Definition of the breakpointError

Toby Dylan Hocking

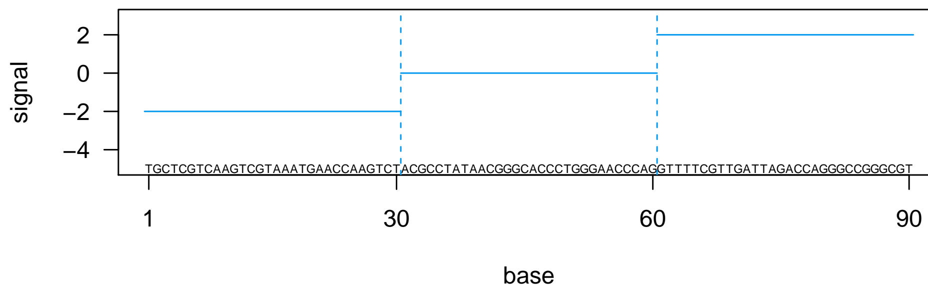
October 25, 2016

This vignette discusses the mathematical definition of the breakpointError which can be calculated using this R package. Given a latent piecewise constant signal $\mu \in \mathbb{R}^D$ defined on bases $1, \dots, D$, we can calculate the positions $B \subseteq \{1, \dots, D-1\}$ after which it changes. We use these breakpoint positions B to define a precise breakpointError function that can be used to quantify the accuracy of a set of breakpoint guesses $G \subseteq \{1, \dots, D-1\}$. The breakpointError was originally introduced as the exact breakpoint error by Hocking [2012, Chapter 4].

1 Setup: recovering breakpoints from noisy observations

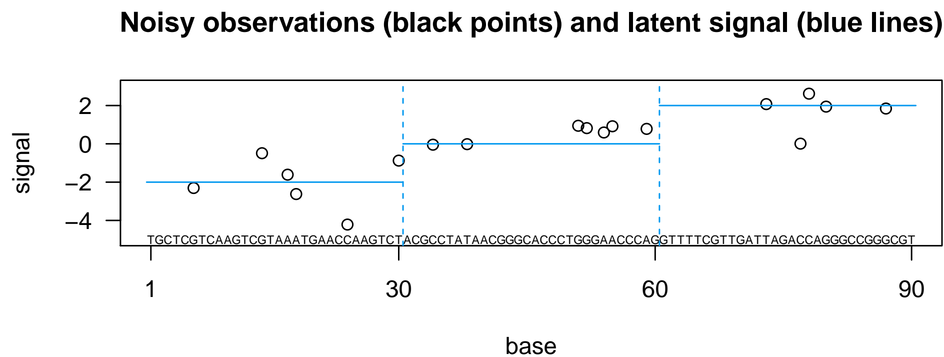
We assume there is a chromosome with D base pairs. Let $\mathcal{X} = \{1, \dots, D\}$ be all the base pairs, and let $\mathbb{B} = \{1, \dots, D-1\}$ be all bases after which a break is possible. In simulations, we assume there is some latent piecewise constant signal $\mu \in \mathbb{R}^D$ defined at each of those bases.

```
> plotsig("Latent signal (horizontal lines) and breakpoints (vertical dashed lines)")
> print(latent.segs)

  first last mean
1     1   30   -2
2    31   60    0
3    61   90    2

> ## The segment from base i to base j is drawn from i-1/2 to j+1/2.
> with(latent.segs,segments(first-1/2,mean,last+1/2,mean,col=signal.colors["latent"]))
> ## And if there is a break after base i, it should be drawn at i+1/2.
> abline(v=latent.breaks+1/2,col=signal.colors["latent"],lty="dashed")
```



We sample some noisy signal $y \in \mathbb{R}^d$ at base positions $p \in \mathcal{X}^d$.

```
> set.seed(1)
> d <- 18
> base <- sort(sample(seq_along(latent), d))
> signal <- rnorm(d, latent[base])
> plotsig("Noisy observations (black points) and latent signal (blue lines)")
> points(base, signal)
> with(latent.segs,segments(first-1/2,mean,last+1/2,mean,col=signal.colors["latent"]))
> abline(v=latent.breaks+1/2,col=signal.colors["latent"],lty="dashed")
```



In the plot above, the latent signal μ is drawn using blue lines, and the noisy signal (p, y) is drawn using black points. We will use models to estimate the latent signal, given only the noisy observations.

We will focus on the cghseg model. First, order the vectors of observations such that the positions are in increasing order $p_1 < \cdots < p_d$. Then, we define the estimated signal with k segments as

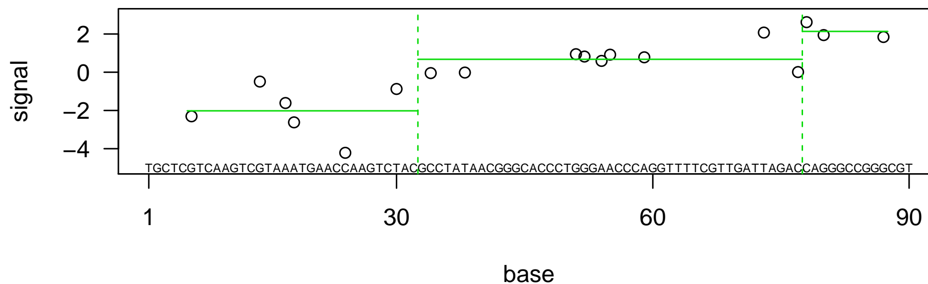
$$\begin{aligned} \hat{y}^k = & \operatorname*{arg\,min}_{x \in \mathbb{R}^d} && ||y - x||_2^2 \\ & \text{subject to} && k - 1 = \sum_{j=1}^{d-1} 1_{x_j \neq x_{j+1}}. \end{aligned} \tag{1}$$

Note that we can quickly calculate $\hat{y}^k$ for $k \in \{1, \dots, k_{\max}\}$ using pruned dynamic programming [Rigaill, 2010]. This is implemented in the R package **cghseg**, and the **breakpointError** package includes the run.cghseg function which returns a list L of results. The estimated segments can be found as a data.frame in L$segments.

```
> plotsig("Noisy observations (black points) and estimated signal (green lines)")
> points(base, signal)
> kmax <- 6
> L <- run.cghseg(signal, base, kmax)
> k <- 3
> yhat <- subset(L$segments,segments==k)
> print(yhat)
```

```
  first.index last.index first.base last.base      mean segments
4           1          6        5.5      32.5 -2.019129        3
5           7         15       32.5      77.5  0.676018        3
6          16         18       77.5      87.5  2.135967        3
```

```
> with(yhat,segments(first.base,mean,last.base,mean,col=signal.colors["estimate"]))
> abline(v=yhat$first.base[-1],col=signal.colors["estimate"],lty="dashed")
```



The cghseg model tells us the points after which a break occurs, not the bases. So we define the estimated breakpoint locations shown as vertical green dashed lines using the mean

$$\phi(\hat{y}^k, p) = \big\{ \lfloor (p_j + p_{j+1})/2 \rfloor \text{ for all } j \in \{1, \dots, d-1\} \text{ such that } \hat{y}_j^k \neq \hat{y}_{j+1}^k \big\}. \tag{2}$$

Note that this is a function $\phi : \mathbb{R}^d \times \mathcal{X}^d \to 2^{\mathbb{B}}$ that gives the positions after which there is a break in $\hat{y}^k$.

```
> print(L$breaks[[k]])
```

```
[1] 32 77
```

For the cghseg model with k segments, let $G_k = \phi(\hat{y}^k, p) \subseteq \mathbb{B}$ denote the estimated positions after which a break occured. These can be found in L$breaks as a list of $k_{\max}$ vectors.

```
> str(L$breaks)

List of 6
 $ : int(0)
 $ : int 32
 $ : int [1:2] 32 77
 $ : int [1:3] 21 27 77
 $ : int [1:4] 21 27 44 77
 $ : int [1:5] 17 21 27 44 77
```

We would like to compare these estimated breakpoints to the exact set of breakpoints in the simulated signal

$$B = \phi\left(\mu, \begin{bmatrix} 1 & \cdots & D \end{bmatrix}'\right) = \{j \in \mathbb{B} : \mu_j \neq \mu_{j+1}\}. \tag{3}$$

```
> print(latent.breaks)

[1] 30 60
```

The breakpointError package defines a function $E : 2^{\mathbb{B}} \rightarrow \mathbb{R}^+$ based on the latent breakpoints. Given a guess of the breakpoints $G \subseteq \mathbb{B}$, we quantify its error with $E(G)$. We can then select the number of segments $k \in \{1, \dots, k_{\max}\}$ which minimizes the error $E(G_k)$.

2 Properties of an ideal error function for breakpoint detection

Given some guess of the breakpoint locations $G \subseteq \mathbb{B}$, we would like to to define a function $E(G)$ that quantifies how bad the breakpoint location guess was. We would like the function $E : 2^{\mathbb{B}} \rightarrow \mathbb{R}^+$ to satisfy:

- **(correctness)** Guessing exactly right costs nothing: $E(B) = 0$.
- **(precision)** A guess closer to a real breakpoint is less costly:
  if $B = \{b\}$ and $0 \leq i < j$, then $E(\{b + i\}) \leq E(\{b + j\})$ and $E(\{b - i\}) \leq E(\{b - j\})$.
- **(FP)** False positive breakpoints are bad: if $b \in B$ and $g \notin B$, then $E(\{b\}) < E(\{b, g\})$.
- **(FN)** Undiscovered breakpoints are bad: $b \in B \Rightarrow E(\{b\}) < E(\emptyset)$.

When the latent signal is available in simulations, we can use the exact breakpoint locations B to define the breakpointError E, which satisfies all 4 properties.

3 Definition of the breakpointError for simulated signals

In this section, we use the exact breakpoint locations B to define a breakpoint detection error function.

We define the error of a breakpoint location guess $g \in \mathbb{B}$ as a function of the closest breakpoint in B. So first we put the breaks in order, by writing them as $B_1 < \cdots < B_n$, with each $B_i \in \mathbb{B}$. Then, we define a set of intervals $R_B = \{r_1, \dots, r_n\}$ that form a partition of $\mathbb{B}$. For each breakpoint B_i we define the region $r_i = [\underline{r}_i, \overline{r}_i] \in \mathbb{IB}$, where $\mathbb{IB} \subset 2^{\mathbb{B}}$ denotes the set of all intervals of $\mathbb{B}$. We take the notation conventions from the interval analysis literature [Nakao et al., 2010].

We define the right limit of region i as

$$\overline{r}_i = \begin{cases} D-1 & \text{if } i = n \\ \lfloor (B_{i+1} + B_i)/2 \rfloor & \text{otherwise} \end{cases} \tag{4}$$

and the left limit as

$$\underline{r}_i = \begin{cases} 1 & \text{if } i = 1 \\ \overline{r}_{i-1} + 1 & \text{otherwise.} \end{cases} \tag{5}$$

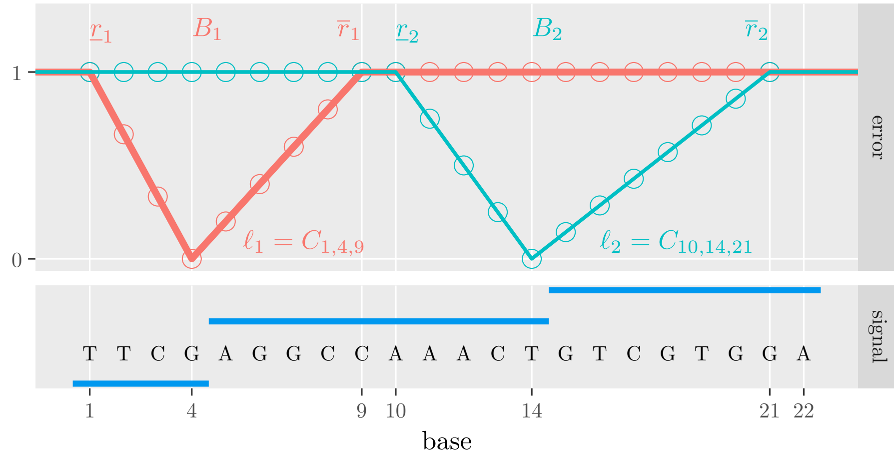
Intuitively, if we observe a breakpoint guess $g \in r_i$, then its closest breakpoint is B_i. To define the best guess in each region, we use piecewise linear functions $C_{\underline{r},b,\overline{r}} : \mathbb{R} \rightarrow [0, 1]$ defined as follows:

$$C_{\underline{r},b,\overline{r}}(g) = \begin{cases} 0 & \text{if } g = b \\ (b-g)/(x-\underline{r}) & \text{if } \underline{r} < g < b \\ (g-b)/(\overline{r}-x) & \text{if } b < g < \overline{r} \\ 1 & \text{otherwise.} \end{cases} \tag{6}$$

For each breakpoint i we measure the precision of a guess $g \in \mathbb{B}$ using

$$\ell_i(g) = C_{\underline{r}_i, B_i, \overline{r}_i}(g). \tag{7}$$

These functions are shown in the figure below for a small signal with 2 breakpoints. Additionally, the breakpoints B_i and regions r_i are labeled. The signal $\mu \in \mathbb{R}^{22}$ has 2 breakpoints: $B = \{4, 14\}$.



Now, we are ready to define the exact breakpointError of a set of guesses $G \subseteq \mathbb{B}$. First, let $G \cap r$ be the subset of guesses G that fall in region r.

Then, we define the false negative rate for region r as

$$\text{FN}(G, r) = \begin{cases} 1 & \text{if } G \cap r = \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

and the false positive rate for region r as

$$\text{FP}(G, r) = \begin{cases} 0 & \text{if } G \cap r = \emptyset \\ |G \cap r| - 1 & \text{otherwise} \end{cases} \tag{9}$$

and the imprecision of the best guess in region r as

$$I(G, r, \ell) = \begin{cases} 0 & \text{if } G \cap r = \emptyset \\ \min_{g \in G \cap r} \ell(g) & \text{otherwise.} \end{cases} \tag{10}$$

When there are no breakpoints, we have $B = \emptyset$ and $R_B = \emptyset$. But we still would like to quantify the false positives, so let $G \setminus \big(\cup R_B\big)$ be the set of guesses G outside of the breakpoint regions R_B. Finally, we define the exact breakpointError of guess G with respect to the true breakpoints B as

$$E(G) = \big|G \setminus (\cup R_B)\big| + \sum_{i=1}^{|B|} \text{FP}(G, r_i) + \text{FN}(G, r_i) + I(G, r_i, \ell_i). \tag{11}$$

To calculate the exact breakpoint error, we first sort lists of $n = |B|$ and $m = |G|$ items. Using the quicksort algorithm, this requires $O(n \log n + m \log m)$ operations in the average case [Cormen et al., 1990]. Once sorted, the components of the cost can be calculated in linear time $O(n+m)$. So, overall the calculation of the error can be accomplished in best case $O(n + m)$, average case $O(n \log n + m \log m)$ operations. It is implemented in the errorDetails function in `berr/pkg/src/breakpointError.c`.

4 R functions for calculating the breakpointError

There are several ways to calculate the breakpointError. The simplest is via the breakpointError function, which takes the guesses, latent breakpoints, and latent signal size. It returns the breakpointError of the guesses as a numeric scalar.

```
> breakpointError(L$breaks[[3]], latent.breaks, length(latent))
```

```
[1] 0.7195402
```

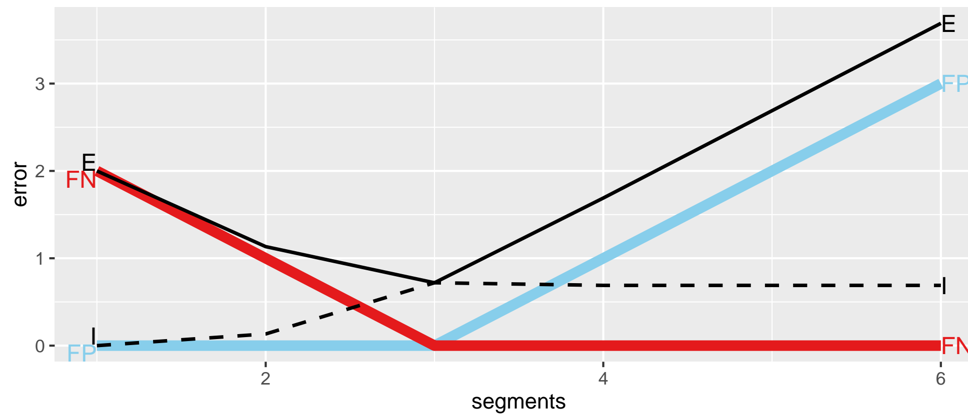
```
> sapply(L$breaks, breakpointError, latent.breaks, length(latent))
```

```
[1] 2.0000000 1.1333333 0.7195402 1.6896552 2.6896552 3.6896552
```

Note that the minimum breakpointError occurs for $k = 3$ segments, or 2 breakpoints, which is expected for the signal with 3 segments that we saw earlier.

For a bit more detail, the errorComponents function can be used to get the FP, FN, and I components for each model.

```
> e <- errorComponents(L$breaks, latent.breaks, length(latent))
> library(reshape2)
> dcast(e,segments~type,value.var="error")

  segments FP FN         E         I
1        1  0  2 2.0000000 0.0000000
2        2  0  1 1.1333333 0.1333333
3        3  0  0 0.7195402 0.7195402
4        4  1  0 1.6896552 0.6896552
5        5  2  0 2.6896552 0.6896552
6        6  3  0 3.6896552 0.6896552

> library(ggplot2)
> p <- ggplot(e,aes(segments,error))+
+   geom_line(aes(size=type,colour=type,linetype=type))+
+   scale_linetype_manual(values=fp.fn.linetypes)+
+   scale_colour_manual(values=fp.fn.colors)+
+   scale_size_manual(values=fp.fn.sizes)
> library(directlabels)
> dl <- direct.label(p+guides(linetype="none",colour="none",size="none"),
+                    dl.combine("first.qp","last.qp"))
> print(dl)
```



For extreme detail, the errorDetails function can be used. It returns a list with several components, which count the error of each break region and guess.

```
> str(errorDetails(L$breaks[[2]], latent.breaks, length(latent)))

List of 8
 $ breaks            : int [1:2] 30 60
 $ guess             : int 32
 $ false.positive    : num [1:2] 0 0
 $ false.negative    : num [1:2] 0 1
 $ imprecision       : num [1:2] 0.133 0
 $ guess.unidentified: num 0
 $ left              : int [1:2] 1 46
 $ right             : int [1:2] 45 89
```

```
> str(errorDetails(L$breaks[[4]], latent.breaks, length(latent)))

List of 8
 $ breaks            : int [1:2] 30 60
 $ guess             : int [1:3] 21 27 77
 $ false.positive    : num [1:2] 1 0
 $ false.negative    : num [1:2] 0 0
 $ imprecision       : num [1:2] 0.103 0.586
 $ guess.unidentified: num [1:3] 0 0 0
 $ left              : int [1:2] 1 46
 $ right             : int [1:2] 45 89
```

The guess.unidentified component will be positive only when there are no real breaks.

```
> str(errorDetails(c(2,6,7), c(), 10))

List of 8
 $ breaks            : int(0)
 $ guess             : int [1:3] 2 6 7
 $ false.positive    : num(0)
 $ false.negative    : num(0)
 $ imprecision       : num(0)
 $ guess.unidentified: num [1:3] 1 1 1
 $ left              : int(0)
 $ right             : int(0)
```

References

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, Cambridge, Massachusetts, second edition, 1990.

T. D. Hocking. *Learning algorithms and statistical software, with applications to bioinformatics*. PhD thesis, Ecole Normale Superieure de Cachan, France, 2012.

M. T. Nakao, A. Neumaier, S. M. Rump, S. P. Shary, and P. van Hentenryck. Standardized notation in interval analysis. `http://www.mat.univie.ac.at/ neum/papers.html`, 2010.

G. Rigaill. Pruned dynamic programming for optimal multiple change-point detection. arXiv:1004.0887, 2010.