algorithms

MDPI

Article

Finite Sample Complexity Analysis of Binary Segmentation

Toby Dylan Hocking

Département d'Informatique, Université de Sherbrooke, 2500 Boulevard de l'Université, Sherbrooke, QC J1K-2R1, Canada; toby.dylan.hocking@usherbrooke.ca

Abstract

Binary segmentation is the classic greedy algorithm which recursively splits a sequential data set by optimizing some loss or likelihood function. Binary segmentation is widely used for changepoint detection in data sets measured over space or time, and as a sub-routine for decision tree learning. In theory it should be extremely fast for N data and K splits, $O(NK)$ in the worst case, and $O(N \log K)$ in the best case. In this paper we describe new methods for analyzing the time and space complexity of binary segmentation for a given finite N, K, and minimum segment length parameter. First, we describe algorithms that can be used to compute the best and worst case number of splits the algorithm must consider. Second, we describe synthetic data that achieve the best and worst case and which can be used to test for correct implementation of the algorithm. Finally, we provide an empirical analysis of real data which suggests that binary segmentation is often close to optimal speed in practice.

Keywords: change-point detection; segmentation; heuristics

1. Introduction

Data sequences are measured over time or space in many fields of study such as genomics [1], oceanography [2], finance [3], and neuroscience [4]. In the analysis of such data, an important step involves detection of abrupt changes in the data distribution over time or space. In this context, dynamic programming algorithms can be used to compute a globally optimal set of changes and segment-specific parameters for some loss or likelihood (even though the optimization problem is non-convex). For computing a model with K splits for N data, classic dynamic programming algorithms include the $O(KN^2)$ time algorithm of Auger and Lawrence [5] and the $O(N^2)$ algorithm of Jackson et al. [6]. Since these classic algorithms are too slow for large N, new algorithms using pruning rules have been proposed to decrease computation time to $O(N \log N)$, while maintaining optimality [7]. However even these new optimal algorithms are not fast enough for some applications, so an alternative is the classic binary segmentation heuristic [8]. Binary segmentation is not guaranteed to compute an optimal set of changepoints, but it can be extremely fast. An entire regularization path of models from 0 to K splits can be computed in worst case $O(NK)$, best case $O(N \log K)$ time. Our paper provides a new method for analyzing the amount of computation that binary segmentation must do for a finite N and K (in the best and worst cases, as well as in real data sets).

1.1. Related Work

The classic binary segmentation heuristic algorithm is originally attributed to Scott and Knott [8]. It involves recursively searching a sequential data set for the best split point,

Academic Editor: Jesper Jansson

Received: 15 June 2026
Revised: 31 August 2026
Accepted: 2 September 2026
Published: 3 September 2026

given previously chosen split points (Figure 1). More recent variants of binary segmentation have been proposed, such as searching random intervals [3,9]. Another idea is to limit the search to one changepoint each in a number of pre-defined seed intervals [10]. Because a known limitation is detection of short segments in longer ones [11], more computationally intensive variants of binary segmentation have been proposed which involve searching for two split points simultaneously [1,12]. Binary segmentation is also an important sub-routine in decision tree learning algorithms such as CART [13] and Maximum Margin Interval Trees [14]. In each of the above algorithms there is a model complexity parameter (number of splits/segments), which needs to be appropriately chosen given the data. This model selection problem can be solved using several unsupervised [15,16] as well as supervised methods which can be used in situations where labeled training data are available [17,18].

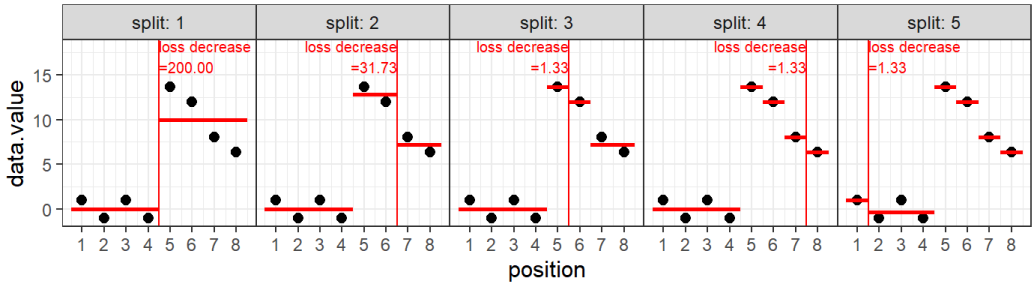

Figure 1. Demonstration of a binary segmentation algorithm on a simple synthetic data set (details in Section 2.1) for which a tie-breaking rule is required to achieve the best case number of splits to consider. Black dots show data and red lines show the binary segmentation models. After the first two splits, the next three splits all have an equal loss decrease value (1.33), so the best case time complexity can be achieved with a tie-breaking rule that chooses a split which results in the smallest number of candidate splits to consider afterwards.

The dynamic programming algorithm we propose to compute the best case number of splits for binary segmentation is similar to the idea of the optimal binary search tree, for which there are many classic algorithms [19]. For example Knuth [20] described $O(n^3)$ and $O(n^2)$ optimal algorithms for a tree with n nodes, as well as $O(n \log n)$ heuristic/approximate algorithms which are very fast but not guaranteed to compute the optimal tree. Later Mehlhorn [21] proved that the important heuristic was bisection, which always produces nearly optimal trees.

1.2. Contributions and Organization

Our main contributions are new algorithms for finite sample time and space complexity analysis of binary segmentation. We propose algorithms for computing the best and worst case number of splits that must be considered, which is proportional to overall computation time. In Section 2 we describe the classic binary segmentation algorithm, and provide new synthetic data examples that can be used for testing that the algorithm is correctly implemented. In Section 3 we propose new algorithms which can be used to compute the best and worst case number of splits for finite sample complexity analysis. To the best of our knowledge this is the first time such finite sample complexity analysis has been described for binary segmentation with a minimum segment length parameter. In Section 4 we provide an empirical study of binary segmentation in several synthetic and real data sets. Section 5 concludes with a discussion of the significance and novelty of our findings.

2. Binary Segmentation Algorithm

In this section we provide an overview of how the binary segmentation algorithm works. We assume there is a sequence of N data for which we want to compute K

splits/iterations (max number of segments is $K+1$). We further assume that each iteration splits a segment into two segments, each of minimum size m (this is the minimum segment length parameter). Finally, we assume there is a loss function ℓ that binary segmentation seeks to minimize (for example, common choices are the square loss or L1 loss for real-valued data, and the Poisson loss for non-negative integer count data). The binary segmentation algorithm performs the following computations: (a) for any new segments, find the split which results in the best loss decrease; (b) store that segment/split in a container for future lookup; (c) look in that container for the segment with best loss decrease and split it into two new segments; (d) repeat until the max number of splits K is reached.

To analyze the time complexity of step (a) we use the number of candidate splits that must be considered to find the best loss decrease (Table 1). Throughout the paper we use various synonyms for "number of candidate splits that must be considered": "number of splits the algorithm must consider," "number of candidate split points," "number of candidate split evaluations," etc. Note that this is a valid choice for finite sample time complexity analysis since the loss of each candidate split can be computed in amortized constant time for common loss functions including square/Poisson (and log-time for L1 loss). The positions after which a change may occur in N data are $\{m, \ldots, N-m\}$ and the size of this set (number of candidate splits that must be considered) is

$$g(N) = (N - 2m + 1)_+, \tag{1}$$

where $(\cdot)_+$ is the positive part function (identity if argument is positive, otherwise zero). For example, with min segment length $m = 2$, we have $g(4) = 1$ because, starting with four data, there is only one possible split (in the middle) that results in segments with at least two data. We have $g(3) = g(2) = 0$ with $m = 2$ because segments of size 3 and 2 can not be split to yield two segments of at least size 2, so there are no candidate splits to consider.

Table 1. Counts of best and worst case number of candidates to compute for $N = 64$ data and min segment length $m = 1$. Each row shows the number of candidates for a given split K and tree depth j, along with the cumulative total.

		Best Case: Equal Splits		**Worst Case: Unequal Splits**	
j	K	**This Iteration**	**Total**	**This Iteration**	**Total**
1	0	$N-1=63$	$N-1=63$	$N-1=63$	$N-1=63$
2	1	$N-2=62$	$2N-3=125$	$N-2=62$	$2N-3=125$
	2	$N/2-2=30$		$N-3=61$	$3N-6=186$
3	3	$N/2-2=30$	$3N-7=185$	$N-4=60$	$4N-10=246$
⋮	⋮	⋮	⋮	⋮	⋮
4	7	$N/4-2=14$	$4N-15=241$	$N-8=56$	$8N-36=476$
⋮	⋮	⋮	⋮	⋮	⋮
7	63	$N/32-2=0$	$7N-127=321$	$N-64=0$	$64N-2080=2016$

To analyze the time and space complexity of steps (b) and (c) we need to specify what type of container is used. For efficiency the container should be a red-black tree or similar data structure which allows for constant time lookup of the item with best loss decrease, and insertion of a new item which is logarithmic in the container size. For finite sample complexity analysis of steps (b) and (c) we therefore need to examine the container size before each insert (Table 2). Note that in both steps (a) and (b) the "best" loss decrease must be identified. In order for the algorithm to achieve the best case number of candidate splits considered, it must correctly perform tie-breaking, as explained in the next section.

Table 2. Table of best and worst case storage, for $N = 64$ data with min segment length $m = 1$ and max number of segments/splits. Sizes before inserts columns show the size of the container before each insert operation (to analyze time complexity of the insert operation); size after iteration columns show the size of the container after each iteration (to analyze space complexity). Equal splits results in largest container size before inserts of $O(N)$, and worst case space complexity of $O(N)$. Unequal splits result in the smallest container size before inserts (always zero), and best case storage (container size 1 after each iteration).

	Equal Splits		**Unequal Splits**	
Iteration I	**Sizes Before Inserts**	**Size After Iteration**	**Sizes Before Inserts**	**Size After Iteration**
1	0	1	0	1
2	0, 1	2	0	1
3	1, 2	3	0	1
⋮	⋮	⋮	⋮	⋮
32	30, 31	32	0	1
33	none	31	0	1
34	none	30	0	1
⋮	⋮	⋮	⋮	⋮
62	none	2	0	1
63	none	1	0	1

2.1. Tie-Breaking Rules and Test Cases

In this section we provide examples of simple synthetic data which have splits of equal loss, and thus require a tie-breaking rule in order to achieve the best case number of splits. We begin by discussing the case of ties in split points within a segment, then we discuss the more complex case of ties in different segments that could be split.

2.1.1. Ties in Split Points on a Segment

When the binary segmentation algorithm considers a new segment to split, it must optimize over all possible split points, by taking the split that results in minimum loss. If there are several splits which each have the same minimum loss value, then to minimize the number of candidates to optimize over in the next step, the binary segmentation algorithm should first break ties using the number of split candidates which would need to be computed, then the distance from the start/end of the segment. A simple example is the data sequence $[1, 2, \ldots, 8]$ with the L1 loss and min segment length $m = 1$. The optimal L1 loss values for the seven possible splits are [12, 10, 8, 8, 8, 10, 12], so there are three splits which result in the same min loss. Each of these three splits has the same number of split candidates which would need to be computed during the next iteration, $g(3) + g(5) = g(4) + g(4) = 6$. The best case number of splits for future iterations is achieved if the algorithm chooses the split in the middle (two child segments of size 4). As discussed later, choosing the split in the middle is a heuristic, which is not guaranteed to achieve the best case number of candidate splits, but is sufficient to get within a constant factor. More generally, let $\underline{p}$ be the segment start, $\overline{p}$ be the segment end, and $c \in \{\underline{p}, \ldots, \overline{p} - 1\}$ be a candidate split point. The distance from split c to the nearest start or end is

$$D_c = \min\{c - \underline{p}, \overline{p} - c - 1\}. \tag{2}$$

Let $\mathcal{C}$ be the set of splits to tie-break, with min loss and min number of split candidates which would need to be computed during the next iteration. The best case tie-breaking rule chooses the split with maximal value for the distance to nearest start or end,

$$c^* = \arg\max_{c \in \mathcal{C}} D_c. \tag{3}$$

In the example above, the segment start $\underline{p} = 1$, end $\overline{p} = 8$, and set of splits to tie-break $\mathcal{C} = \{3, 4, 5\}$. Distances are $D_3 = 2$, $D_4 = 3$, $D_5 = 2$, so the best split is therefore $c^* = 4$, which results in two segments of size 4. This requires computing six new candidates in the next step (three on each of the two segments with four data), then four in the following steps (one on each of the four segments with two data), for a total of 10 candidates. This is preferable to the other options, which result in a larger number of candidates (4 and 2 with segments of size 3 and 5; then 1, 2, 2 for segments of size 2, 2, 3; then 1 for final segment of size 2; total number of candidates 12).

2.1.2. Ties in Different Segments That Could Be Split

A flexible and efficient implementation of binary segmentation requires a container to store the different segments that could be split during each iteration. The segments in this container should be sorted by loss decrease values; the next segment to split should result in the largest loss decrease. Similar to the discussion above about ties in split points on a segment, it is possible for there to be several segments which have the same loss decrease value. In this case, to achieve the best case number of splits to optimize over in subsequent iterations, the algorithm should again break ties in the same way. A simple synthetic data set for which this tie-breaking rule is necessary using the L2 square loss can be constructed in the following manner. First, let $a, b, x, \epsilon \in \mathbb{R}$ be real number parameters which will determine our synthetic data $[a, -a, a, -a|b + \epsilon + x, b + \epsilon|b - \epsilon, b - \epsilon - x]$. The idea is to choose a b and ϵ large enough so that the first two splits happen at the vertical bar positions, resulting in three segments. Then a and x can be chosen such that all three segments have an equal value for the best loss decrease. We chose $a = 1$, $b = 10$, $x = \sqrt{8/3}$, $\epsilon = 2$, which results in the data shown in Figure 1. The algorithm proceeds as follows:

Split 1:]7 candidates are considered; the best has a change after position 4.

Split 2: Each of the two new children segments created from split 1 require computing the loss of 3 split candidates, for a total of 6 candidates considered. Splitting the optimal segment results in a change after position 6, whereas the other segment with a loss decrease of $4/3$ stays in the container.

Split 3: Each of the two new children segments created from split 2 require computing the loss of 1 split candidate, for a total of 2 candidates considered. Both new segments have a loss decrease of $4/3$, so the container now has three segments with the same loss decrease value. Tie-breaking is performed by minimizing the number of split candidates which would be required to be computed in the next iteration; the result is a change after position 5. Two segments with the same loss decrease value remain in the container.

Split 4: Both of the two new children segments created from split 3 are of size 1, so cannot be split, and do not require any new split candidates to be computed. The next split results in a change after position 7, and one segment remains in the container.

Split 5: Again no new split candidates are computed, and instead the last segment in the container is split, resulting in a change after position 1, and an empty container.

Looking at the example above, we see that the total number of split candidates considered is 15, which is indeed the best case for $N = 8$ data, $K = 5$ splits, and a min segment length of $m = 1$. In contrast if we had not implemented tie-breaking, and had instead chosen split 3 after position 1, then we would have computed two additional split candidates, for a total of 17. The worst case would be achieved for the data sequence $[-1, 1, -1, 1, -1, 1, -1, 1]$ which for $K = 5$ splits would require computing the loss for a total of 25 split candidates. In the next section we describe methods that can be used to

compute the best/worst case number of iterations for any data size N, number of splits K, and min segment length m.

3. Finite Sample Complexity Analysis

We begin this section with an analysis of the number of candidate splits that must be considered when computing the maximum number of splits, $K = N - 1$. For simplicity we first assume a min segment length of $m = 1$ and that N is a multiple of 2; we can let $N = 2^J$ for some $J \in \{1, 2, \dots\}$, for example $J = 6 \Rightarrow N = 64$. Then for any $j \in \{1, \dots, J+1\}$ if we do $K = 2^{j-1} - 1$ splits then the number of candidates that must be considered is shown in Table 1. In the best case, each split results in segments of equal size, and we have $Nj - 2^j + 1 = N(1 + \log_2 K) - 2K + 1 \leq 1 + N + N \log_2 K \Rightarrow O(N \log K)$ total candidates to consider. In the worst case, each split results in segments of unequal sizes, and we have $NK - K(1+K)/2 \leq NK \Rightarrow O(NK)$ total candidates to consider.

To store the previously computed best loss/split for each segment, we use a container with insert operations which are logarithmic in container size—if the container has p items then an insert takes $O(\log p)$ time. In Table 2 we therefore analyze the container size in the two extreme cases (equal and unequal splits). The "sizes before inserts" columns show p before each insert (to analyze time complexity), and the "size after iteration" columns show p after inserts (to analyze space complexity). For unequal splits, the total time is linear, and the total space is constant, because the container always stays at size 1. For equal splits, the total time over all inserts is $-\log(K-1) + \sum_{p=1}^{K-1} 2 \log p \in O(K \log K)$, which is smaller than the $O(N \log K)$ time for the split computation, so the container insert step is not a limiting factor for time complexity. However for equal splits, the space complexity achieves the worst case, a max container size of $N/2$ segments for which we have already computed the best split point, but we have not yet split, at iteration $N/2$. The asymptotic size of the container is $O(N)$ (same as the input data), and there is no larger storage to consider, so the space complexity is reasonable even for very large data sizes N.

3.1. New Dynamic Programming Algorithm for Computing Best Case Number of Candidate Splits

In the previous section we provided an overview of how the finite sample complexity analysis works for the special case of the data size being a multiple of 2. The worst case analysis (counting number of candidates for unequal splits) also works for the general case. In particular, for min segment length m, the worst case is obtained when m data are split off by themselves in each iteration. The worst case number of candidates is therefore $\sum_{i=0}^{K-1} g(N - im)$. However, the best case analysis does not work in the general case (equal splits do not always result in the best case for small K). In this section we describe a new algorithm which can be used to compute the best case number of splits that must be considered for any data size N, number of splits K, and min segment length m.

Recall from (1) that $g(N)$ is the number of candidate splits which must be considered in order to find the optimal split on the segment of size N. Let $f(N, T)$ be the best number of candidates computed after the segment of size $N \geq 1$ is split $T \geq 0$ times; the resulting segments are stored in the container, and the best feasible segment is then split, resulting in a model with at most $K = T + 1$ splits (there may be fewer if min segment length $m > 1$ and some segments are not feasible to split). For T times, $2T + 1$ is the number of segments which are created and then searched for the best split point.

The geometric interpretation of $f(N, T)$ involves a binary tree with T splits and $2T + 1$ nodes, in which each node represents a segment with a certain size, and the size of each parent always equals the sum of sizes over the two children (Figure 2). Computing $f(N, T)$ involves searching the space of all trees with root size N, T splits, and $2T + 1$ nodes, in

https://doi.org/10.3390/a19090754

order to find the tree with min cost, as quantified by summing $g(s)$ over all node sizes s in the tree.

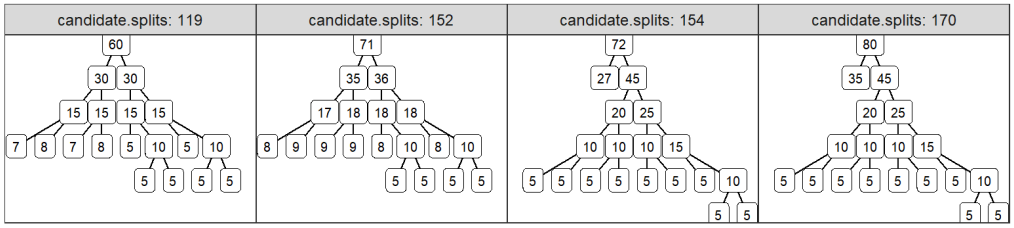

Figure 2. Optimal binary trees constructed to determine optimal number of candidate splits for $N \in \{60, 71, 72, 80\}$ data, which are split $T = 9$ times, with min segment length $m = 5$. Smaller N values result in a balanced first split, whereas larger N values result in an unbalanced first split (one small child with no splits, one large child with all remaining splits).

For example, for $T = 0$, there is only one segment (the original with all data) which is searched, with $f(N, 0) = g(N)$ candidates considered, resulting in a model with $K = 1$ split (2 segments). This corresponds to a binary tree with one node and no splits.

For $T = 1$, there are three segments searched (the original plus the two created after the first split), resulting in a model with $K = 2$ splits (3 segments); the number of candidates to compute is

$$f(N, 1) = g(N) + \min_{s \in \{m, \ldots, N-m\}} f(s, 0) + f(N - s, 0). \tag{4}$$

The min above is over the split point s. It can be efficiently computed in linear $O(N)$ time since each $f(N, 0) = g(N)$ can be evaluated in constant time. This corresponds to a binary tree with three nodes and one split.

For $T = 2$, there are five segments searched (the original plus the four created after the first two splits), resulting in a model with $K = 3$ splits (4 segments); the number of candidates to compute is

$$f(N, 2) = g(N) + \min_{s \in \{m, \ldots, N-2m\}} f(s, 0) + f(N - s, 1) \tag{5}$$

The split values s have an upper bound of $N - 2m$ to respect the min segment length m. Assuming the each of the $f(N, 1)$ values have been pre-computed, the min above can be efficiently computed in linear $O(N)$ time. This corresponds to a binary tree with five nodes and two splits.

For $T = 3$, there are seven segments searched (the original plus the six created after the first three splits), resulting in a model with $K = 4$ splits (5 segments); the number of candidates to compute is

$$f(N, 3) = g(N) + \min \begin{cases} \min_{s \in \{2m, \ldots, N-2m\}} f(s, 1) + f(N - s, 1) \\ \min_{s \in \{m, \ldots, N-3m\}} f(s, 0) + f(N - s, 2). \end{cases} \tag{6}$$

This corresponds to a binary tree with seven nodes and three splits. Above, the first min is over tree topologies. The first line in the min corresponds to splitting the left and right nodes once each, whereas the second line in the min corresponds to not splitting the left node, but splitting the right node twice.

The key idea of our proposed algorithm is that $f(N, T)$ can be recursively computed. For some ideal smaller number of changes $d \in \{0, \ldots, T - 1\}$, and for some ideal split size s, we have $f(N, T) = g(N) + f(s, d) + f(N - s, T - d - 1)$. To compute this we need to

search over all possible numbers of changes d and split sizes s. For any number of changes d, we have the following lower bound on the size s due to the min segment length m,

$$s \geq (d+1)m. \tag{7}$$

For the same reason we have the following upper bound,

$$s \leq N + (d-T)m. \tag{8}$$

For example consider computing $f(15,2)$ with min segment length of $m = 3$. In that case we need to search over $f(s,0) + f(15-s,1)$ for seven sizes $s \in \{3,4,5,6,7,8,9\}$. Additionally when the number of changes in the two child segments is the same, the symmetry of the problem implies the following stronger upper bound,

$$d = (T-1)/2 \Rightarrow s \leq \lfloor N/2 \rfloor. \tag{9}$$

For example consider computing $f(15,3)$ with min segment length of $m = 3$. In that case we need to search over $f(s,0) + f(15-s,2)$ for four sizes $s \in \{3,4,5,6\}$, and $f(s,1) + f(15-s,1)$ for just two sizes $s \in \{6,7\}$. When there are d changes the overall upper bound on s is therefore

$$\bar{s}(N,T,d,m) = \begin{cases} N + (d-T)m & \text{if } d < (T-1)/2, \\ \lfloor N/2 \rfloor & \text{if } d = (T-1)/2. \end{cases} \tag{10}$$

For N data, split T times, with min segment length m, and any smaller number of changes $d < T$, we therefore have the following set of sizes s to optimize over,

$$\mathcal{S}(N,T,d,m) = \{(d+1)m, \ldots, \bar{s}(N,T,d,m)\}. \tag{11}$$

The results above are combined into the theorem below, which gives an efficient and optimal method for computing the best case number of splits that need to be computed during binary segmentation.

Theorem 1. *Assume binary segmentation with T splits, starting with a segment of N data, with min segment length m. The best case number of candidate splits that must be computed can be determined as follows. First, for all N we initialize $f(N,0) = g(N)$. Then, for all $T > 0$ and for all N we use the following dynamic programming update rule.*

$$f(N,T) = g(N) + \min_{d \in \{0,\ldots,\lfloor (T-1)/2 \rfloor\}} \min_{s \in \mathcal{S}(N,T,d,m)} f(s,d) + f(N-s, T-d-1). \tag{12}$$

Proof. First, T takes values between 0 and $N-2$, which corresponds to models from 1 to $N-1$ splits (the max value is smaller if the min segment length $m > 1$). In (11), the lower bound $(d+1)m$ comes from (7): there must be m data on a segment, so at least $2m$ data on a segment which must be split once, etc. The upper bound (10) comes from symmetry, meaning there is no need to check both $f(4,0) + f(6,0)$ and $f(6,0) + f(4,0)$, because they are of equal cost, etc. Similarly the upper bound on d in (12) comes from symmetry, meaning there is no need no check both $f(5,2) + f(5,0)$ and $f(5,0) + f(5,2)$, because they are of equal cost, etc. Finally, the update rule avoids under-counting and double-counting candidates, by including a $g(s)$ term for each node in the corresponding binary tree. □

Pseudo-Code and Complexity Analysis of Proposed Algorithm

The dynamic programming algorithm is summarized in Algorithm 1, which has two for loops (over depth and size). Inside each iteration of those two for loops is a call to the getMin sub-routine (Algorithm 2), which also has two for loops (over depth and size), in which there are constant time operations (cost lookup and storage). Therefore the getMin sub-routine (Algorithm 2) is $O(NT)$ time, and the overall dynamic programming (Algorithm 1) is $O(N^2T^2)$ time. Since $K = T + 1$, we have overall $O(N^2K^2)$ time. Because of this quadratic time complexity, it is only reasonable to run for relatively small number of data N and splits K. Asymptotically faster algorithms are an interesting direction for future research.

Algorithm 1: Computing the best case number of splits in binary segmentation.

Input: Number of data N, number of times to split T, min segment length m
Initialize data structure to store optimal cost values: F
for *depth* d *from 0 to* T **do**
  for *size* $s \in \mathcal{S}(N, T, d, m)$ **do**
    min_cost $\leftarrow$ if $d = 0$ then 0 else getMin(s, d, m, F)
    $F[s, d] \leftarrow$ min_cost $+ (s - 2m + 1)_+$
  end
end
Output: $F[N, T]$, smallest number of splits that must be considered.

Algorithm 2: getMin sub-routine.

Input: Number of data N, number of times to split T, min segment length m, optimal cost values F
Initialize min_cost $\leftarrow \infty$
for *depth* d *from 0 to* $\lfloor (T-1)/2 \rfloor$ **do**
  for *size* $s \in \mathcal{S}(N, T, d, m)$ **do**
    cost $\leftarrow F[s, d] + F[N - s, T - d - 1]$
    if cost $<$ min_cost then min_cost $\leftarrow$ cost
  end
end
Output: min_cost, smallest number of splits that must be considered.

4. Empirical Results

In this section we analyze the results of running our proposed Algorithm 1, and show how it can be used to compare the best case with the empirical number of candidate splits computed in real data sets.

4.1. Analysis of Optimal Binary Trees

In order to visualize the structure of the optimal binary trees when the number of splits K is less than the number of data N, we ran Algorithm 1 using $N \in \{60, 71, 72, 80\}$ data, min segment length $m = 5$, and number of splits/iterations $I = 9$. We expected results qualitatively similar to the best case binary trees produced when the number of splits is set to the max value, $K = N - 1$ (in these trees each split creates two children of equal size, as explained in Section 3). Interestingly, we observed that this is only the case when N is relatively small (Figure 2, $N = 60$ and 71). When N is relatively large, we observed qualitatively different best case trees in which the first split creates one small node with no

children, and one large node with the rest of the children (Figure 2, $N = 72$ and 80). These results show that computing the tree in which children are always of nearly equal size is not sufficient (and is heuristic) for computing the best case number of candidate splits.

4.2. Analysis of Number of Candidate Splits Considered in Real Data Sets

In this section we examine the number of candidate splits which must be computed in real data sets, in order to determine if binary segmentation is closer to the best or worst case in practical scenarios. We expected that binary segmentation should be closer to the best case in real data sets, since the worst case only happens with pathological data sets (such as data which bounce up and down). Since the proposed Algorithm 1 is quadratic time, we only ran it for small data sizes, so we only know the true lower bound on the number of candidate splits for small data sizes. For larger data sizes, we computed an approximate lower bound using a fast heuristic algorithm which performs a depth-first search of the binary tree with nearly equal sized children. The depth-first search is a heuristic, meaning that it is not guaranteed to return the best case number of candidates to consider, but it does make sense as an approximate lower bound (since it recursively divides each segment into nearly equal parts). Our empirical results show that the heuristic is optimal (returns the best case number of candidates) when binary segmentation is run to the max number of splits ($K = N - 1$). Our empirical results also show that the heuristic is only larger by a constant factor when binary segmentation is run to a smaller number of splits.

4.2.1. Peak Detection Benchmark

We used data from the McGill ChIP-seq benchmark [22], which does not have a license but is freely downloadable from https://rcdata.nau.edu/genomic-ml/chip-seq-chunk-db/ (accessed on 2 Sept 2026). This benchmark consists of 2752 real genomic data sequences of size N from 87 to 263,169. Since these data are non-negative counts, we used the Poisson loss. We used R package binsegRcpp, and specified the max number of segments as either a constant (19), or equal to the number of data. We used min segment length parameter $m = 1$. The main goal in analyzing these data is to determine the positions of peaks, which are regions where the signal jumps abruptly up and then back down again. We therefore suspected that there may be some evidence of the worst case number of candidate splits to consider (because the data bounce up and down). However, we observed that the empirical number of splits that the algorithm considered closely matches the best case (Figure 3). We used Algorithm 1 to compute the best case number of candidate splits for $N = 100$ to 200, and observed that the heuristic is exact when max segments equals number of data, but the heuristic is an over-estimate of the lower bound when max segments is smaller than the number of data (as expected, see discussion in Section 3).

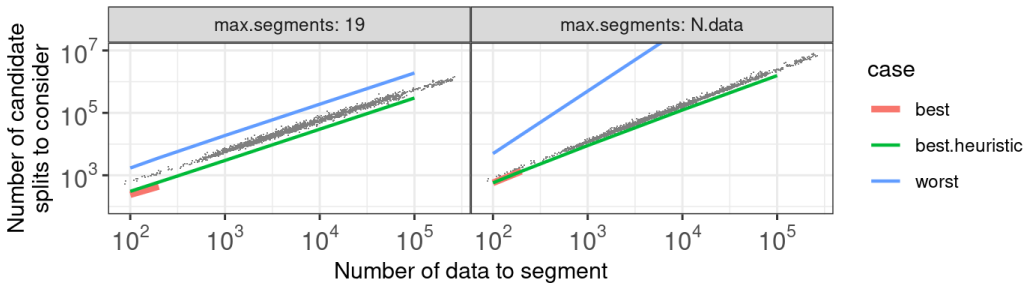

Figure 3. Analysis of 2752 real genomic count data sets from McGill benchmark of data size N from 87 to 263,169, using binary segmentation with the Poisson loss. Number of candidate splits to consider in real data (grey dots) achieves the asymptotic best case, $O(N \log N)$.

4.2.2. Copy Number Benchmark

We used data from R package neuroblastoma [23], which is freely downloadable from the Comprehensive R Archive Network (CRAN) under the GPL-3 license. This benchmark consists of 13,721 real genomic data sets of size N from 11 to 5937. We used the square loss since the data were real numbers. We used R package binsegRcpp, and specified the max number of segments as either a constant (10), or equal to the number of data. We used min segment length parameter $m = 1$. In these data we used Algorithm 1 to compute exact lower bounds for all data sizes N from 11 to 100. We observed that the heuristic was always exact when max segments equals number of data, and it was exact for five sizes, $N \in \{11, 12, 13, 18, 19\}$, when max segments was set to 10. As in the previous benchmark, we observed that the empirical number of splits that the algorithm considered closely matches the best case (Figure 4). In fact we observed 45 real data sets for which the empirical number of splits considered was less than the heuristic/inexact lower bound. Overall these data provide convincing evidence for fast performance in practical applications of binary segmentation. In particular, in these real genomic data sets, binary segmentation is close to the best case asymptotic time complexity.

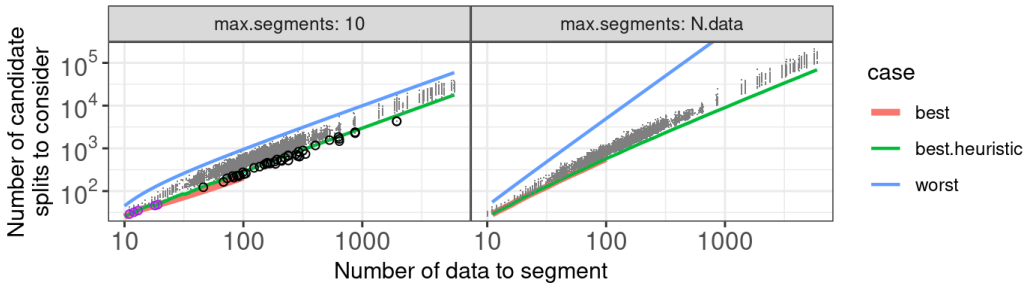

Figure 4. Binary segmentation with the square loss was used on 13,721 real genomic data sets (grey dots) from the neuroblastoma benchmark with data sizes N from 11 to 5937. Number of candidate splits to consider achieves the asymptotic best case, $O(N \log N)$, and in 45 instances (black circles) requires fewer candidate splits than predicted by the best case heuristic. For max segments = 10, we used dynamic programming to compute the best case number of splits for all data sizes between 11 and 100 (orange line), and the heuristic (green line) was exact for only 5 data sizes $N \in \{11, 12, 13, 18, 19\}$ (purple circles).

4.3. Analysis of Computation Time in Synthetic Data

All timing results in this section depend on the particular computer hardware used; we used a laptop with a 2.40 GHz Intel(R) Core(TM)2 Duo CPU P8600.

4.3.1. Similar Asymptotic Trends for Computation Time and Number of Candidate Splits

To validate that candidate split count scales linearly with real computation time, we performed the following experiment. We created synthetic best and worst case data sets for various data sizes N from 10^1 to 10^6. Best case data are linear, $x_i = i$; worst case data are alternating up and down, $x_i = 1$ if i is even and 0 if i is odd. On each data set, we ran binary segmentation, as implemented in the `wbs::sbs()` function in R, which computes the full path of splits using the square loss and min segment length $m = 1$. In this case, $K = N - 1$, so we expected time and number of splits to be $O(N \log N)$ in the best case, and $O(N^2)$ in the worst case. We used R package `atime` to compare timings and number of splits for various data sizes N, stopping after a time limit of 1 s [24]. Figure 5 shows that the empirical measurements (colored curves) closely match the asymptotic references (black curves). These results clearly demonstrate that the number of splits considered by the algorithm scales linearly with real computation time.

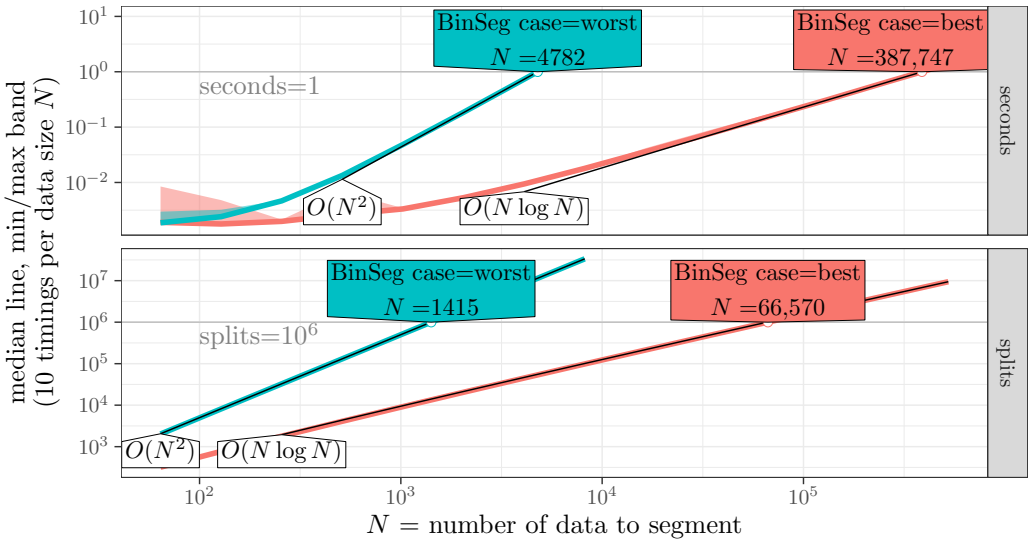

Figure 5. Binary segmentation with max changes ($K = N - 1$ with min segment length $m = 1$) was run for data sets of increasing sizes N, until going over the time limit of 1 s. Colored curves show timings (top) and number of candidate splits (bottom), for best and worst case synthetic data. Black lines show asymptotic references, which suggest that best case is $O(N \log N)$ time/splits, and worst case is $O(N^2)$. $N =$ values in colored boxes show throughput (data sizes computable given a limit of 1 s or 10^6 splits).

4.3.2. Comparing Standard and Wild Binary Segmentation

To demonstrate the speed of binary segmentation relative to other changepoint methods, we performed the following experiment. We created synthetic best case data ($x_i = i$) for various data sizes N from 10^1 to 10^6. On each data set, we ran standard binary segmentation (sbs) alongside wild binary segmentation (wbs), as implemented in the `sbs()` and `wbs()` functions in the R package `wbs` [3]. Both functions compute a full path of splits using the square loss and min segment length $m = 1$. We ran `wbs()` using four diffirent settings for the number of intervals parameter M (two constants: $M = 10$ and the default $M = 5000$; two linearly increasing regimes: $M = N/2$ and $N/10$). We expected the best case $O(N \log N)$ time complexity. We used R package `atime` to compare timings for various data sizes N, stopping after a time limit of 1 s [24]. Figure 6 shows that sbs has a convex timing curve that is consistent with the expected $O(N \log N)$ time complexity. The wbs timing curves are non-convex, with an abrupt decrease in slope at 0.1 s, after which curves increase to their asymptotic regime. It is clear that when M is constant (M = 10 and the default M = 5000), the wbs asymptotic slope is consistent with sbs, the expected best case $O(N \log N)$ time complexity. Surprisingly, when M increases linearly (M = N/2 and N/10), the wbs asymptotic slope is larger, indicating $O(N^2)$ time, asymptotically slower than sbs. The sbs algorithm is substantially faster than all wbs variants, for all N values. Asymptotically, the closest wbs competitor is wbs with M = 10, which is $\approx 10\times$ slower than sbs. These results clearly demonstrate that in best case data, standard binary segmentation is substantially faster than wild binary segmentation.

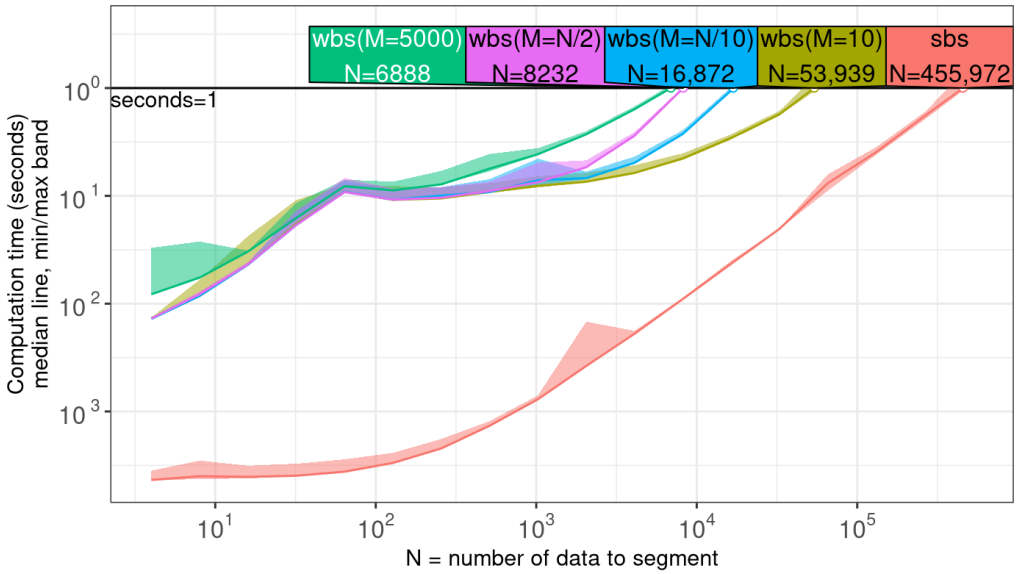

Figure 6. Standard binary segmentation (sbs) and wild binary segmentation (wbs), with max changes ($K = N - 1$ with min segment length $m = 1$), was run for data sets of increasing sizes N, until going over the time limit of 1 s. In these synthetic best case data, sbs is fastest, with asymptotic slope consistent with the expected $O(N \log N)$ time complexity for sbs, and for wbs with constant M parameter. For wbs with increasing M parameter (N/2 and N/10), the larger asymptotic slope is consistent with a larger asymptotic complexity class, $O(N^2)$. N = values in colored boxes show throughput (data sizes computable given a limit of 1 s).

5. Discussion and Conclusions

This paper presented new methods for analyzing the time complexity of the classic binary segmentation algorithm for changepoint detection. Our work analyzed the speed of the classic algorithm in terms of the number of candidate splits that must be considered in each iteration of the algorithm, and in terms of the number of previously computed splittable segments that must be stored. We proposed a new dynamic programming algorithm for computing a lower bound on the number of candidate splits that must be considered, thereby providing a method for finite sample analysis of the time complexity of the algorithm. We analyzed several real data sets in terms of the number of candidate splits that were computed, and compared those empirical observations to the upper and lower bounds. We observed that in those real data sets, the empirical number of candidate splits is only a constant factor larger than the lower bound, indicating that the binary segmentation algorithm achieves the asymptotic best case in practical real data scenarios. Our analysis provides convincing evidence that binary segmentation can be quickly computed for large real data sets, and large model sizes. Our work therefore suggests that the speed of binary segmentation is close to best case in several real data settings, $O(N \log K)$ time for N data and K segments. Our empirical comparisons indicate slower speeds for wild binary segmentation [3]. One drawback/limitation of our work was the time complexity of our proposed dynamic programming algorithm, which was quadratic in both number of data and splits. For future work, we plan to investigate faster algorithms for computing the best case lower bound, using ideas from the optimal binary search tree literature [19].

Funding: This research received no external funding.

Data Availability Statement: An R package with C/C++ code that implements our proposed algorithm is available at https://github.com/tdhock/binsegRcpp (accessed 2 Sept 2026). In addition, the GitHub repository https://github.com/tdhock/binseg-model-selection contains code for reproducing the figures in this paper.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Olshen, A.B.; Venkatraman, E.; Lucito, R.; Wigler, M. Circular binary segmentation for the analysis of array-based DNA copy number data. *Biostatistics* **2004**, *5*, 557–572.
2. Killick, R.; Eckley, I.A.; Ewans, K.; Jonathan, P. Detection of changes in variance of oceanographic time-series using changepoint analysis. *Ocean Eng.* **2010**, *37*, 1120–1126.
3. Fryzlewicz, P. Wild binary segmentation for multiple change-point detection. *Ann. Stat.* **2014**, *42*, 2243–2281.
4. Jewell, S.W.; Hocking, T.D.; Fearnhead, P.; Witten, D.M. Fast nonconvex deconvolution of calcium imaging data. *Biostatistics* **2020**, *21*, 709–726.
5. Auger, I.; Lawrence, C. Algorithms for the optimal identification of segment neighborhoods. *Bull. Math. Biol.* **1989**, *51*, 39–54.
6. Jackson, B.; Scargle, J.; Barnes, D.; Arabhi, S.; Alt, A.; Gioumousis, P.; Gwin, E.; Sangtrakulcharoen, P.; Tan, L.; Tsai, T. An algorithm for optimal partitioning of data on an interval. *IEEE Signal Process Lett.* **2005**, *12*, 105–108.
7. Maidstone, R.; Hocking, T.; Rigaill, G.; Fearnhead, P. On optimal multiple changepoint algorithms for large data. *Stat. Comput.* **2017**, *27*, 519–533.
8. Scott, A.; Knott, M. A cluster analysis method for grouping means in the analysis of variance. *Biometrics* **1974**, *30*, 507–512.
9. Baranowski, R.; Chen, Y.; Fryzlewicz, P. Narrowest-over-threshold detection of multiple change points and change-point-like features. *J. R. Stat. Soc. Ser. B (Stat. Methodol.)* **2019**, *81*, 649–672.
10. Kovács, S.; Li, H.; Bühlmann, P.; Munk, A. Seeded binary segmentation: A general methodology for fast and optimal change point detection. *arXiv* **2020**, arXiv:2002.06633.
11. Venkatraman, E. Consistency Results in Multiple Change-Point Situations. Unpublished Ph.D. Thesis, Department of Statistics, Stanford University, Stanford, CA, USA, 1992.
12. Levin, B.; Kline, J. The cusum test of homogeneity with an application in spontaneous abortion epidemiology. *Stat. Med.* **1985**, *4*, 469–488.
13. Breiman, L.; Friedman, J.H.; Olshen, R.A.; Stone, C.J. *Classification and Regression Trees*; Routledge: London, UK, 1984.
14. Drouin, A.; Hocking, T.; Laviolette, F. Maximum Margin Interval Trees. In *Advances in Neural Information Processing Systems 30*; Guyon, I., Luxburg, U.V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., Garnett, R., Eds.; Curran Associates, Inc.: New York, NY, USA, 2017; pp. 4947–4956.
15. Yao, Y.C. Estimating the number of change-points via Schwarz' criterion. *Stat. Probab. Lett.* **1988**, *6*, 181–189.
16. Zhang, N.R.; Siegmund, D.O. A Modified Bayes Information Criterion with Applications to the Analysis of Comparative Genomic Hybridization Data. *Biometrics* **2007**, *63*, 22–32.
17. Rigaill, G.; Hocking, T.; Vert, J.P.; Bach, F. Learning sparse penalties for change-point detection using max margin interval regression. In Proceedings of the 30th International Conference on Machine Learning, Atlanta, GA, USA, 16–21 June 2013; pp. 172–180.
18. Truong, C.; Gudre, L.; Vayatis, N. Penalty learning for changepoint detection. In *Proceedings of the 2017 25th European Signal Processing Conference (EUSIPCO)*; IEEE: New York, NY, USA, 2017; pp. 1569–1573.
19. Nagaraj, S. Optimal binary search trees. *Theor. Comput. Sci.* **1997**, *188*, 1–44. https://doi.org/https://doi.org/10.1016/S0304-3975(96)00320-9.
20. Knuth, D.E. Optimum binary search trees. *Acta Inform.* **1971**, *1*, 14–25. https://doi.org/10.1007/BF00264289.
21. Mehlhorn, K. Nearly optimal binary search trees. *Acta Inform.* **1975**, *5*, 287–295.
22. Hocking, T.D.; Goerner-Potvin, P.; Morin, A.; Shao, X.; Pastinen, T.; Bourque, G. Optimizing ChIP-seq peak detectors using visual labels and supervised machine learning. *Bioinformatics* **2016**, *33*, 491–499.
23. Hocking, T.D. *Neuroblastoma: Neuroblastoma Copy Number Profiles*; R Package Version 1.0; 2013.
24. Amoakohene, D.; Chetia, A.; Steinmacher, I.; Hocking, T. Asymptotic benchmarking using the atime package. *R J.* **2026**, *18*, 170–188.