The bumphunter user's guide

Kasper Daniel Hansen khansen@jhsph.edu
Martin Aryee aryee.martin@mgh.harvard.edu
Rafael A. Irizarry rafa@jhu.edu

Modified: November 23, 2012. Compiled: October 13, 2014

Introduction

This package implements the statistical procedure described in [2] (with some small modifications). Notably, batch effect removal and the application of the bootstrap to linear models of Efron and Tibshirani [1] need additional code.

For any given type of data, it is usually necessary to make a number of choices and/or transformations before the bump hunting methodology is ready to be applied. Typically, these modifications resides in other packages. Examples are charm (for CHARM-like methylation microarrays), bsseq (for whole-genome bisulfite sequencing data) and minfi (for Illumina 450k methylation arrays). In some cases (specifically bsseq) only parts of the methodology as implemented in the bumphunter package is applied, although the conceptual approach is still build on bump hunting.

In other words, this package is mostly intended for developers wishing to adapt the general methodology to their specific applications.

The core of the package is encapsulated in the bumphunter method which uses the underlying bumphunterEngine to do the heavy lifting. However, bumphunterEngine consists of a number of useful functions that does much of the specific tasks involved in bump hunting. This document attempts to describe the overall workflow as well as the specific functions. The relevant functions are clusterMaker, getSegments, findRegions.

```
> library(bumphunter)
```

Note that this package is written with genomic data as an illustrative example but most of it is easily generalizable to other data types.

Other functions

Most of the `bumphunter` package is code for bump hunting. But we also include a number of convenience functions we have found useful, which are not necessarily part of the bump hunting exercise. At the time of writing, this include `annotateNearest`.

The Statistical Model

The bump hunter methodology is meant to work on data with several biological replicates, similar to the `lmFit` function in `limma`. While the package is written using genomic data as an illustrative example, most of it is generalizable to other data types (with some one-dimensional location information).

We assume we have data Y_{ij} where i represents (biological) replicate and l_j represents genomic location. The use of j and l_j is a convenience notation, allowing us to discuss the "next" observation $j+1$ which may be some distance $lj+1-l_j$ away. Note that we assume in this notation that all replicates have been observed at the same set of genomic locations.

The basic statistical model is the following:

$$Y_{ij} = \beta_0(l_j) + \beta_1(l_j)X_j + \varepsilon_{ij}$$

with i representing subject, l_j representing the jth location, X_j is the covariate of interest (for example $X_j = 1$ for cases and $X_j = 0$ otherwise), ε_{ij} is measurement error, $\beta_0(l)$ is a baseline function, and $\beta_1(l)$ is the parameter of interest, which is a function of location. We assume that $\beta_1(l)$ will be equal to zero over most of the genome, and we want to identify stretched where $\beta_1(l) \neq 0$, which we call *bumps*.

We want to share information between nearby locations, typically through some form of smoothing.

Creating clusters

For many genomic applications the locations are clustered. Each cluster is a distinct unit where the model fitting will be done separately, and each cluster does not depend on the data, only on the locations l_j. Typically there is some maximal distance, and we do not want to smooth between observations separated by more than this distance. The choice of distance is very application dependent.

"Clusters" are simply groups of locations such that two consecutive locations in the cluster are separated by less than some distance `maxGap`. For genomic applications, the biggest possible clusters are chromosomes.

The function `clusterMaker` defines such grouping locations.

Example: We first generate an example of typical genomic locations

```
> pos <- list(pos1=seq(1,1000,35),
+             pos2=seq(2001,3000,35),
+             pos3=seq(1,1000,50))
> chr <- rep(paste0("chr",c(1,1,2)), times = sapply(pos,length))
> pos <- unlist(pos, use.names=FALSE)
```

Now we run the function to obtain the three clusters from the positions. We use the default gap of 300 base pairs (bps), i.e. any two points more than 300 bps away are put in a new cluster. Also, locations from different chromosomes are separated.

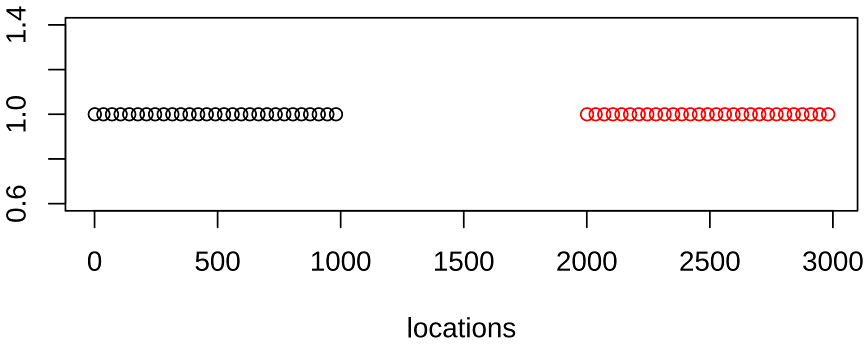
```
> cl <- clusterMaker(chr, pos, maxGap = 300)
> table(cl)
```

```
cl
 1  2  3 
29 29 20 
```

The output is an indexing variable telling us which cluster each location belongs to. Locations on different chromosomes are always on different clusters.

Note that data from the first chromosome has been split into two clusters:

```
> ind <- which(chr=="chr1")
> plot(pos[ind], rep(1,length(ind)), col=cl[ind],
+      xlab="locations", ylab="")
```



Breaking into segments

The function `getSegments` is used to find segments that are positive, near zero, and negative. Specifically we have a vector of numbers θ_j with each number associated with a genomic location l_j (thinks either test statistics or estimates of $\beta_i(l)$). A segment is a list of consecutive locations such that all θ_l in the segment are either "positive", "near zero" or "negative". In order to define "positive" etc we need a `cutoff` which is one number L (in which case "near zero" is $[-L, L]$) or two numbers L, U (in which case "near zero" is $[L; U]$).

Example: we are going to create a simulated $\beta_1(l)$ with a couple of real bumps.

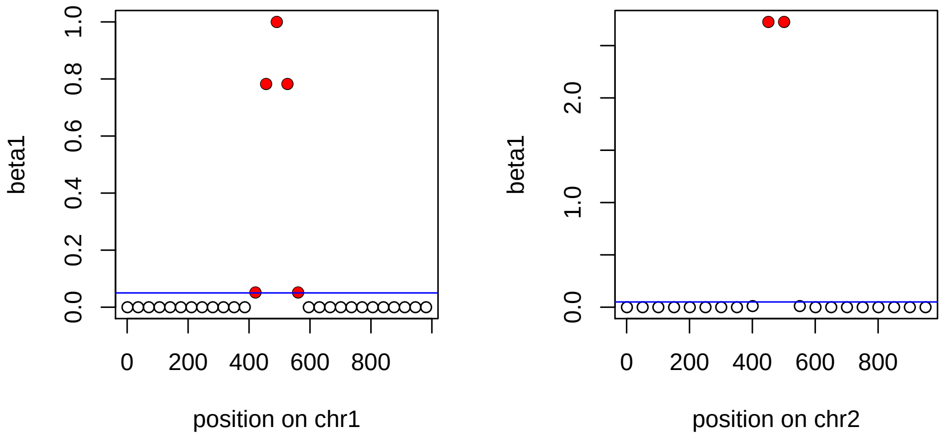
```
> Indexes <- split(seq_along(cl), cl)
> beta1 <- rep(0, length(pos))
> for(i in seq(along=Indexes)){
+     ind <- Indexes[[i]]
+     x <- pos[ind]
+     z <- scale(x, median(x), max(x)/12)
+     beta1[ind] <- i*(-1)^(i+1)*pmax(1-abs(z)^3,0)^3 ##multiply by i to vary size
+ }
```

We now find bumps of this functions by

```
> segs <- getSegments(beta1, cl, cutoff=0.05)
```

Now we can make, for example, a plot of all the positive bumps

```
> par(mfrow=c(1,2))
> for(ind in segs$upIndex){
+     index <- which(cl==cl[ind[1]])
+     plot(pos[index], beta1[index],
+          xlab=paste("position on", chr[ind[1]]),
+          ylab="beta1")
+     points(pos[ind], beta1[ind], pch=16, col=2)
+     abline(h = 0.05, col = "blue")
+ }
```



This function is used by regionFinder which is described next.

regionFinder

This function packages up the results of `getSegments` into a table of regions with the location and characteristics of bumps.

```
> tab <- regionFinder(beta1, chr, pos, cl, cutoff=0.05)
> tab
```

```
   chr start  end      value      area cluster indexStart indexEnd  L
3 chr1  2281 2701 -1.2636037 16.426848       2         38       50 13
2 chr2   451  501  2.7262463  5.452493       3         68       69  2
1 chr1   421  561  0.5336474  2.668237       1         13       17  5
  clusterL
3       29
2       20
1       29
```

In the plot in the preceding section we show two of these regions in red.

Note that `regionFinder` and `getSegments` do not really contain any statistical model. All it does is finding regions based on segmenting a vector θ_j associated with genomic locations l_j.

Bumphunting

`Bumphunter` is a more complicated function. In addition to `regionFinder` and `clusterMaker` it also implements a statistical model as well as permutation testing to assess uncertainty.

We start by creating a simulated data set of 10 cases and 10 controls (recall that `beta1` was defined above).

```
> beta0 <- 3*sin(2*pi*pos/720)
> X <- cbind(rep(1,20), rep(c(0,1), each=10))
> error <- matrix(rnorm(20*length(beta1), 0, 1), ncol=20)
> y <- t(X[,1])%x%beta0 + t(X[,2])%x%beta1 + error
```

Now we can run `bumphunter`

```
> tab <- bumphunter(y, X, chr, pos, cl, cutoff=.5)
> tab
```

```
$table
    chr start  end      value       area cluster indexStart indexEnd
13 chr1  2351 2701 -1.6355668 17.9912351       2         40       50
```

```
8  chr2   451  501  2.7237451  5.4474901       3         68        69
2  chr1   491  526  1.4692630  2.9385259       1         15        16
14 chr2   901  901 -0.9320635  0.9320635       3         77        77
12 chr1  2036 2036 -0.7418113  0.7418113       2         31        31
3  chr1   736  736  0.6984470  0.6984470       1         22        22
11 chr1   911  911 -0.6843332  0.6843332       1         27        27
7  chr2   301  301  0.6444073  0.6444073       3         65        65
6  chr2   201  201  0.6095003  0.6095003       3         63        63
9  chr1   141  141 -0.5861420  0.5861420       1          5         5
4  chr1   981  981  0.5838711  0.5838711       1         29        29
5  chr1  2876 2876  0.5540799  0.5540799       2         55        55
10 chr1   351  351 -0.5533324  0.5533324       1         11        11
1  chr1   386  386  0.5502981  0.5502981       1         12        12
    L clusterL
13 11       29
8   2       20
2   2       29
14  1       20
12  1       29
3   1       29
11  1       29
7   1       20
6   1       20
9   1       29
4   1       29
5   1       29
10  1       29
1   1       29

$coef
              [,1]
 [1,] -0.366362256
 [2,]  0.094598901
 [3,] -0.386382529
 [4,] -0.274731401
 [5,] -0.586141987
 [6,] -0.201445895
 [7,] -0.193903955
 [8,] -0.010976310
 [9,]  0.028613927
[10,] -0.089806673
[11,] -0.553332357
```

```
[12,]  0.550298052
[13,]  0.307700473
[14,]  0.168614073
[15,]  1.295059624
[16,]  1.643466286
[17,] -0.158516646
[18,] -0.130426445
[19,]  0.135686027
[20,]  0.281868129
[21,]  0.382731419
[22,]  0.698447049
[23,] -0.313192294
[24,]  0.280784544
[25,] -0.367361043
[26,]  0.191818806
[27,] -0.684333202
[28,] -0.384884748
[29,]  0.583871117
[30,]  0.363181599
[31,] -0.741811306
[32,] -0.229147567
[33,]  0.123795446
[34,]  0.185098890
[35,] -0.215696767
[36,] -0.406131206
[37,]  0.312903536
[38,] -0.287451571
[39,] -0.217894850
[40,] -0.706520749
[41,] -1.834160944
[42,] -2.159037019
[43,] -2.356109681
[44,] -1.758863338
[45,] -2.209116354
[46,] -1.969546915
[47,] -2.106921701
[48,] -1.463597489
[49,] -0.815192008
[50,] -0.612168926
[51,]  0.480472732
[52,] -0.019938569
[53,]  0.119017050
```

```
[54,]  0.295718629
[55,]  0.554079930
[56,] -0.205140340
[57,] -0.110435158
[58,]  0.103557520
[59,] -0.030781576
[60,] -0.027979144
[61,]  0.023034736
[62,] -0.211490838
[63,]  0.609500315
[64,]  0.271758742
[65,]  0.644407271
[66,]  0.060264242
[67,]  0.090264148
[68,]  2.570123053
[69,]  2.877367074
[70,] -0.491800509
[71,] -0.293039815
[72,]  0.014243206
[73,]  0.343615447
[74,] -0.273329406
[75,] -0.368317505
[76,] -0.230140387
[77,] -0.932063525
[78,] -0.007320682

$fitted
              [,1]
 [1,] -0.366362256
 [2,]  0.094598901
 [3,] -0.386382529
 [4,] -0.274731401
 [5,] -0.586141987
 [6,] -0.201445895
 [7,] -0.193903955
 [8,] -0.010976310
 [9,]  0.028613927
[10,] -0.089806673
[11,] -0.553332357
[12,]  0.550298052
[13,]  0.307700473
[14,]  0.168614073
```

```
[15,]  1.295059624
[16,]  1.643466286
[17,] -0.158516646
[18,] -0.130426445
[19,]  0.135686027
[20,]  0.281868129
[21,]  0.382731419
[22,]  0.698447049
[23,] -0.313192294
[24,]  0.280784544
[25,] -0.367361043
[26,]  0.191818806
[27,] -0.684333202
[28,] -0.384884748
[29,]  0.583871117
[30,]  0.363181599
[31,] -0.741811306
[32,] -0.229147567
[33,]  0.123795446
[34,]  0.185098890
[35,] -0.215696767
[36,] -0.406131206
[37,]  0.312903536
[38,] -0.287451571
[39,] -0.217894850
[40,] -0.706520749
[41,] -1.834160944
[42,] -2.159037019
[43,] -2.356109681
[44,] -1.758863338
[45,] -2.209116354
[46,] -1.969546915
[47,] -2.106921701
[48,] -1.463597489
[49,] -0.815192008
[50,] -0.612168926
[51,]  0.480472732
[52,] -0.019938569
[53,]  0.119017050
[54,]  0.295718629
[55,]  0.554079930
[56,] -0.205140340
```

```
[57,] -0.110435158
[58,]  0.103557520
[59,] -0.030781576
[60,] -0.027979144
[61,]  0.023034736
[62,] -0.211490838
[63,]  0.609500315
[64,]  0.271758742
[65,]  0.644407271
[66,]  0.060264242
[67,]  0.090264148
[68,]  2.570123053
[69,]  2.877367074
[70,] -0.491800509
[71,] -0.293039815
[72,]  0.014243206
[73,]  0.343615447
[74,] -0.273329406
[75,] -0.368317505
[76,] -0.230140387
[77,] -0.932063525
[78,] -0.007320682

$pvaluesMarginal
[1] NA
```

```
> names(tab)
```

```
[1] "table"           "coef"            "fitted"
[4] "pvaluesMarginal"
```

```
> tab$table
```

```
    chr start  end      value       area cluster indexStart indexEnd
13 chr1  2351 2701 -1.6355668 17.9912351       2         40       50
8  chr2   451  501  2.7237451  5.4474901       3         68       69
2  chr1   491  526  1.4692630  2.9385259       1         15       16
14 chr2   901  901 -0.9320635  0.9320635       3         77       77
12 chr1  2036 2036 -0.7418113  0.7418113       2         31       31
```

```
3   chr1   736  736  0.6984470  0.6984470       1         22        22
11  chr1   911  911 -0.6843332  0.6843332       1         27        27
7   chr2   301  301  0.6444073  0.6444073       3         65        65
6   chr2   201  201  0.6095003  0.6095003       3         63        63
9   chr1   141  141 -0.5861420  0.5861420       1          5         5
4   chr1   981  981  0.5838711  0.5838711       1         29        29
5   chr1  2876 2876  0.5540799  0.5540799       2         55        55
10  chr1   351  351 -0.5533324  0.5533324       1         11        11
1   chr1   386  386  0.5502981  0.5502981       1         12        12
    L clusterL
13 11       29
8   2       20
2   2       29
14  1       20
12  1       29
3   1       29
11  1       29
7   1       20
6   1       20
9   1       29
4   1       29
5   1       29
10  1       29
1   1       29
```

Briefly, the `bumphunter` function fits a linear model for each location (like `lmFit` from the `limma` package), focusing on one specific column (coefficient) of the design matrix. This coefficient of interest is optionally smoothed. Subsequently, a permutation can be used to test is formed for this specific coefficient.

The simplest way to use permutations to create a null distribution is to set `B`. If the number of samples is large this can be set to a large number, such as 1000. Note that this will be slow and we have therefore provided parallelization capabilities. In cases were the user wants to define the permutations, for example cases in which all possible permutations can be enumerated, these can be supplied via the `permutation` argument.

Note that the function permits the matrix `X` to have more than two columns. This can be useful for those wanting to fit models that try to adjust for confounders, such as age and sex. However, when `X` has columns other than those representing an intercept term and the covariate of interest, the permutation test approach is not recommended. The function will run but give a warning. A method based on the bootstrap for linear models of Efron and Tibshirani [1] may be more appropriate but this is not currently implemented.

Faster bumphunting with multiple cores

bumphunter can be speeded up by using multiple cores. We use the foreach package which allows different parallel ”back-ends” that will distribute the computation across multiple cores in a single machine, or across multiple machines in a cluster. The most straightforward usage, illustrated below, involves multiple cores on a single machine. See the foreach documentation for more complex use cases, as well as the packages doParallel and doSNOW (among others). Finally, we use doRNG to ensure reproducibility of setting the seed within the parallel computations.

In order to use the foreach package we need to register a backend, in this case a multicore machine with 2 cores.

```
> library(doParallel)
> registerDoParallel(cores = 2)
```

bumphunter will now automatically use this backend

```
> tab <- bumphunter(y, X, chr, pos, cl, cutoff=.5, B=250, verbose = TRUE)
> tab
```

```
a 'bumps' object with 14 bumps
```

References

[1] B. Efron and R.J. Tibshirani.

[2] Andrew E Jaffe, Peter Murakami, Hwajin Lee, Jeffrey T Leek, M Daniele Fallin, Andrew P Feinberg, and Rafael A Irizarry. Bump hunting to identify differentially methylated regions in epigenetic epidemiology studies. *International Journal of Epidemiology*, 41(1):200–209, 2012.

Cleanup

This is a cleanup step for the vignette on Windows; typically not needed for users.

```
> bumphunter:::foreachCleanup()
```

SessionInfo

- R version 3.1.1 Patched (2014-09-25 r66681), `x86_64-unknown-linux-gnu`
- Locale: `LC_CTYPE=en_US.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_US.UTF-8`, `LC_COLLATE=C`, `LC_MONETARY=en_US.UTF-8`, `LC_MESSAGES=en_US.UTF-8`, `LC_PAPER=en_US.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_US.UTF-8`, `LC_IDENTIFICATION=C`
- Base packages: base, datasets, grDevices, graphics, methods, parallel, stats, stats4, utils
- Other packages: BiocGenerics 0.12.0, GenomeInfoDb 1.2.0, GenomicRanges 1.18.0, IRanges 2.0.0, S4Vectors 0.4.0, bumphunter 1.6.0, doParallel 1.0.8, doRNG 1.6, foreach 1.4.2, iterators 1.0.7, locfit 1.5-9.1, pkgmaker 0.22, registry 0.2, rngtools 1.2.4
- Loaded via a namespace (and not attached): R.methodsS3 1.6.1, XVector 0.6.0, codetools 0.2-9, compiler 3.1.1, digest 0.6.4, grid 3.1.1, lattice 0.20-29, matrixStats 0.10.0, stringr 0.6.2, tools 3.1.1, xtable 1.7-4