+coords, three_kp+fw_sw, three_kp, acc,jerk,turn_a, three_kp+coords]. The setup was the same as the experiment described hereafter, see that and the schema 5.2 below for further details. The results are shown in 6.2.

5.4.3 Experiment 3: Comparison to Binseg and Pelt

In this test the change point placement of our method is compared to two of the best change point methods: Binseg and Pelt. Binseg and Pelt are general versions of state of the art change point detection methods as identified by [2]. These are change point selection methods which can use many different cost functions for valuing change points. The methods were implemented using the Ruptures library [32].

The first comparison is performed on six partitions of Mice_trajectories_8 and then on the other four trials described in 5.1. Our method and Binary Segmentation get the true number of change points as goal, Pelt is optimized to approximate the true number of change points. An orderly overview of the test setup is provided below 5.2, here follows only a brief explanation of some of the choices shown below.

Mice_trajectories_8 is split in partitions to avoid strong effects of one part of the data having too big of an influence on the overall score. On the flip side, to avoid one partition having too large an influence on the score, from each method the best parameter combination (as determined by the highest average of the covering metric and f1, margin=5 metric) is tested again on the full trial and on four other trials. The scores from the partitions are taken together and averaged to show the final results.

Different number of keypoints are used with different features to find the best combination for each method. As single keypoint the Nose is used, as this is has the most variation between behaviours. For our method, the turn angle feature is padded with a row of zeros, such that the even number of features requirement for the trajectory similarity metric is met. For Pelt and binary segmentation this row is not added. The window sizes are chosen to be about the shortest sequence size at smallest and to be about twice the average sequence size at largest (of Mice_trajectories_8). Maximum window sizes larger than 80 start getting computationally heavy but will not neces-

sarily yield better results as many sub-sequences will start to blend into the same window. Besides that, it is expected that the distinctness of a pattern will be detectable in window sizes of a maximum of 80.
The cost functions for binary Segmentation and Pelt are chosen based on the results of a pre-test. This pre-test was performed on 2000 timepoints of Mice_trajectories_1, this subset has 123 change points and is free of null values. The best three across these input variations were chosen out of all cost functions from the rupture change point library [32]. These approximately represent the best results that can be expected by using each of the cost functions. Furthermore, the other cost functions were left out with an eye on computational time limits.
Pelt cannot find a pre-determined number of change points, but uses a penalty instead. The best penalty for Pelt was determined using an optimization method which adjusts the penalty in the direction of of the true number of change points. The best penalty is approximated by multiplying the penalty by the (predicted number of change points / true number of change points). When the chosen penalty overshoots the true number of change points, the step size of the penalty is reduced. This method does not guarantee that the true number of change points is found, as Pelt doesn't always find every possible number of change points, but it does provide an approximation. Since Pelt is a popular and high performing method, we wanted to include it in the comparison to give some sense of comparative performance.

```
For each partition in the dataset [6 partitions of 2500 timepoints]
    For each nr of keypoints in [Nose, [Center of gravity, Nose, Tail-base]]
        For each feature set in [Coordinates, forward-sideward, [Speed, Acceleration, Jerk, Turn angle]]
            1. Test our method:
                for each window size in [4, 8, 12, 20, 30, 40, 50, 60, 80]
                    compute change points our method
                    compute metric scores

            2. Test Binary Segmentation and Pelt
                for each cost function in [Mahalanobis, l2, rbf]
                    compute change points Binary Segmentation
                    compute metric scores Binary Segmentation

                    while Pelt penalty is unoptimized and fewer than 30 penalties are tested:
                        Adjust Pelt Penalty
                        compute change points Pelt
                    compute metric scores optimized Pelt
```

Figure 5.2: Pseudo-code showing the setup to test multiple parameter configurations for our method, Pelt and Binary Segmentation.

Chapter 6

Results

Here the results of the three experiments described in the experiments (section 5) are shown: Firstly the comparison of change point choices between our method and GSBS [10] (section 6.1). Secondly the effect of the maximum window size on the change point performance and runtime (section 6.2). Thirdly the comparison of our method with Pelt and Binary Segmentation (section 6.3).

6.1 Comparison to GSBS

Here the results of comparing change point placement between GSBS and our method are presented.

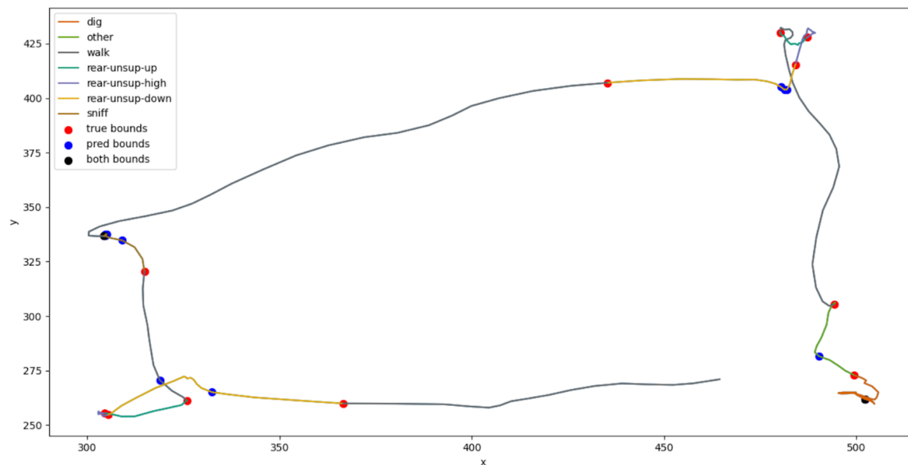

Figure 6.1: **GSBS**: Change point placement of **gsbs** as viewed on the trajectory, the number of change points to find is set equal to the true number of change points. The red bounds represent the true change points, the blue bounds represent the change points predicted by **gsbs** The black bounds represent an exactly accurate prediction.

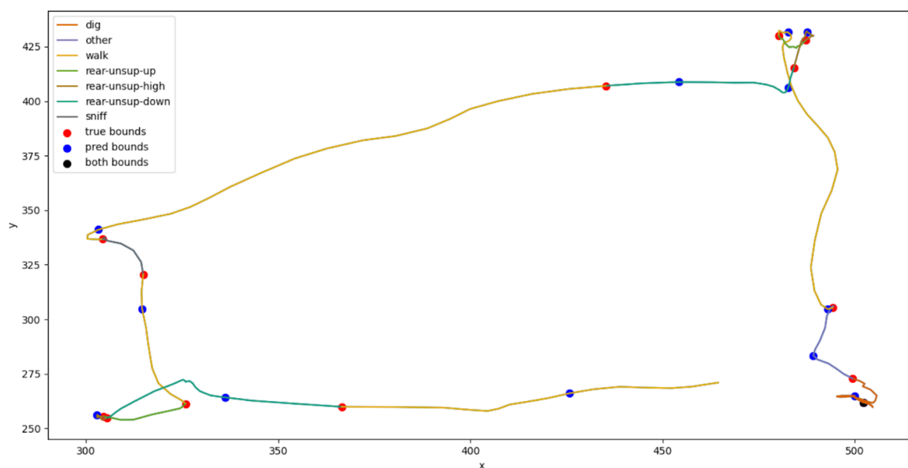

Figure 6.2: **Our method**: Change point placement of **our method**, the number of change points to find is set equal to the true number of change points. The red bounds represent the true change points, the blue bounds represent the change points predicted by **our method** The black bounds represent an exactly accurate prediction.

Here can be seen that the change point placement of our method is closer to the true change point placement.

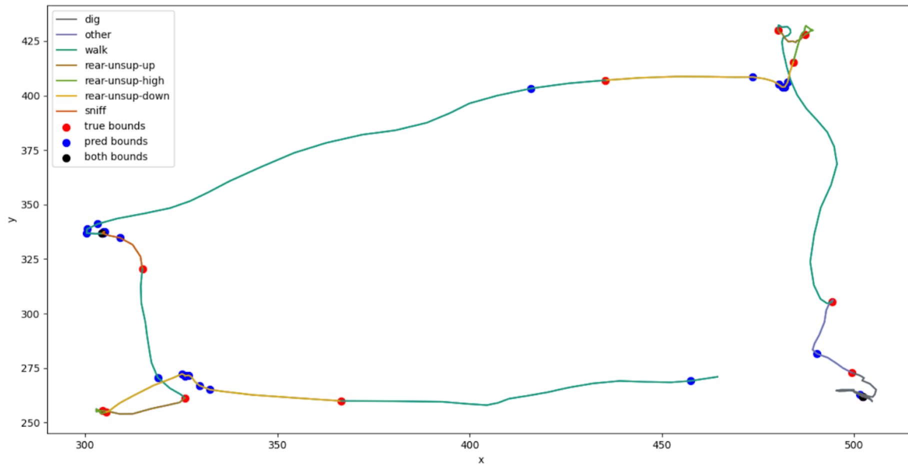

Figure 6.3: **GSBS**: Change point placement of **gsbs** as viewed on the trajectory, the number of change points to find is set equal to the twice the true number of change points. The red bounds represent the true change points, the blue bounds represent the change points predicted by **gsbs** The black bounds represent an exact accurate prediction.

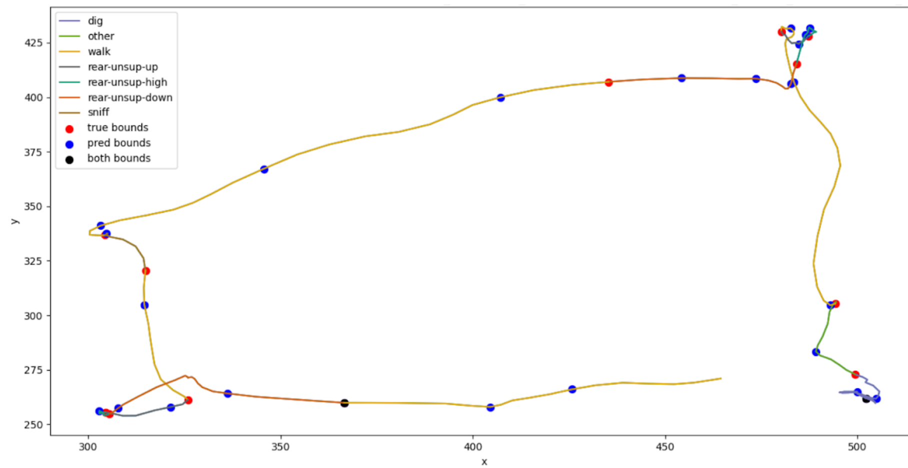

Figure 6.4: **Our method**: Change point placement of **our method** as viewed on the trajectory, the number of change points to find is set equal to the twice the true number of change points. The red bounds represent the true change points, the blue bounds represent the change points predicted by **our method** The black bounds represent an exact accurate prediction.

Here can be seen that gsbs doesn't come much closer to finding true change points for the most part. Our method comes closer than gsbs.

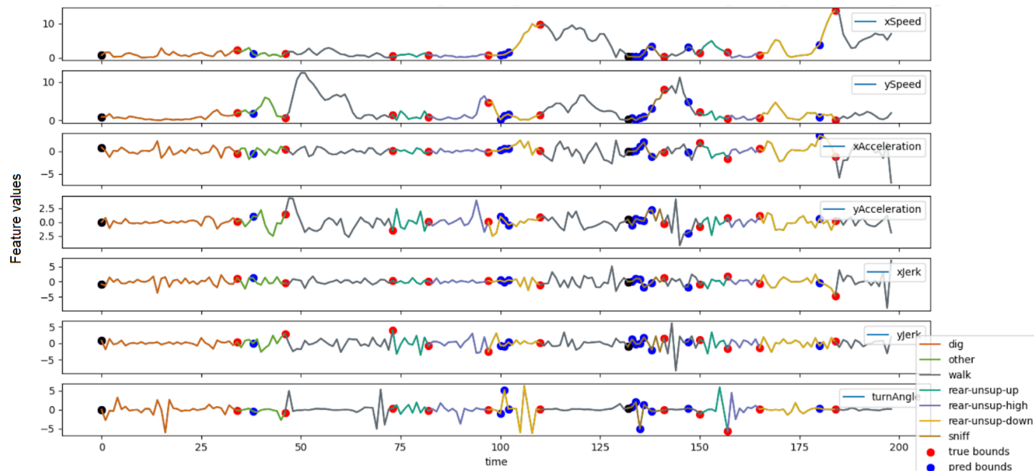

Figure 6.5: **GSBS**: Change point placement of **gsbs** as viewed on the feature values through time, the number of change points to find is set equal to the true number of change points. The red bounds represent the true change points, the blue bounds represent the change points predicted by **gsbs** The black bounds represent an exact accurate prediction.

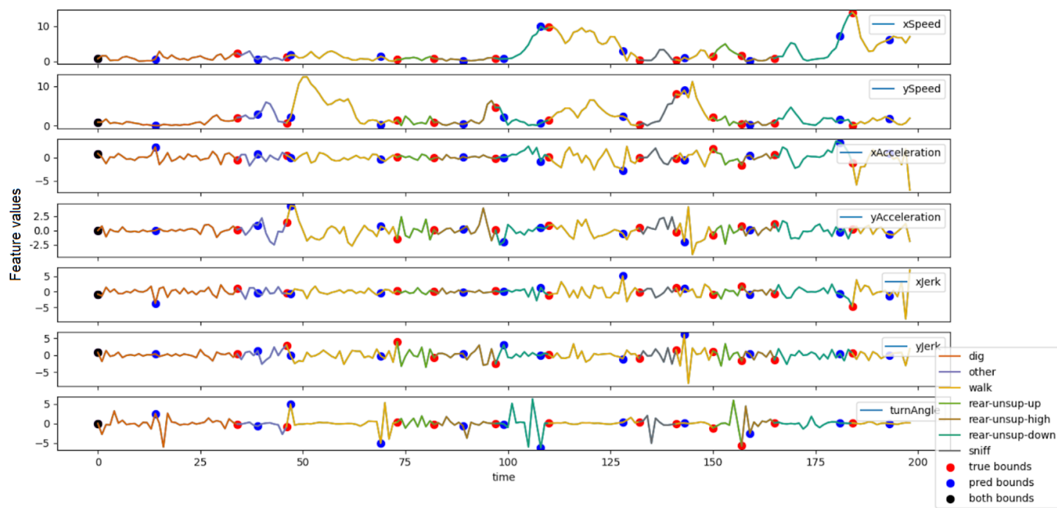

Figure 6.6: **Our method**: Change point placement of **our method** as viewed on the feature values through time, the number of change points to find is set equal to the true number of change points. The red bounds represent the true change points, the blue bounds represent the change points predicted by **our method** The black bounds represent an exact accurate prediction.

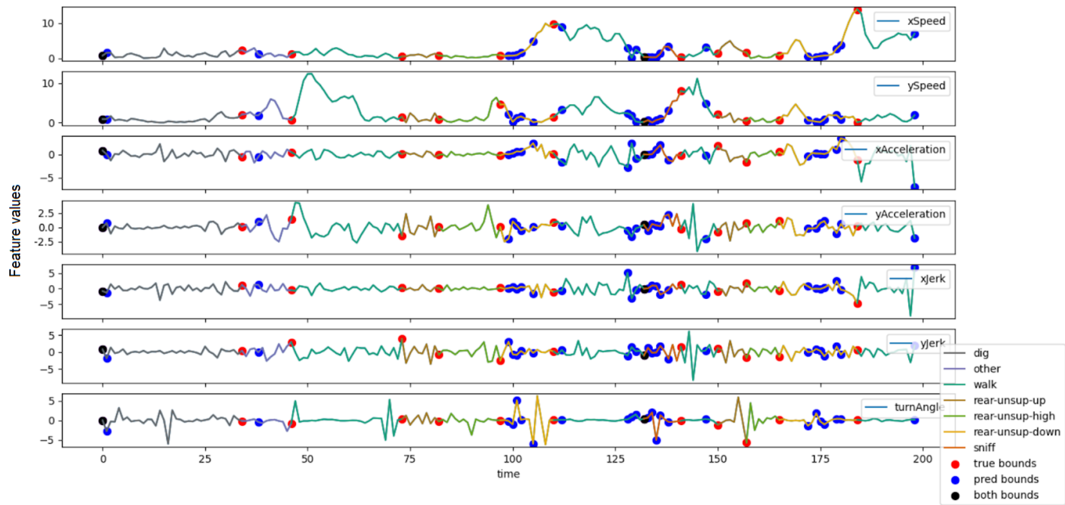

Figure 6.7: **GSBS**: Change point placement of **gsbs** as viewed on the feature values through time, the number of change points to find is set equal to twice the true number of change points. The red bounds represent the true change points, the blue bounds represent the change points predicted by **gsbs** The black bounds represent an exact accurate prediction.

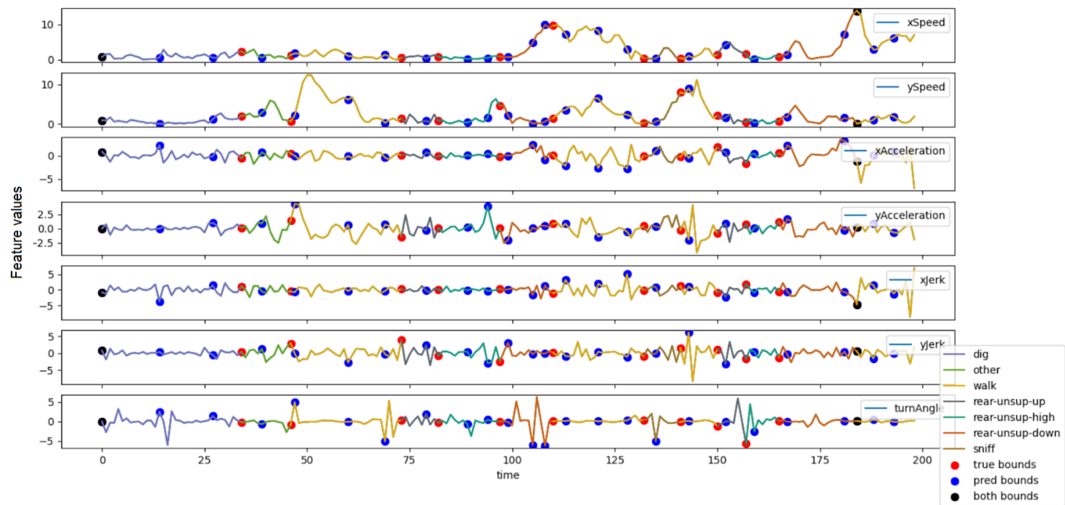

Figure 6.8: **Our method**: Change point placement of **our method** as viewed on the feature values through time, the number of change points to find is set equal to twice the true number of change points. The red bounds represent the true change points, the blue bounds represent the change points predicted by **our method** The black bounds represent an exact accurate prediction.

6.2 Effect of the Maximum Window Size

Here the effect of the maximum window size on the meanCovF1, and the runtime are presented. The effect of the maximum window size on the covering metric, f1 (margin=5) and f1 (margin=1) can be seen in the appendix A.1.

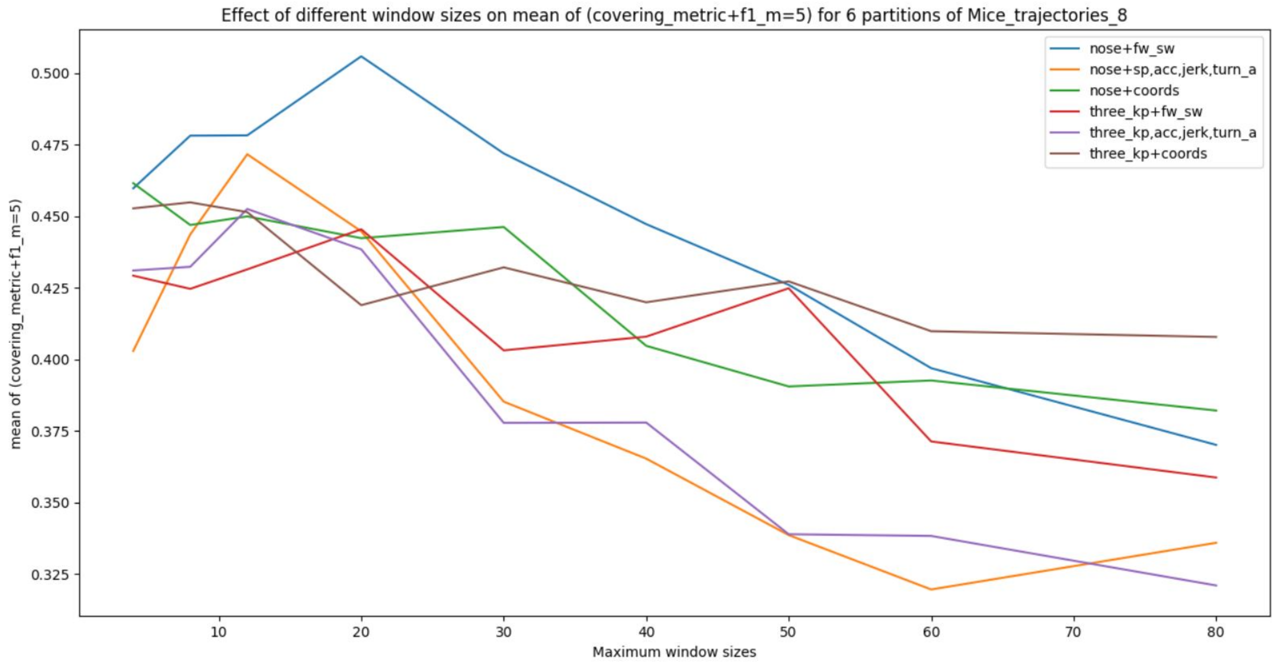

Figure 6.9: The effect of window size on the mean of the covering metric and f1 metric (margin=5) of our method, with five different input data forms.

The mean of the covering metric and f1 (margin=5) generally decreases for larger window sizes. Coordinates show the most stable results for all window sizes. speed,acc,jerk and turn_angle show the strongest decrease in performance with larger window size. Window sizes around 10-30 achieve the highest results for each method.

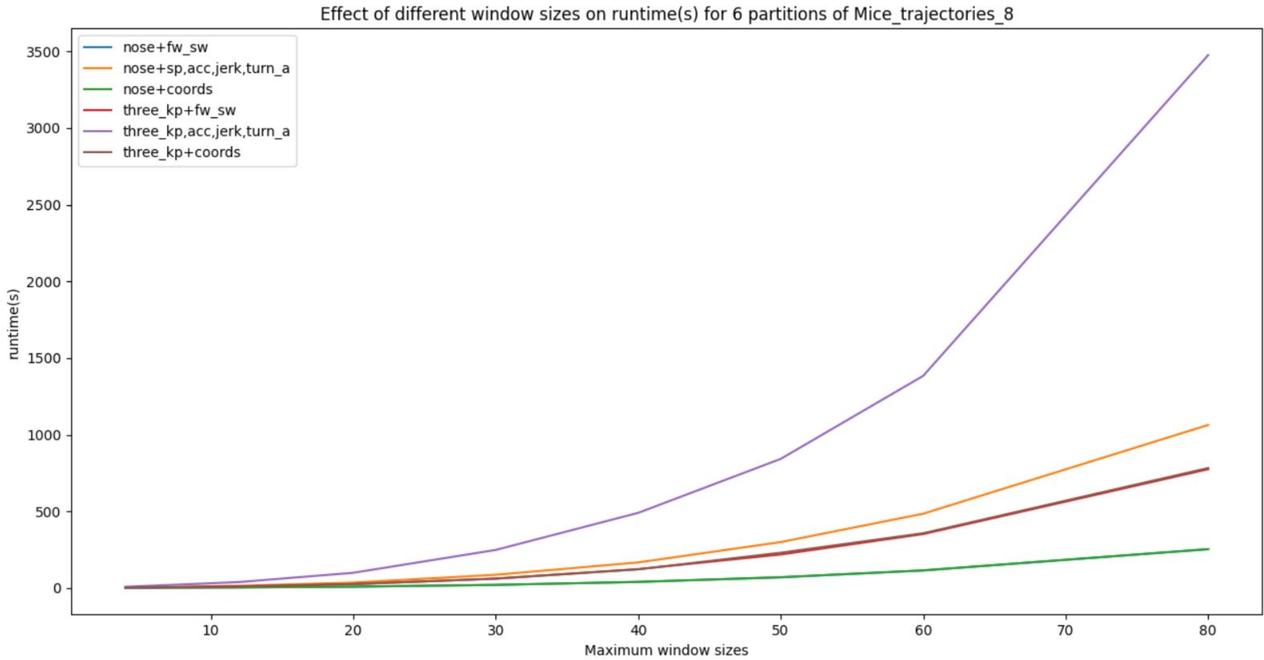

Figure 6.10: The effect of window size on the runtime of our method, with five different input data forms.

The runtime shows a general increase with window sizes and number of keypoint + number of features.

6.3 Comparison to Binseg and Pelt

Here the results of the comparison to the best change point methods are presented, the setup of the experiment can be found in the experimental setup 5.4.3. figure 6.11 shows the results of the best three versions of each method on the six partitions of Mice_trajectories_8, ordered by smallest covF1Mean first, in ascending order. For a full list of the results of all parameter combinations tested, see figure A.4 in the appendix. Figure 6.12 shows the best result of each method obtained out of all configurations. Figure 6.13 shows the results of the best configurations from Figure A.4 when tested on the five full mouse trials.

Method	Features	Keypoints	Mean covering and f1,margin=5	Covering metric	f1, margin=5	f1, margin=1	runtime (s)
Pelt_mahalanobis,penalty_optimized	coordinates	nose	0.4191	0.4711	0.3672	0.1082	469.0358
Pelt_l2,penalty_optimized	coordinates	Three_keypoints	0.42	0.4662	0.3738	0.1177	381.6666
Binseg_mahalanobis	coordinates	nose	0.4223	0.4562	0.3884	0.1136	26.3645
Binseg_mahalanobis	fw_sw	Three_keypoints	0.4227	0.4901	0.3553	0.1502	32.7007
Pelt_l2,penalty_optimized	coordinates	nose	0.4227	0.4721	0.3733	0.1001	245.552
Binseg_l2	fw_sw	Three_keypoints	0.431	0.5026	0.3595	0.1362	2.0984
Our_method,max_win_size=8	fw_sw	nose	0.4781	0.5799	0.3763	0.1401	**1.803**
Our_method,max_win_size=12	fw_sw	nose	0.4782	0.5742	0.3822	0.1517	3.3134
Our_method,max_win_size=20	fw_sw	nose	**0.5058**	**0.5948**	**0.4169**	**0.1531**	8.191

Figure 6.11: Best three parameter configurations our method, Pelt and Binary Segmentation, based on average best results, ordered by smallest meanCovF1 first. These are the average results on six partitions of each 2500 timepoints of Mice_trajectories_8. The runtime of Pelt is influenced by its optimization strategy: the penalty is optimized for which it is run anywhere between 1 and 30 times.

Method	Mean covering and f1,margin=5	Covering metric	f1, margin=5	f1, margin=1
Pelt,penalty_optimized	0.4227	0.5376	0.3738	0.1624
Binseg	0.431	0.5406	0.3884	0.1772
Our_method	**0.5058**	**0.5948**	**0.4353**	**0.1845**

Figure 6.12: Best average result of each metric for each method, out of all configurations tested on six partitions of 2500 timepoints each of mice_trajectories_8.

The best methods based on the meanCovF1 are tested on the full trial (Mice_trajectories_8]). Furthermore they are also tested on four other mouse trials of similar size (Mice_trajectories_1, Mice_trajectories_2, Mice_trajectories_3, Mice_trajectories_4). For further details see the experimental setup (figure 5.1).

Mice_trajectories_8

Method	Features	Keypoints	Mean covering and f1,margin=5	Covering metric	f1, margin=5	f1, margin=1	runtime (s)
Our_method,max_win_size=20	fw_sw	nose	**0.4592**	**0.5626**	0.3557	0.1342	98.7916
Binseg_l2	fw_sw	Three_keypoints	0.4226	0.5076	0.3375	**0.1344**	**71.2813**
Pelt_l2, penalty_optimized	coordinates	nose	0.4022	0.4449	**0.3594**	0.0969	4043.5953

Mice_trajectories_1

Method	Features	Keypoints	Mean covering and f1,margin=5	Covering metric	f1, margin=5	f1, margin=1	runtime (s)
Our_method,max_win_size=20	fw_sw	nose	0.5283	0.5243	0.5323	0.1792	**44.32**
Binseg_l2	fw_sw	Three_keypoints	**0.5329**	**0.5296**	**0.5362**	**0.2172**	47.2354
Pelt_l2, penalty_optimized	coordinates	nose	0.5035	0.5122	0.4948	0.1552	7382.4838

Mice_trajectories_2

Method	Features	Keypoints	Mean covering and f1,margin=5	Covering metric	f1, margin=5	f1, margin=1	runtime (s)
Our_method,max_win_size=20	fw_sw	nose	0.5539	0.6059	0.5019	0.139	42.7936
Binseg_l2	fw_sw	Three_keypoints	0.4568	0.4651	0.4485	**0.2010**	**40.7546**
Pelt_l2, penalty_optimized	coordinates	nose	**0.5674**	**0.6218**	**0.5129**	0.1392	43291.5394

Mice_trajectories_3

Method	Features	Keypoints	Mean covering and f1,margin=5	Covering metric	f1, margin=5	f1, margin=1	runtime (s)
Our_method,max_win_size=20	fw_sw	nose	0.3119	0.3623	0.2614	0.0912	87.8677
Binseg_l2	fw_sw	Three_keypoints	0.3414	0.4450	0.2378	0.1037	**81.2173**
Pelt_l2, penalty_optimized	coordinates	nose	**0.4924**	**0.6493**	**0.3354**	**0.1159**	21593.0182

Mice_trajectories_4

Method	Features	Keypoints	Mean covering and f1,margin=5	Covering metric	f1, margin=5	f1, margin=1	runtime (s)
Our_method,max_win_size=20	fw_sw	nose	0.2449	0.4121	0.0777	0.0583	92.2903
Binseg_l2	fw_sw	Three_keypoints	0.1638	0.2687	0.0588	0.0196	**58.6152**
Pelt_l2, penalty_optimized	coordinates	nose	**0.4862**	**0.7567**	**0.2157**	**0.0784**	9223.8104

Figure 6.13: Results of best configurations of our method, Pelt and Binary Segmentation tested on a dataset of five mouse trials. Mice_trajectories_8 is also the trial that the configurations are based on. Mice_trajectories_1, 2 and 3 contain null values, see experimental setup for further information (section 5.4.3).

Chapter 7

Discussion

In this chapter the results shown in chapter 6 are discussed. The results are discussed in the order that they appear in, first the change point choice comparison to GSBS (section 7.1), then the effect of the maximum window size (section 7.2) and lastly the score comparisons to Pelt and Binseg (section 7.3).

7.1 Comparison to GSBS

In figures 6.1 and 6.2 it can be seen that the choices made by our method are closer to the actual change points than GSBS. Looking at figures 6.3 and 6.4 we see that our method keeps making good decisions when placing more change points than there are true change points. The choices approach change points that were left unfound in earlier iterations. We see that GSBS doesn't improve its choices for the most part.
The poor choices of GSBS are likely caused by its correlation-based cost function, in combination with the greedy best first strategy. With this combination GSBS maximizes the directional similarity of timepoints within the same segment in each iteration. Since in trajectory data there are many directional changes, also within a segment, this approach does not necessarily lead to detection of the desired segments. The feature plots of GSBS figures 6.5 and 6.7 show that gsbs places many change points very close to existing change points, on strong directional differences. This is because change points of segments of few timepoints in the same direction lead to a higher increase in directional similarity than change points of segments that include different directions. Even though the latter may eventually improve total directional similarity the most, the short term gain of change points close to previous change points is always preferred by GSBS.
In our method, timepoints close to each other generally also have similar windows and thus similar scores. As can be seen in the figures 6.2 and 6.4 our method doesn't often place change points very close to each other. This

is likely because after each change point is placed, the windows of nearby timepoints are decreased to avoid overlap with the change point. This eliminates the most similar windows to those of the change point and thus makes surrounding timepoints less desirable, allowing other change points to be selected earlier. Looking at the placements of the change points in the feature plots of our method (figures 6.6 and 6.8), we see that they are generally placed between segments with different patterns, showing that the trajectory similarity-based cost function indeed detects pattern changes.
These results suggest that trajectory similarity cost functions can work well to find segments in trajectory segmentation. The combination with decreasing windows sizes in further iterations is a good strategy to avoid change points being cluttered around a few, high score positions.
A possible downside of the decreasing window sizes is that change points that are actually close to other change points could be less likely to be detected in early iterations. This is because, in our method, longer sequences tend to have higher scores: If many window sizes are much larger than the small window of an actual change point, then the change points' pattern change on the small window could be comparatively insignificant. Another potential problem this can bring is that incorrect placement in an early iteration can lead to more and more inaccurate placements, as each change point influences the next potential change points.

7.2 Effect of Maximum Window Size

In this section we discuss the effects that the maximum window size can have on metric scores (and thus indirectly change point placement) (section 7.2.1) and how this effect is different based on the pre-processing (features and number of keypoints) performed on the data. We also discuss the effect on the runtime 7.2.2.

7.2.1 Metric Scores

The meanCovF1 graph 6.9 shows a general downward trend in metric score with an increased maximum window size. The best scores for all pre-processing forms are achieved in the 10-30 size range. The best results are achieved by nose+fw_sw, but this performance quickly declines with an increase in window size. The speed, acc, jerk and turn_angle features show good performance in the 10-20 range, but show a quick decline thereafter. The other data forms show a fairly stable result with a smaller decline in performance for larger maximum window sizes.
The decline in performance with larger maximum window size suggests that, at least on this dataset trial, shorter windows are preferred. In this trial the 10-30 range is also the range where a substantial portion of the larger patterns can fit inside a window, without too many smaller patterns fitting

inside a single window, thereby possibly making these window sizes the most suitable for this data. For this trial this maximum window size range could thus be a good trade-off between detecting shorter and longer patterns.
Several factors might influence the performance. One factor is the number of keypoints and features. When there is only a single keypoint and a single feature, the change points are selected in the exact order of highest pattern scores of the feature. If the pattern change scores in the feature align well with the actual change points of the data, then this feature can give very accurate results. If the scores do not align well, the performance will be poor. By using multiple features with multiple keypoints, the influence of a single feature on the change point score is decreased, both for well aligned and poorly aligned features. Multiple features and keypoints can thus average out results, making a features' good scores worse, but a features' bad scores better. This could explain the relatively stable results for the three_kp+fw_sw and three_kp+coords feature sets. In the case where there are a large number of features with very different scores at different timepoints, then the good and bad scores of different features will together average to a more neutral score, such that there are fewer timepoints that clearly represent change points.
Another factor that can influence performance are the properties of the features themselves. The high scores for forward-sideward with the nose keypoint suggest that this feature aligns well with the actual change point for a certain maximum window size (10-30), but less well for larger windows. In combination with the previous factor this explains the score curve of this data form. The score curve of the feature set consisting of speed, acc, jerk and turn_angle can similarly be explained. The feature scores fit well with the data for a certain maximum window size range (10-20), but fit less well with larger window sizes. The strong decline in score could be explained in combination with the previous factor: For large window sizes the features have less accurate pattern change scores and when combined together with all the less accurate scores of the other features, inaccuracies pile up and there are few timepoints that clearly represent the best change points.
It must be noted that the trajectory similarity method used in the cost function, ERP, is designed for coordinates. This could possibly cause it the perform less well for features that are very different from coordinates.
As also discussed in the previous section, the general bias of our algorithm to give higher scores to larger window sizes can cause less accurate change point placement when maximum window sizes become very large, because small patterns may not be detected properly anymore. This may be a factor resulting in the overall downward trend of the scores for increasing window size.

7.2.2 Runtime

In the runtime graph it can be seen that a larger maximum window size always results in a larger runtime. It also shows that in general, the more features and keypoints there are, the larger the runtime time.
The increase in runtime for larger maximum window size can be explained in the following way: With a maximum window size of $x + 1$, there is one extra, larger window computed per timepoint than for maximum window size x. The number of computations thus increases with an increasing maximum window size, it can then also be expected that the runtime increases.
A larger runtime for data with more features and keypoints can also logically be explained: For data with a number of dimensions equal to $n_keypoints \times n_features \times 2$, ERP has to be run $n_keypoints \times n_features \times max_win_size$ times. The same dimensions should then also get about the same runtime. This is true for both three_kp+fw_sw compared with three_kp+coords and nose+fw_sw compared with nose+coords: It is nearly indistinguishable in this graph, but in both cases the coordinates had a very slightly larger runtime, making the graphs nearly identical. The small difference here could be caused by either random processing differences or coordinates having larger numbers which could result in a higher computational time.

7.3 Best Methods Comparison

In this section the results of computing the metric scores of our method, Pelt and Binary Segmentation are discussed. First results on the partitions of the primary dataset, Mice_trajectories_8 are discussed (section 7.3.1) (a complete table of results is shown in the appendix A.4), then the results of the the best method configurations on the full dataset of five trials is discussed (section 7.3.2).

7.3.1 Main trial partitions

From the tests on the partitions (figures 6.11 and 6.12) it can be seen that our method outperforms Pelt and Biseg in many of its configurations. Notably the best three configurations of our method perform better on all three metrics. Also looking at the best scores of each method, Our method outperforms the others, while binary segmentation and Pelt are more on par. Looking at the appendix, it can be seen that a large number of the configurations of our method have better performance than all configurations of the other methods.
The use of multiple partitions can result in different scores than on a full dataset. The number of change points per partition is different and the distribution and length of behaviours is also slightly different. However, the argument is that this way the most robust configuration can be selected that

can then be used to test on the full dataset of five trials.

It should be noted that during the optimization phase Pelt with rbf was sometimes unable to obtain a correct number of change points. This means Pelt cannot find the correct number of change points here and thus has a lower score, but it also isn't a fair comparison. It is also possible that there was a mistake in the optimization code which caused it to be unable to find the specific number of change points. The result has been included for sake of completeness and can be found in the appendix A.4.
The runtime of Pelt is larger than is to be expected from a single run, as the runtime is an accumulation of the optimization phase, where Pelt is run anywhere between 1 and 30 times .

7.3.2 Full dataset

This part of the discussion is based on the results shown in figure 6.13. Looking at the results for Mice_trajectories_8, we see that our method still has the best performance. This is to be expected as this was also the data that the configurations were selected on. Even so, all of the methods showed a decline in performance, with our method showing the most performance decrease and Binary Segmentation the least. In Mice_trajectories_1 our method also performs the best, but is very closely tied to Binary Segmentation. The best performance for our method is in Mice_trajectories_2. Nevertheless it is slightly outperformed by Pelt. Looking at the results of the other two trials, it can be seen that the performance of both our method and Binseg is decreased compared to the other trials.
The performance decrease on Mice_trajectories_8 could be because some partitions of the data have timepoints that the methods more strongly see as change points (larger pattern change for our method): These now get selected instead of other change points that were originally in another partition.
Looking at the differences between the trials, it can be seen that our method achieves lower performance in the trials where the average sequence size is further from that of Mice_trajectories_8, on which the maximum window size was based. This suggests the maximum window size can play a large role in the accuracy of change point predictions. This is in-line with the result shown in figure 6.2, where only a limited range of window sizes had the best results. This is especially true for the forward-sideward features, which were used in this configuration. It may therefore be possible to achieve significantly better scores by optimizing the window size for each trial, which was not done in this experiment due to time constraints.
The greedy best-first approach of our method may cause it to select other change points first over true change points in the case where only few change points need to be found. This is supported by the fact that Binseg also uses

a greedy best-first approach and also had decreased performance. The results from the comparison to GSBS suggest that our method may converge to find (close to) all change points when it can select change points beyond the true number of change points.
The null entries (which where replaced by the mean) seem to have little effect on the performance for all of the methods. Only Binary Segmentation has a possibly identifiable dip in performance shown in Mice_trajectories_2, but it is unsure if the null values are the cause of the performance dip.

The advantage of our method is that the maximum window size can be adjusted to fit well with the data, in which case it can often achieve (close to) the best results. The maximum window size that you can find is limited by the computational time available.
Yet this advantage is also a disadvantage, because Pelt and Binary Segmentation don't require a parameter to be optimized in order to give a result. Pelt seems the most robust and receives the most stable scores between different numbers of desired change points. Binary segmentation is less stable which may also in part be caused by its greedy best first strategy.

Overall, our method has good performance when the window size is adjusted to fit the data, but is less accurate when unoptimized. It generally has better performance than Binary Segmentation, but Pelt has better performance on average. It's downside however is that it always needs a penalty to be optimized. Optimizing this penalty can lead to an increase of 100 fold for computational time.

Chapter 8

Future work

In this chapter three different types of improvements are discussed for our method. Firstly some aspects of the method are suboptimal and leave space for improvement (section 8.1), secondly the method needs some more extensive testing on different trajectory data and needs to be compared to other trajectory segmentation methods (section 8.2). The last section discusses possible extensions of the method 8.3, this hopefully serves as inspiration for any potential reader.

8.1 Possible improvements

We identify two simple places for improvement: Using and testing different trajectory similarity methods and costs and also better balancing of min_fit and max_fit.

In this research only ERP has been used, but there exist many other trajectory similarity methods that could be used. One that seems promising is STLC [28] showed robustness in tests performed by [30]. The cost function of our method could also be extended to work with different cost metrics like L1 that was used for Pelt and Binary segmentation in the experiments. This way the method could become a versatile method for any type of time series segmentation problem.
Min_fit and max_fit are currently not balanced, as discussed in chapters 4 and 7. This potentially makes it so the difference between the windows before and after the potential change point is always more important in determining change points. This could cause larger windows to be more strongly preferred over shorter windows than is desired when trying to fit the best window to a pattern. We suspect a different kind of optimization would be needed to both minimize min_fit and maximize max_fit at the same time. We identify the use of pareto optimality as a potential strategy.

8.2 Trajectory testing with trajectory segmentation methods

In this research the method has only been tested on one type of (highly complex) trajectory data. To see that the method generalizes well to other types of trajectory data, it should be tested on other trajectory datasets. Further testing is also required to establish how well the method functions when optimized on large datasets with few change points (such as Mice_trajectories_4). Furthermore, in the experiments our method was only compared in performance to change point detection methods. To establish how good it really is as a trajectory segmentation algorithm, it should also be tested against state of the art trajectory segmentation algorithms.

8.3 Possible extensions

Here we identify two area's where the use of our method could be greatly amplified. One is by using Monte Carlo tree search (MCTS) to approximate the best order to select change points, the other is to use this method to create a hierarchy of the data to create more insight into the structure of the data.

8.3.1 Monte Carlo Tree Search

After the pattern change scores of all windows are computed there is a matrix where all these values are stored. Currently the highest score is selected as a change point first, then windows are pruned and this is repeated until a desired number of change points is found. This process doesn't guarantee that the best change points over the entire dataset are found. This leaves room to simulate this process many times with MCTS. MCTS is designed to figure out best solutions by simulating many choices and evaluating them. It is often used in (video)games with many options available per turn, For example, it was used by IBM to defeat the world's best chess player way back in 1997 [13]. In our method all potential change points are the options in each iteration and their value at that time can be used as values in MCTS. Given that change points are actually represented by the largest pattern change, this strategy will approach the best overall distribution of pattern change, without needing to compute every single possible order.

8.3.2 Hierarchy detection

The strategy of our method where largest pattern changes are detected first, after which windows are pruned, could be used to create a hierarchy of sequences and subsequences. Take a dataset of a mouse for example, it might

perform a behaviour like walking, but walking actually consists of smaller subsequences (e.g taking any two or any one step). If changes between sequences in general have a larger pattern change than subsequences, then the changes between the sequences are detected first. The order in which change points are detected then represents a hidden hierarchy. After some number of change points are detected it should start placing change points one layer of the hierarchy lower. Potentially this point where one layer transitions into the other can be detected, after which a hierarchy of multiple layers of pattern complexity can be created by just using the change points. This could then also be used to automatically predict a set of potential numbers of change points, out of which the best can be selected based on the data. We like to see this as a meta-change point problem; finding the number of changes in a number of change points.

Chapter 9

Conclusions

In this research we proposed a novel trajectory segmentation algorithm that can find segments by calculating the points of largest pattern change. The amount of pattern change is calculated by computing the trajectory similarity with ERP of two windows around each timepoint. Window sizes are adjusted to find the window that best fits each pattern. After all pattern change scores are calculated, change points are selected in a greedy-best first manner.

Different window sizes were tested on a multi-trajectory dataset of a mouse, with multiple feature sets. The results show that the best results can be obtained using few keypoints and the forward-sideward features. Good results can be obtained by optimizing the maximum window size. A window size unoptimized for the data can lead to poorer results. Using coordinates only or a few features allows for a similar score on multiple window sizes but is less likely to yield peak performance.

We compared the method on a mouse dataset with five trials to two of the best change point methods: Pelt and Binary Segmentation. The results show that when the window size is optimized, our method achieves the best results. The performance using the window size does not generalize well to other data where the number of change points is a lot smaller, and the average sequence size a lot larger. Here the results of our method declined and Pelt showed a better performance. Data with a slightly smaller average sequence size did show a good performance, however Pelt also had the best result here. It is unknown whether fitting on data with large sequence sizes will yield good results, and further testing is required here.

In conclusion, our method is the best performing method if the window size can be optimized for the data and is generally better than Binary Segmentation in other scenario's. If little is known about the data and the average sequence size, then Pelt gives better results. Overall, our method is on par with the best change point methods and has the potential to be a high performing method when window size optimization is possible.

Bibliography

[1] Samaneh Aminikhanghahi and Diane J. Cook, *A survey of methods for time series change point detection*, Knowledge and Information Systems **51** (2017), no. 2, 339–367 (en).

[2] Gerrit J. J. van den Burg and Christopher K. I. Williams, *An Evaluation of Change Point Detection Algorithms*, arXiv:2003.06222 [cs, stat] (2020) (en), arXiv: 2003.06222.

[3] Lei Chen, *On The Marriage of Lp-norms and Edit Distance*, international conference on Very large data bases **13** (2004), 12 (en).

[4] Sina Dabiri and Kevin Heaslip, *Inferring transportation modes from GPS trajectories using a convolutional neural network*, Transportation Research Part C: Emerging Technologies **86** (2018), 360–371 (en).

[5] Yang Du, Chunfeng Yuan, Bing Li, Lili Zhao, Yangxi Li, and Weiming Hu, *Interaction-Aware Spatio-Temporal Pyramid Attention Networks for Action Classification*, Computer Vision – ECCV 2018 (Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, eds.), vol. 11220, Springer International Publishing, Cham, 2018, Series Title: Lecture Notes in Computer Science, pp. 388–404 (en).

[6] Yuki Endo, Hiroyuki Toda, Kyosuke Nishida, and Akihisa Kawanobe, *Deep Feature Extraction from Trajectories for Transportation Mode Estimation*, (2016), 12 (en).

[7] Mohammad Etemad, Amilcar Soares, Elham Etemad, Jordan Rose, Luis Torgo, and Stan Matwin, *SWS: an unsupervised trajectory segmentation algorithm based on change detection with interpolation kernels*, GeoInformatica (2020) (en).

[8] Yuan Gao, Longfei Huang, Jun Feng, and Xin Wang, *Semantic trajectory segmentation based on change-point detection and ontology*, International Journal of Geographical Information Science **34** (2020), no. 12, 2361–2394 (en).

[9] Akram Gasmelseed and Nasrul Humaimi Mahmood, *STUDY OF HAND PREFERENCES ON SIGNATURE FOR RIGHT- HANDED AND LEFT-HANDED PEOPLES*, **1** (1963), no. 5, 6 (en).

[10] Linda Geerligs, Marcel van Gerven, and Umut Güçlü, *Detecting neural state transitions underlying event segmentation*, preprint, Neuroscience, May 2020.

[11] Brendan Guillouet, *bguillouet/traj-dist*, 2018.

[12] Nima Hatami, Yann Gavet, and Johan Debayle, *Classification of Time-Series Images Using Deep Convolutional Neural Networks*, arXiv:1710.00886 [cs] (2017) (en), arXiv: 1710.00886.

[13] _ IBM, *IBM100 - Deep Blue*, March 2012, Publisher: IBM Corporation.

[14] Longlong Jing and Yingli Tian, *Self-supervised Visual Feature Learning with Deep Neural Networks: A Survey*, arXiv:1902.06162 [cs] (2019) (en), arXiv: 1902.06162.

[15] R. Killick, P. Fearnhead, and I. A. Eckley, *Optimal Detection of Changepoints With a Linear Computational Cost*, Journal of the American Statistical Association **107** (2012), no. 500, 1590–1598 (en).

[16] Taesup Kim, Sungjin Ahn, and Yoshua Bengio, *Variational Temporal Abstraction*, arXiv:1910.00775 [cs, stat] (2019) (en), arXiv: 1910.00775.

[17] Jeremias Knoblauch and Theodoros Damoulas, *Spatio-temporal Bayesian On-line Changepoint Detection with Model Selection*, arXiv:1805.05383 [cs, stat] (2018) (en), arXiv: 1805.05383.

[18] Xiangjie Kong, Menglin Li, Kai Ma, Kaiqi Tian, Mengyuan Wang, Zhaolong Ning, and Feng Xia, *(PDF) Big Trajectory Data: A Survey of Applications and Services*, October 2018.

[19] Sanjay Krishnan, Animesh Garg, Sachin Patil, Colin Lea, Gregory Hager, Pieter Abbeel, and Ken Goldberg, *Transition state clustering: Unsupervised surgical trajectory segmentation for robot learning*, The International Journal of Robotics Research **36** (2017), no. 13-14, 1595–1618 (en).

[20] Christopher A. Kurby and Jeffrey M. Zacks, *Segmentation in the perception and memory of events*, Trends in Cognitive Sciences **12** (2008), no. 2, 72–79 (en).

[21] Qing Li, Kehui Yao, and Xinyu Zhang, *A change-point detection and clustering method in the recurrent-event context*, Journal of Statistical Computation and Simulation **90** (2020), no. 6, 1131–1149 (en).

[22] Timmy Ma and Natalia L. Komarova, *Object-Label-Order Effect When Learning From an Inconsistent Source*, Cognitive Science **43** (2019), no. 8 (en).

[23] Nehal Magdy, Mahmoud A. Sakr, Tamer Mostafa, and Khaled El-Bahnasy, *Review on trajectory similarity measures*, 2015 IEEE Seventh International Conference on Intelligent Computing and Information Systems (ICICIS) (Cairo, Abbassia, Egypt), IEEE, December 2015, pp. 613–619.

[24] Matthias Minderer, Chen Sun, Ruben Villegas, Forrester Cole, Kevin Murphy, and Honglak Lee, *Unsupervised Learning of Object Structure and Dynamics from Videos*, arXiv:1906.07889 [cs] (2020) (en), arXiv: 1906.07889.

[25] Meinard Müller, *Information Retrieval for Music and Motion*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007 (en).

[26] Information Technology Noldus, *Mice trajectory dataset*, November 2020.

[27] Ashish Sen and Muni S. Srivastava, *On Tests for Detecting Change in Mean*, The Annals of Statistics **3** (1975), no. 1.

[28] Shuo Shang, Lisi Chen, Zhewei Wei, Christian S. Jensen, Kai Zheng, and Panos Kalnis, *Trajectory similarity join in spatial networks*, Proceedings of the VLDB Endowment **10** (2017), no. 11, 1178–1189 (en).

[29] Derek Sleeman, Laura Moss, Andy Aiken, Martin Hughes, John Kinsella, and Malcolm Sim, *Detecting and resolving inconsistencies between domain experts' different perspectives on (classification) tasks*, Artificial Intelligence in Medicine **55** (2012), no. 2, 71–86 (en).

[30] Han Su, Shuncheng Liu, Bolong Zheng, Xiaofang Zhou, and Kai Zheng, *A survey of trajectory distance measures and performance evaluation*, The VLDB Journal **29** (2020), no. 1, 3–32 (en).

[31] Han Su, Kai Zheng, Kai Zeng, Jiamin Huang, Shazia Sadiq, Nicholas Jing Yuan, and Xiaofang Zhou, *Making sense of trajectory data: A partition-and-summarization approach*, 2015 IEEE 31st International Conference on Data Engineering (Seoul, South Korea), IEEE, April 2015, pp. 963–974 (en).

[32] Charles Truong, Laurent Oudre, and Nicolas Vayatis, *Selective review of offline change point detection methods*, Signal Processing **167** (2020), 107299 (en).

[33] Di Wang, Tomio Miwa, and Takayuki Morikawa, *Big Trajectory Data Mining: A Survey of Methods, Applications, and Services*, Sensors **20** (2020), no. 16, 4571 (en).

[34] Xuanhan Wang, Lianli Gao, Peng Wang, Xiaoshuai Sun, and Xianglong Liu, *Two-Stream 3-D convNet Fusion for Action Recognition in Videos With Arbitrary Size and Length*, IEEE Transactions on Multimedia **20** (2018), no. 3, 634–644 (en).

[35] Zhongqiu Wang, Guan Yuan, Haoran Pei, Yanmei Zhang, and Xiao Liu, *Unsupervised learning trajectory anomaly detection algorithm based on deep representation*, International Journal of Distributed Sensor Networks **16** (2020), no. 12, 155014772097150 (en).

[36] Yu Zheng, *Trajectory Data Mining: An Overview*, ACM Transactions on Intelligent Systems and Technology **6** (2015), no. 3, 1–41 (en).

Appendix A

Appendix

A.1 Effect of Maximum Window Size

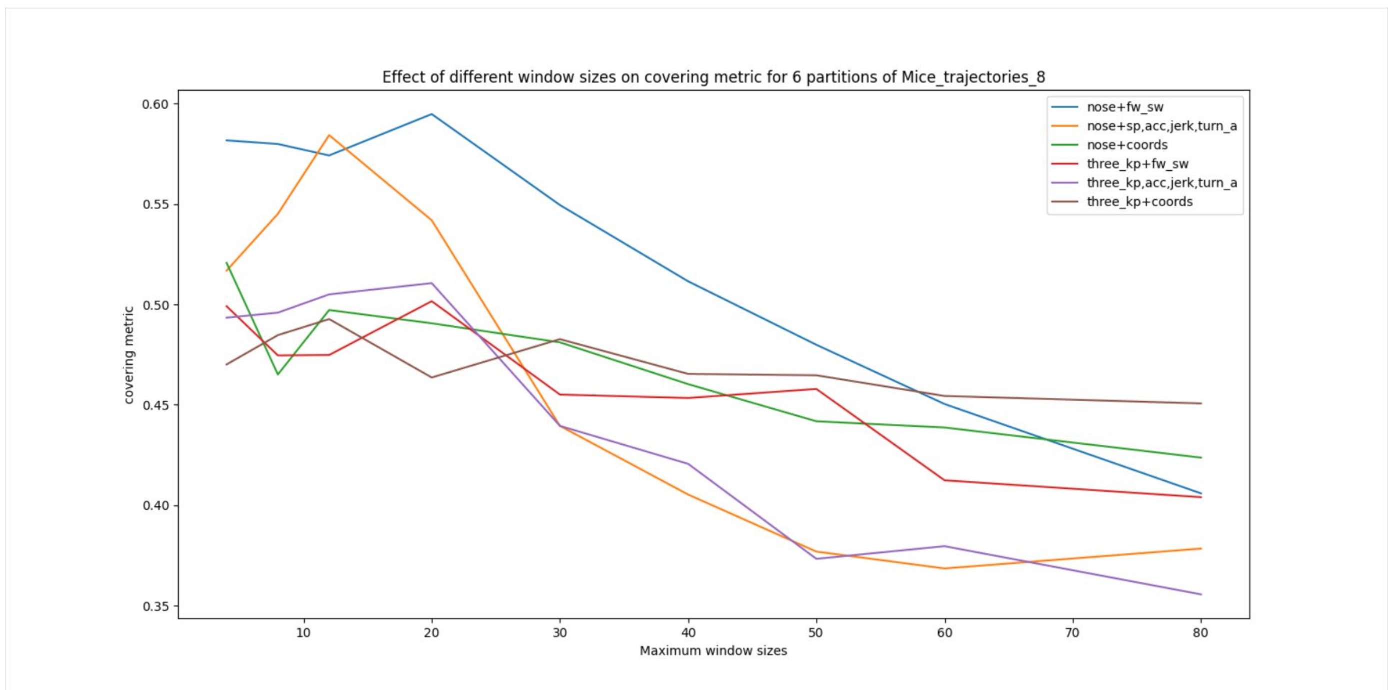

Figure A.1: The effect of window size on the covering metric of our method, with five different input data forms.

The effect on the covering metric shows a similar trend as the meanCovF1M5. The nose+fw_sw and the nose+speed, acc, jerk, turn_angle show by far the best performance, around window sizes 10-30.

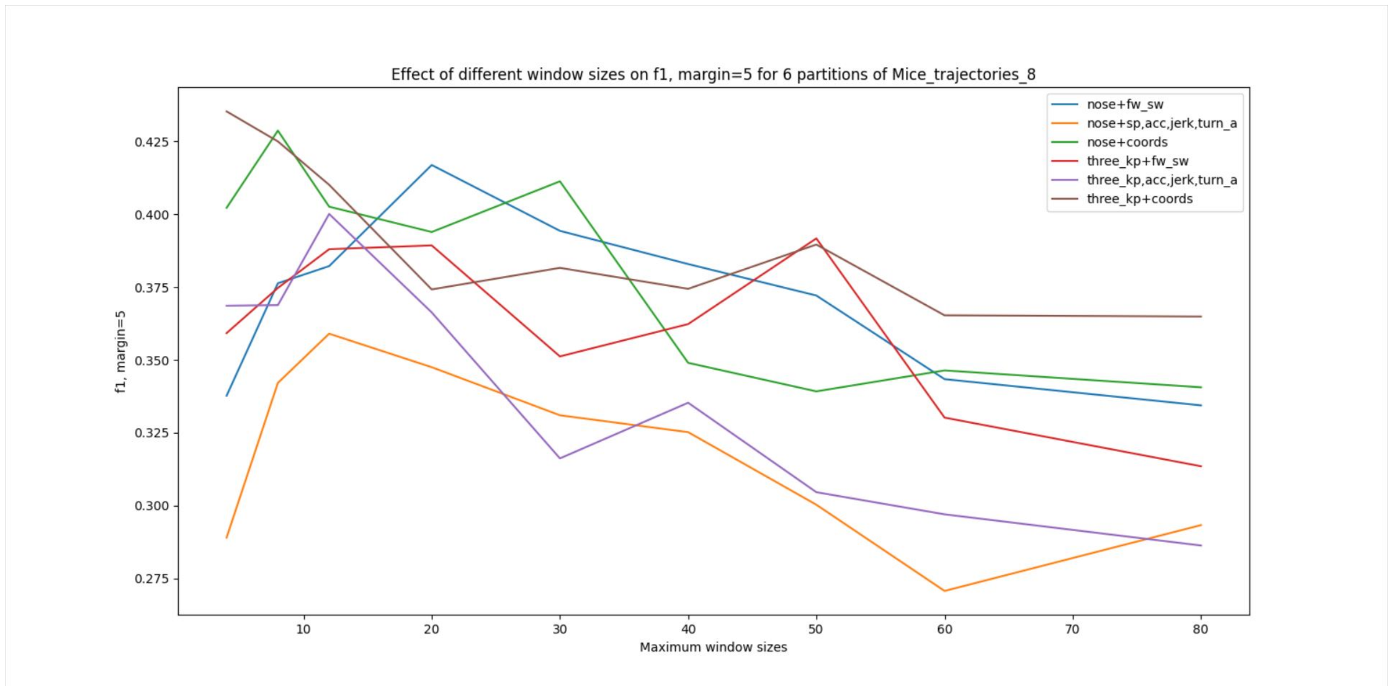

Figure A.2: The effect of window size on the f1 metric (margin=5) of our method, with five different input data forms.

The different data forms show a slight decline in score with larger window sizes with the best scores around sizes 10 - 30. It shows more variability in the scores and overall a weaker decreasing trend than the covering metric. There seems to be a larger difference between the different dataforms regardless of the window size.

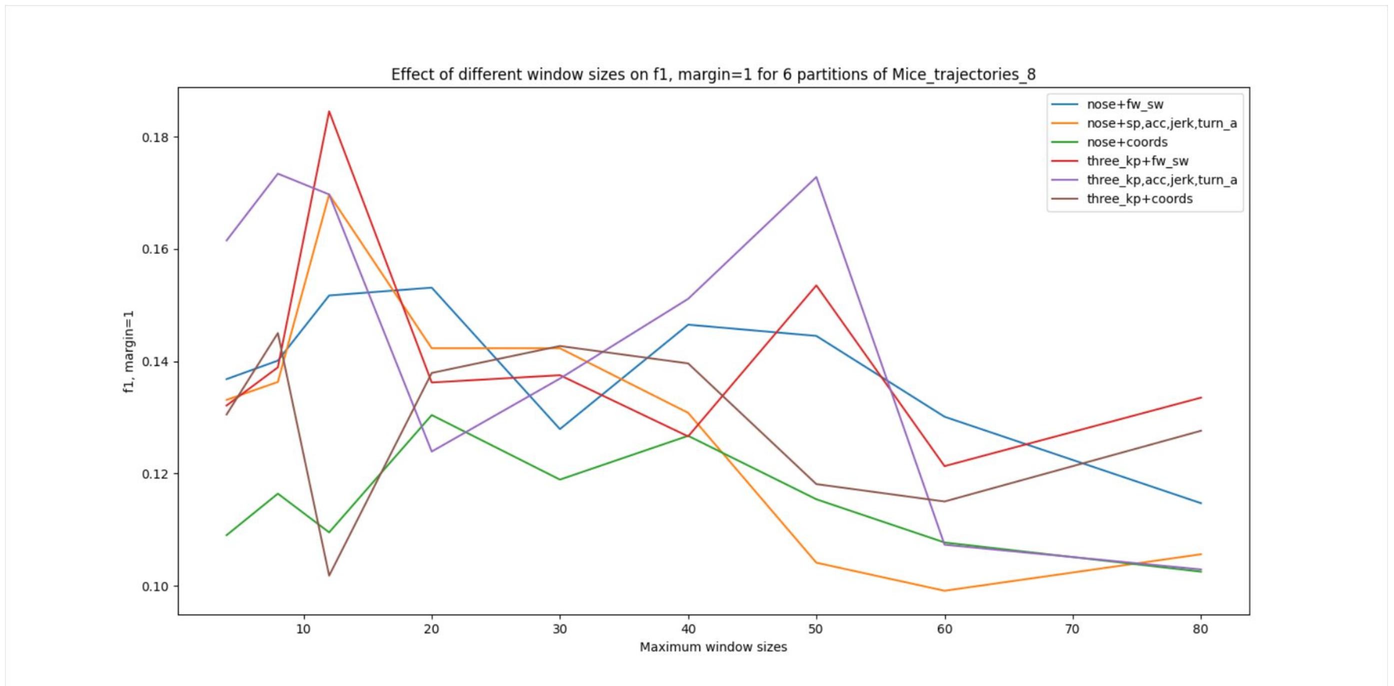

Figure A.3: The effect of window size on the f1 metric (margin=1) of our method, with five different input data forms.

The window sizes show no clear trend for the effect of the f1 (margin=5) score. The most stable results between datasets are obtained with window sizes between 20-40, with the ranges 4-20 and 40 - 60 showing more variability. The three keytpoints with fw_sw and speed, acc, jerk, turn_angle and the nose with fw_sw seem to perform the best in both these ranges. nose + speed, acc, jerk, turn_angle only performs well in the 20-40 range and the others show poor performance in both ranges.

A.2 Table of All Results on Six Partitons of Mice_trajectories_1

Method	Features	Keypoints	Mean covering and f1,margin=5	Covering metric	f1, margin=5	f1, margin=1	runtime (s)
Our_method,max_win_size=20	fw_sw	nose	0.5058	0.5948	0.4169	0.1531	8.191
Our_method,max_win_size=12	fw_sw	nose	0.4782	0.5742	0.3822	0.1517	3.3134
Our_method,max_win_size=8	fw_sw	nose	0.4781	0.5799	0.3763	0.1401	1.803
Our_method,max_win_size=30	fw_sw	nose	0.4719	0.5495	0.3943	0.1279	20.0802
Our_method,max_win_size=12	speed,sAcc,sJerk,zero_pad_turn_angle	nose	0.4716	0.5843	0.359	0.1697	14.1441
Our_method,max_win_size=4	coordinates	nose	0.4615	0.5207	0.4022	0.109	0.6994
Our_method,max_win_size=4	fw_sw	nose	0.4597	0.5817	0.3377	0.1368	0.7131
Our_method,max_win_size=8	coordinates	Three_keypoints	0.4548	0.4847	0.425	0.145	5.3483
Our_method,max_win_size=4	coordinates	Three_keypoints	0.4527	0.4701	0.4353	0.1305	1.9041
Our_method,max_win_size=12	speed,sAcc,sJerk,zero_pad_turn_angle	Three_keypoints	0.4525	0.505	0.4001	0.1697	38.4203
Our_method,max_win_size=12	coordinates	Three_keypoints	0.4514	0.4927	0.4101	0.1018	10.2176
Our_method,max_win_size=12	coordinates	nose	0.4499	0.4972	0.4026	0.1095	3.2741
Our_method,max_win_size=40	fw_sw	nose	0.4472	0.5115	0.3829	0.1465	39.2171
Our_method,max_win_size=8	coordinates	nose	0.4469	0.4651	0.4287	0.1164	1.7183
Our_method,max_win_size=30	coordinates	nose	0.4462	0.4811	0.4113	0.1189	19.8793
Our_method,max_win_size=20	fw_sw	Three_keypoints	0.4454	0.5016	0.3893	0.1362	24.9266
Our_method,max_win_size=20	speed,sAcc,sJerk,zero_pad_turn_angle	nose	0.4447	0.5419	0.3475	0.1423	35.064
Our_method,max_win_size=8	speed,sAcc,sJerk,zero_pad_turn_angle	nose	0.4436	0.5451	0.3421	0.1363	7.6754
Our_method,max_win_size=20	coordinates	nose	0.4423	0.4906	0.3939	0.1304	8.3164
Our_method,max_win_size=20	speed,sAcc,sJerk,zero_pad_turn_angle	Three_keypoints	0.4384	0.5106	0.3663	0.1239	98.8815
Our_method,max_win_size=8	speed,sAcc,sJerk,zero_pad_turn_angle	Three_keypoints	0.4323	0.4959	0.3688	0.1734	22.8185
Our_method,max_win_size=30	coordinates	Three_keypoints	0.4321	0.4827	0.3816	0.1427	60.2102
Our_method,max_win_size=12	fw_sw	Three_keypoints	0.4314	0.4748	0.388	0.1845	9.8142
Our_method,max_win_size=4	speed,sAcc,sJerk,zero_pad_turn_angle	Three_keypoints	0.431	0.4934	0.3686	0.1615	7.6719
Binseg_l2	fw_sw	Three_keypoints	0.431	0.5026	0.3595	0.1362	2.0984
Our_method,max_win_size=4	fw_sw	Three_keypoints	0.4292	0.4991	0.3592	0.1321	2.1108
Our_method,max_win_size=50	coordinates	Three_keypoints	0.4272	0.4647	0.3896	0.1181	229.8233
Our_method,max_win_size=50	fw_sw	nose	0.426	0.4799	0.3721	0.1445	69.2028
Our_method,max_win_size=50	fw_sw	Three_keypoints	0.4248	0.4579	0.3917	0.1535	218.3565
Our_method,max_win_size=8	fw_sw	Three_keypoints	0.4246	0.4746	0.3747	0.1389	5.5005
Pelt_l2,penalty_optimized	coordinates	nose	0.4227	0.4721	0.3733	0.1001	245.552
Binseg_mahalanobis	fw_sw	Three_keypoints	0.4227	0.4901	0.3553	0.1502	32.7007
Binseg_mahalanobis	coordinates	nose	0.4223	0.4562	0.3884	0.1136	26.3645
Pelt_l2,penalty_optimized	coordinates	Three_keypoints	0.42	0.4662	0.3738	0.1177	381.6666
Our_method,max_win_size=40	coordinates	Three_keypoints	0.4199	0.4654	0.3744	0.1396	121.5135
Pelt_mahalanobis,penalty_optimized	coordinates	nose	0.4191	0.4711	0.3672	0.1082	469.0358
Binseg_mahalanobis	speed,sAcc,sJerk,turn_angle	nose	0.419	0.5406	0.2974	0.1335	21.4547
Our_method,max_win_size=20	coordinates	Three_keypoints	0.4189	0.4636	0.3742	0.1379	25.2438
Binseg_l2	coordinates	Three_keypoints	0.416	0.4597	0.3722	0.1224	1.7331
Our_method,max_win_size=60	coordinates	Three_keypoints	0.4098	0.4544	0.3653	0.115	356.7921
Our_method,max_win_size=40	fw_sw	Three_keypoints	0.4079	0.4534	0.3623	0.1266	122.7163
Our_method,max_win_size=80	coordinates	Three_keypoints	0.4078	0.4507	0.3649	0.1276	782.3264
Binseg_l2	coordinates	nose	0.4067	0.4553	0.3581	0.1118	1.7773
Binseg_mahalanobis	fw_sw	nose	0.4063	0.5102	0.3025	0.1382	39.906
Our_method,max_win_size=40	coordinates	nose	0.4047	0.4603	0.349	0.1267	39.5618
Pelt_mahalanobis,penalty_optimized	fw_sw	Three_keypoints	0.4038	0.499	0.3086	0.0977	400.4446
Our_method,max_win_size=30	fw_sw	Three_keypoints	0.4031	0.4551	0.3512	0.1375	61.414
Our_method,max_win_size=4	speed,sAcc,sJerk,zero_pad_turn_angle	nose	0.4029	0.5168	0.289	0.1331	3.2803
Binseg_mahalanobis	speed,sAcc,sJerk,turn_angle	Three_keypoints	0.4005	0.475	0.326	0.1772	23.6949
Our_method,max_win_size=60	fw_sw	nose	0.3969	0.4504	0.3434	0.1301	114.1901
Our_method,max_win_size=60	coordinates	nose	0.3926	0.4387	0.3464	0.1077	114.4821
Binseg_l2	fw_sw	nose	0.3914	0.5042	0.2786	0.1319	2.0904
Our_method,max_win_size=50	coordinates	nose	0.3905	0.4418	0.3392	0.1154	70.0729
Pelt_mahalanobis,penalty_optimized	speed,sAcc,sJerk,turn_angle	nose	0.3894	0.5376	0.2412	0.1212	7544.3926
Pelt_mahalanobis,penalty_optimized	fw_sw	nose	0.3888	0.5278	0.2498	0.1003	10339.5285
Pelt_l2,penalty_optimized	fw_sw	nose	0.3884	0.5368	0.2401	0.0911	453.7514
Our_method,max_win_size=30	speed,sAcc,sJerk,zero_pad_turn_angle	nose	0.3852	0.4395	0.331	0.1423	85.8639
Our_method,max_win_size=80	coordinates	nose	0.3821	0.4237	0.3406	0.1025	253.4546
Pelt_l2,penalty_optimized	fw_sw	Three_keypoints	0.378	0.484	0.272	0.0914	315.7322
Our_method,max_win_size=40	speed,sAcc,sJerk,zero_pad_turn_angle	Three_keypoints	0.3779	0.4206	0.3353	0.1511	489.8605
Our_method,max_win_size=30	speed,sAcc,sJerk,zero_pad_turn_angle	Three_keypoints	0.3778	0.4395	0.3162	0.1369	247.9522
Pelt_mahalanobis,penalty_optimized	speed,sAcc,sJerk,turn_angle	Three_keypoints	0.3748	0.4642	0.2854	0.1624	1820.7078
Our_method,max_win_size=60	fw_sw	Three_keypoints	0.3713	0.4124	0.3302	0.1213	351.5383
Our_method,max_win_size=80	fw_sw	nose	0.3701	0.4059	0.3344	0.1147	252.5281
Our_method,max_win_size=40	speed,sAcc,sJerk,zero_pad_turn_angle	nose	0.3653	0.4053	0.3252	0.1308	166.4787
Binseg_l2	speed,sAcc,sJerk,turn_angle	Three_keypoints	0.3595	0.444	0.2751	0.1445	2.3901
Our_method,max_win_size=80	fw_sw	Three_keypoints	0.3587	0.404	0.3135	0.1335	774.6639
Pelt_mahalanobis,penalty_optimized	coordinates	Three_keypoints	0.3563	0.4147	0.2979	0.107	169.2066
Binseg_l2	speed,sAcc,sJerk,turn_angle	nose	0.3547	0.4709	0.2386	0.1353	2.0289
Binseg_mahalanobis	coordinates	Three_keypoints	0.3454	0.4017	0.2891	0.1005	22.0969
Our_method,max_win_size=50	speed,sAcc,sJerk,zero_pad_turn_angle	Three_keypoints	0.3389	0.3733	0.3046	0.1728	841.9065
Our_method,max_win_size=50	speed,sAcc,sJerk,zero_pad_turn_angle	nose	0.3386	0.3769	0.3003	0.1041	299.0873
Our_method,max_win_size=60	speed,sAcc,sJerk,zero_pad_turn_angle	Three_keypoints	0.3383	0.3796	0.297	0.1073	1383.7217
Our_method,max_win_size=80	speed,sAcc,sJerk,zero_pad_turn_angle	nose	0.3359	0.3784	0.2933	0.1056	1063.2358
Pelt_l2,penalty_optimized	speed,sAcc,sJerk,turn_angle	nose	0.3243	0.4792	0.1693	0.1023	713.0171
Our_method,max_win_size=80	speed,sAcc,sJerk,zero_pad_turn_angle	Three_keypoints	0.321	0.3556	0.2863	0.1029	3476.5788
Our_method,max_win_size=60	speed,sAcc,sJerk,zero_pad_turn_angle	nose	0.3196	0.3685	0.2707	0.0991	484.3461
Binseg_rbf	fw_sw	nose	0.2985	0.3888	0.2082	0.0922	67.6212
Binseg_rbf	coordinates	nose	0.2945	0.3509	0.2381	0.0871	19.5385
Pelt_rbf,penalty_optimized	coordinates	Three_keypoints	0.2874	0.3305	0.2444	0.0842	55.4534
Binseg_rbf	fw_sw	Three_keypoints	0.2874	0.3898	0.1849	0.0823	62.0772
Binseg_rbf	coordinates	Three_keypoints	0.2871	0.3396	0.2345	0.0889	17.9278
Pelt_rbf,penalty_optimized	coordinates	nose	0.2754	0.3387	0.2121	0.0651	91.9633
Pelt_l2,penalty_optimized	speed,sAcc,sJerk,turn_angle	Three_keypoints	0.2444	0.3595	0.1293	0.09	3699.9513
Binseg_rbf	speed,sAcc,sJerk,turn_angle	nose	0.2438	0.2991	0.1885	0.0723	56.2586
Binseg_rbf	speed,sAcc,sJerk,turn_angle	Three_keypoints	0.2293	0.2889	0.1696	0.0756	30.9719
Pelt_rbf,penalty_optimized	fw_sw	Three_keypoints	0.1053	0.1497	0.061	0.0543	36207.3607

Figure A.4: