Identification of Transcription Factor Binding Sites

N. R. Peterson, M. J. Hubisz, A. L. Martins, and A. Siepel

July 23, 2011

This example will go through the steps of predicting transcription factor binding sites. In this case, we will look for binding sites for the NRSF_01 transcription factor. We begin by initializing RTFBS, extracting example files and loading the Position Weight Matrix that represents the NRSF binding site motif.

```
> require("rtfbs")
> # extract sequence and position weight matrix files from RTFBS package
> exampleArchive <- system.file("extdata", "NRSF.zip", package="rtfbs")
> unzip(exampleArchive, c("input.fas", "pwm.meme"))
> # read the position weight matrices from meme text formatted file
> pwm <- read.pwm("pwm.meme")
> # delete the file since we now have the PWMs in memory
> unlink("pwm.meme")
```

Next, we read in a FASTA file containing sequences that we want to search for potential binding sites. If this were a large FASTA file, we would use the pointer.only=TRUE option to read.ms. This causes the MS object to be stored on the C side, and handled only "by reference" (i.e., using a pointer) in R. It can be much more efficient than passing the entire object between R and C on every function call. These sequences will be represented as a Multiple Sequences (MS) object.

```
> # read in the sequences
> ms <- read.ms("input.fas")
> # delete the file since we now have the sequences in memory
> unlink("input.fas")
```

If the sequences in the FASTA file are very long, it might make sense to split them into smaller sequences, in order to account for local changes in GC content across each sequence. This can be done in two different ways.
1. Splitting the sequences by a given window size (i.e., every 1000 bases)
2. Splitting the sequences into regions as defined by a features file (GFF or BED)

Here we use the first option, and choose to split every 500 bases. Note, this step is optional, and may not be necessary if the original sequences are already short.

```
> msSplit <- split.ms(ms, 500)
```

Splitting by a features file may be useful if the candidate regions (where you wish to search for matches to the motif) are a subset of the sequences that were loaded with read.ms. Searching only the candidate regions can greatly improve power by reducing the number of false positives found in non-candidate regions. To split based on locations in a GFF or BED file, we would first read the file into R as a features object using the read.feat(FullPath) function. Then, when calling split.ms, we would provide the features object instead of a numeric window size.

Now we have our MS object split into segments that are each at most 500 bases long. The next step is to account for GC content. This step is also optional, but it is recommended to avoid false positives due to variation in local GC content. The sequences are grouped into N quantiles according to their GC content. The number of quantiles is a user provided parameter, but in this example we will be using N=4. This step will yield a list containing 4 ms objects. Almost all of the functions in RTFBS operate on single MS objects, not a list of MS objects. To process a list of objects, we could use lapply or loop over each element of the list.

```
> msGroups <- groupByGC.ms(msSplit, 4);
```

The next task is creating a Markov model to represent the null model for each MS object. Since we have split the MS by GC content, there will be a separate null model for each GC quantile. These Markov models will be required in later functions such as score.ms and simulate.ms. We can specify the order of the Markov model, which indicates how many previous bases are used to compute the probability of the next base. In this case we will choose a 3rd order Markov model. build.mm will then compute transition matrices from order 0 to order 3 (the lower order models will be used to simulate/score bases on the boundaries of sequences).

```
> markovModels <- list()
> for (i in 1:length(msGroups))
+   markovModels[[i]] <- build.mm(msGroups[[i]], 3)
```

We can now score all potential binding sites for each MS object, using a Position Weight Matrix and the corresponding Markov model that we just built.

```
> listOfScoresReal <- list()
> for (i in 1:length(msGroups))
+   listOfScoresReal[[i]] <- score.ms(msGroups[[i]], pwm, markovModels[[i]])
```

The scores computed by score.ms represent the log of the probability of observing a sequence assuming it is a binding site, minus the log of the probability of observing the same sequence using the Markov model. By default, score.ms returns all binding sites with scores > 0, though this threshold can be specified by the user.

Now we need to choose a score threshold which will control the number of false positives. We will compute the False Discovery Rate (FDR) by simulating sequences using our Markov models and scoring them. These sequences do not have any real binding sites so this will give us the distribution of scores expected under the null hypothesis. To simulate a sequence, we provide the Markov model, and a simulation length. The simulation length should be approximately the same as the total sequence length of the MS used to create the Markov Model. The total length of the MS object can be determined by running sum(lengths.ms(MSobject)). In this example, we are using 500,000 as our simulation length. This will generate a sequence that is 500,000 bases long based on the Markov model.

```
> simulatedSeqs <- list()
> for (i in 1:length(msGroups))
+   simulatedSeqs[[i]] <- simulate.ms(markovModels[[i]], 500000)
```

Now we use the score.ms() function on our simulated sequences, using the same PWM and corresponding Markov model as before.

```
> listOfScoresSimulated <- list()
> for (i in 1:length(msGroups))
+   listOfScoresSimulated[[i]] <- score.ms(simulatedSeqs[[i]], pwm, markovModels[[i]])
```

Now we can calculate the FDR for each binding site. This function will produce a mapping between each binding site score and its calculated FDR for each MS object. The mapply function is used to calculate the FDR for each MS object.

```
> fdrMap <- list()
> for (i in 1:length(msGroups))
+   fdrMap[[i]] <- calc.fdr(msGroups[[i]], listOfScoresReal[[i]],
+                              simulatedSeqs[[i]], listOfScoresSimulated[[i]])
```

Plotting the FDR vs. score enables us to choose an appropriate FDR threshold value for our binding sites. You can use plot.fdr() to plot the data returned by the calc.fdr function. Or, you can give plot.fdr a list of results from calc.fdr to view multiple curves at once (i.e., curves for each GC bin).

```
> plot.fdr(fdrMap, col=rainbow(4))
```

The FDR plot is shown in Figure 1.

Now we can output candidate binding sites passing a chosen threshold. The output.sites function can take either a score threshold, which might be chosen by looking at the FDR plots. Alternatively, it can take an FDR threshold along with the FDR map computed by calc.fdr. Then it will return a data frame giving all binding sites that pass this threshold.

```
> bindingSites <- data.frame()
> for (i in 1:length(msGroups)) {
+   bindingSites <- rbind(bindingSites,
+                           output.sites(listOfScoresReal[[i]],
+                                        fdrScoreMap=fdrMap[[i]],
+                                        fdrThreshold=0.1))
+ }
```

That's it! Now we have our candidate binding sites at a 10% FDR threshold. output.sites returns the potential binding sites stored as a features object. The results from output.sites can be printed to a file using write.feat().

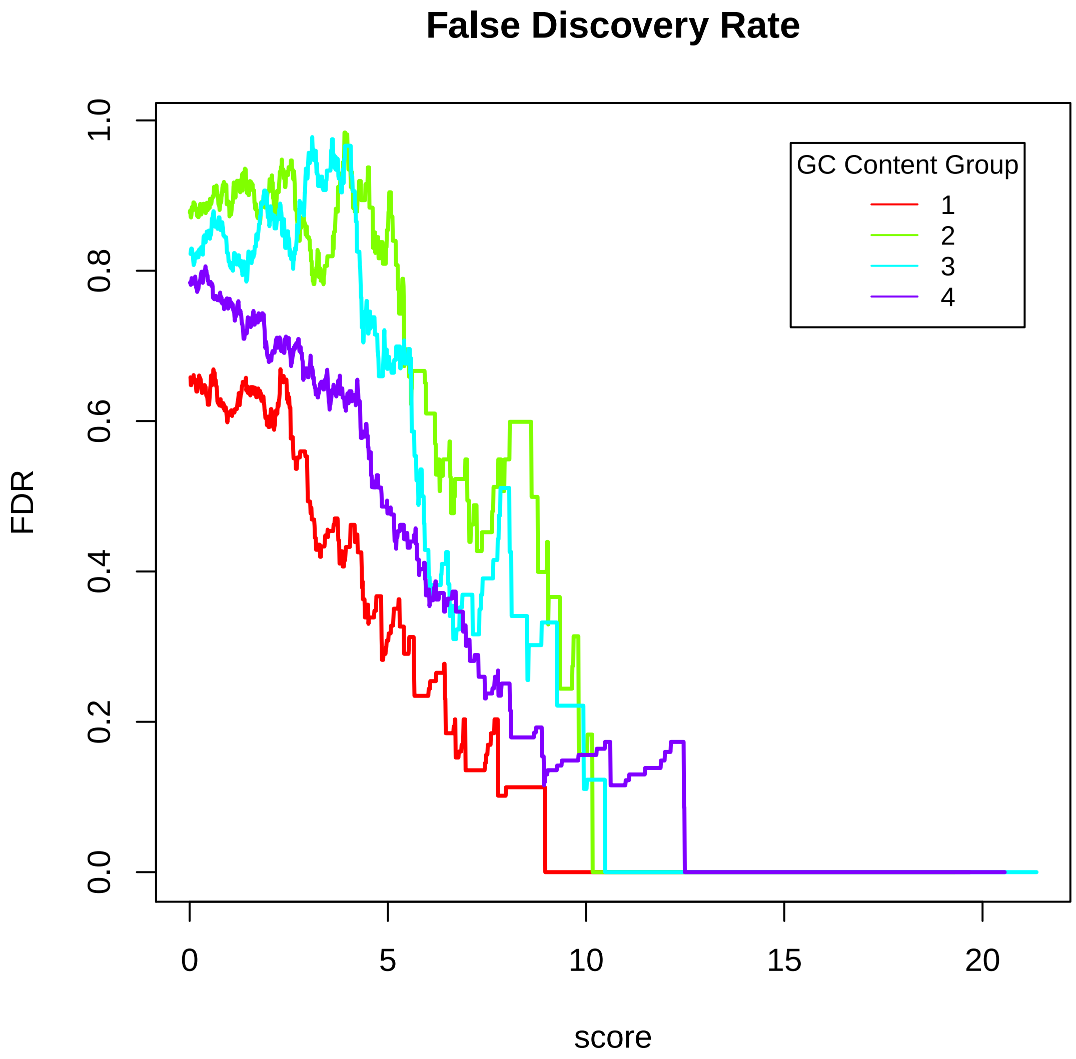

Figure 1: False Discovery Rate compared to Score