UBMI-IFC, UNAM
Coyoacan, CDMX

Sequece Characterization Test 1

Using Genomic-Benchmarks Data

Author: Fuentes David
Tutors:
- PhD. Poot Augusto
- MSc. Pedraza Carlos

Table of contents

1 Introduction

In this analysis, we explore methods to process, characterize, visualize and classify a preliminary set of activating cis-regulatory sequences from promoters and enhancers. We started by downloading genomic data, preparing it in a compatible format, and applying several custom and existing tools for sequence characterization. The goal was to produce a robust dataset, prepare it for machine learning classification, and conduct exploratory analyses that highlight distinctive sequence features. Below, we outline the data acquisition, preparation, and characterization steps, as well as the libraries and custom functions employed.

Note:

Each time the programming language changes, the code-cell will display a header indicating the corresponding language.

2 Data Preparation

2.1 Downloading data

Although the initial intention was to use sequences from various public databases (like GeneHancer, RefSeq, ENCODE and EPD among others), it was suggested to run some tests with the easy-access data from genomic-benchmarks (*Grešová et al., 2023*), so the first step was to explore all available datasets after downloading the corresponding python package through '`pip install genomic-benchmarks`'.

After that, we have to '`import`' the needed functions to explore the available datasets. It feels noteworthy to mention the importance of internet connectivity as a determining factor of the run-times of the following functions.

Python Code

```python
# To list available datasets
from genomic_benchmarks.data_check import \
    list_datasets

# To inspect each dataset to select two
from genomic_benchmarks.data_check import info as \
    info_gb

# To download each dataset
from genomic_benchmarks.loc2seq import \
    download_dataset

# To position ourselves in the correct directory
import os
```

Due to some cell-output inconveniences explained briefly in the *Custom Functions* Annex and a personal preference for modularity, next we will import '`custom_functions`'. Refer to 'Custom Functions Annex', Section 1, for more details.

```python
# To load the 'custom-functions' module
os.chdir("/path/to/Project/scripts")
from custom_functions import *
```

When displaying the available datasets, we can highlight the presence of four datasets containing human regulatory elements: '*non-TATA* promoters', 'enhancers *Ensembl*', 'enhancers *Cohn*' and '*Ensembl* regulatory':

```python
list_datasets()
```

```
['human_nontata_promoters', 'drosophila_enhancers_stark',
'human_enhancers_cohn', 'human_ensembl_regulatory',
'dummy_mouse_enhancers_ensembl', 'demo_human_or_worm',
'human_ocr_ensembl', 'demo_coding_vs_intergenomic_seqs',
'human_enhancers_ensembl']
```

```python
info_gb("human_nontata_promoters", version=0)
```

```
Dataset `human_nontata_promoters` has 2 classes:
negative, positive.

 All lengths of genomic
intervals equals 251.

 Totally 36131 sequences
have been found, 27097 for training and 9034 for
testing.
          train  test
negative  12355  4119
positive  14742  4915
```

```python
info_gb("human_ensembl_regulatory", version=0)
```

```
Dataset `human_ensembl_regulatory` has 3 classes:
enhancer, ocr, promoter.

 The length of genomic
intervals ranges from 71 to 802, with average
429.91753643694585 and median 401.0.

 Totally
289061 sequences have been found, 231348 for
training and 57713 for testing.
          train   test
enhancer  85512  21378
ocr       69902  17476
promoter  75934  18859
```

```python
info_gb("human_enhancers_cohn", version=0)
```

```
Dataset `human_enhancers_cohn` has 2 classes:
negative, positive.

 All lengths of genomic
intervals equals 500.

 Totally 27791 sequences
have been found, 20843 for training and 6948 for
testing.
          train  test
negative  10422  3474
positive  10421  3474
```

```
info_gb("human_enhancers_ensembl", version=0)
```

```
Dataset `human_enhancers_ensembl` has 2 classes:
negative, positive.

 The length of genomic
intervals ranges from 2 to 573, with average
268.8641324705183 and median 269.0.

 Totally
154842 sequences have been found, 123872 for
training and 30970 for testing.
          train   test
negative  61936  15485
positive  61936  15485
```

```
info_gb("human_ocr_ensembl", version=0)
```

```
Dataset `human_ocr_ensembl` has 2 classes:
negative, positive.

 The length of genomic
intervals ranges from 71 to 593, with average
326.3452470873675 and median 315.0.

 Totally
174756 sequences have been found, 139804 for
training and 34952 for testing.
          train   test
negative  69902  17476
positive  69902  17476
```

```
os.chdir("/path/to/Project/datasets/GenomicBenchmarks")

download_dataset("human_nontata_promoters", version=0)
download_dataset("human_enhancers_cohn", version=0)
```

2.2 Formatting data

The downloaded data consisted of multiple '.txt' files organized into two directories, and since at least the '`getShape`' function from the '`DNAshapeR`' package required FASTA files to work, it felt right to integrate all sequences of each cis-regulatory element in a single FASTA file. For this, I gave them all a simple header and appended them together with *AWK*.

Bash Code
```
cd /path/to/Project/datasets/GenomicBenchmarks/

awk 'BEGIN{counter=0}
     {print ">promoter_"counter"|train|positive";
     print $0; counter+=1}' \
     human_nontata_promoters/train/positive/*.txt \
     > promoters_train_positive.fasta

awk 'BEGIN{counter=0}
     {print ">enhancer_"counter"|train|positive";
     print $0; counter+=1}' \
     human_enhancers_cohn/train/positive/*.txt \
     > enhancers_train_positive.fasta
```

3 Data Characterization

3.1 Libraries used

The following procedures require several R libraries, alongside custom functions developed for sequence characterization. The libraries and their respective roles in the analysis are outlined below:

R Code
```
# For useful tools like 'filter'
library(dplyr)
library(plyr)
# For statistic analysis
library(stats)
library(irlba)      # Memory-efficient PCA
# library(nortest)    # Lillifors normality test
# For genome-functions.R
library(stringr)
library(stringi)
library(primes)
# For parallel computing
library(doParallel)
library(foreach)
# For biological functions:
library(Biostrings) # Local/Global Alignments
library(DNAshapeR)  # DNA Shape Features
# For plotting
library(paletteer)  # Color Palettes
library(cowplot)    # Plot Grids
library(ggplot2)
library(see)        # Half Violin Plots
# For pretty tables
library(knitr)
library(kableExtra)
# For my own functions
source("/path/to/Project/scripts/genome-functions.R")
source("/path/to/Project/scripts/custom-functions.R")
```

3.2 Characterizing sequences

Here, we characterize a subset from each of our datasets: 1638 sequences per regulatory element; 3276 in total. However, there's a circumstance about the sequences that has to be noted:

- Both datasets have considerably different elements length-wise:
  1. All promoters have a length of 251 nucleotides.
  2. All enhancers have a length of 500 nucleotides.
- All features per sequence must be numerical and two-dimensional since we want it to be fed eventually to a simple classifier (like a Support Vector Machine).

```
proj_path <- "path/to/Project/datasets/GenomicBenchmarks"
prom_fastaname <- "promoters_train_positive.fasta"
enha_fastaname <- "enhancers_train_positive.fasta"
```

```r
prom_path <- paste(proj_path, prom_fastaname, sep = "/")
enha_path <- paste(proj_path, enha_fastaname, sep = "/")

# Scanning sequences
prom_seqs <- scan(prom_path,
                  character(), quote="")[seq(2,29484,2)]
enha_seqs <- scan(enha_path,
                  character(), quote="")[seq(2,20842,2)]
```

Given some previous tests done to ‘`sequences_characterizer`’ I came to the conclusion that parallel computing might provide a higher and more complex set of data in a feasible time span.

```r
# Prepairing clusters for parallel computing
corescluster <- makeCluster(6)
registerDoParallel(corescluster)

# Characterizing sequences and exporting to CSV
list_seqs <- list(promoters = prom_seqs,
                  enhancers = enha_seqs)
reg_elems <- c("promoters", "enhancers")
libs <- c("stringr", "stringi", "primes")

for (reg_elem in reg_elems) {
  foreach(i = 1:6) %dopar% {
    required_libs(libs)
    i_start <- ((i - 1) * 273) + 1
    i_final <- i * 273

    if (i > 1) { # Only the first CSV has headers
      write.table(
        sequences_characterizer(
          list_seqs[[reg_elem]][i_start:i_final],
          optim = TRUE, k_max = 6),
        paste(
          "datasets/GB-Testing/test", reg_elem,
          "-training_", i, ".csv", sep = ""),
          row.names = FALSE, col.names = FALSE,
          sep = ",")

    } else {
      write.csv(
        sequences_characterizer(
          list_seqs[[reg_elem]][i_start:i_final],
          optim = TRUE, k_max = 6),
        paste(
          "datasets/GB-Testing/", reg_elem,
          "-training_", i, ".csv", sep = ""),
          row.names = FALSE)
}}}
```

3.3 Concatenating CSV’s

It was decided to produce many files instead of appending over the same CSV table in order to apply a sort of quality control checkup after the parallel computing was done. This, because when forcing the process RAM overload was possible and the function could die mid-process. Since our data was separated in six tables per cis-regulatory element, here we only join each set together in a single CSV.

Bash Code

```bash
cat datasets/GB-Testing/testpromoters-training_*.csv \
    > datasets/GB-Testing/test-1638-promoters-6mers.csv
cat datasets/GB-Testing/testenhancers-training_*.csv \
    > datasets/GB-Testing/test-1638-enhancers-6mers.csv
```

3.4 Data description

Table 1: Overview of each column generated so far by ’`sequences_characterizer`’

Column	Description	Section Processed
A	Percentage of Alanines	per Sequence
T	Percentage of Thymines	
C	Percentage of Cytosines	
G	Percentage of Guanines	
temp	Melting Temperature	
shan	Shannon Coefficient	
kN.M_prod	KSG Product	per Kmer: each possible kmer of size *N* is identified on a scale of 1 to 4^N denoted by *M*. `i.e. k3.1 (or the first kmer of size 3) would be AAA; k4.2 would be AAAC`
kN.M_barc	Barcode Profile	
kN.M_pals	Palindrome Profile	
kN.M_revc	Reverse Complement Profile	

Prior to our primary analysis, it feels reasonable to explain the columns per sequence produced by our tabulator ‘`sequences_characterizer`’. From here we’ll first describe the ones computed sequence-wise, the ones computed kmer-wise, then the ones computed over each kmer distribution, and finally the ones corresponding to DNA-Shape; this feature comes at last considering ‘`getShape`’ function makes its’ own dataframe. The ones in **black** are already integrated in the table, the ones in **red** are yet to be adjoined:

Per sequence

- From *‘genome-functions.R’*:
  - **A, T, C, G** - *Nucleotide Percentages* per sequence.
  - **temp** - *Melting Temperature*: Temperature at which DNA’s double helix dissociates into single strands. *It’s*

dependent on GC percentage and sequence length.
  - **shan** - *Shannon Entropy Coefficient*: Statistical quantifier of information in a system. Measures the uncertainty a set of data has. In this case is a nucleotide-diversity metric. *It's dependent on nucleotide percentages.*
- From *'Biostrings'*:
  - **la_sc** - *Local Alignment Score*
  - **la_id** - *Local Alignment Identity*
  - **ga_sc** - *Global Alignment Score*
  - **ga_id** - *Global Alignment Identity*

Per kmer

Kmer characterization came with many challenges, principally to provide each kmer with a distinct signature value dependent on their sequence structure. This under the assumption that .

- From *'genome-functions.R'*:
  - **kN.M_prod** - *KSG Product*: Its obtained by multiplying **Kmer-Percentage * Shannon Entropy * GC-Percentage** (*in concept*). The justification for this comes from the desire to give each kmer a distictive signal based on their sequence. However many kmers can produce the same value depending on the function, for example, the kmers **'AAAT', 'ATAA', 'CCGC'** and **'TGGG'** all have the *same Entropy Coefficient* (0.811), while **'GGAT', 'CAAG', 'AACC'** and **'TGCT'** all have the *same GC Percentage* (0.5). Additionally, it is a little tricky to *keep the product from becoming zero* when the sequence is either devoid of C/G nucleotides (e.g. **gc_percentage('ATTA') =0**), or lacks nucleotide diversity (e.g. **shannon_entropy('GGGG')=0**). Refer to 'Theory Annex', Sections 2 & 3, for more details.
  - **kN.M_barc** - *Barcode Profile*: Its obtained by generating a vector with size equal to the kmer amount (which is equivalent to **'sequence length'** - **'kmer length'** + 1) and filling it with one prime number per cell in ascending order. Afterwars we use a logical (**1's & 0's**) vector of the same size to represent each kmer's positions in the sequence. *In concept*, if we multiply both vectors we will get a set of prime numbers representing the positions of each kmer inside the sequence. If we multiply as well the elements inside this set, we will obtain an exlusive value for each kmer in each sequence, and only the sequences with the same kmers in the same place will be divisible by the corresponding prime.
  - **kN.M_pals** - *Palindromes' Profile*:
  - **kN.M_revc** - *Reverse Complement Profile*:

Per kmer distribution

Per non-defined kmer

- From *'DNAshapeR'*:
  Generates a set features as vectors (EP, MGW, HelT, Roll & ProT by default) per provided FASTA file. Splits each sequence in sliding kmers of size 5 (pentamers) not defined by me, which therefore makes vector size dependent on sequence-length. The features intended to be used are:
  - **sh_ep** - Shape EP
  - **sh_mgw** - *Minor Groove Width*
  - **sh_helt** - *Helix Twist*
  - **sh_prot** - *Propeller Twist*
  - **sh_roll** - *Roll*

4 Exploration analysis

4.1 Primary analysis

Firstly we read our CSV tables into our conviniently named dataframes.

We could have concatenated both tables together, however I prefer to keep them in different files.

R Code
```r
setwd("/path/to/Project/")
testdir_path <- "datasets/GB-Testing/"
prom_csvpath <- "test-1638-promoters-6mers.csv"
enha_csvname <- "test-1638-enhancers-6mers.csv"

proms <- read.csv(paste0(testdir_path,
                         prom_csvname),
                         check.names = F)
enhas <- read.csv(paste0(testdir_path,
                         enha_csvname),
                         check.names = F)
CREs <- c("Promoter","Enhancer")
```

First we get an overviwew of the dimensions of our data: NOTE: Replace this with some kind of table

```r
deparse(substitute(proms))
deparse(substitute(enhas))
dim(proms)
dim(enhas)
```

```
[1] "proms"                    [1] "enhas"
[1]  1638 21830                [1]  1638 21830
```

It's noticeable the fact that we have way more columns than rows in this test table. Let's get a general glimpse of the first three promoters' rows by displaying the first 3 lines and first 18 columns pertaining to all *sequence-wise* variables computed so far plus the first 12 *kmer-wise* variables, which align with the data related to the first 3 kmers (**'AA', 'AC'** & **'AG'**):

```r
teatable(proms[1:3,1:18])
```

A	T	C	G	temp	shan
0.1673307	0.2231076	0.3306773	0.2788845	87.21315	1.956136
0.2629482	0.2788845	0.2549801	0.2031873	81.00598	1.990374
0.3625498	0.2031873	0.2470120	0.1872510	80.02590	1.948719

k2.1_prod	k2.1_barc	k2.1_pals	k2.1_revc	k2.2_prod	k2.2_barc
9	1.959765	2.177403e+09	1.374020e+12	17.11198	1.531862
17	2.633165	1.138401e+17	2.971115e+17	26.44579	2.624612
38	5.347025	3.981015e+36	8.100763e+29	24.89016	1.855662

k2.2_pals	k2.2_revc	k2.3_prod	k2.3_barc	k2.3_pals	k2.3_revc
3.845422e+15	3.525451e+11	26.44579	2.445328	4.788062e+13	1.305836e+26
2.607331e+20	6.308689e+14	24.89016	2.513656	1.320117e+14	6.435389e+19
1.156904e+25	3.573059e+12	34.22397	3.431445	6.011592e+17	7.178542e+17

We'll get a similar panorama when looking at the first three enhancers' rows (although it seems like *'barcode'* values appear to be significantly larger):

```
teatable(enhas[1:3,1:18])
```

A	T	C	G	temp	shan
0.240	0.236	0.234	0.290	85.0392	1.993988
0.204	0.286	0.210	0.300	84.4652	1.978249
0.250	0.280	0.258	0.212	82.8252	1.992923

k2.1_prod	k2.1_barc	k2.1_pals	k2.1_revc	k2.2_prod	k2.2_barc
36	10.432345	1.885437e+37	8.632413e+30	24.89016	6.027803
20	6.153015	5.673403e+20	1.061673e+53	23.33452	4.984254
31	9.936447	1.640775e+32	5.563175e+39	46.66905	9.301359

k2.2_pals	k2.2_revc	k2.3_prod	k2.3_barc	k2.3_pals	k2.3_revc
1.579134e+29	1.707234e+31	76.22611	13.43518	3.557032e+41	5.830130e+44
4.485619e+32	1.810188e+35	73.11484	13.29662	3.273426e+39	3.755909e+39
9.951038e+37	3.612544e+25	60.66976	12.53356	4.255600e+33	3.010280e+57

```
mean_prom <- colMeans(proms)
mean_enha <- colMeans(enhas)
sd_prom <- apply(proms, 2, sd)
sd_enha <- apply(enhas, 2, sd)

names_CREs <- rep(CREs, each = length(proms))
cre_summary <- data.frame(Type = factor(names_CREs),
                          Field = rep(names(proms),2),
                          Means = c(mean_prom,mean_enha),
                          StDevs = c(sd_prom,sd_enha))
```

```
ldf <- length(proms)
coffeetable(cre_summary[c(1:10,(ldf+1):(ldf+10)),])
```

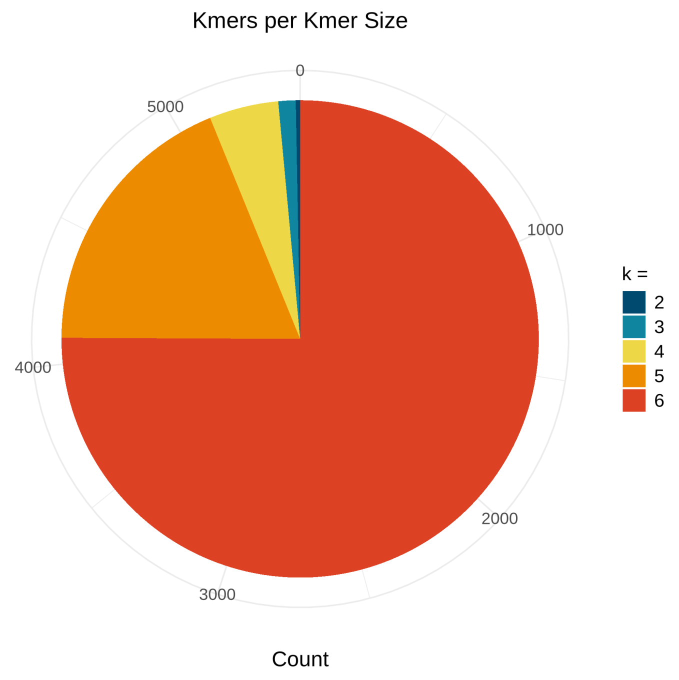
	Type	Field	Means	StDevs
1	Promoter	A	1.911256e-01	7.145510e-02
2		T	1.995826e-01	7.599340e-02
3		C	2.962655e-01	8.067840e-02
4		G	3.130263e-01	8.524230e-02
5		temp	8.720208e+01	5.379461e+00
6		shan	1.891724e+00	9.437000e-02
7		k2.1_prod	1.196276e+01	8.995306e+00
8		k2.1_barc	1.500971e+00	1.229003e+00
9		k2.1_pals	2.634405e+51	1.065029e+53
10		k2.1_revc	9.789346e+62	3.960331e+64
21831	Enhancer	A	2.653712e-01	5.853000e-02
21832		T	2.678205e-01	5.815610e-02
21833		C	2.351111e-01	5.648440e-02
21834		G	2.316972e-01	5.435370e-02
21835		temp	8.269434e+01	3.785299e+00
21836		shan	1.959202e+00	3.892580e-02
21837		k2.1_prod	4.093346e+01	1.835697e+01
21838		k2.1_barc	1.208127e+01	5.751107e+00
21839		k2.1_pals	2.567318e+112	1.038419e+114
21840		k2.1_revc	3.321646e+121	1.344343e+123

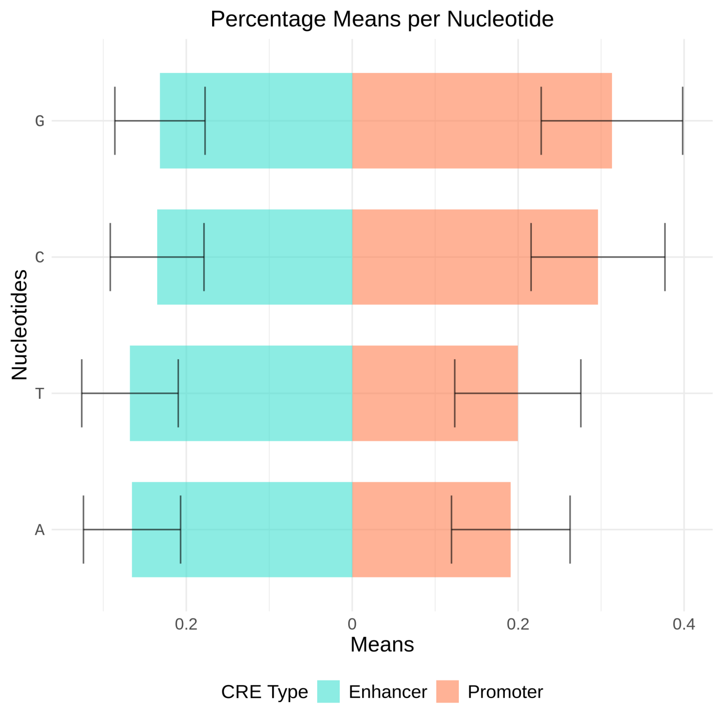
This text should be later deleted

```
# Get only 'prod' columns of each kmer

k_Ns <- c(n_ki(2), n_ki(3), n_ki(4), n_ki(5), n_ki(6))
```

```
k_inds <- c(0)
k_stt <- c(0)
k_fin <- c(0)
for (k_n in k_Ns) {
  k_inds <- c(k_inds, tail(k_inds,1) + 1)
  k_inds <- c(k_inds, tail(k_inds,2)[1] + k_n)
  k_stt <- c(k_stt, tail(k_fin,1) + 1)
  k_fin <- c(k_fin, tail(k_fin,1) + k_n)
}
k_inds <- k_inds[-1]
k_indz <- array(c(k_stt[-1], k_fin[-1]), dim = c(5, 2),
                dimnames = list(1:5, c("start","final")))
# print(rev(k_inds)[1])
# print((ldf-6)/4)
# print(ldf)
```

```
kmer_set_sizes <- data.frame(sizes = k_Ns,
  k_sets = c("2","3","4","5","6"))

ggplot(kmer_set_sizes, aes(x = "", y = sizes,
                           fill = k_sets))+
  geom_bar(width = 1, stat = "identity") +
  labs(y = "Count", x = "",
       title = "Kmers per Kmer Size",
       fill = "k =") +
  coord_polar("y", start = 0) +
  theme_minimal() +
  scale_fill_paletteer_d("PNWColors::Bay") +
  theme(legend.position = "right",
        axis.title = element_text(size = 15),
        text = element_text(size = rel(4.25)),
        legend.text = element_text(size = 13),
        legend.title = element_text(size = 13.5),
        axis.text.y = element_text(family = "mono"),
        plot.title = element_text(size = 16, hjust = 0.5))
```



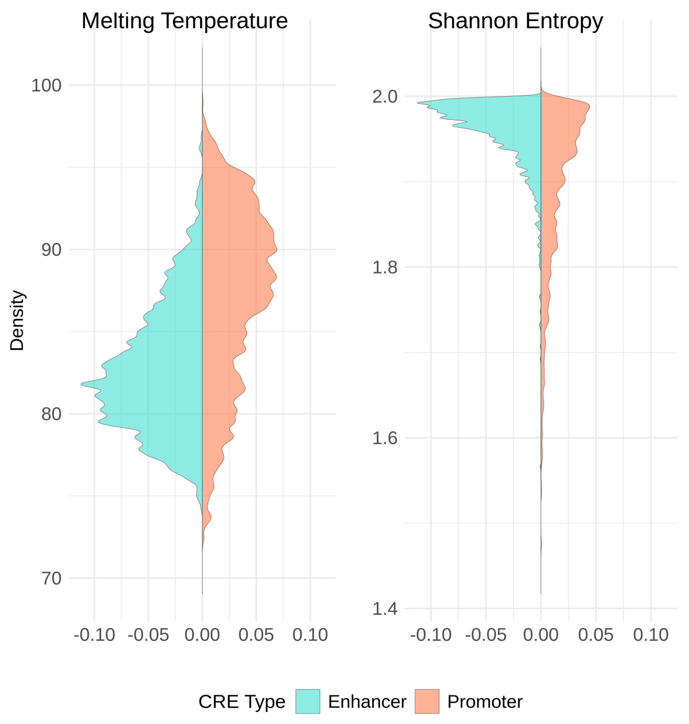


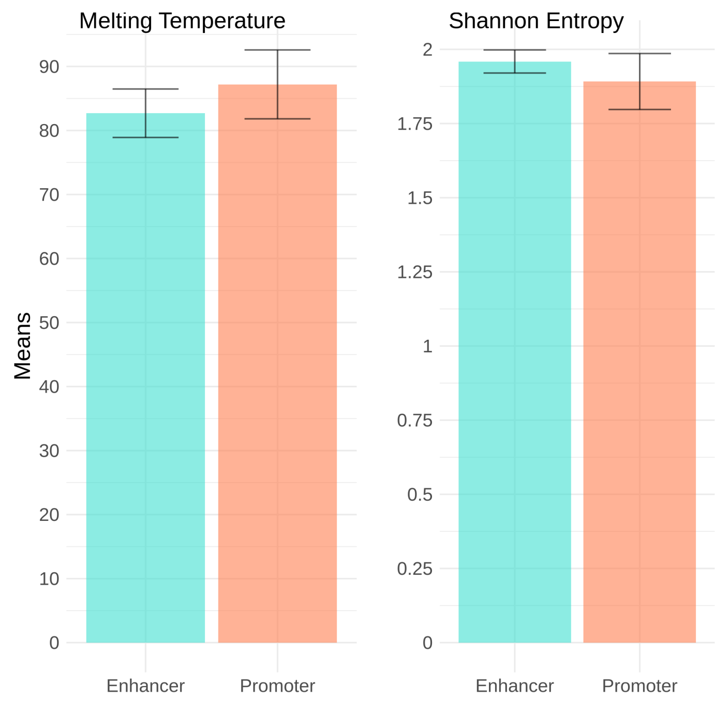
```r
pe_arr <- function(seq_data)
  return(array(seq_data, dim = c(rev(k_inds)[1], 2),
    dimnames = list(1:rev(k_inds)[1], c("prom","enha"))))

indxs <- list(
  prod = pe_arr(c(seq(7,ldf-3,4), ldf+seq(7,ldf-3,4))),
  barc = pe_arr(c(seq(8,ldf-2,4), ldf+seq(8,ldf-2,4))),
  pals = pe_arr(c(seq(9,ldf-1,4), ldf+seq(9,ldf-1,4))),
  revc = pe_arr(c(seq(10,ldf,4), ldf+seq(10,ldf,4))))
```

```r
# Comment
pyrplot_(cre_summary[c(1:4, ldf + (1:4)), ],
      x_label = "Nucleotides", y_breaks = seq(-1,1,0.2),
      title = "Percentage Means per Nucleotide")
```

```r
data_e <- cbind(Type = rep("Enhancer",
                length(enhas$temp)), enhas[,5:6])
data_p <- cbind(Type = rep("Promoter",
                length(proms$temp)), proms[,5:6])
subset_tm_sh <- rbind(data_e, data_p)
```

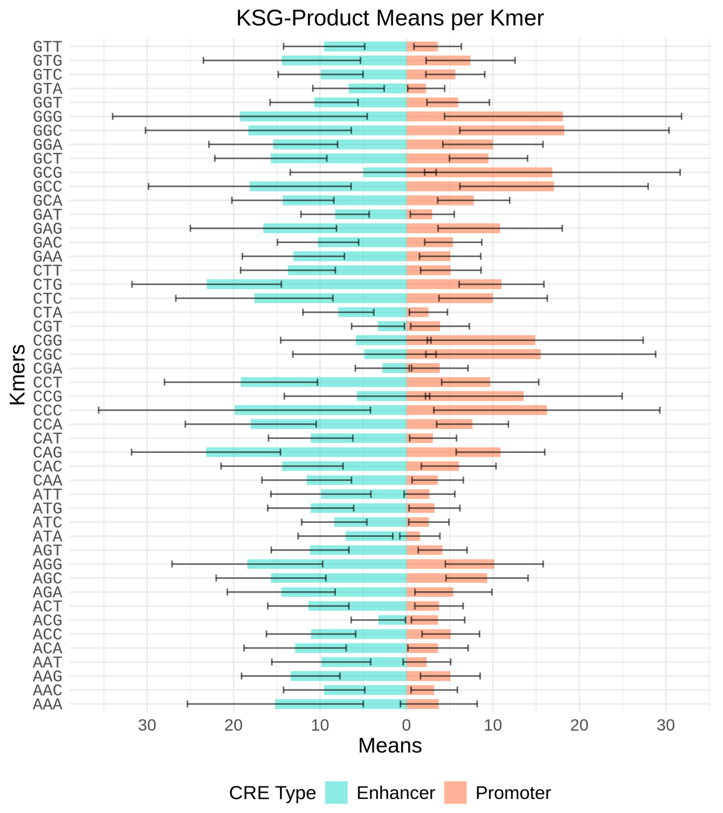
```r
# Comment
# Saving Melting Temperature Violin Plot
tm_violin <- hvioplot_(data = subset_tm_sh,
                       y_var = "temp",
                       y_label = "Density")

# Saving Shannon Violin Plot
sh_violin <- hvioplot_(data = subset_tm_sh,
                       y_var = "shan",
                       fill_legend_title = "CRE Type")

# Get legend
legend_violin <- get_legend_bypass(sh_violin +
  guides(color = guide_legend(nrow = 1)) +
  theme(legend.position = "bottom",
        legend.title = element_text(size = 13.5),
        legend.text = element_text(size = 13)))

# Merging of Violin-Plots
grid_violin <- plot_grid(tm_violin, sh_violin, NULL,
                 ncol = 3, rel_widths = c(1,1,0.1),
                 labels = c("Melting Temperature",
                 "Shannon Entropy"), label_size = 16,
                 label_fontface = "plain", vjust = 1,
                 hjust = c(-0.38, -0.48))

# Adding legend at the bottom
plot_grid(grid_violin, legend_violin,
          ncol = 1, rel_heights = c(12,1))
```



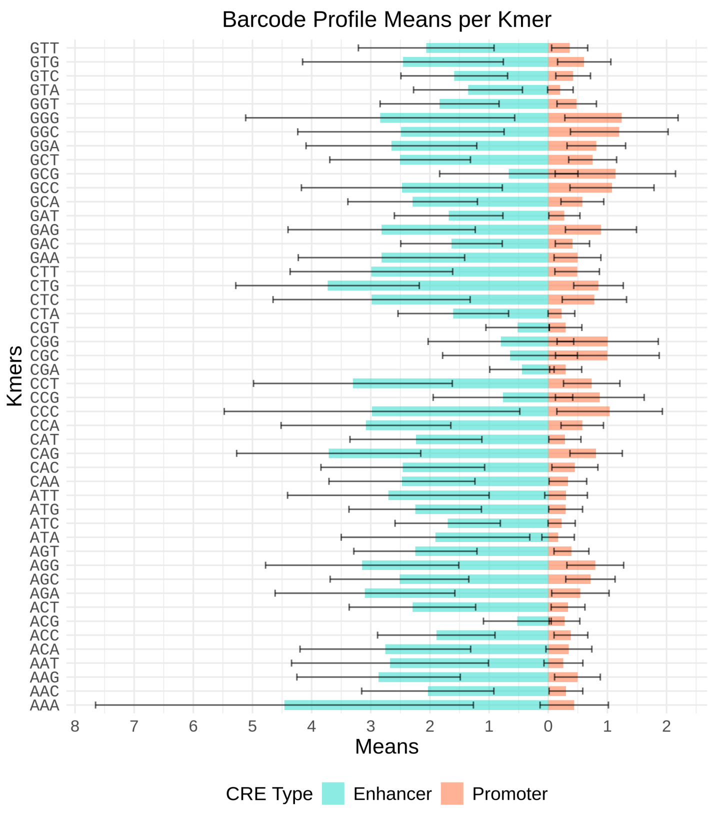


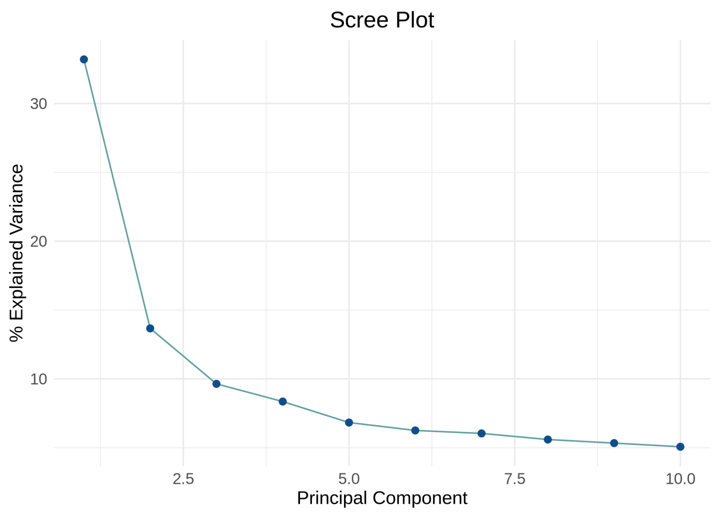
```
# Comment
# Saving Melting Temperature Bar-Plot
temp_plot <- barplot_(filter(cre_summary, Field=="temp"),
                      y_breaks = seq(0, 100, 10),
                      y_axis_title = "Means")

# Saving Shannon Coefficient Bar-Plot
shan_plot <- barplot_(filter(cre_summary, Field=="shan"),
                      y_breaks = seq(0, 2, 0.25),
                      fill_legend_title = "CRE Type")

# Merging of Bar-Plots
plot_grid(temp_plot, shan_plot,
    ncol = 2, rel_widths = c(1,1),
    vjust = 1, hjust = c(-0.35, -0.48),
    label_size = 16, label_fontface = "plain",
    labels = c("Melting Temperature", "Shannon Entropy"))
```

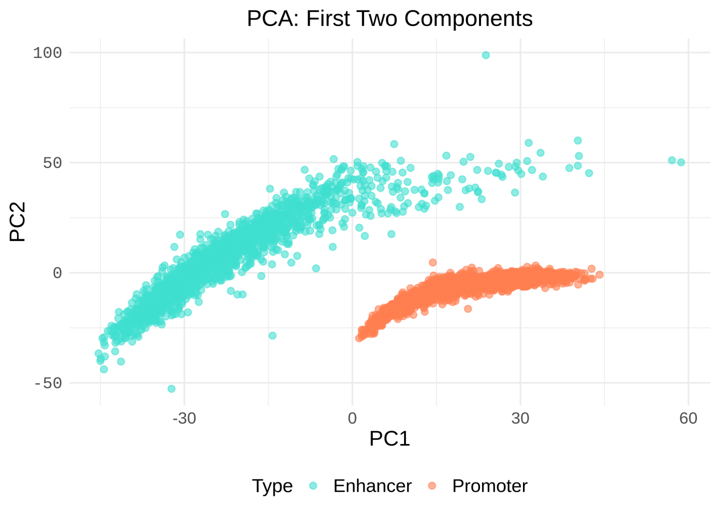
```
kmer_names <- combi_kmers(k=3)[1:48]

pyrplot_(cre_summary[indxs$prod[17:64,],], kmer_names,
        x_label = "Kmers", y_breaks = seq(-30,30,10),
        title = "KSG-Product Means per Kmer")
```



```
pyrplot_(cre_summary[indxs$barc[17:64,],], kmer_names,
        x_label = "Kmers", y_breaks = seq(-10,10,1),
        title = "Barcode Profile Means per Kmer")
```







With this last pyramid plot, I noticed one crucial bias all my profiler functions had (*Barcode*, *Palindrome* and *Reverse Complement*), sequence length. This would make them somewhat useless for the purpose of this actual test, nevertheless they would still be of use for the following analysis.

4.2 Principal Components Analysis

Since **Principal Components Anal ysis** (PCA) should be one of the elementary tests when approaching a high-dimensional dataset given its' simplicity, straightforward understanding and computational speed, it was the next method applied to our data.

Yet, there was a catch for this: I wanted to test whether the KSG-Product would be sufficient to characterize this data, and if so, at least how many kmers would be needed to achieve that.

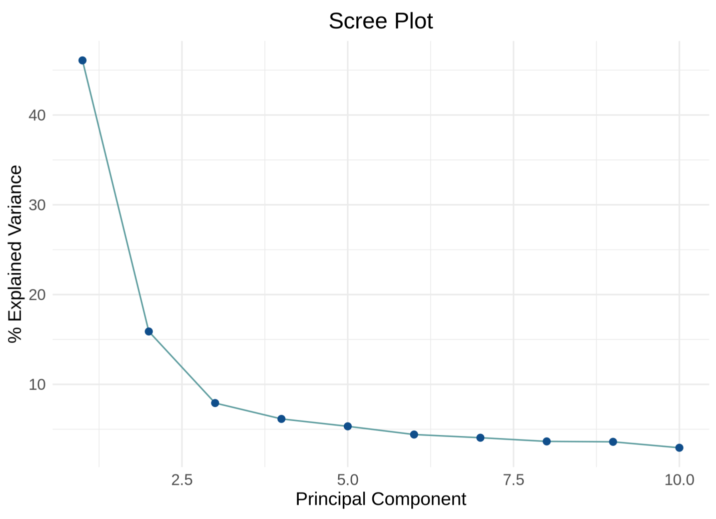
The PCA procedure is described (for readability purposes) through the following function: '`pca_plot`'. Note that instead of using ' `prcomp`' from '`stats`', the alternative '`irlba`' was chosen due to its' optimization (via truncated singular value decomposition) for very large, sparse and high-dimensional datasets; the concrete method being the **implicitly restarted Lanczos bidiagonalization algorithm** (IRLBA).

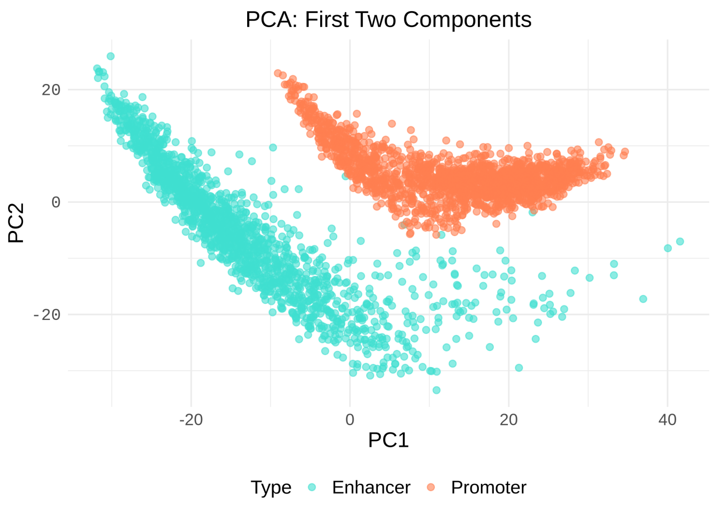
```
# 'Type' tags for the resulting elements
CRE_type <- c(rep("Enhancer", dim(enhas)[1]),
              rep("Promoter", dim(proms)[1]))

pe_data <- rbind(enhas, proms)
fulldata_pca <- simple_pca(pe_data, CRE_type)
fulldata_pca$scree_plot; hspace()
fulldata_pca$pca_plot
```

Here we observe the data separate through the first 2 principal components, however this is something expected since most variables (namely *Tm* and all *kmer profilers*) in our table are biased towards sequence length.

In consequence, for the following approach we will exclude those variables and use only the ones we know to be independent of sequence length.

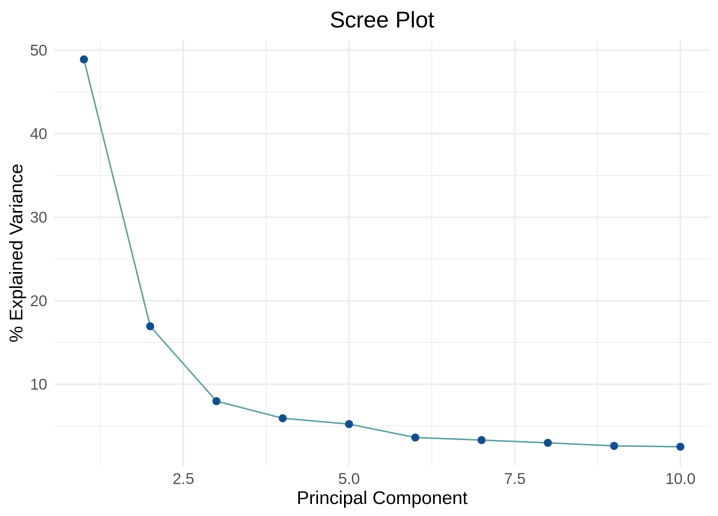
```
# Selection of sequence-length independent  columns
seqlen_indep <- c(1:4,6, indxs$prod[,1])

# Data scaling
pe_data<- rbind(enhas[,seqlen_indep],
                proms[,seqlen_indep])

seqlen_indep_pca <- simple_pca(pe_data, CRE_type)
seqlen_indep_pca$scree_plot; hspace()
seqlen_indep_pca$pca_plot
```



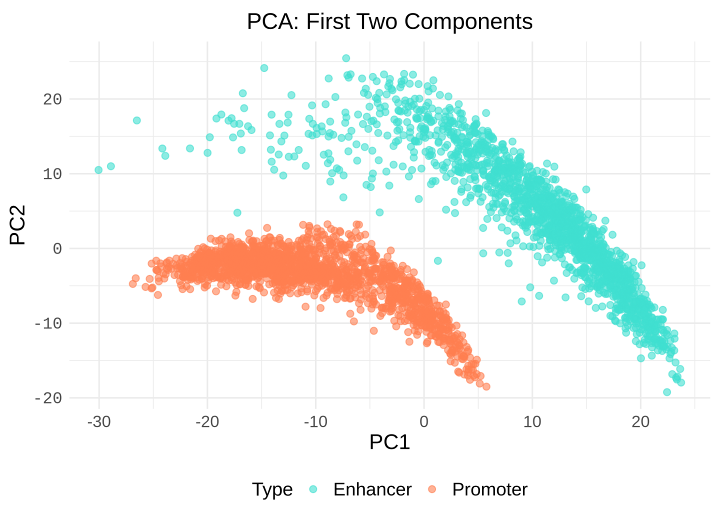


Suprisingly, the data does show a clear separation between groups a further step would be to try with the k-means algorithm, however I believe it's fair to try and reduce the number of kmers analized so that we may further reduce the number of computations.

```
indz <- c(k_indz[1,"start"]:k_indz[4,"final"])
less_kmers <- indxs$prod[indz, 1]
seqlen_indep <- c(1:4,6, less_kmers)

pe_data <- rbind(enhas[,seqlen_indep],
                 proms[,seqlen_indep])

lessk_pca <- simple_pca(pe_data, CRE_type)
lessk_pca$scree_plot
```



```
lessk_pca$pca_plot
```


PCA: First Two Components
PC2
PC1
Type
Enhancer
Promoter