The ChIPanalyser User's Guide

Patrick Martin

Introduction - What is this package?

ChIPanalyser provides a quick an easy method to predict and explain TF binding. The package uses a statistical thermodynamic framework to model the binding of proteins to DNA.

The model assumes that there are four main driver to TF binding:

- Chromatin State affinity
- Binding Energy
- Number of bound Molecules
- Scaling Factor modulating binding affinity

While binding energy is given by PWM matrices, the other parameters will be inferred by maximizing or minimizing a goodness of fit score between ChIP data and ChIPanalyser predictions. To do so, ChIPanalyser provides a genetic algorithm to infer optimal values for each parameter.

Methods - The Chromatin State Model

The chromatin state model is described by the following equation derived from statistical thermodynamics:

$$P(N, a, \lambda, \omega)_j = \frac{N \cdot a_j \cdot e^{(\frac{1}{\lambda} \cdot \omega_j)}}{N \cdot a_j \cdot e^{(\frac{1}{\lambda} \cdot \omega_j)} + L \cdot n \cdot [a_i \cdot e^{(\frac{1}{\lambda} \cdot \omega_j)}]_i}$$

- N , the average number of bound molecules
- a_j , chromatin state affinity at site j
- ω , the binding energy required for a TF to bind to site j - in the form of a Position Weight Matrix Score
- λ , a scaling factor for the Position Weight Matrix score
- L , the length of the genome of interest
- n , the ploidy level of the organism

Chromatin state affinity is defined as the following:

$$a_j = \sum_k \alpha_k \cdot c_j^k$$

with α the chromatin state affinity scores for a given TF and c the chromatin state at site j.

Shortly, the model above generates ChIP like profiles and describes the affinity of a TF to chromatin states. The affinity scores a, the number of bound molecules N, and lambda λ are inferred by optimising the fit between predicted profiles and ChIP data using a genetic algortihm.

Using ChIPanalyser

Loading data

To demonstrate the use of ChIPanalyser and the genetic algorithm, we provide some internal data sets. We complement this data with external data source such a DNA sequence sets taken from the `BSgenome`package suit.

ChIPanalyser

First, we load ChIPanlyser and the internal data

```
library(ChIPanalyser)
# Input data
data(ChIPanalyserData)
# PFM Matrix
PFM <- file.path(system.file("extdata",package="ChIPanalyser"),"BEAF-32.pfm")
```

External Data

To reduce the size of the internal data size, we complement our data with the *Drosophila* DNA sequence taken from the BSgenome package suit.

```
library(BSgenome.Dmelanogaster.UCSC.dm6)

DNASequenceSet <- getSeq(BSgenome.Dmelanogaster.UCSC.dm6)
```

Input data : What is it?

Now we that have loaded some data, what exactly are we looking at? The environment should contain the following new objects:

- *chip* - GRanges object containing ChIP scores. Essentially, ChIP data that will be used to *train* our model.
- *cs* - GRanges object containing Chromatin State (CS) information. Essentially, where each CS can be found in the genome.
- *top* - GRanges object containing regions of interest. Esssentially, the genomic regions we will use for training and testing.
- *PFM* - Path to file containing the Position Frequency Matrix. Essentially, this represents the binding affinity of our TF (here we are using BEAF-32)
- *DNASequenceSet* - DNAStringSet containing the DNASequenceSet of the organism we are looking at.
- *geneRef* - GRanges object containing gene reference taken from the (Genome UCSC website)[https://genome.ucsc.edu/]

Setting Parameters

The next step consitst in setting initial parameters to run the genetic algorithm.

```
# Number of individuals per generation
pop <- 10

# Number of generations
gen <- 2
```

```
# Mutation Probability
mut <- 0.3

# Children - Number of ofspring passed to the next generation
child <- 2

# Method - Goodness of fit metric used to optimise the Genetic algorithm
method <- "MSE"
```

Please note that the parameters presented here are for the sake of the vignette. We recommend using the following parameters for a first trial with your own data. Please be advised that these parameters will depend on the nature of the data you are using.

```
# Number of individuals per generation
pop <- 100

# Number of generations
gen <- 50

# Mutation Probability
mut <- 0.3

# Children - Number of ofspring passed to the next generation
child <- 10

# Method - Goodness of fit metric used to optimise the Genetic algorithm
method <- "MSE"
```

We can also define which parameters we wish to optimise. In this example, we will optimise the number of bound molecules (N), lambda (λ), the PWM Threshold and 11 different chromatin states.

```
# Parameters to optimised
params <- c("N","lambda","PWMThreshold", paste0("CS",seq(1:11)))
```

Alternatively, we can also set custom ranges for the parameters. We will use custom ranges here to reduce computational time.

```
params_custom <- vector("list", 14)
names(params_custom) <- c("N","lambda","PWMThreshold", paste0("CS",seq(1:11)))

# vector in the format of min value, max value and number of values
params_custom$N <- c(1,1000000,5)

params_custom$lambda <- c(1,5,5)

# Bound between 0 and 1
params_custom$PWMThreshold <- c(0.1,0.9,5)

# Bound between 0 and 1
CS <- c(0,1,5)
CS_loc <- grep("CS",names(params_custom))
for(i in CS_loc){
    params_custom[[i]] <- CS
}
```

Initializing ChIPanalyser

Building Initial objects

The first step of any ChIPanalyser anaylsis is to create parameter object. These object contain input paramters and hold basic data. Here, we will load the PFM as PWM and compute the Base Pair Frequency from the DNASequenceSet.

```
GPP <- genomicProfiles(PFM=PFM,PFMFormat="JASPAR", BPFrequency=DNASequenceSet)
```

Generating a starting population

For clairty, we will show what a starting population looks like. This step is not stricly required as the `evolve` function can take the number of individuals in the poppulation (defined above by `pop`) and the parameters to be optimised (defined above by `params` or `params_custom`).

```
start_pop <- generateStartingPopulation(pop, params_custom)
```

Pre-processing ChIP data

We will pre-proccess the ChIP data by reducing noise and converting GRanges to a ChIPscore object. This object contains normalised and smoothed ChIP scores that will be used to train and test the model.

```
chipProfile <- processingChIP(chip,loci = top)

# Splitting data into training and testing
# We recommend setting dist to 20/80. However, here we only have 4 loci.
splitdata <- splitData(chipProfile,dist = c(50,50),as.proportion = TRUE)

trainingSet <- splitdata$trainingSet
testingSet <- splitdata$testingSet
```

Evolution

We can run the Genetic algorithm provided by ChIPanalyser using a single function. The function will generate intermediate files that allow you to check the status of the algorithm while it is running. It should be noted that the intermediate files are updated at each generation. This method also provides a lambda database. This allows for a faster run time on larger data sets as all values for lambda are pre-computed.

If you do not want intermediate files, set the `checkpoint` argument to `FALSE`.

If you do not want to pre-compute lambda values, set the `lambda` argument to `FALSE`.

For the purpose of this vignette, we will not save intermediate files.

```
evo <- evolve(population = pop,
    DNASequenceSet = DNASequenceSet,
    ChIPScore = trainingSet,
    genomicProfiles = GPP,
    parameters = params_custom,
    mutationProbability = mut,
    generations = gen,
    offsprings = child,
    chromatinState = cs,
    method = method,
    filename = "This_TF_is_Best_TF",
    checkpoint = FALSE,
    cores= 1)
```

```
## Generating Lambda DataBase

## Generation: 1

## Generation Fitness: 0.0153067666992702

## Generation: 2

## Generation Fitness: 0.0153067666992702
```

The output of this function returns a list of 3 elements:

- *database* - a data frame with all parameters that have been computed through out each generation
- *poppulation* - list containing the last generated population
- *fitest* - list containing goodness of metric for best perfomring individuals

Fitest of them all

Once we have the best perfomring paramters, we can plug them into a single run of ChIPanalyser and use these parameters on a tesing set.

Get fitest individual

The first step is to extract the best performing individual and its associated traits from the population.

```
SuperFit <- getHighestFitnessSolutions(evo$population,
    child = 1,
    method = method)
single<-evo[["population"]][SuperFit]
```

Running ChIPanalyser with fitest individual

We can use these parameters as input to the `singleRun` function to obtain the ChIP like profiles. This run represents the predicted run and TF binding affinity of our TF of choice. We will set fitness to `all`. This means that the function will return all goodness of fit metrics available.

```
# Set chromatin states for single run - create CS Granges with affinity scores
cs_single <- setChromatinStates(single,cs)[[1]]

superFit <- singleRun(indiv = single,
    DNAAffinity = cs_single,
    genomicProfiles = GPP,
    DNASequenceSet = DNASequenceSet,
    ChIPScore = testingSet,
    fitness = "all")
```

```
## Warning in ks.test.default(predicted, locusProfile): p-value will be
## approximate in the presence of ties

## Warning in ks.test.default(predicted, locusProfile): p-value will be
## approximate in the presence of ties
```

This final sections returns a list containing 3 elements:

- *occupnacy* - genomicProfiles object containing occupancy scores for each region
- *ChIP* - genomicProfiles object containing ChIP like scores for each region
- *gof* - goodness of fit scores for each regions and mean scores over all regions.

Plotting

Finally, we plot the resulting profiles.

```
par(mfrow = c(2,1))
plotOccupancyProfile(predictedProfile = superFit$ChIP,
    ChIPScore = testingSet,
    chromatinState = cs_single,
    occupancy = superFit$occupancy,
    goodnessOfFit = superFit$gof,
    geneRef = geneRef,
    addLegend = TRUE)
```