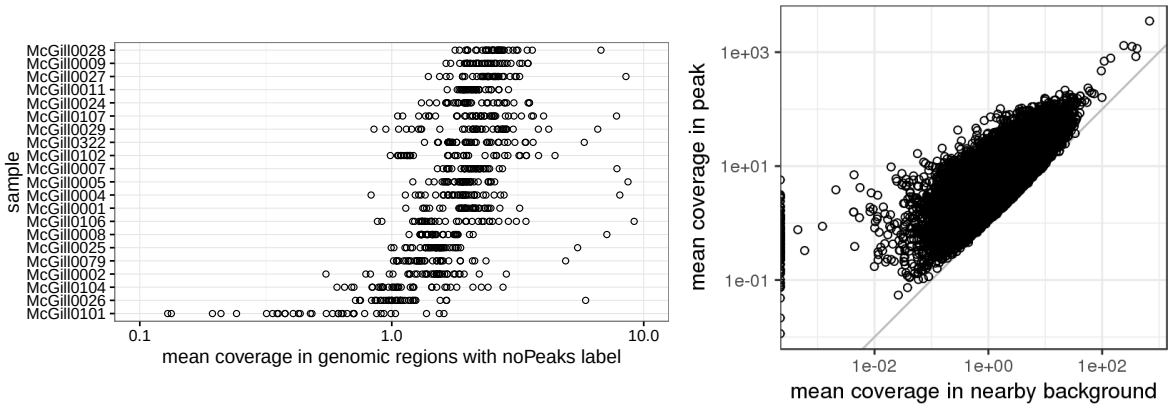
Supplementary material for "Constrained dynamic programming and supervised penalty learning algorithms for peak detection in genomic data"

November 4, 2019

Supplementary Figure 1: background level varies across a sample



Hidden Markov Models have a uniform background/noise mean level, rather than a new segment mean for every background segment (as in the model we proposed). We have observed different mean levels in different background/noise regions in the same ChIP-seq sample, which suggests that the uniform background/noise mean model would not be appropriate in these data. **Left**: we computed mean coverage in each sample and genomic region with the noPeaks label (which means a biologist has observed that there are no peaks in that region, so it must only contain background noise), and observed that the mean is highly variable between regions of the genome. For example in sample McGill0028, the mean coverage in background/noise regions ranges from 1.78 to 6.76, which suggests that a uniform/constant mean would not be a good model of these data. **Right**: for one sample, we computed peaks throughout the genome, then computed mean coverage in each peak and mean coverage in nearby background. We observed that the mean of the nearby background increases as the mean of the peak increases, which suggests that a uniform/constant mean would not be a good model of these data.

Supplementary Figure 2: details of example where CDPA fails

Consider computing the 5 segment up-down constrained model for the set of 6 data points $\mathbf{y} = [3, 9, 18, 15, 20, 2]$ using the Poisson loss $\ell(y_i, m) = m - y_i \log m$.

- The unconstrained PDPA computes the model $\mathbf{m} = [3, 9, 16.5, 16.5, 20, 2]$ which has a total Poisson loss of ≈ -109.8827. Its two increasing changes followed by two decreasing changes are not feasible for the up-down constrained problem.
- The up-down constrained GPDPA computes the model $\mathbf{m} = [6, 6, 18, 15, 20, 2]$ which has a total Poisson Loss of ≈ -108.4495. Each up change is followed by a down change, so it is feasible for the up-down constrained problem.
- The CDPA returns no feasible model with 5 segments.

To see why the CDPA fails, we give the detailed calculations of the GPDPA and CDPA below. The first cost function computed by the GPDPA is the Poisson loss of the first data point:

$$C_{1,1}(u_1) = \ell(3, u_1) = u_1 - 3 \log u_1 \tag{1}$$

The minimum of $C_{1,1}$ is at 3, so its min-less operator is convex for $u_2 \leq 3$, and constant for $u_2 \geq 3$:

$$C^{\leq}_{1,1}(u_2) = \begin{cases} C_{1,1}(u_2) = u_2 - 3 \log u_2 & \text{if } u_2 \in [2, 3],\ u_1 = u_2 \\ C_{1,1}(3) = -0.296 & \text{if } u_2 \in [3, 20],\ u_1 = 3 \end{cases} \tag{2}$$

The second cost function is the total Poisson loss of the first two data points:

$$C_{1,2}(u_1) = \ell(9, u_1) + C_{1,1}(u_1) = 2u_1 - 12 \log u_1 \tag{3}$$

The minimum of $C_{1,2}$ is at 6, so its min-less operator is convex for $u_2 \leq 6$, and constant for $u_2 \geq 6$:

$$C^{\leq}_{1,2}(u_2) = \begin{cases} C_{1,2}(u_2) = 2u_2 - 12 \log u_2 & \text{if } u_2 \in [2, 6],\ u_1 = u_2 \\ C_{1,2}(6) = -9.501 & \text{if } u_2 \in [6, 20],\ u_1 = 6 \end{cases} \tag{4}$$

The best cost in 2 segments up to data point 2 is:

$$C_{2,2}(u_2) = \ell(9, u_2) + C^{\leq}_{1,1}(u_2) = \begin{cases} 2u_2 - 12 \log u_2 & \text{if } u_2 \in [2, 3],\ u_1 = u_2 \\ u_2 - 9 \log u_2 - 0.296 & \text{if } u_2 \in [3, 20],\ u_1 = 3 \end{cases} \tag{5}$$

Note in the equation above that a non-decreasing change between data points 1 and 2 is enforced by the min-less operator $C^{\leq}_{1,1}$.

The best cost in 2 segments up to data point 3 is defined as:

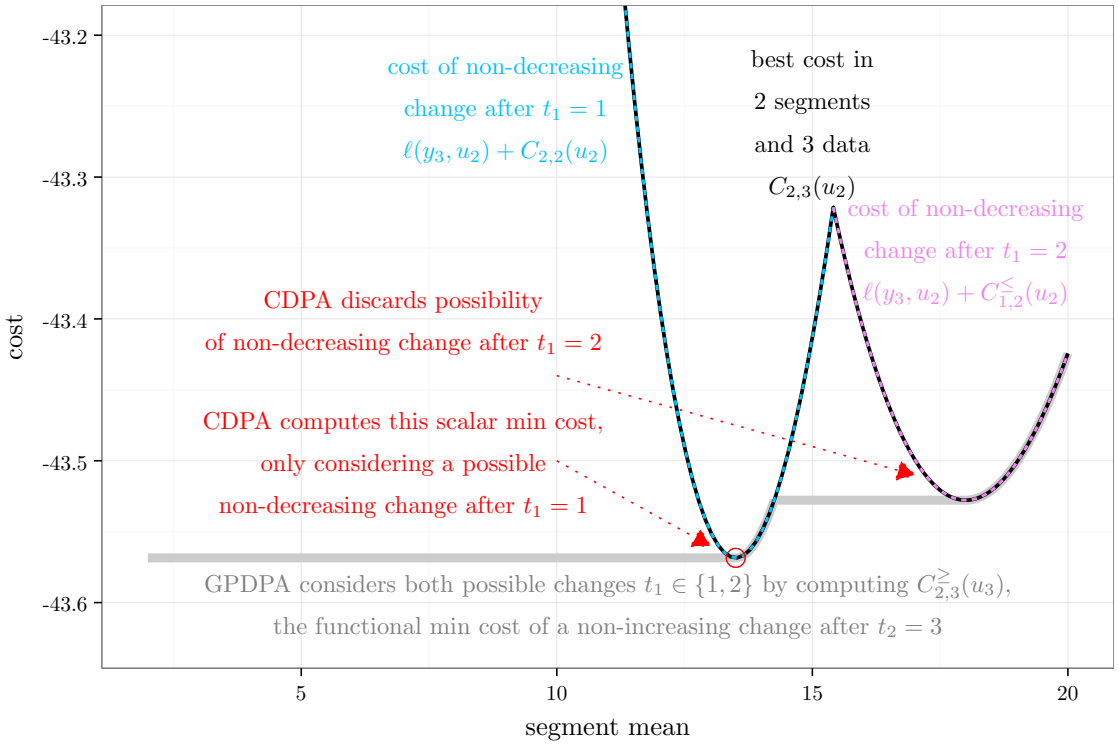
$$C_{2,3}(u_2) = \ell(18, u_2) + \min \begin{cases} C_{2,2}(u_2) & \text{if } t_1 = 1 \\ C^{\leq}_{1,2}(u_2) & \text{if } t_1 = 2 \end{cases} \tag{6}$$

The GPDPA computes the roots of $C_{2,2}(u_2) - C^{\leq}_{1,2}(u_2)$ in order to find their intersection at $u_2 \approx 15.41$, so the best cost in 2 segments up to data point 3 simplifies to:

$$C_{2,3}(u_2) = \ell(18, u_2) + \begin{cases} C_{2,2}(u_2) & \text{if } u_2 \in [2, 15.41],\ t_1 = 1 \\ C^{\leq}_{1,2}(u_2) & \text{if } u_2 \in [15.41, 20],\ t_1 = 2 \end{cases} \tag{7}$$

$$= \begin{cases} 3u_2 - 30\log u_2 & \text{if } u_2 \in [2, 3],\ u_1 = u_2,\ t_1 = 1 \\ 2u_2 - 27\log u_2 - 0.296 & \text{if } u_2 \in [3, 15.41],\ u_1 = 3,\ t_1 = 1 \\ u_2 - 18\log u_2 - 9.501 & \text{if } u_2 \in [15.41, 20],\ u_1 = 6,\ t_1 = 2 \end{cases} \tag{8}$$

This cost function is shown as the black curve in the figure below. The part on the left ($u_2 \leq 15.41$) in blue is the cost of a non-decreasing change after the first data point ($t_1 = 1$). The part on the right in violet ($u_2 \geq 15.41$) is the cost of a non-decreasing change after the second data point ($t_2 = 2$).



The GPDPA then computes the functional min cost $C^{\geq}_{2,3}(u_3)$ for all possible values of the mean parameter u_3 (shown as grey function in plot above):

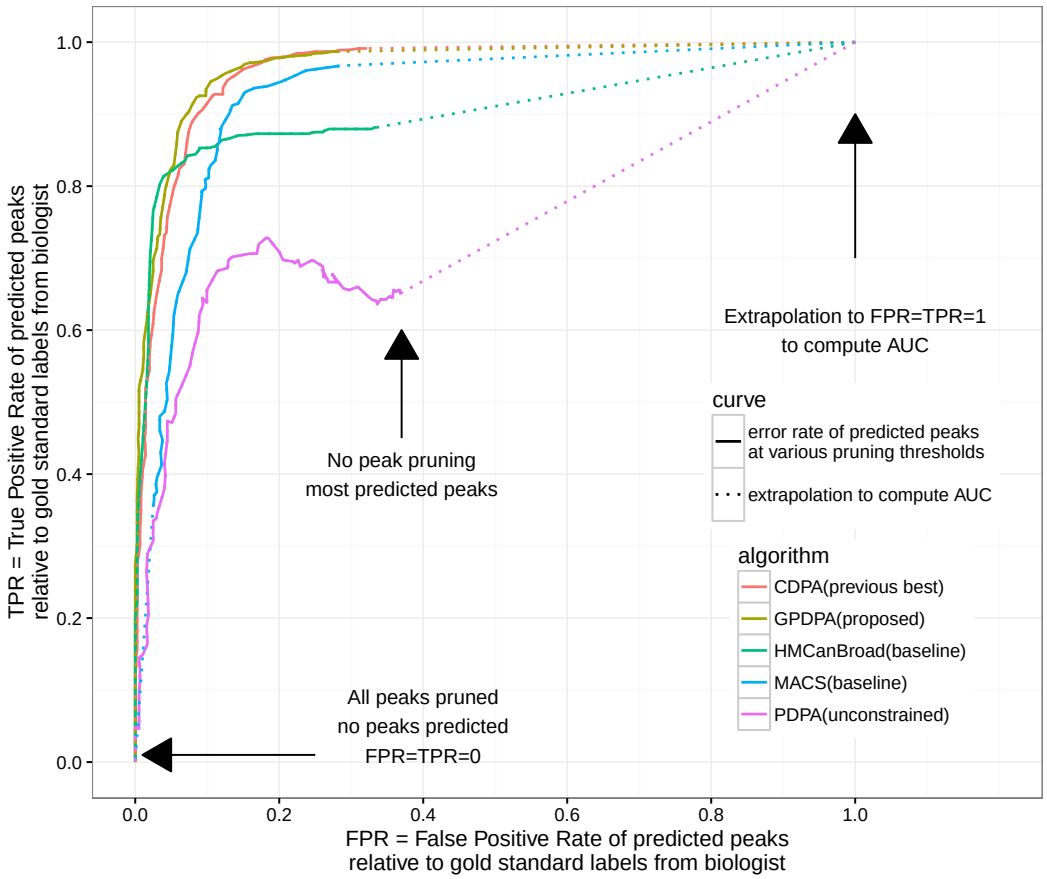
$$C^{\geq}_{2,3}(u_3) = \begin{cases} -43.57 & \text{if } u_3 \in [2, 13.5],\ u_2 = 13.5,\ t_1 = 1 \\ 2u_3 - 27\log u_3 - 0.296 & \text{if } u_3 \in [13.5, 14.25],\ u_2 = u_3,\ t_1 = 1 \\ -43.53 & \text{if } u_3 \in [14.25, 18],\ u_2 = 18,\ t_1 = 2 \\ u_3 - 18\log u_3 - 9.501 & \text{if } u_3 \in [18, 20],\ u_2 = u_3,\ t_1 = 2 \end{cases} \tag{9}$$

Since the optimal cost is computed for both possible changepoints, and all possible mean values, the GPDPA is able to compute the optimal solution (which occurs at $u_3 = 15, u_2 = 18, u_1 = 6, t_1 = 2$, but is unknown until the algorithm computes the total cost of all the data points $C_{5,6}$). In contrast, the CDPA computes a scalar min cost of a non-decreasing change after the first data point (red circle in the plot above, $u_2 = 13.5, u_1 = 3, t_1 = 1$), and discards the possibility of a non-decreasing change after the second data point (which ends up being optimal). The CDPA is thus a greedy algorithm.

Supplementary Figure 3: ROC curves for peak detection accuracy

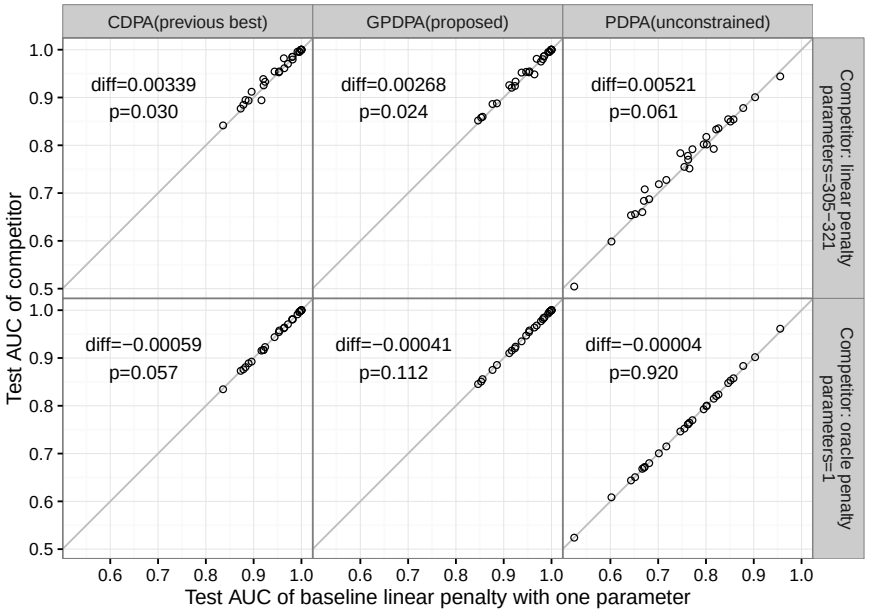
We wanted to compare the peak detection accuracy of our proposed algorithm with others from the bioinformatics literature, which typically report many false positive peaks using default parameter settings. Therefore in a typical analysis, to control the false positive rate, the default peak list is pruned by only considering the top p peaks, according to some likelihood or significance threshold. For example, the MACS algorithm uses a q-value threshold parameter, and HMCanBroad uses a finalThreshold parameter (higher thresholds result in more false positives).

To account for this pruning step in our evaluation, we used Receiver Operating Characteristic (ROC) curve analysis. For each threshold parameter, we computed the false positive rate and true positive rate using the labels, which results in one point on the ROC curve. The area under the curve (AUC) is computed by varying the threshold parameter over its entire range (from a complete list of peaks with many false and true positives, to a completely pruned/empty list of peaks with FPR=TPR=0). Note that even with the largest number of predicted peaks in each model, not all labels achieve their max possible TP and FP, because the largest peak list does not necessarily predict peaks in all labels. We therefore linearly extrapolate each ROC curve to TPR=FPR=1 in order to compute AUC (dotted lines in plot below).



Note that the ROC curves are not necessarily monotonic, because the peak pruning is not necessarily hierarchical. For example the GPDPA computes optimal models from 0 to 9 peaks for each problem; the peak present in the optimal model with 1 peak may not be present in the optimal model with 2 or more peaks.

Supplementary Figure 4: test AUC comparison between penalty functions

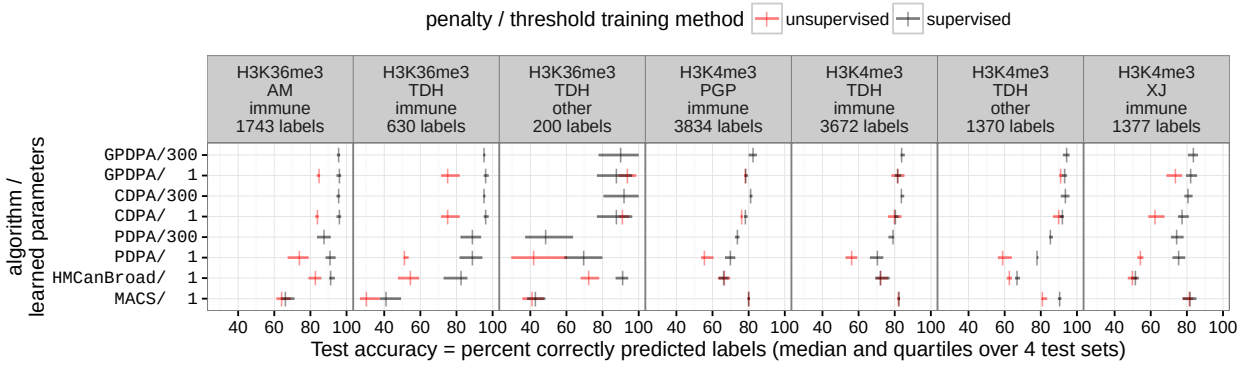


We proposed to select the number of segments in changepoint models using two kinds of model complexity functions $\mathcal{C}$. The oracle model complexity is a relatively complex expression motivated by statistical arguments; the linear model complexity simply measures the number of segments. In this section we compare these methods in terms of test AUC.

We consider learning a one-parameter penalty function consisting of a constant penalty λ, given either the linear or oracle model complexity function. We expected the oracle model complexity to result in higher test AUC, because of its statistical motivation. The second row of Supplementary Figure 4 plots the distribution of test AUC values for one model complexity function versus the other. Contrary to our expectation, for all three changepoint algorithms (CDPA, GPDPA, PDPA), the linear and oracle penalties showed no significant difference in test AUC (mean difference from 0.00004 to 0.0006, p-value > 0.05 in paired t_{27}-test). These data indicate that, despite the statistical arguments that motivate the oracle model complexity, it is not more accurate than the simple linear model complexity for peak detection in real genomic data.

We also compare learning a penalty function with either one or multiple parameters, given the linear model complexity function. We expected higher test AUC for the penalty function with multiple parameters. The first row of Supplementary Figure 4 plots the distribution of test AUC values for one penalty function versus the other. In agreement with our expectation, for all three changepoint algorithms, the multi-parameter penalty function has a significantly larger test AUC (mean difference from 0.003 to 0.005, p-value < 0.07). These data indicate that learning a multi-parameter penalty function should be preferred for accurate peak detection in genomic data.

Supplementary Figure 5: test accuracy for supervised and unsupervised models



Four-fold cross-validation was used to estimate peak prediction accuracy. Each panel shows one of seven ChIP-seq data sets, labeled by pattern/experiment (Broad H3K36me3), labeler (AM), cell types (immune), and number of labels. It is clear that supervised models are generally more accurate than unsupervised, and multi-parameter models are generally more accurate than single parameter models.

1 Supplementary Text 1

1.1 GPDPA pseudocode

In this section we give pseudocode for our proposed Generalized Pruned Dynamic Programming Algorithm (GPDPA). We propose the following data structures and sub-routines for the computation:

- FunctionPiece: a data structure which represents one piece of a $C_{k,t}(u)$ cost function (for one interval of mean values u). It has coefficients which depend on the convex loss function ℓ (for the square loss it has three real-valued coefficients a, b, c which define a function au^2+bu+c). It also has two real-valued elements for min/max mean values $[\underline{u}, \overline{u}]$ of this interval, meaning the function $C_{k,t}(u) = au^2 + bu + c$ for all $u \in [\underline{u}, \overline{u}]$. Finally it stores a previous segment endpoint t' (integer) and mean u' (real).

- FunctionPieceList: an ordered list of FunctionPiece objects, which exactly stores a cost function $C_{k,t}(u)$ for all values of last segment mean u.

- OnePiece$(y, \underline{u}, \overline{u})$: a sub-routine that initializes a FunctionPieceList with just one FunctionPiece $\ell(y, u)$ defined on $[\underline{u}, \overline{u}]$.

- MinLess(t, f): an algorithm that inputs a changepoint and a FunctionPieceList, and outputs the corresponding min-less operator $f^{\leq}$ (another FunctionPieceList), with the previous changepoint set to $t' = t$ for each of its pieces. This algorithm also needs to store the previous mean value u' for each of the function pieces (see pseudocode below).

- MinOfTwo(f_1, f_2): an algorithm that inputs two FunctionPieceList objects, and outputs another FunctionPieceList object which is their minimum.

- ArgMin(f): an algorithm that inputs a FunctionPieceList and outputs three values: the optimal mean $u^* = \arg\min_u f(u)$, the previous segment end t' and mean u'.

- FindMean(u, f) an algorithm that inputs a mean value and a FunctionPieceList. It finds the FunctionPiece in f with mean $u \in [\underline{u}, \overline{u}]$ contained in its interval, then outputs the previous segment end t' and mean u' stored in that FunctionPiece.

The above data structures and sub-routines are used in the following pseudo-code, which describes the GPDPA for solving the up-down constrained changepoint problem with K segments.

Algorithm 1 Generalized Pruned Dynamic Programming Algorithm (GPDPA)

1: Input: data set $\mathbf{y} \in \mathbb{R}^n$, maximum number of segments $K \in \{2, \dots, n\}$.
2: Output: matrices of optimal segment means $U \in \mathbb{R}^{K \times K}$ and ends $T \in \{1, \dots, n\}^{K \times K}$
3: Compute min $\underline{y}$ and max $\overline{y}$ of $\mathbf{y}$.
4: $C_{1,1} \leftarrow \text{OnePiece}(y_1, \underline{y}, \overline{y})$
5: for data points t from 2 to n:
6: $\quad C_{1,t} \leftarrow \text{OnePiece}(y_t, \underline{y}, \overline{y}) + C_{1,t-1}$
7: for segments k from 2 to K: for data points t from k to n: // dynamic programming
8: $\quad \text{MinLessOrMore} \leftarrow \text{MinLess}$ if k is even, else MinMore
9: $\quad \text{min\_prev} \leftarrow \text{MinLessOrMore}(t-1, C_{k-1,t-1})$
10: $\quad \text{min\_new} \leftarrow \text{min\_prev}$ if $t = k$, else $\text{MinOfTwo}(\text{min\_prev}, C_{k,t-1})$
11: $\quad C_{k,t} \leftarrow \text{min\_new} + \text{OnePiece}(y_t, \underline{y}, \overline{y})$
12: for segments k from 1 to K: // decoding for every model size k
13: $\quad u^*, t', u' \leftarrow \text{ArgMin}(C_{k,n})$
14: $\quad U_{k,k} \leftarrow u^*$; $T_{k,k} \leftarrow t'$ // store mean of segment k and end of segment $k-1$
15: $\quad$ for segment s from $k-1$ to 1: // decoding for every segment $s < k$
16: $\quad\quad$ if $u' < \infty$: $u^* \leftarrow u'$ // equality constraint active, $u_s = u_{s+1}$
17: $\quad\quad t', u' \leftarrow \text{FindMean}(u^*, C_{s,t'})$
18: $\quad\quad U_{k,s} \leftarrow u^*$; $T_{k,s} \leftarrow t'$ // store mean of segment s and end of segment $s-1$

Algorithm 1 begins by computing the min/max on line 3. The main storage of the algorithm is $C_{k,t}$, which should be initialized as a $K \times n$ array of empty FunctionPieceList objects. The computation of $C_{1,t}$ for all t occurs on lines 4–6.

The dynamic programming updates occur in the for loops on lines 7–11. Line 9 uses the MinLess (or MinMore) sub-routine to compute the temporary FunctionPieceList min_prev (which represents the function $C^{\leq}_{k-1,t-1}$ or $C^{\geq}_{k-1,t-1}$). Line 10 sets the temporary FunctionPieceList min_new to the cost of the only possible changepoint if $t = k$; otherwise, it uses the MinOfTwo sub-routine to compute the cost of the best changepoint for every possible mean value. Line 11 adds the cost of data point t, and stores the resulting FunctionPieceList in $C_{k,t}$.

The decoding of the optimal segment mean U (a $K \times K$ array of real numbers) and end T (a $K \times K$ array of integers) variables occurs in the for loops on lines 12–18. For a given model size k, the decoding begins on line 13 by using the ArgMin sub-routine to solve $u^* = \arg\min_u C_{k,n}(u)$ (the optimal values for the previous segment end t' and mean u' are also returned). Now we know that u^* is the optimal mean of the last (k-th) segment, which occurs from data point $t'+1$ to n. These values are stored in $U_{k,k}$ and $T_{k,k}$ (line 14). And we already know that the optimal mean of segment $k-1$ is u'. Note that the $u' = \infty$ flag means that the equality constraint is active (line 16). The decoding of the other segments $s < k$ proceeds using the FindMean sub-routine (line 17). It takes the cost $C_{s,t'}$ of the best model in s segments up to data point t', finds the FunctionPiece that stores the cost of u^*, and returns the new optimal values of the previous segment end t' and mean u'. The mean of segment s is stored in $U_{k,s}$ and the end of segment $s-1$ is stored in $T_{k,s}$ (line 18).

The time complexity of Algorithm 1 is $O(KnI)$ where I is the complexity of the MinLess and MinOfTwo sub-routines, which is linear in the number of intervals (FunctionPiece objects) that are used to represent the cost functions. There are pathological synthetic data sets for which the number of intervals $I = O(n)$, implying a time complexity of $O(Kn^2)$. However, the average number of intervals in real data sets is empirically $I = O(\log n)$, so the average time complexity of Algorithm 1 is $O(Kn \log n)$.

1.2 Min-less algorithm

The following sub-routines are used to implement the MinLess sub-routine.

- GetCost(p, u): an algorithm that takes a FunctionPiece object p, and a mean value u, and computes the cost at u. For a square loss FunctionPiece p with coefficients $a, b, c \in \mathbb{R}$, we have GetCost$(p, u) = au^2 + bu + c$.
- OptimalMean(p): an algorithm that takes one FunctionPiece object, and computes the optimal mean value. For a square loss FunctionPiece p we have OptimalMean$(p) = -b/(2a)$.
- ComputeRoots(p, d): an algorithm that takes one FunctionPiece object, and computes the solutions to $p(u) = d$. For the square loss we propose to use the quadratic formula. For other convex losses that do not have closed form expressions for their roots, we propose to use Newton's root finding method. Note that for some constants d there are no roots, and the algorithm needs to report that.
- f.push_piece$(\underline{u}, \overline{u}, p, u')$: push a new FunctionPiece at the end of FunctionPieceList f, with coefficients defined by FunctionPiece p, on interval $[\underline{u}, \overline{u}]$, with previous segment mean set to u'.
- ConstPiece(c): sub-routine that initializes a FunctionPiece p with constant cost c (for the square loss it sets $a = b = 0$ in $au^2 + bu + c$).

Pseudocode for the MinLess sub-routine is given below. The MinMore sub-routine is similar.

Algorithm 2 MinLess algorithm.

Input: The previous segment end t_{prev} (an integer), and f_{in} (a FunctionPieceList).
Output: FunctionPieceList f_{out}, initialized as an empty list.
prev_cost $\leftarrow \infty$
new_lower_limit $\leftarrow$ LowerLimit($f_{\text{in}}[0]$).
$i \leftarrow 0$; // start at FunctionPiece on the left
while $i <$ Length(f_{in}): // continue until FunctionPiece on the right
  FunctionPiece $p \leftarrow f_{\text{in}}[i]$
  if prev_cost $= \infty$: // look for min in this interval.
    candidate_mean $\leftarrow$ OptimalMean(p)
    if LowerLimit(p) $<$ candidate_mean $<$ UpperLimit(p):
      new_upper_limit $\leftarrow$ candidate_mean // Minimum found in this interval.
      prev_cost $\leftarrow$ GetCost(p, candidate_mean)
      prev_mean $\leftarrow$ candidate_mean
    else: // No minimum in this interval.
      new_upper_limit $\leftarrow$ UpperLimit(p)
    f_{out}.push_piece(new_lower_limit, new_upper_limit, p, ∞)
    new_lower_limit $\leftarrow$ new_upper_limit
    $i \leftarrow i + 1$
  else: // look for equality of p and prev_cost
    (small_root, large_root) $\leftarrow$ ComputeRoots(p, prev_cost)
    if LowerLimit(p) $<$ small_root $<$ UpperLimit(p):
      f_{out}.push_piece(new_lower_limit, small_root, ConstPiece(prev_cost), prev_mean)
      new_lower_limit $\leftarrow$ small_root
      prev_cost $\leftarrow \infty$
    else: // no equality in this interval
      $i \leftarrow i + 1$ // continue to next FunctionPiece
if prev_cost $< \infty$: // ending on constant piece
  f_{out}.push_piece(new_lower_limit, UpperLimit(p), ConstPiece(prev_cost), prev_mean)
Set all previous segment end $t' = t_{\text{prev}}$ for all FunctionPieces in f_{out}

Consider Algorithm 2 which contains pseudocode for the computation of the min-less operator. The algorithm initializes prev_cost (line 3), which is a state variable that is used on line 8 to decide whether the algorithm should look for a local minimum or an intersection with a finite cost. Since prev_cost is initially set to ∞, the algorithm begins by following the convex function pieces from left to right until finding a local minimum. If no minimum is found in a given convex FunctionPiece (line 15), it is simply pushed on to the end of the new FunctionPieceList (line 16). If a minimum occurs within an interval (line 10), the cost and mean are stored (lines 11–12), and a new convex FunctionPiece is created with upper limit ending at that mean value (line 16). Then the algorithm starts looking for another FunctionPiece with the same cost, by computing the smaller root of the convex loss function (line 20). When a FunctionPiece is found with a root in the interval (line 21), a new constant FunctionPiece is pushed (line 22), and the algorithm resumes searching for a minimum. At the end of the algorithm, a constant FunctionPiece is pushed if necessary (line 28). The complexity of this algorithm is $O(I)$ where I is the number of FunctionPiece objects in f_{in}.

The algorithm which implements the min-more operator is analogous. Rather than searching from left to right, it searches from right to left. Rather than using the small root (line 21), it uses the large root.

1.3 Implementation details

Some implementation details that we found to be important:

Weights for data sequences that contain repeats it is computationally advantageous to use a run-length encoding of the data, and a corresponding loss function. For example if the data sequence 5,1,1,1,0,0,5,5 is encoded as $n = 4$ counts y_t 5,1,0,5 with corresponding weights w_t 1,3,2,2 then the Poisson loss function for mean μ is $\ell(y_t, w_t, \mu) = w_t(\mu - y_t \log \mu)$.

Mean cost The text defines $C_{k,t}$ functions as the total cost. However for very large data sets the cost values will be very large, resulting in numerical instability. To overcome this issue we instead implemented update rules using the mean cost. For weights $W_t = \sum_{i=1}^{t} w_i$, the update rule to compute the mean cost is

$$C_{k,t}(\mu) = \frac{1}{W_t}\left[\ell(y_t, \mu) + W_{t-1} \min\{C_{k,t-1}(\mu),\, C^{\leq}_{k-1,t-1}(\mu)\}\right]$$

Intervals in log(mean) space For the Poisson model of non-negative count data $y_t \in \{0, 1, 2, \dots\}$ there is no possible mean μ value less than 0. We thus used $\log(\mu)$ values to implement intervals in FunctionPiece objects. For example rather than storing $\mu \in [0, 1]$ we store $\log \mu \in [-\infty, 0]$.

Root finding For the ComputeRoots sub-routine for the Poisson loss, we used Newton root finding. For the larger root we solve $a \log \mu + b\mu + c = 0$ (linear as $\mu \to \infty$) and for the smaller root we solve $ax + be^x + c = 0$ ($x = \log \mu$, linear as $x \to -\infty$ and $\mu \to 0$). We stop the root finding when the cost is near zero (absolute cost value less than 10^{-12}).

Storage Since the dynamic programming update rule for $C_{k,t}$ only depends on $C^{\geq}_{k-1,t-1}$ and $C_{k,t-1}$, these are the only functions that need to be in memory, and the rest of the cost functions can be stored on disk (until the decoding step).