Identifying DMRs in Promoters and Genes between AML Cells and Monocytes

Leo Burmedi, Pierre De Marinis, Konstantin Fischer, Daniel Ürge

6/10/2019

1. Introduction

A few introductory elements as to how and why our analysis was performed.

1.1 Abstract

Acute myeloid leukemia is a disease of cells deriving from the bone marrow. Haematopoietic stem cells localized in the bone marrow develop into myeloid stem cells and later into myeloblasts which act as the progenitors of mature myeloid white blood cells like monocytes and granulocytes. During these steps, called haematopoiesis, the immature myeloid stem cells differentiate into mature white blood cells, orchestrated by complex methylation systems in the cells' genome. This process requires proper expression and functioning of multiple proteins. Some proteins important for the differentiation of the myeloid stem cells or myeloblasts may eventually malfunction or be silenced by epigenetic editing and cause a stasis of the cells in the immature und stem cell-like state (Fialkow, 1976).

Genesis of single immature myeloblasts may not be a problem. The problem will be additional mutations or epigenetic edits that influence the proliferation of the myeloblasts resulting in massive proliferation of the immature white blood cells competing for space with mature and functioning blood cells. This results in typical symptoms of acute myeloid leukemia like short breath and dizziness for the erythrocytes get crowded out but massively proliferating myeloblasts (Fialkow et al., 1991). By epigenetic modifications the expression of genes can be regulated by the cells. One way of epigenetically modifying the genome is by methylation of the CpG-islands of genes and promoters. In case of the promoters a hypermethylation results in the condensing of the chromatin and thereby in the silencing of the associated gene. Especially in diseases, abnormally methylated regions of key player genes can be observed, so called differentially methylated regions (DMR), which highly differ in diseased cells compared to healthy cells (Rakyan et al., 2011).

Epigenetic silencing of genes may play a special role in a lot of cancerous diseases, so too in acute myeloid leukemia. By studying the epigenetic profiles of certain cancer types, scientists promise themselves to find a way of better understanding the development and factors of cancer diseases.

In our analysis we compared the genome-wide methylation data of myeloblasts from acute myeloid leukemia patients (bone marrow) with the genome-wide methylation data of monocytes from healthy patients (bone marrow) to identify differentially methylated regions in the genome of myeloblasts in contrast to white blood cells, that differentiated correctly. We analyzed the differentially methylated regions for a better understanding of what goes wrong in the differentiation process of myeloblasts into monocytes in diseased patients, maybe identifying key player genes hypo- or hypermethylated compared to healthy methylation patterns, and thereby striking the path for a better understanding of acute myeloid leukemia.

1.2 Annotation for Naming Data Frames and Functions

Here is a brief overview of how things are named in our code.

Code	Annotation
g_[text]	dataframe derived from the genes_data_frame dataset
p_[text]	dataframe derived from the promoters_data_frame dataset
l_[text]	dataframe intended for use in logistic regression
f_[text]	function, fg/fp/fl indicate that the function only works on certain data sets (genes/promoters/logistic for example)
_Resource	resource dataset, includes the remaining rows after quality conrol with Ensembl ID, symbol and the fold change
_T	dataframe, that is than used for the t-test, includes number of NAs (n), means (m), standard deviations (sd) for each row
i,j, y	vectrors only used by function and loops
_cov	coverage values
_bed, _bet	beta values
_Mvalues	dataset with all remaining genes/ promoters after QC, normalization and imputation

In general every data set has a few one letter codes at the front indicating their origin and usage, whereas the text following this usually gives clearer indications.

Data frames and variables generated within functions that are used in apply can deviate from this standard system because they are never saved in the ennvironment. A similar behaviour can be seen for the imputation models.

1.3 Why We Suspect DMRs in our Data

We take a brief look at the overall distribution of methylation levels between our cohorts as a simple visual indication of why DMRs are to be expected.

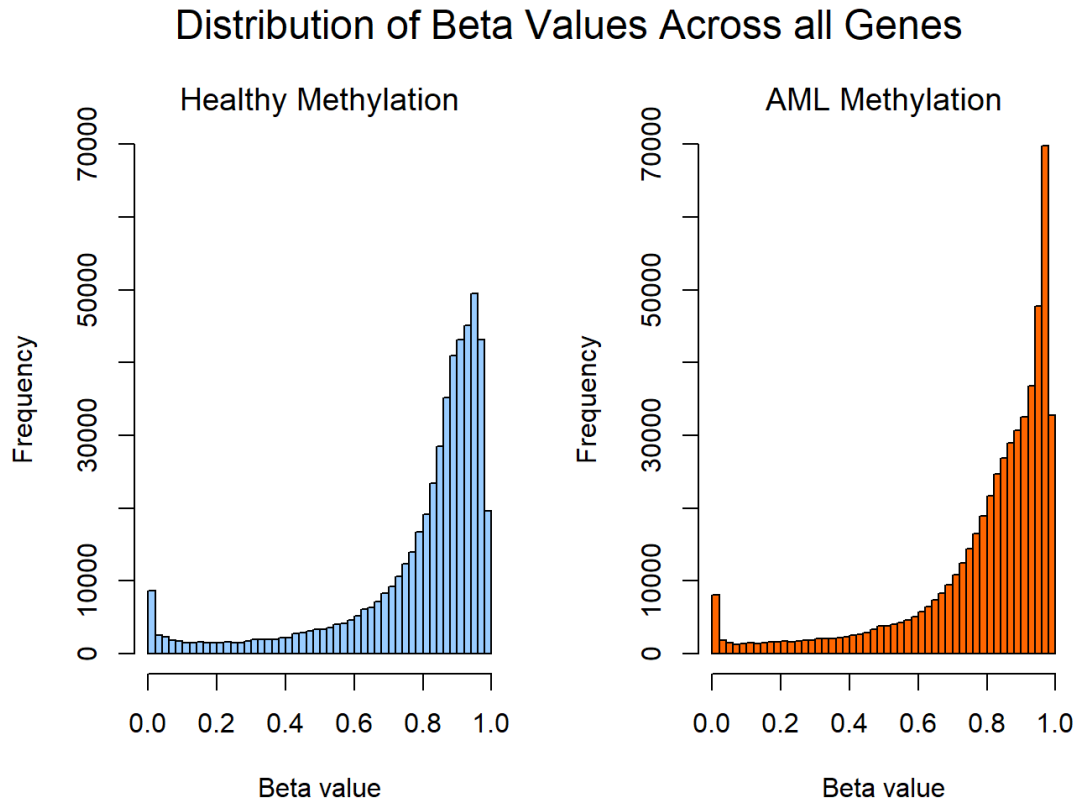
We start by loading our data and continue by plotting the distribution of Beta values for each cohort

```
## Loading the dataframe and naming in input_data is neccesary first for the code to run through properly.
AML_Mono_list <- readRDS("AML_Mono_list.RDS")
input_data <- AML_Mono_list
genes_data_framexy <- input_data$genes
promoters_data_framexy <- input_data$promoters

## Removing uneccesary data sets
remove(input_data)
```

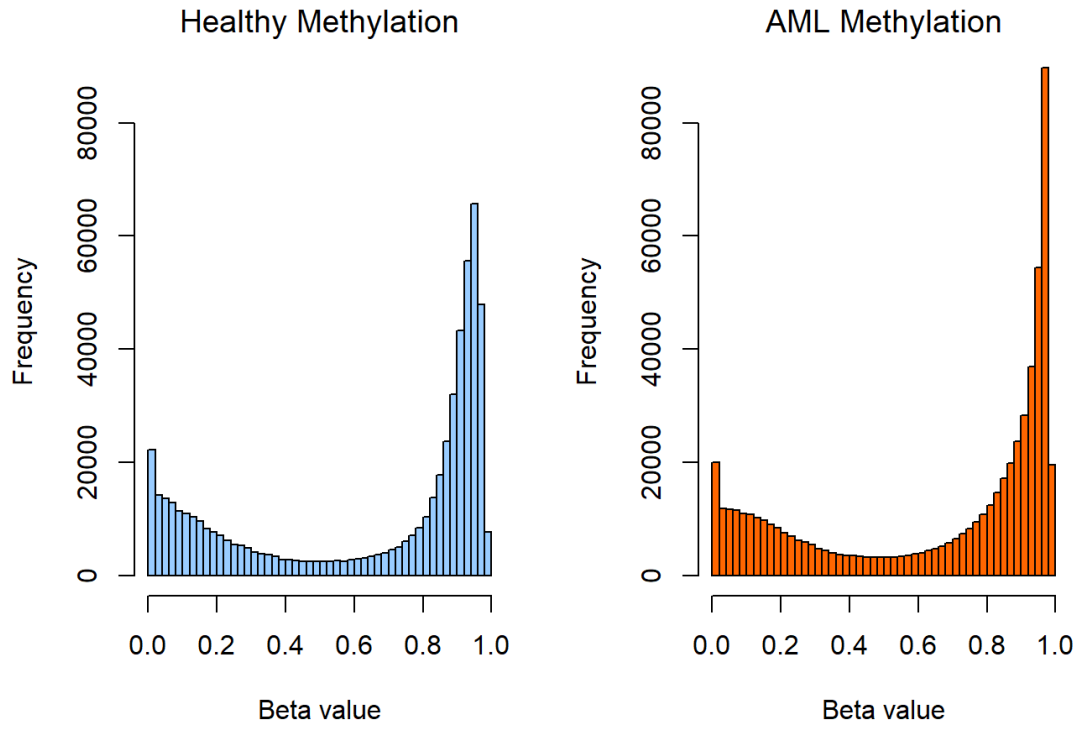
From these available data sets in the AML_Mono_list we have only extracted the genes and the promoters data sets. We ignore the rest because they don't contain as much relevant information for us. Genes and promoters are distinct functional groups we know to be influenced by methylation, where promoters are especially affected. The sam cannot be said for the sequence units in the tiling and CpG Island data sets.

Continuing on we now plot how the distribution of beta values between the cohorts differ.

```
## Creating dataframes for genes and promoters with all beta values without NAs.
    g_AML_bedall <- data.frame(Beta = c(t(na.omit(genes_data_framexy[,11:20]))))
    g_Mono_bedall <- data.frame(Beta = c(t(na.omit(genes_data_framexy[,21:30]))))
## Transforming the above created dataframes into a data frame where coverage is the only variable.
    p_AML_bedall <- data.frame(Beta = c(t(na.omit(promoters_data_framexy[,11:20]))))
    p_Mono_bedall <- data.frame(Beta = c(t(na.omit(promoters_data_framexy[,21:30]))))

## Plotting the beta value distributions for genes
    ## Determining maximum value of the AML histogramms
      g_AML_bedmax <- max(hist(g_AML_bedall$Beta, breaks = 50, plot = FALSE)$counts)
      p_AML_bedmax <- max(hist(p_AML_bedall$Beta, breaks = 50, plot = FALSE)$counts)

    ## Rest of the plotting for genes
    par(mfrow=c(1, 2), mar = c(4,4,4,2))

    hist(g_Mono_bedall$Beta, main = "", xlab = "Beta value", col = "#99CCFF", breaks = 50, ylim = c(0, g_AML_bed
max))
    title("Healthy Methylation", line = 0.5, font.main = 1)

    hist(g_AML_bedall$Beta, main = "", xlab = "Beta value", col = "#FF6600", breaks = 50)
    title("AML Methylation", line = 0.5, font.main = 1)

    mtext("Distribution of Beta Values Across all Genes", cex = 1.5, side = 3, line = - 1.5, outer = TRUE)
```




```
    ## Rest of the Plotting for promoters
    par(mfrow=c(1, 2), mar = c(4,4,4,2))

    hist(p_Mono_bedall$Beta, main = "", xlab = "Beta value", col = "#99CCFF", breaks = 50, ylim = c(0, p_AML_bed
max))
    title("Healthy Methylation", line = 0.5, font.main = 1)

    hist(p_AML_bedall$Beta, main = "", xlab = "Beta value", col = "#FF6600", breaks = 50)
    title("AML Methylation", line = 0.5, font.main = 1)

    mtext("Distribution of Beta Values Across all Promoters", cex = 1.5, side = 3, line = - 1.5, outer = TRUE)
```



```
## Removing uneccesary data sets
    remove(g_AML_bedmax, g_AML_bedall, g_Mono_bedall)
    remove(p_AML_bedmax, p_AML_bedall, p_Mono_bedall)
```

It can be seen that the two plots have a slightly different distribution for both promoters and genes. This is an easy indication for DMRs between each of the cohorts.

2. Quality control and Data Transformation

Our analysis ends up relying on three data sets, the 'Mvalues', 'T' and 'Resource' data sets. All three data sets need to be cleaned of uneccesary and unreliable data. All the data sets rely on normalization, and additionally the Mvalues data set requires Imputation as well. The Mvalues data set contains all the normalized methylation data for our patients and sequences. The T data set includes all the t-test input statistics required for a t-test, and the Resource data set includes the fold change, ensembl ID and Symbol corresponding to every sequence.

Quality Control and Data Transformation will take us through the creation process for all three of these. The goal of Qualtity control is to make the dataset as suited as possible for the analysis.

This includes: removing uncessary variables, getting rid NAs and removing information that can produce fallacious results.

2.1 Removing Chromosomes X and Y

Chromosome X is removed because it can be hypermethylated for female patients and thus influence the analysis. Y Chromosomes are removed because there are not enough male patients to make reliable statistical statements.

```
## Removing chromosome X and Y.
    genes_data_frame <- data.frame(genes_data_framexy[!(genes_data_framexy$Chromosome == "chrX" | genes_data_fra
mexy$Chromosome == "chrY"),])
    promoters_data_frame <-     data.frame(promoters_data_framexy[!(promoters_data_framexy$Chromosome == "chrX"
 | promoters_data_framexy$Chromosome == "chrY"),])

## Removing uneccesary data sets
    remove(genes_data_framexy)
    remove(promoters_data_framexy)
```

2.2 Removing Unneccesary Variables

For the Mvalues and T data sets we only need the beta and the coverage values of the patients, so we create two new data sets that only include these values. We still keep the old data sets around though because the Resource data set will need the Symbols column.

```
    g_patients <- data.frame(genes_data_frame[11:50])
    p_patients <- data.frame(promoters_data_frame[11:50])
```

2.3 Setting Coverage Thresholds and Handling NAs

If a coverage value is too low that means there are not enough reads for given sequence. If it is too high however that might be due to overlapping of repeated sequences by the alignment. This can render our analysis unreliable so we need to find sensible thresholds. The lower coverage threshold was set by 25 based on literature (Ziller et al., 2015). For the higher however we came up with the solution to find the value where we would lose the lowest percentage of data by losing the highest percentage of coverage values. To check if this is even possible we made two plots to evaluate what our distribution of coverage values look like. Since the two datasets have a different distribution we need to find seperate upper thresholds.

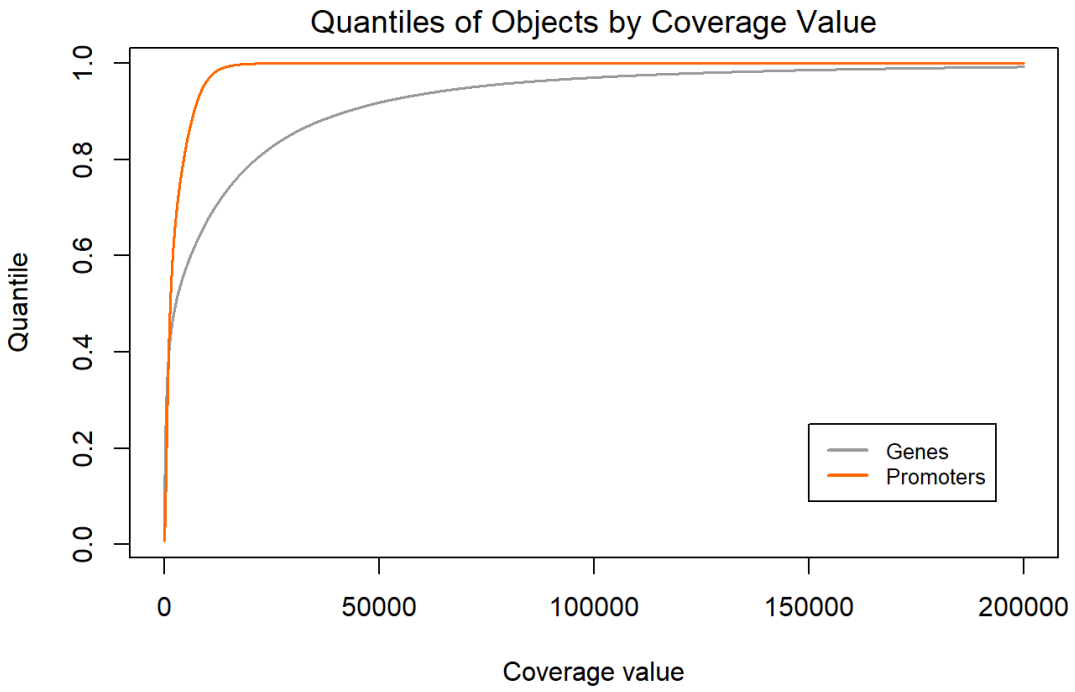
Once we find the threshold we will convert all beta values with an unreliable corresponding coverage value to NAs. Next we will remove all rows with more than 3 NAs for one cohort.

2.3.1 Creating Data Frames With all Coverage Values of Genes and Promoters

```
    g_coverage_all <- data.frame(Coverage = c(t(genes_data_frame[,31:50])))
    p_coverage_all <- data.frame(Coverage = c(t(promoters_data_frame[,31:50])))
```

2.3.2 Plotting What Percentile of Objects is Below a Given Threshold

```
## Creating our sequence of arbitrary but plausible values for the x axis
    x = seq(0, 200000, 1)
## Calculating the coverage value for the given quantile (x).
    g_Y_axis = ecdf(g_coverage_all$Coverage) (x)
    p_Y_axis = ecdf(p_coverage_all$Coverage) (x)

    ## Rest of the plotting for genes
    plot(x, g_Y_axis, type = "l", main = "", xlab = "Coverage value", ylab = "Quantile", col = "#999999", lwd =
1.5)
    lines(x, p_Y_axis, col = "#FF6600", lwd = 1.5)
    title("Quantiles of Objects by Coverage Value", line = 0.5, font.main = 1)
    legend(150000, 0.25, legend=c("Genes", "Promoters"), col=c("#999999", "#FF6600"), lty = 1:1, lwd = 2, cex=0.
8)
```




```
## Removing uneccesary data sets
    remove(g_Y_axis, x)
    remove(p_Y_axis)
```

This output show us what percentage of objects (y-axis) would be lost if set an arbitrary value (x-axis) as the upper coverage limit. It should be noted that some genes and promoters have coverages above 2e05, but that they are so few as to be omitted.

The curves for both datasets show clear optimums of low coverage and high percentage of kept objects. So this confirms that we can use our above mentioned method and lose a very low percentage of data by cutting away a relatively high percentage of the total coverage value range, so we can proceed with the analysis.

2.3.2 Finding the Upper Coverage Threshold: The Ultraloop

For finding the upper threshold we used the ultraloop. The ultraloop calculates the amount of lines which would remain if we were to cut at a given coverage value. The ultraloop works by turning the trustworthy values (that are under a given upper threshold) into NAs and cutting rows that have lots of NAs. This way, the rows that are cut away are the ones that can be considered trustworthy at that upper coverage threshold. Since they will be trustorthy at any upper coverage threshold higher than the previous one (and we check our potential thresholds from low to high), we don't need to look at them again.

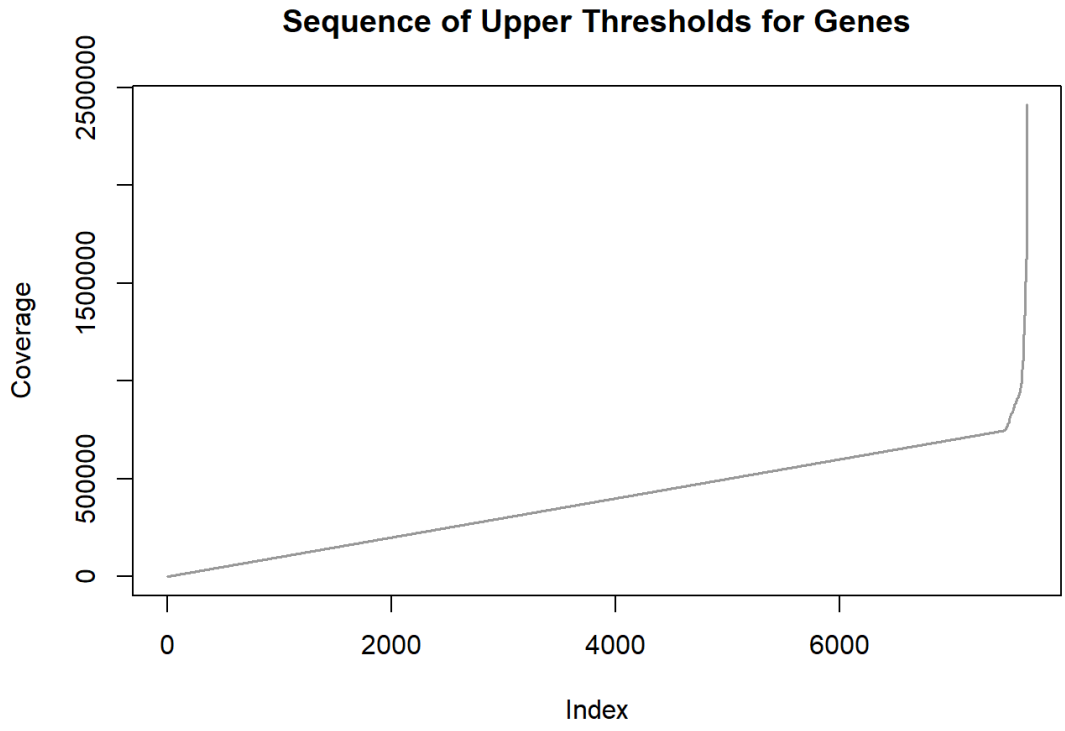
The number of remaining rows is then substracted from the initial number of rows in the dataset. This way the loop gets faster at every iteration, because the size of the daset is significantly reduced. The output of the ultraloop is a dataset with the number of genes/ promoters that would remain at the coverage value that was checked for. This can be than plotted to show us how many

genes remain at each potential upper threshold, allowing us to find the optimum between the two. We do so by using a kneedle (inspired) algorithm to find the highest curvature of the slope.

2.3.2.1 Creating the Sequence for the Ultraloop for Genes

First we want to create the sequence of coverage values the ultraloop will check the remaining number or lines for (g_sequence_ultraloop). It starts with 25 because this is our bottom threshold. We then took the 200 largest coverage vales from the genes dataset and created a sequence which goes from 25 up to the smallest from the top 200 values. Finally a vector was created with the sequence and the top 200 values combined and named g_sequence_ultraloop. Creating this sequence was neccesary for the kneedle analysis to work properly.

```
## Sorting the previously described g_coverage_all dataset in decreasing order. g_coverage all seq is defined as
the 200 largest coverage values from the genes dataset.
g_coverage_all_seq <- sort(g_coverage_all$Coverage, decreasing = TRUE)
g_coverage_all_seq <- g_coverage_all_seq[1:200]
g_coverage_all_seq <- sort(g_coverage_all_seq)

## The g_sequence_ultraloop starts at 25 and goes up to the smallest value of the g_coverage_all_seq dataset in
 steps of 100. Then it goes through the 200 largest coverage values from the genes dataset.
g_sequence_ultraloop <- c(seq(25, min(g_coverage_all_seq), 100), g_coverage_all_seq)

## Plottingour results
plot(g_sequence_ultraloop, type = "l", main = "Sequence of Upper Thresholds for Genes", xlab = "Index", ylab =
"Coverage", col = "#999999", lwd = 1.5)
```




```
## Removing uneccesary data sets
remove(g_coverage_all_seq, g_coverage_all)
```

As we can see by the distinct rise in coverage at an Index of ~7000, we very granularly analyse the beginning area of the coverage and experience much larger jumps in coverage for higher indexed values. This is because the coverage values become more sparse for higher coverages, and at some point it is less intensive to just check how many genes remain at the remaining specific coverage values than to go through in small steps of 100. Furthermore, it is more accurate for the kneedle (inspired) algorithm to have exact data for the first and last coverage value.

2.3.2.2 Creating the Analyse Dataset and Defining the Lower Threshold as ‘g_lowercov’

We created the “analyse” dataset, which is used by the ultaloop. This data set has unique properties that lend themselves to the Ultraloop. The Ultraloop is only used for determining the upper threshold value, and as described above it sees NAs as “trustworthy” values. Therefore it is important that the analyse data set:

- Doesn’t include any coverage values lower than 25, because we are optimizing given that lower threshold

- Turns all it's NAs (which aren't trustworthy) into a value the Ultraloop won't see as trustworthy
- Turns all it's coverage values corresponding to (untrustworthy) NAs into a value the Ultraloop will see as trustworthy

2.3.2.2.1 Setting the beta values with a corresponding coverge values <25 to NAs

```
## Defing analyse dataset as g_patients initially and defining g_lowercov
analyse <- g_patients
g_lowercov <- 25

## Separating data set
analyse_bet <- analyse[, 1:20]
analyse_cov <- analyse[, 21:40]

## Setting the beta values with a corresponding coverge values <25 to NAs using the ifelse function
cov_NAs <- ifelse(analyse_cov < g_lowercov, 1, 0)
analyse_bet[cov_NAs == 1] <- NA

## Removing rows with more than 3 NAs and recombining our data set
analyse <- cbind(analyse_bet, analyse_cov)
analyse <- analyse[!(rowSums(is.na(analyse[1:10])) > 3 |
                       rowSums(is.na(analyse[11:20])) > 3),  ]
```

2.3.2.2.2 NA Formatting for the Analyse Data Set

At this point we need to change the NAs in the beta values to -1 and the coverage values that correspond to NAs into NA as well. This ensures that the NAs thatt exist now, which are areas where data is missing, are not confused for NAs in the ultraloop later, whcih are caused when data is reliable.

```
## Separating data set again to beta values and coverage
analyse_bet <- analyse[ , 1:20]
analyse_cov <- analyse[, 21:40]

## Using ifelse to set NAs in beta to -1 and set coverage values corresponding to NAs to NA
bet_NAs <- ifelse(is.na(analyse_bet), 1, 0 )
analyse_bet[bet_NAs == 1] <- -1
analyse_cov[bet_NAs == 1] <- NA

## Recombining and cleaning data set
analyse <- cbind(analyse_bet, analyse_cov)

## Removing uneccesary data sets
remove(analyse_cov, analyse_bet, cov_NAs, bet_NAs)
```

2.3.2.3 Ultraloop for Genes

The ultraloop goes through the sequence "g_sequence_ultraloop" and creates the dataset described above.

```
## Creating a more aptly named copy of the analyse data set
g_Ultraloop <- analyse
## Defining our output vector
vector_remaining_genes_per_coverage <- c()
## Determining what the original number of rows is
nrow_g <- nrow(g_Ultraloop)

for (y in g_sequence_ultraloop) {

  ## Defining separate dat sets and uppercov
  g_Ultraloop_bet <- g_Ultraloop[ , 1:20]
  g_Ultraloop_cov <- g_Ultraloop[ , 21:40]
  g_uppercov <- y

  ## Using the ifelse functions to record which positions in coverage are trustworthy and turn them into NAs
  cov_NAs <- ifelse(g_Ultraloop_cov < g_uppercov, 1, 0 )
  g_Ultraloop_bet[cov_NAs == 1] <- NA

  ## Recombining the data set and cleaning out of trustworthy rows (>6 NAs), because these will be trustworthy o
n the next iteration of the loop as well
  g_Ultraloop <- cbind(g_Ultraloop_bet, g_Ultraloop_cov)
  g_Ultraloop <-
    g_Ultraloop[!(rowSums(is.na(g_Ultraloop[1:10])) > 6 &
                    rowSums(is.na(g_Ultraloop[11:20])) > 6) , ]

  ## Calculating from the deficit of rows how many would've remained if we had removed the untrustworthy rows at
this coverage threshold, and storing that value
  g_cut <- (nrow_g - nrow(g_Ultraloop))
  vector_remaining_genes_per_coverage <-
    c(vector_remaining_genes_per_coverage, g_cut)

  ## Stopping the loop when the data set has no rows left
  if(g_cut == nrow_g) {
    break()
  }
}

## Removing uneccesary data sets
remove(y, g_Ultraloop, nrow_g, analyse, cov_NAs, g_cut, g_Ultraloop_bet, g_Ultraloop_cov, g_lowercov, g_upperco
v)
```

The output of the loop will be analysed in depth in the kneedle analysis, for now we turn our attention to promoters.

2.3.2.4 Ultraloop for Promoters

We conducted the exact same analysis for promoters, however the code is not shown for space saving purposes. It includes the following steps

- Creating neccesary datasets for the ultraloop for promoters.
- Creating the analyse dataset and defining its variables for promoters.
- Ultraloop code for promoters.

2.3.2.5 Kneedle Analysis for Genes

Now we want to find the highest curviture of the function of the results. For this we used a code based on the kneedle algorithm. This code draws a line from the first point of the function to the last. Then the distance of the line from the function is calculated at each point and plotted. The peak of this new function delivers the highest curviture of slope (Satopaa et al., 2011). We than determine the coverage value in this point and set it as threshold.

We start by formatting the results we obtained from the Ultraloop.

```
## Creating the g_UltraResults_NA3 dataframe
  g_UltraResults_NA3 <- data.frame(vector_remaining_genes_per_coverage)
  g_sequence_ultraloop <- g_sequence_ultraloop[1:nrow(g_UltraResults_NA3)]
  g_UltraResults_NA3 <- cbind(g_UltraResults_NA3, g_sequence_ultraloop)
  g_UltraResults_NA3 <- data.frame(g_UltraResults_NA3)

  colnames(g_UltraResults_NA3) <- c("Remaining_Genes", "Coverage_Value")

## Removing uneccesary data sets
  remove(g_sequence_ultraloop, vector_remaining_genes_per_coverage)
```

Next, we determine what the slope of our line will be, and we determine the exact value of our line for every coverage value that was considered as an upper threshold. Once we have these values we can calculate the difference between our function and our line. From there it is simple to determine the optimum of the kneedle difference.

```
#Determining Kneelde m
  X2 <- max(g_UltraResults_NA3$Coverage_Value)
  X1 <- min(g_UltraResults_NA3$Coverage_Value)
  Y2 <- max(g_UltraResults_NA3$Remaining_Genes)
  Y1 <- min(g_UltraResults_NA3$Remaining_Genes)

  kneedle_m <- ((Y2-Y1)/(X2-X1))

## Determining Kneedle Line and adding it to thee Ultraloop results
  kneedle_line <- ((g_UltraResults_NA3$Coverage_Value-X1)*kneedle_m)
  g_UltraResults_NA3 <- cbind(g_UltraResults_NA3, kneedle_line)

## Determining the Kneedle Line to function difference and adding it to the results
  kneedle_difference <- (g_UltraResults_NA3$Remaining_Genes - g_UltraResults_NA3$kneedle_line)
  g_UltraResults_NA3 <- cbind(g_UltraResults_NA3, kneedle_difference)

## Getting coverage at the knee (highest cuvature)
  g_uppercov_row <- which.max(g_UltraResults_NA3[, 4])
  g_uppercov <- g_UltraResults_NA3[g_uppercov_row,2]

  print(paste0("The optimum upper threshold for the coverage in genes is: ", g_uppercov))
```

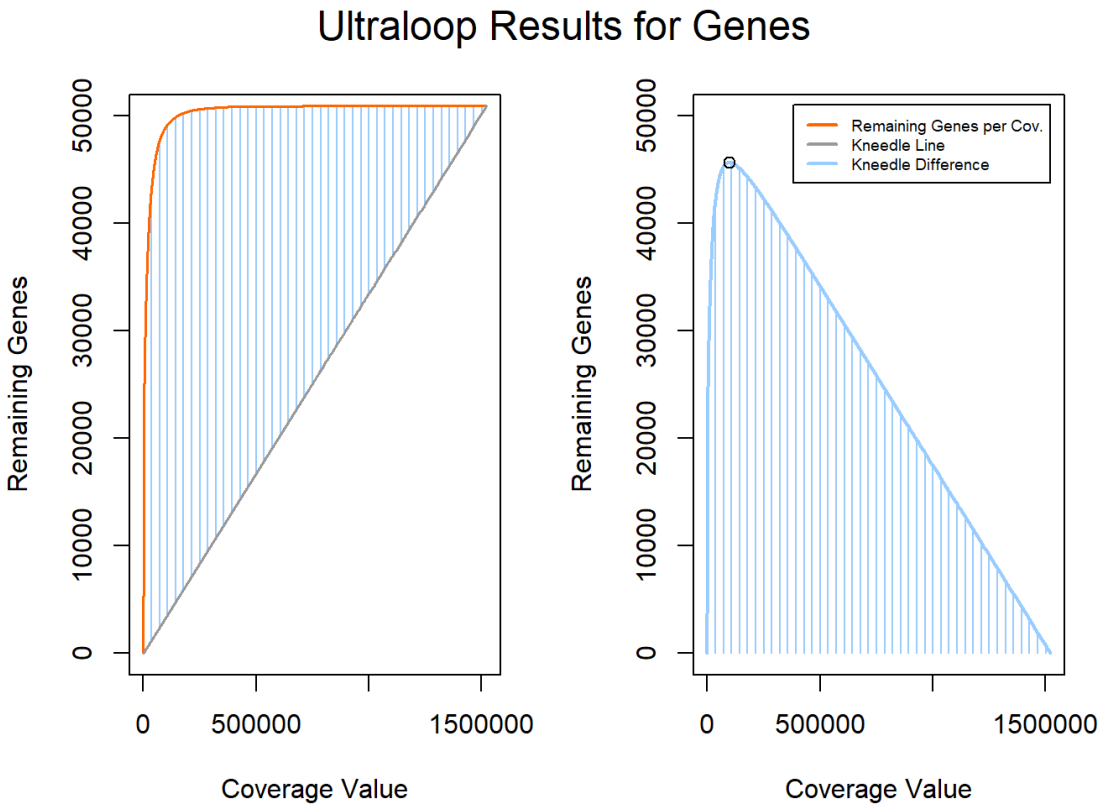
```
## [1] "The optimum upper threshold for the coverage in genes is: 98525"
```

```
## Removing uneccesary data sets
  remove(kneedle_line, kneedle_difference, kneedle_m, g_uppercov_row, X1, X2, Y1, Y2)
```

With the Kneedle analysis complete for genes it's time to plot our results.

2.3.2.6 Plotting Ultraloop Results

```
## Setting up the graphical parameters
par(mfrow=c(1, 2), mar = c(4,4,3,2))

## Plotting the first graph
plot(g_UltraResults_NA3$Coverage_Value, g_UltraResults_NA3$Remaining_Genes, type = "n", main = "", xlab = "Cover
age Value", ylab = "Remaining Genes", ylim  = c(0,50000))
  polygon(c(g_UltraResults_NA3$Coverage_Value, rev(g_UltraResults_NA3$Coverage_Value)), c(g_UltraResults_NA3$Rem
aining_Genes, rev(g_UltraResults_NA3$kneedle_line)), col = "#99CCFF", border = NA, density = 20, angle = 90)
  lines(g_UltraResults_NA3$Coverage_Value, g_UltraResults_NA3$Remaining_Genes, col = "#FF6600", lwd = 1.5)
  lines(g_UltraResults_NA3$Coverage_Value, g_UltraResults_NA3$kneedle_line, col = "#999999", lwd = 1.5)

## Plotting the second Graph
plot(g_UltraResults_NA3$Coverage_Value, g_UltraResults_NA3$kneedle_difference, col = "#99CCFF", lwd = 2, type =
"l", ylim  = c(0,50000), xlab = "Coverage Value", ylab = "Remaining Genes")
  polygon(c(g_UltraResults_NA3$Coverage_Value, rev(g_UltraResults_NA3$Coverage_Value)), c(g_UltraResults_NA3$kne
edle_difference, rep(0, nrow(g_UltraResults_NA3))), col = "#99CCFF", border = NA, density = 20, angle = 90)
  legend(0.38e06, 5.1e4, legend=c("Remaining Genes per Cov.", "Kneedle Line", "Kneedle Difference"), col=c("#FF6
600", "#999999", "#99CCFF"), lty = 1:1, lwd = 2, cex=0.6)
  points(g_uppercov, max(g_UltraResults_NA3[, 4]))

## Adding a Caption
mtext("Ultraloop Results for Genes", cex = 1.5, side = 3, line = - 1.5, outer = TRUE)
```




```
## Removing uneccesary data sets
remove(g_uppercov, g_UltraResults_NA3)
```

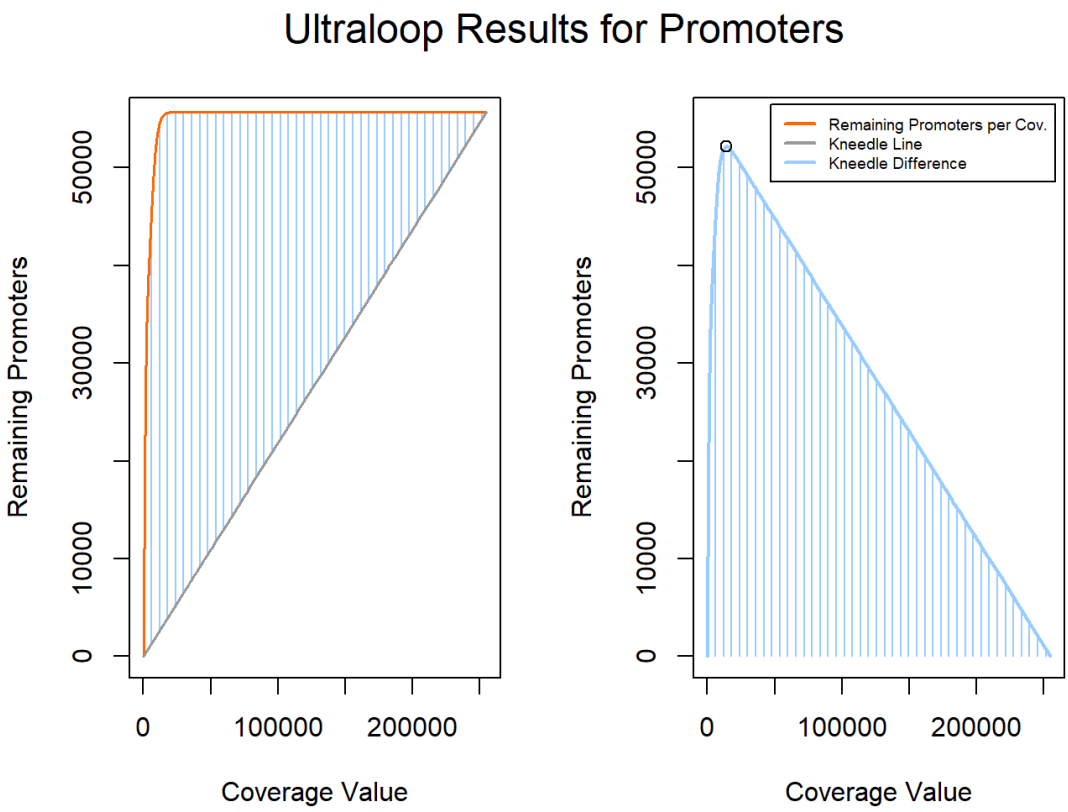
As we can see from the output, the number of genes gained by increasing the upper coverage threshold increases very quickly at first, but then sharply declines later on. This Optimum, which is difficult to visualize in the first graph, becomes much clearer when viewing the distance between the function and kneedle line. The point marked is the optimal upper coverage threshold for genes.

2.3.2.7 Kneedle Analysis for Promoters

Again we used the same procedure for analysing promoters with the kneedle algorithm as for genes and plot the results.

```
## [1] "The optimum upper threshold for the coverage in promoters is: 14175"
```

```
## Setting up the graphical parameters
par(mfrow=c(1, 2), mar = c(4,4,3,2))

## Plotting the first graph
plot(p_UltraResults_NA3$Coverage_Value, p_UltraResults_NA3$Remaining_Promoters, type = "n", main = "", xlab = "C
overage Value", ylab = "Remaining Promoters", ylim  = c(0,55000))
  polygon(c(p_UltraResults_NA3$Coverage_Value, rev(p_UltraResults_NA3$Coverage_Value)), c(p_UltraResults_NA3$Rem
aining_Promoters, rev(p_UltraResults_NA3$kneedle_line)), col = "#99CCFF", border = NA, density = 20, angle = 90)
  lines(p_UltraResults_NA3$Coverage_Value, p_UltraResults_NA3$Remaining_Promoters, col = "#FF6600", lwd = 1.5)
  lines(p_UltraResults_NA3$Coverage_Value, p_UltraResults_NA3$kneedle_line, col = "#999999", lwd = 1.5)

## Plotting the second Graph
plot(p_UltraResults_NA3$Coverage_Value, p_UltraResults_NA3$kneedle_difference, col = "#99CCFF", lwd = 2, type =
"l", ylim  = c(0,55000), xlab = "Coverage Value", ylab = "Remaining Promoters")
  polygon(c(p_UltraResults_NA3$Coverage_Value, rev(p_UltraResults_NA3$Coverage_Value)), c(p_UltraResults_NA3$kne
edle_difference, rep(0, nrow(p_UltraResults_NA3))), col = "#99CCFF", border = NA, density = 20, angle = 90)
  legend(0.47e5, 5.65e4, legend=c("Remaining Promoters per Cov.", "Kneedle Line", "Kneedle Difference"), col=c(
"#FF6600", "#999999", "#99CCFF"), lty = 1:1, lwd = 2, cex=0.6)
  points(p_uppercov, max(p_UltraResults_NA3[, 4]))

## Adding a Caption
mtext("Ultraloop Results for Promoters", cex = 1.5, side = 3, line = - 1.5, outer = TRUE)
```




```
## Removing uneccesary data sets
remove(p_uppercov, p_UltraResults_NA3)
```

Our two outputs, while very similar to genes show us that for one, the coverage values of promoters don't scatter nearly as far, and for another, that the optimum for promoters is more optimal than that for genes.

2.3.3 Cleaning out Unreliable Sequences

Now that the thresholds are identified the beta values with a corresponding coverage value <25 and > 98525 for genes will be set to NAs. For prmoters we used <25 and > 14175 as thresholds based on the kneedle analyisis as well. In both cases we tolerate a maximum of three NAs per cohort.

```
## Separating datasets and defining the threshold as variables
  g_pat_bet <- g_patients[ , 1:20]
  g_pat_cov <- g_patients[, 21:40]

  p_pat_bet <- p_patients[ , 1:20]
  p_pat_cov <- p_patients[, 21:40]

  ## Defining upper and lower coverage values
  g_lowercov <- 25
  g_uppercov <- 98525

  p_lowercov <- 25
  p_uppercov <- 14175

## Setting the unrelaible values to NAs based on the previously described variables (using ifelse)
  g_NAs <- ifelse(g_pat_cov < g_lowercov | g_pat_cov >= g_uppercov, 1, 0 )
  p_NAs <- ifelse(p_pat_cov < p_lowercov | p_pat_cov >= p_uppercov, 1, 0 )

  ## Turning those positions into NA
  g_pat_bet[g_NAs == 1] <- NA
  p_pat_bet[p_NAs == 1] <- NA

  ## Recombining our dataset
  g_pat_covNA <- cbind(g_pat_bet, g_pat_cov)
  p_pat_covNA <- cbind(p_pat_bet, p_pat_cov)

## Leaving out rows that harbour cohorts with too many NAs
    genes_clean <- g_pat_covNA[!(rowSums(is.na(g_pat_covNA[1:10])) > 3 |
                                 rowSums(is.na(g_pat_covNA[11:20])) > 3),  ]

  promoters_clean <- p_pat_covNA[!(rowSums(is.na(p_pat_covNA[1:10])) > 3 |
                                     rowSums(is.na(p_pat_covNA[11:20])) > 3),  ]

## Removing uneccesary data sets
  remove(g_lowercov, g_uppercov, g_NAs, g_pat_bet, g_pat_cov, g_pat_covNA, g_patients)
  remove(p_lowercov, p_uppercov, p_NAs, p_pat_bet, p_pat_cov, p_pat_covNA, p_patients)
```

2.4 Normalization

In oder to be able to perform a Student t-test later we need to nomalize the data. This was done by converting the beta values to M values. This has many advantages: M values are colse to a normal distribution, range from infinity to negative infinity and have a stable variance (Du et al., 2010).

The equation for normaliuation is: $M = \log_2(\frac{\beta}{1-\beta})$

2.4.1 Defining the Function f_BetaToM

This function converts individual beta-values to M-values.

```
## Defining datasets only with patients
genes_clean_reduced <- genes_clean[,1:20]
promoters_clean_reduced <- promoters_clean[,1:20]

## Since beta-values of 0 and 1 render 'Inf' for M-values, we convert 0 and 1 to very close values that do not g
ive the same output
## In this case we go go 10 digits past zero, as this is the maximum number of digits used in our data set
f_BetaToM <- function(x) {

  if(!(is.na(x))){

    if(x == 1){
      x <- 0.9999999999
    }

    if(x == 0) {
      x <- 0.0000000001
    }

    x = log2(x / (1 - x))
  }

  return(x)
}
```

2.4.2 Applying f_BetatoM

Having defined our function for the normalization, we will now apply it across our data sets of patient data. We create the 'M_NA' data sets in the process, which we will be referencinf frequently.

```
g_M_NA <- apply(genes_clean_reduced, c(1,2), f_BetaToM)
g_M_NA <- data.frame(g_M_NA)

p_M_NA <- apply(promoters_clean_reduced, c(1,2), f_BetaToM)
p_M_NA <- data.frame(p_M_NA)

## Removing uneccesary functions
remove(f_BetaToM, genes_clean, genes_clean_reduced)
remove(promoters_clean, promoters_clean_reduced)
```

We have now completed the majority of our quality control and data transformation. All that remains is to create our three data sets that we need for the actual analysis, but the groundwork of cleaning our data is complete.

2.5 Imputation (Mvalues Data Set)

At this point still many NAs remain in the datasets, and since the Mvalues data set can't include NA we want to handle them. We could either get rid of all lines which still include NAs, or impute them. In order to keep as much information as possible we decided to impute the NAs. There are multiple models to do this and we analysed a variety of options. The reason we decided to impute after normalization is that if we were to do it before, we'd be imputing beta values, which are not as easy to statistically impute as M-values.

Rather than simply impute the mean for our NAs we decided to do a more in depth analysis of the effects of varying imputation models on our data. Specifically, we were interested in how the imputation would impact the T-tests we are going to conduct further down the road. We also conclude with a look at which imputation models hold true to the original data structure the best.

In the first part of our analysis, we study the false positive rate by applying our imputation models in test scenarios where all tests are false, and looking at what the histogramms of p-values look like. To clarify, we take two samples of the same distribution, "lose" three values and impute them. We then perform a significance test. As the original distributions were identical, in all cases the p-value distributionn should be equally distributed. We look at the histogramms for the different methods and infer the deviance from this normal distribution for every method.

This is the metric we use to determine which imputation model produces the most realistic rate of false positives. Because of multiple testing correction later, it is especially important to us that the false positive rate remain quantifyable, and therefore, realsitic.

The direct relevance of this model to our data set is that it replicates a scenario in which imputation has the largest influence on our data set. Since our data set allows a maximum of 3 NAs per cohort, the highest degree of imputation would be 3 values in two cohorts that are compared to each other. Since we care about the false positive rate the most, we also model a situation in which a false positive can occur.

2.5.1 Perfect Imputation

As a yardstick to measure all the other methods by, we will start by defining the perfect case scenario.

We take two random samples from an identical standard normal distribution. Then we would lose 3 values and need to impute them. In the perfect scenario the new values would come from the same original standard normal distribution. Therefore we simply do not remove the three NAs in the first place.

```
## Creating a data frame that will serve as the set.seed imputs and dictate the number of iterations the model g
oes through
j <- data.frame(c(seq(1,500000, 1)))
## Creating jcount, a variable that will help us mark whenever 1% of a task has been done (later in the progress
bars)
jcount <- floor(nrow(j)/100)

## We define a function that models what a perfect imputation would look like.
f_model_perfect_imputation <- function(x) {

  ## We set up a progress bar for ease of the user
  if(x %% jcount == 0) {
    cat("|")
  }

  ## We set the seed for reproduceable results (note that we set a different seed for each sample!), and then dr
aw our random samples
  set.seed(x)
  sample1 <- c(rnorm(10))
  set.seed(-x)
  sample2 <- c(rnorm(10))

  ## We "lose and reimpute perfectly" by doing nothiing

  ## We perform a significance test to determine whether our samples originate from the same distribution
  t_testresults <- t.test(sample1, sample2, var.equal = TRUE)

  ## We return the p-value
  return(c(t_testresults["p.value"]))
}

## Here we apply our new function across j and format our results
perfectimputation <- apply(j, 1, f_model_perfect_imputation)
```

```
## ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
perfectimputation <- unlist(perfectimputation)

## A preview of how th false positive rate might be skewed
print(paste0("In perfect imputation ", round(ecdf(perfectimputation)(0.05)*100, digits = 2), " % of p-values lie
under the 5% mark. An alpha of 5% causes ", round(ecdf(perfectimputation)(0.05)/0.05, digits = 2), " times more
 false positives than would be expected."))
```

```
## [1] "In perfect imputation 5.01 % of p-values lie under the 5% mark. An alpha of 5% causes 1 times more false
positives than would be expected."
```

At this point we will not plot the results yet, as they will be more illuminating when they are all simultaneously compared. However, this output shows us that the perfect imputation is nearing the proper equal distribution upon repeated iterations of the model.

2.5.2 Mean Imputation

The classic method for imputing values is to insert the mean of the present values for the NAs. To model this, we will be taking two samples from identical distributions, “losing” three values from them and impute them with the mean. We will theen perform a t-test to try and determine wether or not these two samples are significantly different from one another. Once this process is executed 500.000 times we will see if the distribution of p-values differs from the perfect imputation.

```
## We define a function that models what a mean imputation would look like.
    f_model_mean_imputation <- function(x) {

      ## We set up a progress bar for ease of the user
      if(x %% jcount == 0) {
        cat("|")
      }

      ## We set the seed for reproduceabble results
      set.seed(x)

      ## We take a sampe of seven
      sample1 <- c(rnorm(7))

      ## We impute three more values to make up for the ones we "lost"
      sample_impute <- c(rep(mean(sample1), 3))
      sample1 <- c(sample1, sample_impute)

      ## Same process as above
      set.seed(-x)
      sample2 <- c(rnorm(7))
      sample_impute <- c(rep(mean(sample2), 3))
      sample2 <- c(sample2, sample_impute)

      ## We apply a t-test for significance and return the p-value
      t_testresults <- t.test(sample1, sample2, var.equal = TRUE)
      return(c(t_testresults["p.value"]))
    }

## Here we apply our new function across j and format our results
meanimputation <- apply(j, 1, f_model_mean_imputation)
```

```
## ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
meanimputation <- unlist(meanimputation)

## A preview of how th false positive rate might be skewed
print(paste0("In mean imputation ", round(ecdf(meanimputation)(0.05)*100, digits = 2) , " % of p-values lie unde
r the 5% mark. An alpha of 5% causes ", round(ecdf(meanimputation)(0.05)/0.05, digits = 2), " times more false p
ositives than would be expected."))
```

```
## [1] "In mean imputation 17.73 % of p-values lie under the 5% mark. An alpha of 5% causes 3.55 times more fals
e positives than would be expected."
```

This output shows us that mean imputation causes a great deal more false positives than a normal equal distribution, something we will see in greater detail later.

2.5.3 Statistical Imputation

In this approach that is also common we will assume that the M-distribution is close enough to the normal distribution for the model using the normal distribution as samples to produce the same effects. This assumption, that the m distribution and normal distribution are identical for our purposes, is a repeated assumption in our analysis.

Since we know that the original distribution of our M-values is an M-dsitribution we could attempt to impute NAs statistically, that is to take random values from a normal distribution with the same mean and standard deviation as the sample. In this manner we mimic what the original distribution looked like by estimating its qualities based on the sample. This process is referred to as Rnorm Imputation because it makes use of the function rnorm.

```
## We define a function that models what an rnorm imputation would look like.
  f_model_rnorm_imputation <- function(x) {

    ## We set up a progress bar for ease of the user
    if(x %% jcount == 0) {
      cat("|")
    }

    ## We set the seed for reproduceabble results
    set.seed(x)

    ## We take a sampe of seven
    sample1 <- c(rnorm(7))

    ## We set a different seed for the imputation
    set.seed(x + (nrow(j)))

    ## We impute three more values by mimicing the original distribution
    sample_impute <- c(rnorm(mean = mean(sample1), sd = sd(sample1), 3))
    sample1 <- c(sample1, sample_impute)

    ## We repeat the same process as above
    set.seed(-x)
    sample2 <- c(rnorm(7))
    set.seed(-x - (nrow(j)))
    sample_impute <- c(rnorm(mean = mean(sample2), sd = sd(sample2), 3))
    sample2 <- c(sample2, sample_impute)

    ## We apply a t-test for significance and return the p-value
    t_testresults <- t.test(sample1, sample2, var.equal = TRUE)
    return(c(t_testresults["p.value"]))

  }


## Here we apply our new function across j and format our results
rnormimputation <- apply(j, 1, f_model_rnorm_imputation)
```

```
## ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
rnormimputation <- unlist(rnormimputation)

## A preview of how th false positive rate might be skewed
print(paste0("In rnorm imputation ", round(ecdf(rnormimputation)(0.05)*100, digits = 2), " % of p-values lie und
er the 5% mark. An alpha of 5% causes ", round(ecdf(rnormimputation)(0.05)/0.05, digits = 2), " times more false
positives than would be expected."))
```

```
## [1] "In rnorm imputation 14.13 % of p-values lie under the 5% mark. An alpha of 5% causes 2.83 times more fal
se positives than would be expected."
```

A similar output as mean impuatation.

2.5.4 Rnorm10 Imputation

We hypothesized that imputing the mean into a data set would increase its rate of significant results because it decreases the standard deviation of the sample. This inspiration lead us to extrapolate that perhaps the shifted rate of false positives derives in the altered mean and standard deviation before and after imputation. In search for an improved method of imputation, we designed “Rnorm10 Imputation”, which repeats the Rnorm Imputation process 10 times generating 10 sets of 3 values that ought to be imputed. It then selects for the set of the 10 that alters the mean and standard deviation of the sample as little as possible, whereby mean and standard deviation are weighted equally.

```
## We define a function that models what an rnorm10 imputation would look like.
f_model_rnorm10_imputation <- function(x) {

  ## We set up a progress bar for ease of the user
  if(x %% jcount == 0) {
    cat("|")
  }

  ## We set the seed to get reproduceable results
  set.seed(x)

  ## We take our first sample of seven values
  sample1 <- c(rnorm(7))

  ## We determine what the mean and standard deviation are before imputation
  pre_imputation_stats <- c(mean(sample1), sd(sample1))

  ## We define the data set that our 10 sets of 3 values will be going into
  winnercandidates <- c()

  ## We start the loop that will perform rnorm imputation 10 times
  for (i in 1:10) {

    ## Set seed for reproducibility
    set.seed(x + (i * nrow(j)))

    ## rnorm imputation
    sample_impute <- c(rnorm(mean = mean(sample1), sd = sd(sample1), 3))

    ## we create a vector of our sample and the imputed values
    sample1_post_imputation <- c(sample1, sample_impute)

    ## We collect the new mean and standard deviation post imputation
    post_imputation_stats <- c(mean(sample1_post_imputation), sd(sample1_post_imputation))

    ## We determine how much the mean and sd have changed, and quantify this
    impdiff <- (post_imputation_stats - pre_imputation_stats)
    impdiff <- (sum(abs(impdiff)))

    ## We save our set of three values alongside the impdiff corresponding to them
    candidate <- c(impdiff, sample_impute)
    winnercandidates <- cbind(winnercandidates, candidate)


  }

  ## We format the winnercandidates data frame
  winnercandidates <- data.frame(winnercandidates)

  ## We determine which set of values has the lowest impdiff
  winnercol <- which.min(winnercandidates[1,])

  ## We determine which set of three values corresponds to the lowest impdiff
  winner <-winnercandidates[2:nrow(winnercandidates), winnercol]

  ## Sample1 is imputed with the values corresponding to the lowest impdiff
  sample1 <- c(sample1, winner)

  ## We repeat the identical process for sample2
  set.seed(-x)
  sample2 <- c(rnorm(7))
  preextendstats <- c(mean(sample2), sd(sample2))

  winnercandidates <- c(rep(0, 4))

  for (i in 1:10) {
    set.seed(-x - (i * nrow(j)))
    sample_impute <-
```

```
      c(rnorm(mean = mean(sample2), sd = sd(sample2), 3))
    sample2a <- c(sample2, sample_impute)

    postextendstats <- c(mean(sample2a), sd(sample2a))
    impdiff <- (postextendstats - preextendstats)
    impdiff <- (sum(abs(impdiff)))

    candidate <- c(impdiff, sample_impute)
    winnercandidates <- cbind(winnercandidates, candidate)


  }

  winnercandidates <- data.frame(winnercandidates)
  winnercandidates <-
    winnercandidates[, 2:ncol(winnercandidates)]
  winnercol <- which.min(winnercandidates[1,])
  winner <-
    winnercandidates[2:nrow(winnercandidates), winnercol]

  sample2 <- c(sample2, winner)

  ##  We apply a t-test to determine whether sample1 and sample2 are significantly different
  t_testresults <- t.test(sample1, sample2, var.equal = TRUE)

  ## We return the p-value
  return(c(t_testresults["p.value"]))
}


## Here we apply our new function across j and format our results
rnorm10imputation <- apply(j, 1, f_model_rnorm10_imputation )
```

```
## ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
rnorm10imputation <- unlist(rnorm10imputation)

## A preview of how th false positive rate might be skewed
print(paste0("In rnorm10 imputation ", round(ecdf(rnorm10imputation)(0.05)*100, digits = 2), " % of p-values lie
under the 5% mark. An alpha of 5% causes ", round(ecdf(rnorm10imputation)(0.05)/0.05, digits = 2), " times more
 false positives than would be expected."))
```

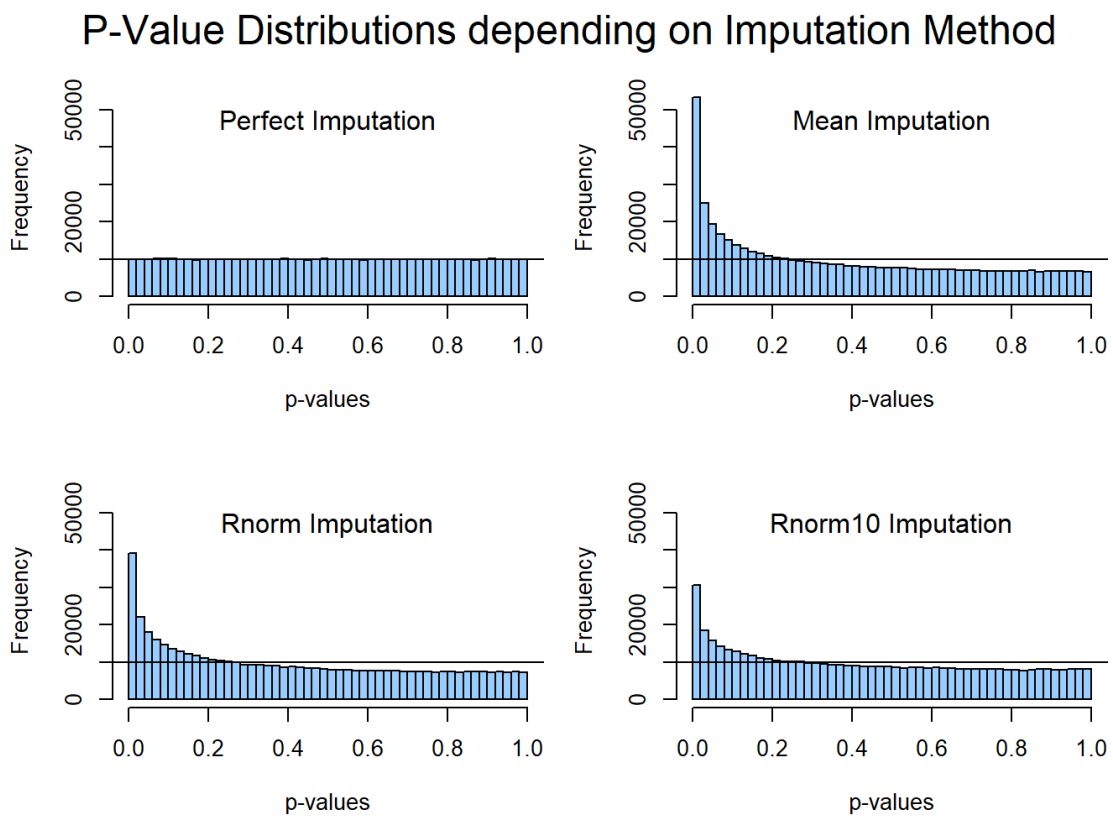
```
## [1] "In rnorm10 imputation 11.49 % of p-values lie under the 5% mark. An alpha of 5% causes 2.3 times more fa
lse positives than would be expected."
```

A similar output as rnorm imputation.

2.5.5 Summarizing and Visualizing our Models

Here we will visualize and summarize the effect of the different imputation types.

```
## We calculate the mode of the mean imputation histogramm and the average value of the perfect imputation histo
gramm, these well help us keep track of the worst and best case scenario in every histogramm
  ## We start by creating histogramm objects and working with their count information
  mean_imputation_hist     <- hist(meanimputation, 40, plot = FALSE)
  perfect_imputation_hist  <- hist(perfectimputation, 40, plot = FALSE)

  ## We take the mean of al the counts for the value an equal distribution would produce at every bin
  perfectaverage <- mean(perfect_imputation_hist$counts)
  ## We take the mode of the mean distribution because we suspect it to be the highest value in all the plots
  meanmode <- max(mean_imputation_hist$counts)


## We set up the layout of the output
par(mfrow = c(2,2), mar = c(4,4,3,1))

## Our four historgamms for our results
hist(perfectimputation, 40, ylim = c(0, meanmode), col = "#99CCFF", main = "", xlab = "p-values")
title("Perfect Imputation", font.main = 1, line = -1.5)
abline(h = perfectaverage)

hist(meanimputation, 40, ylim = c(0, meanmode), col = "#99CCFF", main = "", xlab = "p-values")
title("Mean Imputation", font.main = 1, line = -1.5)
abline(h = perfectaverage)

hist(rnormimputation, 40, ylim = c(0, meanmode), col = "#99CCFF", main = "", xlab = "p-values")
title("Rnorm Imputation", font.main = 1, line = -1.5)
abline(h = perfectaverage)

hist(rnorm10imputation, 40, ylim = c(0, meanmode), col = "#99CCFF", main = "", xlab = "p-values")
title("Rnorm10 Imputation", font.main = 1, line = -1.5)
abline(h = perfectaverage)

mtext("P-Value Distributions depending on Imputation Method", cex = 1.5, side = 3, line = - 1.5, outer = TRUE)
```




```
## Removing uneccesary data
remove(meanmode, perfectaverage, f_model_mean_imputation, f_model_perfect_imputation, f_model_rnorm_imputation,
 f_model_rnorm10_imputation, perfect_imputation_hist, mean_imputation_hist, perfectimputation, meanimputation, r
normimputation, rnorm10imputation)
```

The output shows us that while our differing methods produce differing results, and rnorm10 does actually perform better than any other (non-perfect) method, all of them deviate drastically from the perfect case scenario (we also modeled imputation using the median and a model fo hot-decking, both of which showed similar results). This makes sense, as a sample size of n can only represent it's original distribution so well. Even if we imputed and did not change the mean or standard deviation at all, we would still be increasing n, i.e. our confidence in the information, without adding any information to the data set. It is intuitive that increasing our confidence without adding the basis for said confidence causes false positives.

In the face of these results we decided that no imputation method would be acceptably accurate for our t-tests, and we decided on a "null model" of not imputing at all. After all, a t-test only needs certain inputs that can be collected from a data set with NAs still in it, so it is not necessary to perform imputation for the t-tests. With no imputaton, the rate of false positives cannot be affected, and our multiple testing corrected p-values will be realistic and accurate. While we trade for power in this scenario, we feel the trade is worth it in our context.

However, it is required for all cells to be assigned a value for the PCA, so we still need to find a solution for imputation there. The above plots seem to indicate that rnorm10 is a good option. To create a comparative spectrum we will analyse a perfect case scenario, the rnorm10 scenario, and the standard scenario, which is mean imputation.

2.5.6 Analysing Data Structure for Mean Imputation and rnorm10 Imputation

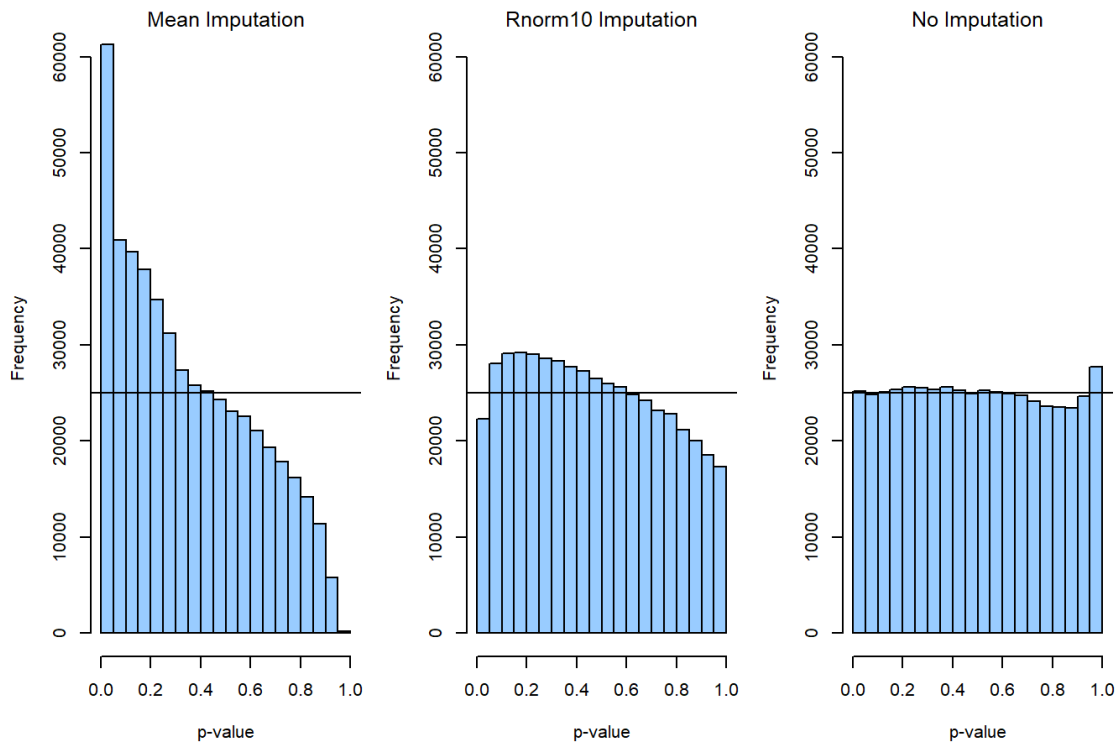
If a data set is normally distributed, it's entire data structure can be given with its mean and standard deviation. We have already compared how the mean and standard deviation are altered by mean and rnorm imputation by comparing them with t-tests earlier. These results have clearly indicated that mean imputation makes two distributions appear more distinct than rnorm10 imputation does, telling us that given normal distribution, rnorm10 reports results truer to the original than mean imputation.

Having analysed these two factors it is time to answer whether or not the the remaining element of data structure, i.e. the distribution of the data, is affected by Imputation and how. After all it doesn't matter how true to the original the mean and standard deviation are if the data does not actually fit a normal distribution. We analyse the normailty of the distributions by performing a Shapiro-Wilk test on our data to determine if it is significantly non-normally distributed. We will consider p-values below 0.1 significant, as was suggested by Royston (Royston, 1982).

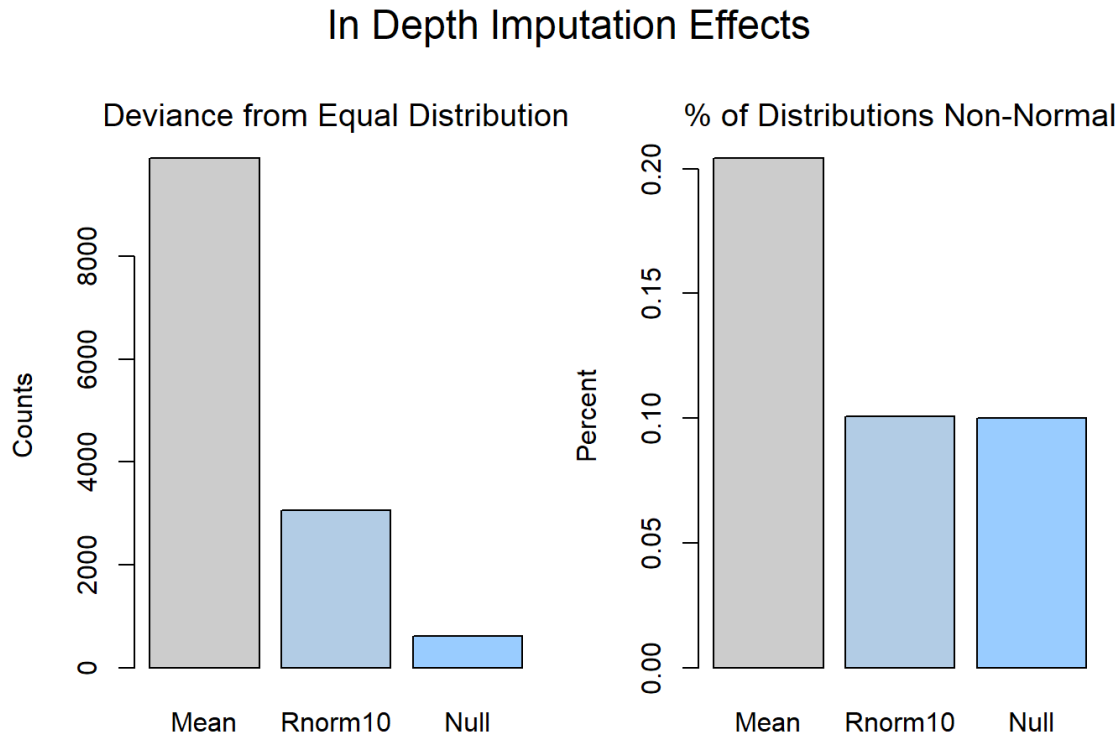
```r
## We define a function that will create a sample and test its normality, while also imputing it twice and testi
ng it's normality then.
f_model_data_structure <- function(x) {

  ## Progress bar
  if(x %% jcount == 0) {
    cat("|")
  }

  ## We get a reproduceable sample1
  set.seed(x)
  sample1 <- c(rnorm(7))

  ## We perform rnorm10 imputation on sample1 to produce an imputed sample called sample_rnorm10
  pre_imputation_stats <- c(mean(sample1), sd(sample1))
  winnercandidates <- c()

  for (i in 1:10) {
    set.seed(x + (i * nrow(j)))

    sample_impute <- c(rnorm(mean = mean(sample1), sd = sd(sample1), 3))
    sample1_post_imputation <- c(sample1, sample_impute)

    post_imputation_stats <- c(mean(sample1_post_imputation), sd(sample1_post_imputation))
    impdiff <- (post_imputation_stats - pre_imputation_stats)
    impdiff <- (sum(abs(impdiff)))

    candidate <- c(impdiff, sample_impute)
    winnercandidates <- cbind(winnercandidates, candidate)

  }

  winnercandidates <- data.frame(winnercandidates)
  winnercol <- which.min(winnercandidates[1,])
  winner <-winnercandidates[2:nrow(winnercandidates), winnercol]

  ## We get rnorm10
  sample_rnorm10 <- c(sample1, winner)

  ## We perform mean imputation on sample1 to produce an imputed sample called sample_mean
  sample_impute <- c(rep(mean(sample1), 3))
  sample_mean <- c(sample1, sample_impute)

  ## We perform our three shapiro-Wilk tests
  data_structure_sample1 <- shapiro.test(sample1)$p.value
  data_structure_rnorm10 <- shapiro.test(sample_rnorm10)$p.value
  data_structure_mean <- shapiro.test(sample_mean)$p.value

  ## We report the p-value of our non-imputed sample, our rnorm10 sample and our mean imputed sample
  return(c(data_structure_sample1, data_structure_mean, data_structure_rnorm10))
}


## Here we apply our new function across j and formatting our resullts
imputation_data_structure <- apply(j, 1, f_model_data_structure)
```

```
## ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
imputation_data_structure <- t(imputation_data_structure)

## Creating histogramm objects for an in depth comparison
mean_imputation_hist     <- hist(imputation_data_structure[, 2], 30, plot = FALSE)
rnorm10_imputation_hist  <- hist(imputation_data_structure[, 3], 30, plot = FALSE)
no_imputation_hist       <- hist(imputation_data_structure[, 1], 30, plot = FALSE)

## Determining what frequencies an equal distribution would give in a histogramm
perfectaverage <- mean(no_imputation_hist$counts)

## Dtermining the highest value in the data set
data_set_mode <- max(c(mean_imputation_hist$counts, rnorm10_imputation_hist$counts, no_imputation_hist$counts))

## Determining how strongly our three distributions bin counts deviate from the value an equal distribution woul
d have in each bin
mean_deviance <- mean(abs(mean_imputation_hist$counts - perfectaverage))
rnorm10_deviance <- mean(abs(rnorm10_imputation_hist$counts - perfectaverage))
no_imputation_deviance <- mean(abs(no_imputation_hist$counts - perfectaverage))

deviance_from_equal <- c(mean_deviance, rnorm10_deviance, no_imputation_deviance)

## Determining what percentage of of vectors would have been considered non-normally distributed
percent_normal_mean <- ecdf(imputation_data_structure[, 2])(0.1)
percent_normal_rnorm10 <- ecdf(imputation_data_structure[, 3])(0.1)
percent_normal_no_imputation <- ecdf(imputation_data_structure[, 1])(0.1)

percent_non_normal <- c(percent_normal_mean, percent_normal_rnorm10, percent_normal_no_imputation)

## Plotting our P-Value Histogramms
par(mfrow = c(1, 3), mar = c(4, 4, 4, 1))

hist(imputation_data_structure[, 2], 30, ylab = "Frequency", xlab = "p-value", col = "#99CCFF", ylim = c(0, data
_set_mode), main = "")
title("Mean Imputation", line = -0.25, font.main = 1)
abline(h = perfectaverage)

hist(imputation_data_structure[, 3], 30,ylab = "Frequency", xlab = "p-value", col = "#99CCFF", ylim = c(0, data_
set_mode), main = "")
title("Rnorm10 Imputation", line = -0.25, font.main = 1)
abline(h = perfectaverage)

hist(imputation_data_structure[, 1], 30, ylab = "Frequency", xlab = "p-value", col = "#99CCFF", ylim = c(0, data
_set_mode), main = "")
title("No Imputation", line = -0.25, font.main = 1)
abline(h = perfectaverage)

mtext("P-Value Distributions for Normality", cex = 1.5, side = 3, line = - 2, outer = TRUE)
```




```
## Plotting our In Depth Results
par(mfrow = c(1, 2), mar = c(4, 4, 5, 1))

barplot(deviance_from_equal, names.arg = c("Mean", "Rnorm10", "Null"), ylab = c("Counts"), font.main = 1, col =
 c("#CCCCCC", "#B2CCE5", "#99CCFF"))
title("Deviance from Equal Distribution", line = 1, font.main = 1)

barplot(percent_non_normal, names.arg = c("Mean", "Rnorm10", "Null"), ylab = c("Percent"), font.main = 1, col =
 c("#CCCCCC", "#B2CCE5", "#99CCFF"))
title("% of Distributions Non-Normal", line = 1, font.main = 1)

mtext("In Depth Imputation Effects", cex = 1.5, side = 3, line = - 1.5, outer = TRUE)
```



```
## Removing uneccesary Data Sets
remove(data_set_mode, deviance_from_equal, f_model_data_structure, imputation_data_structure, j, jcount, mean_de
viance, mean_imputation_hist, no_imputation_hist, no_imputation_deviance, percent_non_normal, percent_normal_mea
n, percent_normal_no_imputation, percent_normal_rnorm10, perfectaverage, rnorm10_deviance, rnorm10_imputation_hi
st)
```

As both of these outputs show us, rnorm10 did not perform perfectly but certainly outperformed mean imputation. The p-value distributions may look a little unusual (dip and peak in non-imputation), this likely has to do with the fact that the Shapiro-Wilk test produces estimated p-values instead of exact ones. However, it is still quite apparent that the rnorm10 distribution looks much closer to an equal distribution than the mean distribution.

A more in depth quantitave look at the data shows us the deviance from the equal distribution and the percentage of non-normal distributions present. The firstgraph shows us the mean absolute difference between thee number of counts per bin and the number that would be expected from an equal distribution. As we can see, mean imputation has about three times as much deviance from an equal distribution than Rnorm10. The second graph shows us what percentage of the generated distributions would be considered non-normal with a significance threshold of 0.1%. Rnorm10 shows an almost identical amount of Non-normal models as the nulll model, which both fall right in line with an alpha of 10%. Mean imputation on the other hand shows twice as many models as would be expected to be non-normal.

In summary, if a sample of 7 normally distributed values is imputed with Rnorm10 Imputation, it will have a significantly more realistic mean and standard deviation representing the original distribution, and it will keep the normal distribution of the sample intact. If another of the methods we tested is used instead, these qualities are not a given.

2.5.7 Applying Rnorm10 Imputation to our data sets

We now apply rnorm10 Imputation to our finalize our Mvalues data sets. We apply rnorm10 imputation identically to how we performed it in the section above, so to save space, the code for this is omitted. At the end, we preview these new Mvalue data sets.

	AML	AML	AML	Mono	Mono	Mono
ENSG00000223972	1.716	0.401	3.198	2.312	1.608	1.864
ENSG00000227232	1.175	0.633	1.121	0.877	1.218	0.904
ENSG00000278267	3.907	5.459	33.219	3.298	33.219	3.248
ENSG00000238009	2.859	2.271	3.547	2.327	2.226	2.423
ENSG00000239945	1.306	1.054	1.232	1.805	2.292	1.716
ENSG00000233750	2.872	4.275	3.898	3.758	3.901	3.687
ENSG00000268903	3.667	4.402	4.134	4.188	4.008	3.663
ENSG00000269981	5.004	5.178	5.464	5.038	4.728	5.030
ENSG00000241860	3.264	2.591	3.987	3.458	3.735	3.754
ENSG00000279928	2.483	3.066	3.250	2.142	1.841	2.216

These are our new Mvalue data sets.

2.6 Creating T datasets

In the following we will collect all the t-test input statistics we need to perform t-tests later on. Note that we are referencing the M_NA data sets, which have not been imputed.

We start by splitting up our data sets and then continue by defining and applying a function that collects the necessary inputs. This includes mean value (m), standard deviation (sd) and number of values (n) for each cohort.

```
## Splitting datasets between our cohorts
g_M_NA_AML <- g_M_NA[, 1:10]
g_M_NA_mon <- g_M_NA[, 11:20]

p_M_NA_AML <- p_M_NA[, 1:10]
p_M_NA_mon <- p_M_NA[, 11:20]

## Defining the function f_MtoT
f_MtoT <- function(x) {


  m <- mean(x, na.rm = TRUE)
  sd  <- sd(x, na.rm = TRUE)
  n <- sum(!(is.na(x)))

  return(c(m,sd,n))
}

## Using the apply function to run f_MtoT
  g_T_AML <- t(apply(g_M_NA_AML, 1, f_MtoT))
  g_T_mon <- t(apply(g_M_NA_mon, 1, f_MtoT))

  p_T_AML <- t(apply(p_M_NA_AML, 1, f_MtoT))
  p_T_mon <- t(apply(p_M_NA_mon, 1, f_MtoT))

## Naming and formatting the g_T and p_T datasets
  g_T <- cbind(g_T_AML, g_T_mon)
  colnames(g_T) <- c("Mean_AML", "SD_AML", "N_AML", "Mean_Mono", "SD_Mono", "N_Mono")
  g_T <- data.frame(g_T)

  p_T <- cbind(p_T_AML, p_T_mon)
  colnames(p_T) <- c("Mean_AML", "SD_AML", "N_AML", "Mean_Mono", "SD_Mono", "N_Mono")
  p_T <- data.frame(p_T)

## Previewing our new data sets
  kable(head(g_T, n = 10), align = "c", digits = 3)
```

	Mean_AML	SD_AML	N_AML	Mean_Mono	SD_Mono	N_Mono
ENSG00000223972	1.549	1.162	9	1.462	0.569	8
ENSG00000227232	0.558	0.357	10	0.601	0.334	10
ENSG00000278267	7.557	9.037	10	7.309	9.136	10
ENSG00000238009	2.372	0.654	10	2.260	0.309	10
ENSG00000239945	0.917	0.740	7	1.903	0.369	8
ENSG00000233750	3.514	0.710	10	3.767	0.498	10
ENSG00000268903	4.543	0.799	10	4.678	0.810	10
ENSG00000269981	4.883	0.574	10	5.378	0.597	10
ENSG00000241860	3.249	1.447	10	2.620	1.059	9
ENSG00000279928	2.585	0.782	10	2.059	0.183	10

```
## Removing unneccesary datasets.
  remove(g_T_AML, g_T_mon, p_T_AML, p_T_mon, f_MtoT, g_M_NA, p_M_NA)
```

2.7 Generating the Resource Data Set

The resource dataset (g_Resource, p_Resource) includes the log2 foldchange, the ensembl ID and the symbols, if available, for genes and promoters each. This dataset is used later to search for the DMR suspects.

The foldchange describes how much the methylation changes among the patient groups in one row. It was calculated by first converting the mean m values back to beta-values in each row for one cohort. The reason why we used the means is that in an M distribution the mean represents the most common i.e normal value, so converting the mean of the M-values will also give us the most common value amongst the beta-values. Then log2 fold change of the newly gained beta-values is calculated.

2.7.1 Collecting the Mean Values of Genes and Promoters for each Cohort

```
## Collecting the mean of the m values, as per the m distribution, the mean represents the most common i.e. norm
al value from each cohort by promoters or genes.
g_RM_AML_m <- data.frame(apply(g_M_NA_AML, 1, mean, na.rm = TRUE))
g_RM_mon_m <- data.frame(apply(g_M_NA_mon, 1, mean, na.rm = TRUE))

p_RM_AML_m <- data.frame(apply(p_M_NA_AML, 1, mean, na.rm = TRUE))
p_RM_mon_m <- data.frame(apply(p_M_NA_mon, 1, mean, na.rm = TRUE))

## Removing Unecessary data sets
remove(g_M_NA_AML, g_M_NA_mon)
remove(p_M_NA_AML, p_M_NA_mon)
```

Converting mean M values back to beta to be able to calculate the foldchange. For this we used the f_MtoBeta function. It makes use of the inverse of the previous equation: $\beta = \frac{2^M}{1+2^M}$

2.7.2 Defining and Applying the f_MtoBeta Function

```
## Defining a function that converts the "normal" M value for each sequence's cohort to the corresponding "norma
l" beta value
f_MtoBeta <- function(x) {

  x = (2^x)/(1 + 2^x)
  return(x)
}

## Here we apply f_MToBeta across all the means of the cohorts we have previously collected
g_RB_AML_m <- apply(g_RM_AML_m, c(1,2), f_MtoBeta)
g_RB_mon_m <- apply(g_RM_mon_m, c(1,2), f_MtoBeta)
p_RB_AML_m <- apply(p_RM_AML_m, c(1,2), f_MtoBeta)
p_RB_mon_m <- apply(p_RM_mon_m, c(1,2), f_MtoBeta)

## Removing uneccesary data sets
remove(f_MtoBeta, g_RM_AML_m, g_RM_mon_m)
remove(p_RM_AML_m, p_RM_mon_m)
```

When we applied the function we noticed a few oddities. If the original gene had a value of 1 for all samples in beta, the result here is 1, even though we changed those 1s to 0.999…9, so R has rounded here. However, genes that were 0 in bet are returned as 0.00…01 here, because we also changed them along the way. R has not rounded here. Very strange but coincidentally convenient for future formatting. Otherwise we wouldve had to set 0.9…9 to 1, and leave 0.0…1 because we can't divide by zero.

2.7.3 Calculating log2 Foldchange and Extracting Symbols

Here we calculate the log2 Fold Change of the means from Mono to AML. We believe that having the fold change show changes from the healthy to unhealthy cohort helps visualize the changes better. In the process we also get the symbols corresponding to very sequence from a far earlier data set.

```
## Calculating log2 foldchange from Mon to AML (so that the "normal" variety / 0 fold change is the healthy one)
g_RB_foldchange <- log2(g_RB_AML_m/g_RB_mon_m)
p_RB_foldchange <- log2(p_RB_AML_m/p_RB_mon_m)

## Extracting symbols for later from genes and promoters
g_R_Symbols <- data.frame(genes_data_frame$symbol)
rownames(g_R_Symbols) <- rownames(genes_data_frame)

p_R_Symbols <- data.frame(promoters_data_frame$symbol)
rownames(p_R_Symbols) <- rownames(promoters_data_frame)

## Removing uneccesary data sets
remove(g_RB_AML_m, g_RB_mon_m, genes_data_frame)
remove(p_RB_AML_m, p_RB_mon_m, promoters_data_frame)
```

2.7.4 Formatting the Resource Data Set

We now perform a final formatting of the data set and give a preview of what it looks likes.

```
## Formatting the g_resource data set: merging with foldchange and symbols
g_Resource <- merge(g_RB_foldchange, g_R_Symbols , by = 0, all = FALSE)
g_Resource <- g_Resource[, c(1,3,2)]
rownames(g_Resource) <- g_Resource$Row.names
colnames(g_Resource) <- c("Ensembl_ID", "Symbols", "Foldchange_Beta")

## Formatting the p_resource data set for promoters.
p_Resource <- merge(p_RB_foldchange, p_R_Symbols , by = 0, all = FALSE)
p_Resource <- p_Resource[, c(1,3,2)]
rownames(p_Resource) <- p_Resource$Row.names
colnames(p_Resource) <- c("Ensembl_ID", "Symbols", "Foldchange_Beta")

## Preview
kable(head(g_Resource, n = 10), align = "c", digits = 3)
```

	Ensembl_ID	Symbols	Foldchange_Beta
ENSG00000000419	ENSG00000000419	DPM1	0.011
ENSG00000000457	ENSG00000000457	SCYL3	0.079
ENSG00000000460	ENSG00000000460	C1orf112	0.033
ENSG00000000938	ENSG00000000938	FGR	0.149
ENSG00000000971	ENSG00000000971	CFH	-0.011
ENSG00000001036	ENSG00000001036	FUCA2	0.006
ENSG00000001084	ENSG00000001084	GCLC	0.045
ENSG00000001167	ENSG00000001167	NFYA	0.023
ENSG00000001460	ENSG00000001460	STPG1	0.051
ENSG00000001461	ENSG00000001461	NIPAL3	0.064

```
## Removing uneccesary data sets
remove(g_RB_foldchange, g_R_Symbols)
remove(p_RB_foldchange, p_R_Symbols)
```

We have now laid the major groundwork for the next steps of our analysis. We will use Mvalues next in the PCA and clustering, the results of which will enable us to sieve through our data for significance and relevance, which will lead us to our Finale data sets. Once there we will cap off our analysis with a logistical regression that confirms our results.

3. PCA and Clustering

Principal component analysis was done to reduce our high dimensional datasets after the QC (50000 genes and 55000 promoters) to a smaller number of genes and promoters with high variance. The variance of a gene or a promotor is linked to biological information. Principal components are created by laying coordinate axes through the dataset in such way that the variance is maximized. In this way we get a lower number of informative dimensions by choosing just informative PC`s. With this method we gain genes with favorable qualities and cohorts can bes separated.

3.1 Performing PCA

First g_Mvalues was transposed, so that all patients are rows and all genes are columns. Then the rownames were changed as preparation for the visualization of the PCA with aa ggbiplot, because the group command in ggbiplot groups by similiar rownames in the matrix. Additionally, columns that lacked standard deviation were removed as a requierment for the PCA. This acceptable to our analysis because sequences that show no variation across both cohorts do no interest us (they can't be DMRs). Lastly, the PCA was done.

```
## Transposing of g_Mvalues, renaming of rownames for grouping in ggbiplot, removing of columns
## containing constant values, running PCA and coercing g_pca to a data frame.
g_T_Mvalues <- t(g_Mvalues)
rownames(g_T_Mvalues) <- c(rep("AML", times = 10), rep("Mono", times = 10))
g_T_Mvalues_clean <- g_T_Mvalues[ , apply(g_T_Mvalues, 2, var) != 0]
g_pca <- prcomp(g_T_Mvalues_clean, center = TRUE, scale. = TRUE)
g_pca_df <- as.data.frame(g_pca$x)
```

In the following lines the same procedure was done for promoters with exception of removing the promoter columns without standard eviation. This was uneccesary, because there weren`t any columns lacking standard deviation..

```
p_T_Mvalues <- t(p_Mvalues)
rownames(p_T_Mvalues) <- c(rep("AML", times = 10), rep("Mono", times = 10))
p_pca <- prcomp(p_T_Mvalues, center = TRUE, scale. = TRUE)
p_pca_df <- as.data.frame(p_pca$x)
```

3.2 Analysing Explained Variance per PC

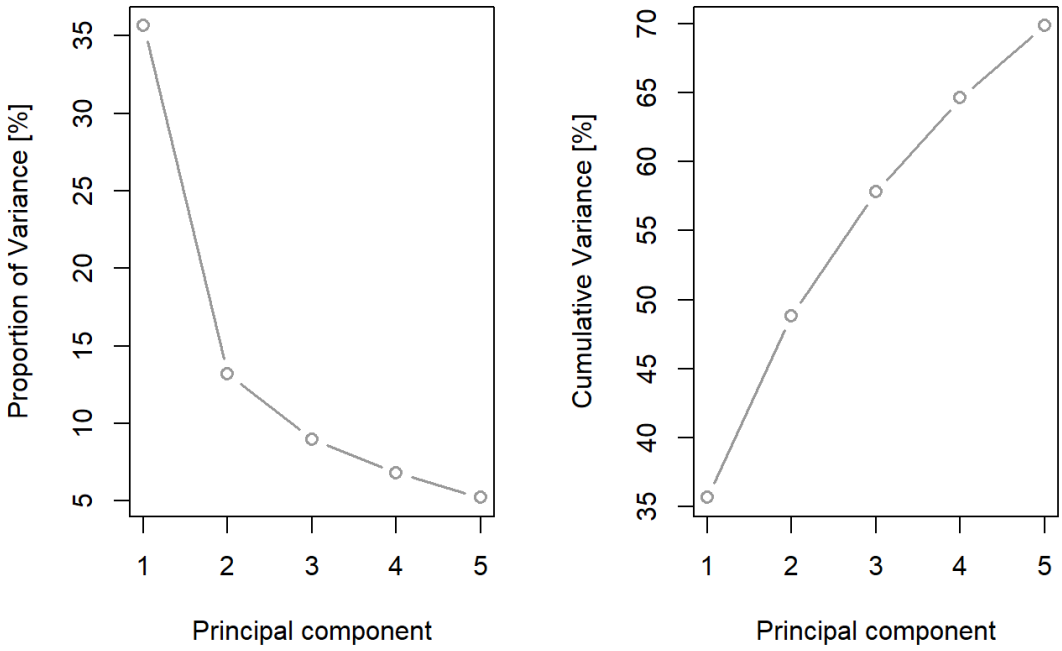
Principal components are ranked by the explained variance. The explained proportion of the variance that each PC explaines from the whole variance can be visualized by a scree plot. The first PC clearly explains most of the variance (around 35%). As the second PC is orthogonal to the first PC the explained variance is lower, because the distance of the second PC to each datapoint is different and this distance is used to calculate the variance. In the first scree plot the proportion of the variance explained can be seen whereas the second scree plot shows the cumultative proportion of variance explained. Both of them show the first five PC's out of twenty. In the further analysis a certain number of PC's has to be choosen which explain most of the variance.

```
## Calculating the proportion of variance explained by every PC
g_pca_var_prop <- ((g_pca$sdev^2)/sum(g_pca$sdev^2))*100

## Giving a preview of the proportion of variance for the first five PCS
g_pca_var_prop <- data.frame(g_pca_var_prop)
g_pca_var_prop <- data.frame(t(g_pca_var_prop))

kable(g_pca_var_prop[, 1:5], align = "c", digits = 1, row.names = FALSE, caption = c("Proportion of Variance in
 Percent"), col.names = c("PC1", "PC2", "PC3", "PC4", "PC5"))
```

Proportion of Variance in Percent

PC1	PC2	PC3	PC4	PC5
35.7	13.2	9	6.8	5.2

```
g_pca_var_prop <- data.frame(t(g_pca_var_prop))

## Plotting the proportion of Variance in two forms
par(mfrow=c(1,2))
   plot(g_pca_var_prop[1:5, 1],xlab="Principal component", ylab="Proportion of Variance [%]", type = "b", col =
"#999999", lwd = 1.5)
   plot(cumsum(g_pca_var_prop[1:5, 1 ]),xlab="Principal component", ylab="Cumulative Variance [%]", type = "b",
 col = "#999999", lwd = 1.5)
```



This output shows us that the first five PCs explain a hefty proportion of the variance. It also shows us that the PCs beyond 5 will explain very little of the Variance. Thus we are analysing the first five PCs in order to filter out the best PC to be used for dimensionality reduction.

3.3 Formatting for Batch Effect Analysis

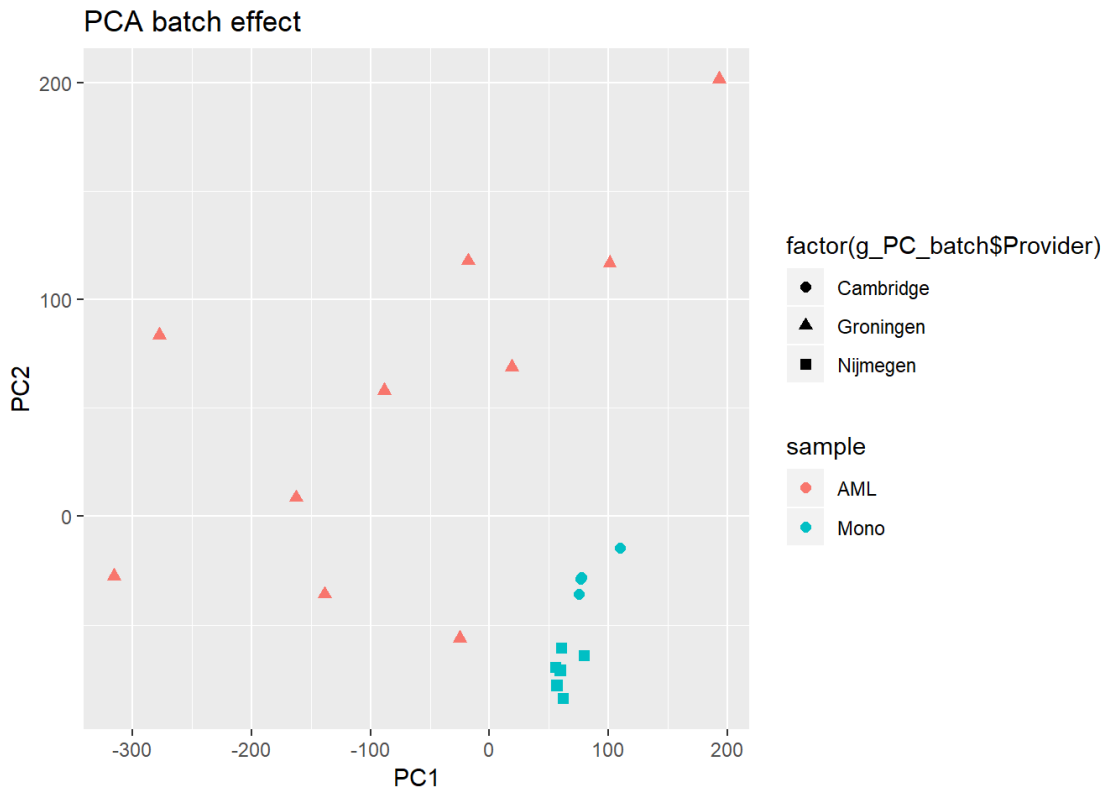
In the next part data frames are created for batch effect detection in genes and promoters. Therefore the dataframes are named g_PC_batch and p_PC_batch. The dataframes contain the first five principal components of genes and promoters, three columns with technical parameters and three columns with biological parameters. The vectors linked to the dataframe after the first five PC's are in the following order: biomaterial provider, first submission date (in days after the first submission date) sequence runs, age of the patients, cell type and sex. Subsequently the columns were named after the referring parameter.

```
 g_PC_batch <- cbind(g_pca_df[,1:5],
 c(rep("Groningen", times=10), rep("Cambridge", times=4), rep("Nijmegen", times=6)),
 c(1032,1005,1032,rep(1005, times=7),1184,1184,1032,1032,788,0,788,80,0,0),
 c(9,10,9,rep(11, times=5),15,11,38,38,38,20,14,4,14,15,17,18),
 c(32.5,57.5,62.5,67.5,72.5,67.5,67.5,42.5,47.5,67.5,67.5,rep(62.5, times=3),rep(47.5,     times=4),42.5,67.5),
 c(rep("myeloid", times=10), rep("monocyte", times=10)),
 c(rep("Female", times=5),"Male",rep("Female", times=4),"Male","Female","Male","Female",rep("Male", times=5),"Fe
male"))
 names(g_PC_batch)[6] <- "Provider"
 names(g_PC_batch)[7] <- "Date"
 names(g_PC_batch)[8] <- "Runs"
 names(g_PC_batch)[9] <- "Age"
 names(g_PC_batch)[10] <- "Cell_type"
 names(g_PC_batch)[11] <- "Sex"

 p_PC_batch <- cbind(p_pca_df[,1:5],
 c(rep("Groningen", times=10), rep("Cambridge", times=4), rep("Nijmegen", times=6)),
 c(1032,1005,1032,rep(1005, times=7),1184,1184,1032,1032,788,0,788,80,0,0),
 c(9,10,9,rep(11, times=5),15,11,38,38,38,20,14,4,14,15,17,18),
 c(32.5,57.5,62.5,67.5,72.5,67.5,67.5,42.5,47.5,67.5,67.5,rep(62.5, times=3),rep(47.5, times=4),42.5,67.5),
 c(rep("myeloid", times=10), rep("monocyte", times=10)),
 c(rep("Female", times=5),"Male",rep("Female", times=4),"Male","Female","Male","Female",rep("Male", times=5),"Fe
male"))
 names(p_PC_batch)[6] <- "Provider"
 names(p_PC_batch)[7] <- "Date"
 names(p_PC_batch)[8] <- "Runs"
 names(p_PC_batch)[9] <- "Age"
 names(p_PC_batch)[10] <- "Cell_type"
 names(p_PC_batch)[11] <- "Sex"
```

3.4 Visualizing Batch Effects

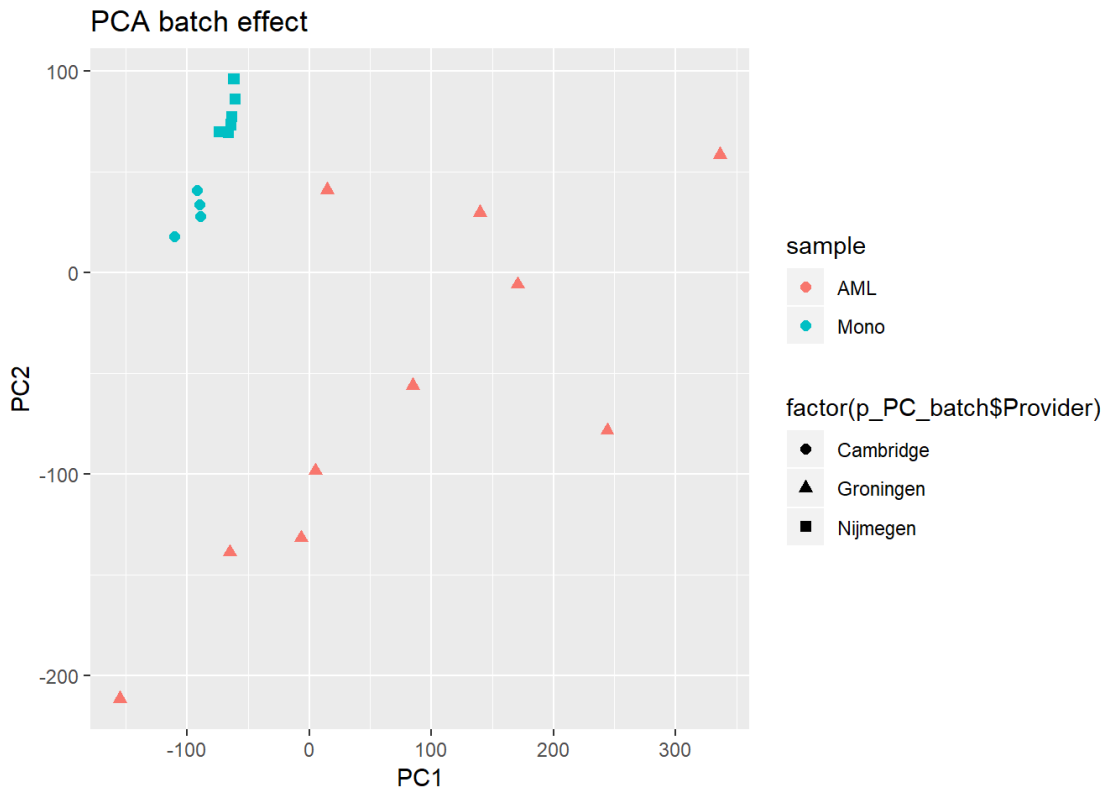
Now we can show the datapoints (patients) in the first two PC's (AML in red and Mono in light blue) to see the general distribution of datapoints generated by the PCA and to look for possible batch effects. As an example biomaterial provider as one of the technical parametersr is visualized. The three different biomaterial providers from Cambridge, Groningen and Nijmegen can be detected by the shape of the datapoints. The visualization is done by using ggplot2. It can be seen that the red points (AML) and light blue points (Mono) are separated. A batch effect is indicated by the separation of the provider groups in the plot. Batch effects in the different PC's will be further analysed with certain statistical tests later on.

```
 g_pca_small <- data.frame(sample=rownames(g_pca$x), X=g_pca$x[,1], Y=g_pca$x[,2] )
 ggplot(g_pca_small, aes(X, Y, group = sample, shape = factor(g_PC_batch$Provider))) +
 geom_point(size = 2, aes(color = sample))+
 ggtitle("PCA batch effect")+
 xlab(paste("PC1"))+
 ylab(paste("PC2"))
```



The same command is run again for promoters and shows the same results.

```
p_pca_small <- data.frame(sample=rownames(p_pca$x), X=p_pca$x[,1], Y=p_pca$x[,2] )
ggplot(p_pca_small, aes(X, Y, group = sample, shape = factor(p_PC_batch$Provider))) +
geom_point(size = 2, aes(color = sample))+
ggtitle("PCA batch effect")+
xlab(paste("PC1"))+
ylab(paste("PC2"))
```



3.5 Batch Effect Analysis

Having performed our PCA and generated the relevant data sets for this step of the analysis it's now time to decide which PCs of the PCA will be used for the continuing analysis. As previously discussed, some PCs are associated with favorable qualities, such as explaining large proportions of the variance or showing internal data structures, like separation of the healthy and cancerous cohorts. Here we determine another favorable qualitiy: not being influenced by batch effects. (In this definition we also include biological effects that are not of relevance to our analysis.)

We have a variety of non-relevant effects that can influence batches in our data. We have the biomaterial providers, sex, age, number of sequencing runs, date of sequencing and cell type. Of all of these, we are looking to find PCs that explain a high proportion of the variance but are not affected by the first five batches mentioned above. We are selecting for PCs that are influenced by cell type however, as these will likely hold sequences that are differentially expressed between our two cohorts (since cell type is monocytes and AML cells respectively).

3.5.1 Performing Batch Effect Tests

In order to more accurately capture the influence of our batches on our PCs we are performing multiple tests that will indicate the significance of the inluence on the batch effect. We are using three different tests depending on the form of th ebatch analysed.

3.5.1.1 Kruskal-Wallis Test

This test is well suited for categorical data with three or more categories. As providers, age and date of sequencing fit this description, this is the test we use.

```
## Defining Vectors for p-values
g_pvalues_prov <- c()
g_pvalues_date <- c()
g_pvalues_age <- c()

p_pvalues_prov <- c()
p_pvalues_date <- c()
p_pvalues_age <- c()

## These loops will run through the PC batch data set and perform a Kruskal Wallis test
## for every PC and all three batches
for (i in 1:5) {
  p <- (kruskal.test(g_PC_batch[,i]~g_PC_batch[,6]))$p.value
  d <- (kruskal.test(g_PC_batch[,i]~g_PC_batch[,7]))$p.value
  a <- (kruskal.test(g_PC_batch[,i]~g_PC_batch[,9]))$p.value
  g_pvalues_prov <- append(g_pvalues_prov, p)
  g_pvalues_date <- append(g_pvalues_date, d)
  g_pvalues_age <- append(g_pvalues_age, a)
}

for (i in 1:5) {
  p <- (kruskal.test(p_PC_batch[,i]~p_PC_batch[,6]))$p.value
  d <- (kruskal.test(p_PC_batch[,i]~p_PC_batch[,7]))$p.value
  a <- (kruskal.test(p_PC_batch[,i]~p_PC_batch[,9]))$p.value
  p_pvalues_prov <- append(p_pvalues_prov, p)
  p_pvalues_date <- append(p_pvalues_date, d)
  p_pvalues_age <- append(p_pvalues_age, a)
}

## Previewing our p-values
kable(g_pvalues_prov, align = "c", col.names = "P-Values for Providers", digits = 3)
```

P-Values for Providers
0.037
0.001
0.032
0.363
0.110

This output from providers shows us that there are few PCs that the providers are significant for, such as PC1 (0.03 < 0.05) or PC2 (0.001 < 0.05).

3.5.1.2 Monte Carlo Permutation Test

This test is well suited for numerical data. Since only sequencing runs fits this description, it'll be the only one we use this test on.

```
## Definign vectors
g_pvalues_runs <- c()
p_pvalues_runs <- c()

## Running each of the for-loops
for (i in 1:5) {
  r <- (permutationTest(g_PC_batch[,i], g_PC_batch[,8], nPermutation = 9999))$pVal.permut
  g_pvalues_runs <- append(g_pvalues_runs, r)
}

for (i in 1:5) {
  r <- (permutationTest(p_PC_batch[,i], p_PC_batch[,8], nPermutation = 9999))$pVal.permut
  p_pvalues_runs <- append(p_pvalues_runs, r)
}

## Previewing our p-values
kable(g_pvalues_runs, align = "c", col.names = "P-Values for Runs", digits = 5)
```

P-Values for Runs
0.5923
0.3949
0.2623
0.2491
0.1550

In contrast to the providers batch effect, the batch effect for runs is non-significant in all PCs.

3.5.1.3 Wilcoxon Test

This test is well suited for categorical data with only two categories. Since sex and cell type fit this description, we will be using this test on them.

```
## Defining Vectors
g_pvalues_CT <- c()
g_pvalues_sex <- c()

p_pvalues_CT <- c()
p_pvalues_sex <- c()


## Running our for loops
for (i in 1:5) {
  c <- (wilcox.test(g_PC_batch[,i] ~ g_PC_batch[,10]))$p.value
  s <- (wilcox.test(g_PC_batch[,i] ~ g_PC_batch[,11]))$p.value
  g_pvalues_CT <- append(g_pvalues_CT, c)
  g_pvalues_sex <- append(g_pvalues_sex, s)
}

for (i in 1:5) {
  c <- (wilcox.test(p_PC_batch[,i] ~ p_PC_batch[,10]))$p.value
  s <- (wilcox.test(p_PC_batch[,i] ~ p_PC_batch[,11]))$p.value
  p_pvalues_CT <- append(p_pvalues_CT, c)
  p_pvalues_sex <- append(p_pvalues_sex, s)
}

## Previewing our p-values
kable(g_pvalues_CT, align = "c", col.names = "P-Values for Cell Type", digits = 3)
```

P-Values for Cell Type

P-Values for Cell Type
0.019
0.001
0.023
0.436
0.912

```
## Removing uneccesary data sets
remove(p, d, a, r, c, s, i)
```

Here we can see that the cell type, i.e. our cohorts have a significant impact on the first three PCs.

3.5.2 Pooling Results

We will now combine all of our vectors of p-values into one matrix where we can quickly compare what PC is affected by which batch effect siggnificantly.

```
# Summarizing values in a data frame
  ## Combining and formatting data set
  g_PC_pvalues <- data.frame(g_pvalues_prov, g_pvalues_date, g_pvalues_age,
                             g_pvalues_runs, g_pvalues_CT, g_pvalues_sex)

  rownames(g_PC_pvalues) = c("PC1", "PC2", "PC3", "PC4", "PC5")
  colnames(g_PC_pvalues) <- c("Provider", "Date", "Age", "Runs", "Cell type", "Sex")

  p_PC_pvalues <- data.frame(p_pvalues_prov, p_pvalues_date, p_pvalues_age,
                             p_pvalues_runs, p_pvalues_CT, p_pvalues_sex)

  rownames(p_PC_pvalues) = c("PC1", "PC2", "PC3", "PC4", "PC5")
  colnames(p_PC_pvalues) <- c("Provider", "Date", "Age", "Runs", "Cell type", "Sex")

  g_PC_pvalues <- t(g_PC_pvalues)
  p_PC_pvalues <- t(p_PC_pvalues)

  ## Previewing our new data set
  kable(g_PC_pvalues, align = "c", caption = "P-Values of Batch Effects on PCs", digits = 3)
```

P-Values of Batch Effects on PCs

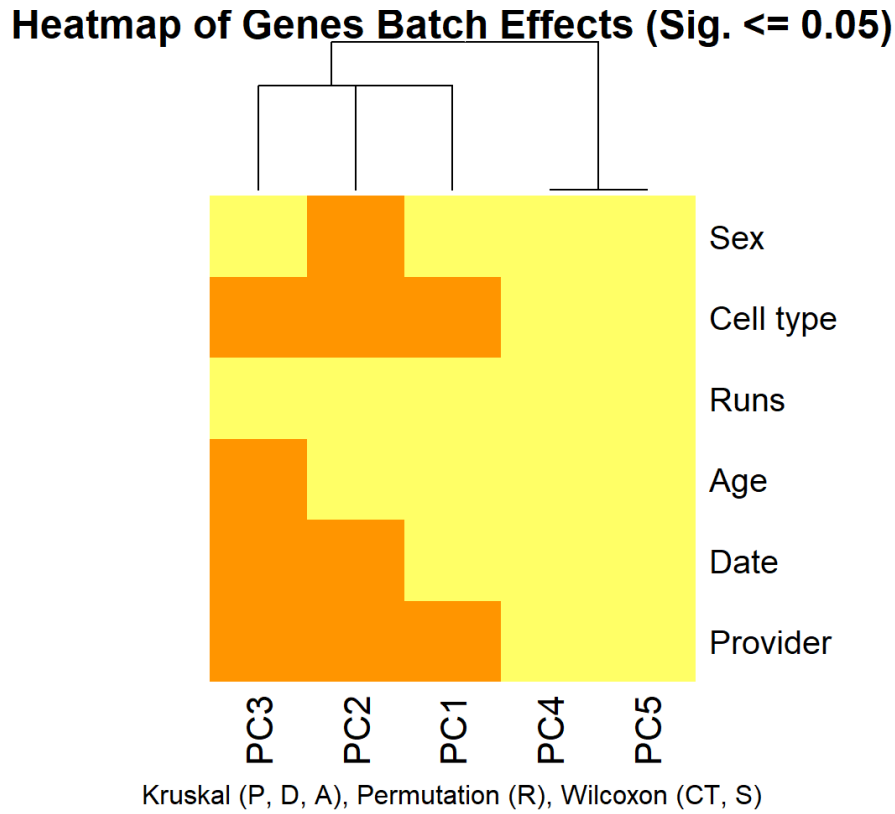
	PC1	PC2	PC3	PC4	PC5
Provider	0.037	0.001	0.032	0.363	0.110
Date	0.500	0.026	0.004	0.671	0.599
Age	0.522	0.186	0.047	0.234	0.129
Runs	0.592	0.395	0.262	0.249	0.155
Cell type	0.019	0.001	0.023	0.436	0.912
Sex	0.305	0.001	0.305	0.910	0.624

```
  ## Removing uneccesary data sets
  remove(g_pvalues_prov, g_pvalues_date, g_pvalues_age, g_pvalues_runs, g_pvalues_CT,
         g_pvalues_sex)
  remove(p_pvalues_prov, p_pvalues_date, p_pvalues_age, p_pvalues_runs, p_pvalues_CT,
         p_pvalues_sex)
```

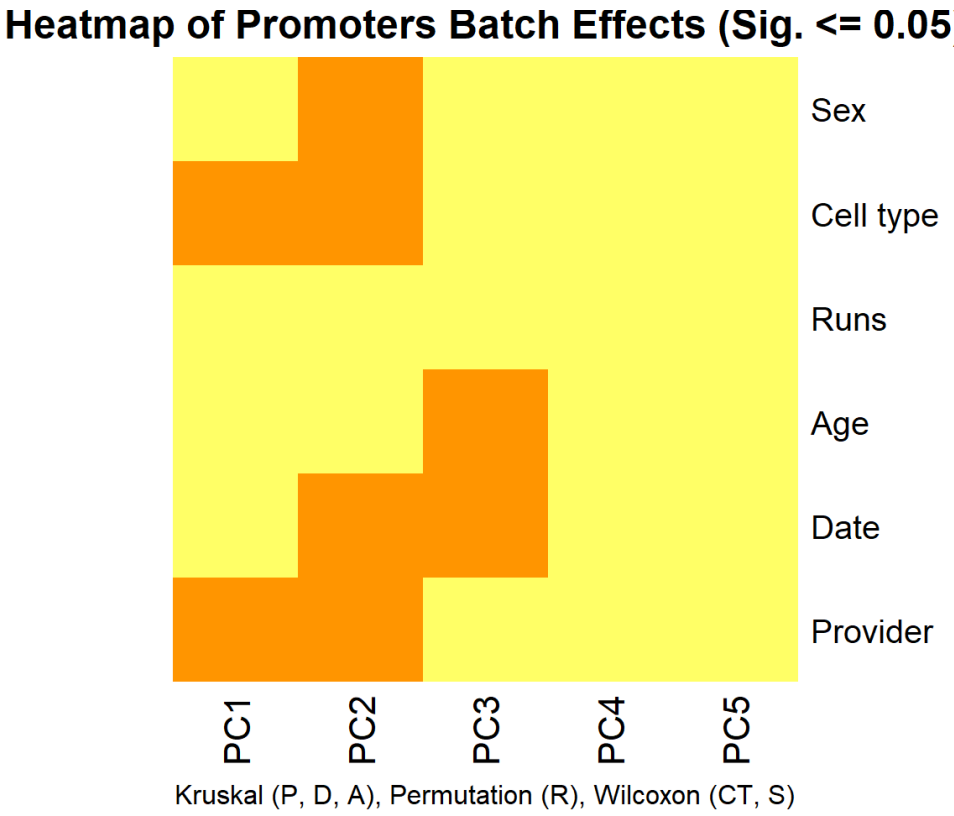
3.5.3 Visualizing our Results and choosing PCs to analyse with

We will now produce a few heatmaps to better understand our results. We will plot actual p-Values as heatmaps once to see overall structures in our data, and then two color heatmaps that simply indicate significance and non-significance.

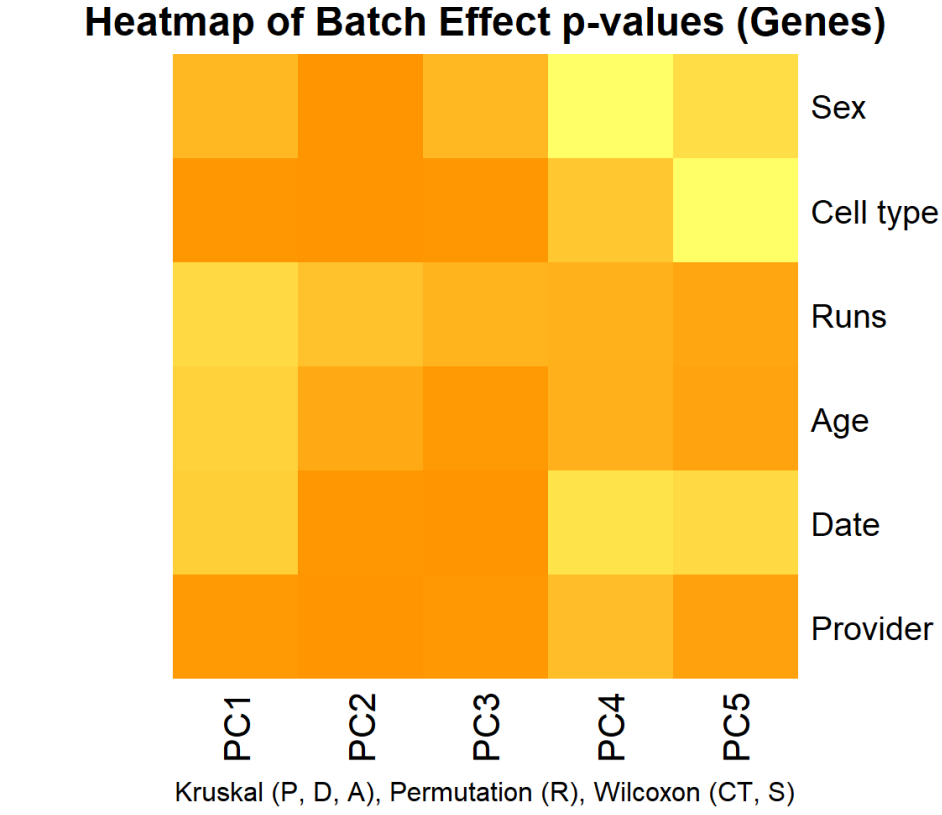
```
## Binary Heatmap
    ## Creating data set
    g_PC_pvalues_bin <- ifelse(g_PC_pvalues <= 0.05, 0, 1)
    p_PC_pvalues_bin <- ifelse(p_PC_pvalues <= 0.05, 0, 1)

    ## Plotting heatmap
    batch_colors <- colorRampPalette(c("#FF9500", "#FFFF66"))

    heatmap(g_PC_pvalues_bin
            , scale = "none"
            , col = batch_colors(2)
            , main = "Heatmap of Genes Batch Effects (Sig. <= 0.05)"
            , xlab = "Kruskal (P, D, A), Permutation (R), Wilcoxon (CT, S)"
            , Rowv = NA
            , Colv =
            , font.main = 1
    )
```



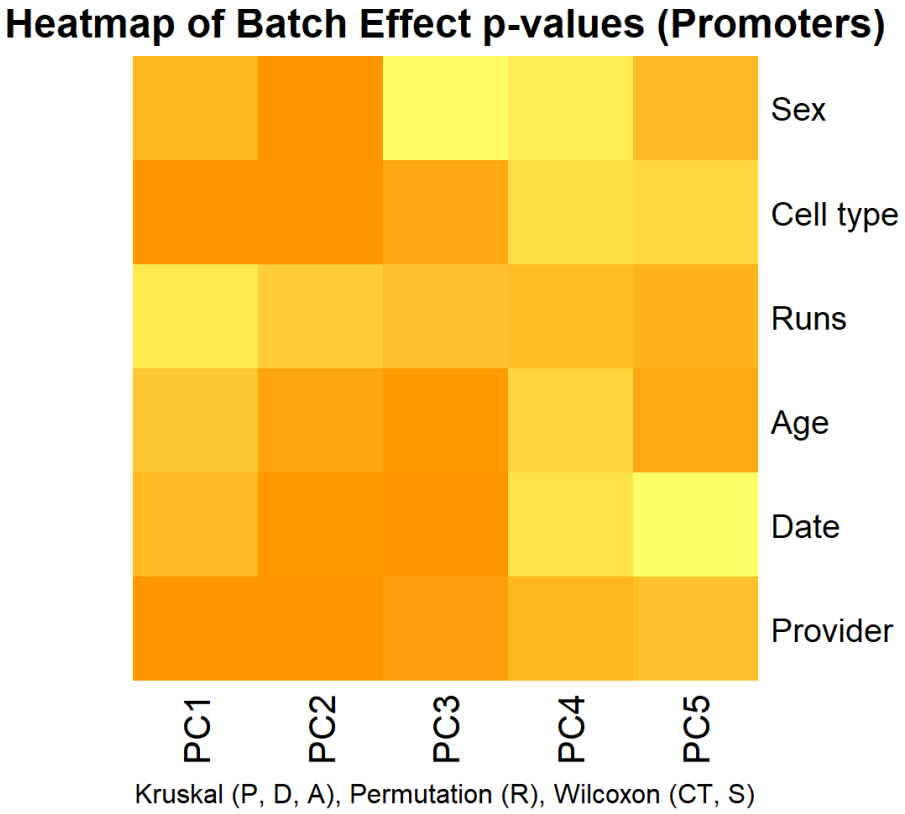

```
    heatmap(p_PC_pvalues_bin
            , scale = "none"
            , col = batch_colors(2)
            , main = "Heatmap of Promoters Batch Effects (Sig. <= 0.05)"
            , xlab = "Kruskal (P, D, A), Permutation (R), Wilcoxon (CT, S)"
            , Rowv = NA
            , Colv = NA
            , font.main = 1
    )
```




```
## Heatmap with continous colors
    heatmap(as.matrix(g_PC_pvalues)
            , scale = "none"
            , col = batch_colors(256)
            , main = "Heatmap of Batch Effect p-values (Genes)"
            , xlab = "Kruskal (P, D, A), Permutation (R), Wilcoxon (CT, S)"
            , Rowv = NA
            , Colv = NA
    )
```



```
heatmap(as.matrix(p_PC_pvalues)
        , scale = "none"
        , col = batch_colors(256)
        , main = "Heatmap of Batch Effect p-values (Promoters)"
        , xlab = "Kruskal (P, D, A), Permutation (R), Wilcoxon (CT, S)"
        , Rowv = NA
        , Colv = NA
)
```




```
## Removing uneccesary data sets
remove(g_PC_pvalues_bin ,batch_colors)
remove(p_PC_pvalues_bin)
```

Looking at the continuous p-values we can see many similar structures between promoters and data sets. Since the data in both cases is made up of identical batches, this make sense.

In contrast the binary heatmaps do show some differences, but show in both cases that PC1 has the lowest number of batch effects while still being influenced by cell type. The only other batch effect in PC1, in both cases, is the provider. While not gone into in further depth, there is only one provider for the AML cohort. It seems a likelihood that the difference between thee AML and monocyte cohort are causing this perceived batch effect. However, this has not been confirmed.

Regardless, PC1 shows the least amount of batch effects in both cases, and sine it also explains the most variance, it is the PC we will be using in our continuing analysis. It has the most favorable qualities.

3.6 Clustering Analysis

The next step includes the code for clustering analysis. This code analyses how well two clusters can be seen in the data depending on how much data from PC1 ordered by the absolute influence of it's loadings is used. We perform this analysis because we hope to reduce our number of genes to those most strongly linked to PC1, because it has the favorable qualities we are looking for. We choosed kmeans clustering for this purpose and the aim of this method would be the separation of the AML cohort and Monocyte cohort.

3.6.1 Formatting for Clustering

As preparation we made the absolute values of the loadings (contribution of each variable to the principal components) of PC1 and created a data frame of them. Then we bound the PC1 ranked data frame with the M values.

```
  g_PC1_df <- data.frame(abs(g_pca$rotation[,1]))
  colnames(g_PC1_df) <- c("g_PC1_df")

  p_PC1_df <- data.frame(abs(p_pca$rotation[,1]))
  colnames(p_PC1_df) <- c("p_PC1_df")

  ## Binding PC1 dataframe with the M values for clustering for sequences
  g_PC1_M <- merge(g_PC1_df, g_Mvalues, by="row.names")
  remove(g_PC1_df)

  p_PC1_M <- merge(p_PC1_df, p_Mvalues, by="row.names")
  remove(p_PC1_df)
```

After that we ordered the column PC1_df with the absolute loading values from the PCA in PC1_M for genes and promoters in a descending order. We also named the rownames in the PC1_M after the names which stand in an extra column created in the merging process. Additionally, we cut off the first two columns (rownames and loadings) as they are not relevant for the further clustering analyses.

```
## ordering and formatting M value data sets appropriately
  g_PC1_M <- g_PC1_M[order(g_PC1_M$g_PC1_df, decreasing = TRUE),]
  rownames(g_PC1_M)=g_PC1_M$Row.names
  g_PC1_M <- g_PC1_M[, 3:22]

  p_PC1_M <- p_PC1_M[order(p_PC1_M$p_PC1_df, decreasing = TRUE),]
  rownames(p_PC1_M)=p_PC1_M$Row.names
  p_PC1_M <- p_PC1_M[, 3:22]
```

3.6.2 Performing Clustering

The next chunk shows the code cluster functions. The name of the function referrs to the fact that the function is applied to genes and attempts to identify two clusters. The variable x is a choosen percentage of dataset to be analysed in the function. The concrete percentage step we choose will be explained later on. The if loop begins in the first two rows with a little code for a progress bar. So when the function is applied one will see how much the function has already calculated. The next part of the loop redefines x as a fraction rather than a percentage. Then a reduced data frame is created which alters with the increasing x. It contains x percent of nrows from the g_PC1_M data frame and all columns. This certain percentage of the dataset will be checked in the next steps for its clustering ability. For kmeans clustering this reduced dataset has to be transposed (g_cluster) and we named the rows after the cohorts. After that kmeans was performed with x as the seed value to gain reproducible output every time we run kmeans. We took two centers because we want to check the separation of our two cohorts AML and Mono. As nstart we took one hundred so the clustering happens one hundred times with random values. The higher the n the higher the accuracy is. The c in the output data frame g_c is referring to clustering.

```
## defining respective cluster functions
  ## Once for genes
  fg_cluster <- function(x) {

    if(x %% jcount == 0) {
      cat("|")
    }

    x <- x/100

    g_PC1_M_reduced <- g_PC1_M[1:floor(nrow(g_PC1_M)*x), ]
    g_cluster <- t(g_PC1_M_reduced)
    rownames(g_cluster) <- c(rep("AML", times = 10), rep("Mono", times = 10))

    set.seed(x)
    g_c <- kmeans(g_cluster, centers = 2, nstart = 100)
    g_c <- c(g_c$cluster, g_c$tot.withinss)

    return(g_c)
  }

  ## Once for promoters
  fp_cluster <- function(x) {

      if(x %% jcount == 0) {
        cat("|")
      }

      x <- x/100

      p_PC1_M_reduced <- p_PC1_M[1:floor(nrow(p_PC1_M)*x), ]
      p_cluster <- t(p_PC1_M_reduced)
      rownames(p_cluster) <- c(rep("AML", times = 10), rep("Mono", times = 10))

      set.seed(x)
      p_c <- kmeans(p_cluster, centers = 2, nstart = 100)
      p_c <- c(p_c$cluster, p_c$tot.withinss)

      return(p_c)
  }
```

The explained cluster functions for promoters and genes are then applied on the data frame j. This data frame contains all numbers from 1 to 100 in the steps of 1. Our input x in the cluster function is therefore defined in j and as we apply the cluster function it will calculate the clusters from one to hundred with increasing x in every loop. jcount referrs to a specific percentage already calculated from all the rows in j. Lastly, the output data frame is converted to have our AML patients and Mono patients in the columns.

```
  j <- data.frame(seq(1, 100, 1))
  jcount <- floor(nrow(j)/100)
  g_c <- apply(j, 1, fg_cluster)
```

```
## ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
  g_c <- t(g_c)

  j <- data.frame(seq(1, 100, 1))
  jcount <- floor(nrow(j)/100)
  p_c <- apply(j, 1, fp_cluster)
```

```
## ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
  p_c <- t(p_c)
```

Having now run through our loadings a percent at a time and tracking cluster assignments, it is time to review the accuracy of these cluster assignments.

3.6.3 Analysing the Accuracy of Clustering

The following part deals with a function which determines the accuracy of the cluster assignment. The next chunk will be expalained with direct comments in the code.

```
  fgc_accuracy <- function(x) {

## In each row of g_c there is a cluster assignment of a certain patient to cluster 1 or 2. Workingrow is define
d as the row with the cluster assignments referring to a certain percent x.
    workingrow <- g_c[x, ]
##Now we determine whether more AML patients are in cluster 1 or Mono patients are in cluster 1
    cluster1_assA <- length(which(workingrow[1:10] == 1))
    cluster1_assM <- length(which(workingrow[11:20] == 1))
#After that each element of the workingrow will be defined as A for AML or M Mono. So if more AML patients are i
n cluster 1 then cluster 1 will be the AML cluster and cluster 2 will be the Mono cluster. Then each element whi
ch is 1 in the workingrow is named A for AML and each element which is 2 is named M for Mono. If less AML patien
ts are in cluster 1 the inverse happens.
    if(cluster1_assA >= cluster1_assM) {
      cluster1_ass <- c("A")
      workingrow[workingrow == 1] <- cluster1_ass
      cluster2_ass <- c("M")
      workingrow[workingrow == 2] <- cluster2_ass

    } else {
      cluster1_ass <- c("M")
      workingrow[workingrow == 1] <- cluster1_ass
      cluster2_ass <- c("A")
      workingrow[workingrow == 2] <- cluster2_ass
    }
#If an element in the AML cluster is rightfully assigned as A it will be set True. If on the other hand an eleme
nt is not belonging to the AML cluster, which would be the case if an element is named M, it is set as False.
    workingrow_AML <- workingrow[1:10]
    workingrow_AML[workingrow_AML == c("A")] <- TRUE
    workingrow_AML[workingrow_AML == c("M")] <- FALSE
#If an element in the Mono cluster is rightfully assigned as M it will be set True. If on the other hand an elem
ent is not belonging to the Mono cluster, which would be the case if an element is named A, it is set as False.
    workingrow_mon <- workingrow[11:20]
    workingrow_mon[workingrow_mon == c("M")] <- TRUE
    workingrow_mon[workingrow_mon == c("A")] <- FALSE
#The gained rows are then converted to logical values (0 for False and 1 for True) and the mean is taken and ref
lects the accuracy of the clustering.
    workingrow <- as.logical(c(workingrow_AML, workingrow_mon))

    return(mean(workingrow))
  }
```

The same analyses of the cluster assignment is repeated for promoters:

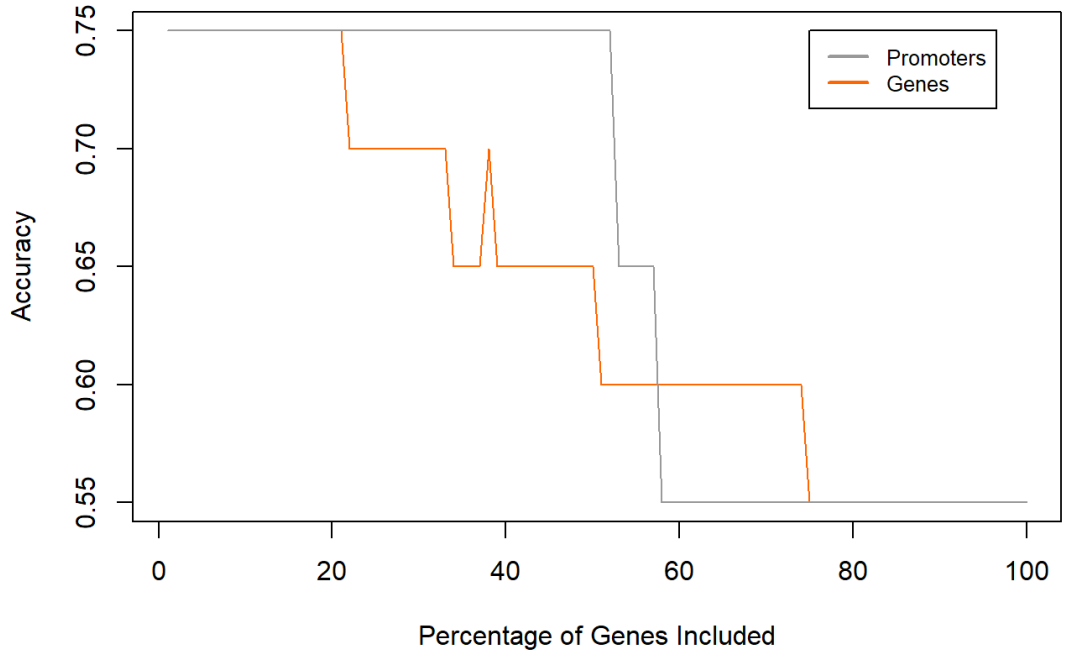
```
fpc_accuracy <- function(x) {

    workingrow <- p_c[x, ]
    cluster1_assA <- length(which(workingrow[1:10] == 1))
    cluster1_assM <- length(which(workingrow[11:20] == 1))

    if(cluster1_assA >= cluster1_assM) {
      cluster1_ass <- c("A")
      workingrow[workingrow == 1] <- cluster1_ass
      cluster2_ass <- c("M")
      workingrow[workingrow == 2] <- cluster2_ass
    } else {
      cluster1_ass <- c("M")
      workingrow[workingrow == 1] <- cluster1_ass
      cluster2_ass <- c("A")
      workingrow[workingrow == 2] <- cluster2_ass
    }

    workingrow_AML <- workingrow[1:10]
    workingrow_AML[workingrow_AML == c("A")] <- TRUE
    workingrow_AML[workingrow_AML == c("M")] <- FALSE

    workingrow_mon <- workingrow[11:20]
    workingrow_mon[workingrow_mon == c("M")] <- TRUE
    workingrow_mon[workingrow_mon == c("A")] <- FALSE

    workingrow <- as.logical(c(workingrow_AML, workingrow_mon))

    return(mean(workingrow))
  }
```

The accuracy functions are now applied on j and the output is named gc_acc, which contains clustering accuracy values between 0 and 1 for each percentage of the loadings of the PC1 genes.

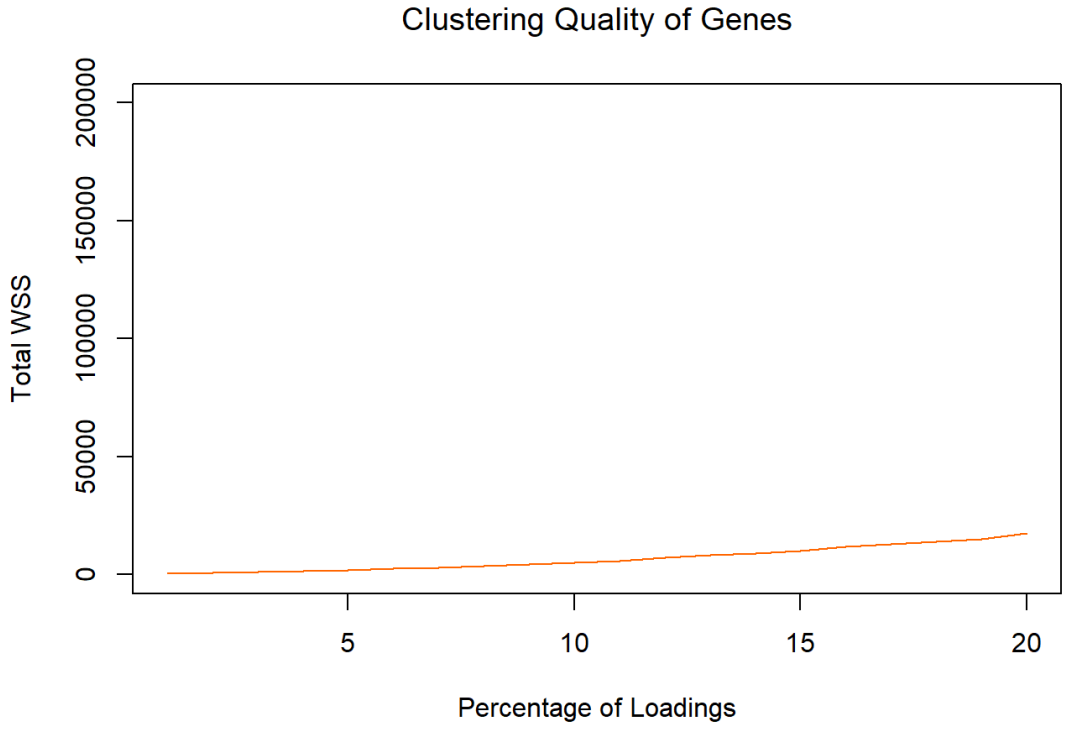
```
  j <- data.frame(seq(1, nrow(g_c), 1))
  gc_acc <- apply(j, 1, fgc_accuracy)

  j <- data.frame(seq(1, nrow(p_c), 1))
  pc_acc <- apply(j, 1, fpc_accuracy)
```

The concluding steps are removing of unecessary data sets and plotting the results.

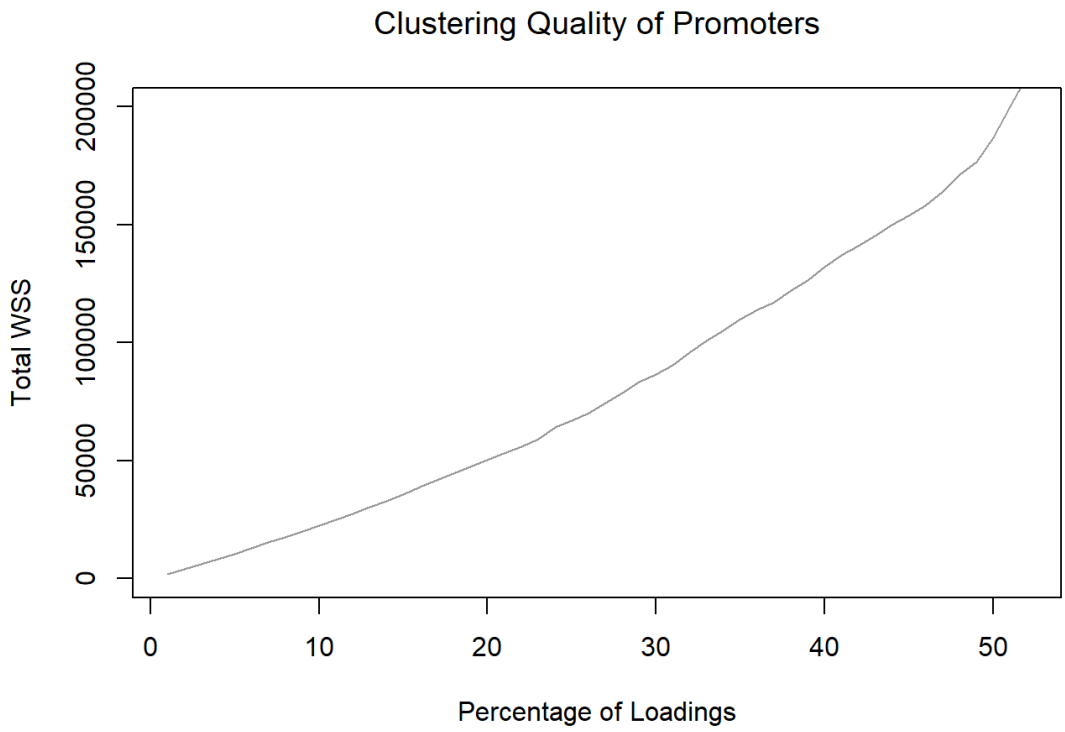
```
    remove(fg_cluster, fgc_accuracy, g_PC1_M, g_PC_batch, g_PC_pvalues, g_pca_small, g_pca_var_prop, g_pca_df, g
_T_Mvalues, g_T_Mvalues_clean, j, jcount)
    remove(fp_cluster, fpc_accuracy, p_PC1_M, p_PC_batch, p_PC_pvalues, p_pca_small, p_pca_df, p_T_Mvalues)

    ## Setting up our graphical parameters
    par(mfrow=c(1,2))

    #Plotting clustering accuracy
    par(mfrow=c(1,1))
    plot(gc_acc, col = "#FF6600", type = "l", ylab = "Accuracy", xlab = "Percentage of Genes Included")
    lines(pc_acc, col = "#999999")
    legend(75, 0.75, legend=c("Promoters", "Genes"), col=c("#999999", "#FF6600"), lty = 1:1, lwd = 2, cex=0.8)
```




```
    #Plotting WSS
    plot(g_c[1:20, 21], col = "#FF6600", type = "l", xlab = "Percentage of Loadings", ylab = "Total WSS", main =
"Clustering Quality of Genes", ylim =   c(0,200000), font.main = 1)
```




```
    plot(p_c[1:52, 21], col = "#999999", type = "l", xlab = "Percentage of Loadings", ylab = "Total WSS", main =
"Clustering Quality of Promoters", ylim =   c(0,200000), font.main = 1)
```




```
## Removing uneccesary Data sets
remove(gc_acc, g_c)
remove(pc_acc, p_c)
```

The plots for the analyses of clustering quality by WSS show that clustering is not suitable for the separation of our data as the graphs are constantly growing. The clustering accuracy is not high enough for any percentage of genes or promoters. Since we don't want to be stringent with a method that performed poorly, our compromise is to keep 20% of the genes and 52% of promoters. This way we do not lose too much data, but also have data which can be clustered with the best possible (and still not great) accuracy.

```
## Getting the names of the sequences in PC1 that we'll be using for their favorable traits
g_pca_abs <- abs(g_pca$rotation[,1:5])
p_pca_abs <- abs(p_pca$rotation[,1:5])

## Formatting and extracting names
g_PC1_ranked <- sort(g_pca_abs[,1], decreasing = TRUE)
g_PC1_ranked <- data.frame(g_PC1_ranked)
g_PC1_names_ranked <- rownames(g_PC1_ranked)
g_PC1_names <- g_PC1_names_ranked[1:floor(length(g_PC1_names_ranked)*0.20)]

p_PC1_ranked <- sort(p_pca_abs[,1], decreasing = TRUE)
p_PC1_ranked <- data.frame(p_PC1_ranked)
p_PC1_names_ranked <- rownames(p_PC1_ranked)
p_PC1_names <- p_PC1_names_ranked[1:floor(length(p_PC1_names_ranked)*0.52)]

## Removing uneccesary data sets
remove(g_PC1_names_ranked, g_PC1_ranked, g_pca, g_pca_abs)
remove(p_PC1_names_ranked, p_PC1_ranked, p_pca, p_pca_abs)
```

4. Determining Significance and Relevance in our Data

Having sorted through our genes for favorable traits, it is now time to turn to the search for DMRs. In order to perform this search we will require the 'T' data sets, the 'Resource' data sets and the list of sequences with favorable traits acquired in the PCA and clustering analysis.

4.1 Loading Relevant Data

First we load relevant packages for further analysis and give a reminder as to what the T and Resource data sets contain. The qvalue package can be installed from https://github.com/StoreyLab/qvalue (https://github.com/StoreyLab/qvalue).

```
kable(head(g_T, n = 4), align = "c")
```

	Mean_AML	SD_AML	N_AML	Mean_Mono	SD_Mono	N_Mono
ENSG00000223972	1.5494422	1.1619752	9	1.4623191	0.5685356	8
ENSG00000227232	0.5582653	0.3566792	10	0.6006172	0.3343029	10
ENSG00000278267	7.5568575	9.0371711	10	7.3089805	9.1358400	10
ENSG00000238009	2.3724431	0.6537955	10	2.2598652	0.3088221	10

```
kable(head(g_Resource, n = 4), row.names = FALSE, align = "c")
```

Ensembl_ID	Symbols	Foldchange_Beta
ENSG00000000419	DPM1	0.0111346
ENSG00000000457	SCYL3	0.0793159
ENSG00000000460	C1orf112	0.0332462
ENSG00000000938	FGR	0.1492468

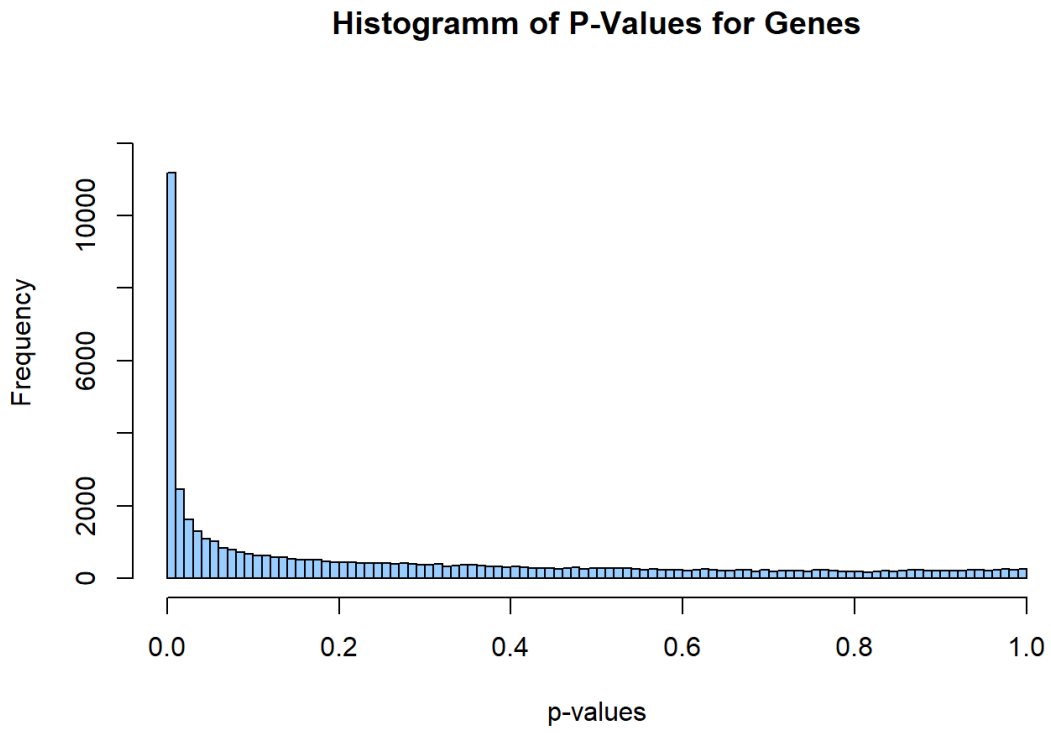
As you can see, the Resource data set contains relevant infomration regarding the name of a sequence and the fold change from the Mono to the AML cohort. The T data set represents all the inputs required for a t-test.

4.2 Performing T-Tests

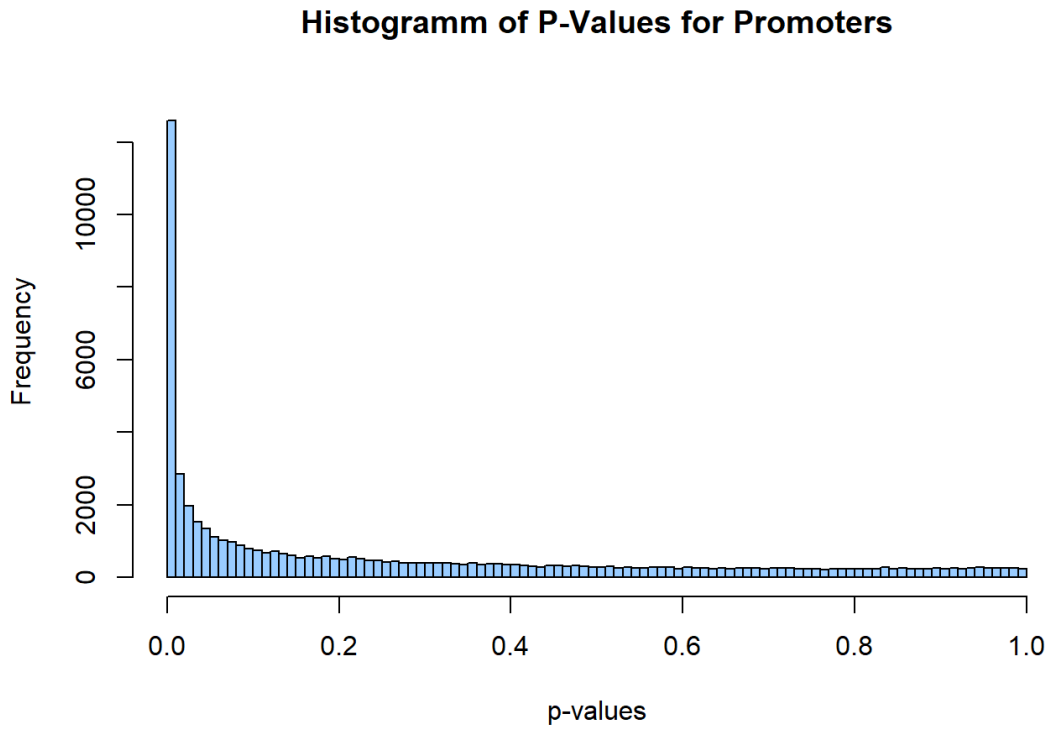
In the following we perform two-sided Student t-tests to determine the significance of differential methylation between our cohorts. To do so we define a function that manually performs the t-test and another that applies it to our T datasets. The resulting p-values are also plotted in a histogramm to innspect if they are well behaved.

```r
## (1) Defining and applying Functions
    ## defining a general function that applies a student t-test based on given input statistics

    f_ttest_alt <- function(m1, m2, s1, s2, n1, n2) {
      if(m1 == m2) {
        p_value = 1

      } else {

        df = (n1 + n2 - 2)
        s12 = sqrt((((n1 - 1)*s1^2)+((n2 - 1)*s2^2))/df)

        t = ((m1 - m2)/(s12*sqrt((1/n1) + (1/n2))))

        p_value = 2*pt(-abs(t),df)

      }
    }

    ## defining two functions that will comb through T data sets and provide inputs for f_ttest_alt

    fg_ttest_apl <- function(x) {
      m1 <- g_T[x, 1]
      m2 <- g_T[x, 4]

      s1 <- g_T[x, 2]
      s2 <- g_T[x, 5]

      n1 <- g_T[x, 3]
      n2 <- g_T[x, 6]

      p_value <- f_ttest_alt(m1, m2, s1, s2, n1, n2)
      return(p_value)
    }

    fp_ttest_apl <- function(x) {
      m1 <- p_T[x, 1]
      m2 <- p_T[x, 4]

      s1 <- p_T[x, 2]
      s2 <- p_T[x, 5]

      n1 <- p_T[x, 3]
      n2 <- p_T[x, 6]

      p_value <- f_ttest_alt(m1, m2, s1, s2, n1, n2)
      return(p_value)
    }

    ## applying our application functions to g_t and p_t
    j <- data.frame(seq(1, nrow(g_T), 1))
    g_p_values <- apply(j, 1, fg_ttest_apl)

    j <- data.frame(seq(1, nrow(p_T), 1))
    p_p_values <- apply(j, 1, fp_ttest_apl)


## (2) Formatting our outputs
    ## We bind our new p_values to the T data sets
    g_T <- cbind(g_T, g_p_values)
    colnames(g_T) <- c(colnames(g_T)[1:(ncol(g_T)-1)], "p_values")

    p_T <- cbind(p_T, p_p_values)
    colnames(p_T) <- c(colnames(p_T)[1:(ncol(p_T)-1)], "p_values")

    ## removal of uneccesary data
    remove(g_p_values, fg_ttest_apl, f_ttest_alt, j)
    remove(p_p_values, fp_ttest_apl)
```

```
## (3) checking to make sure the p value distribution is well behaved
hist(g_T$p_values, 100, main = "Histogramm of P-Values for Genes", xlab="p-values",ylim = c(0, 13000), col = "#9
9CCFF")
```




```
hist(p_T$p_values, 100, main = "Histogramm of P-Values for Promoters", xlab = "p-values", ylim = c(0, 13000), co
l = "#99CCFF")
```



It is evident that both of these p-value distributions are well behaved, which is an important requirement for the next step in the analysis.

4.3 Performing a Q-Value Correction

Since we just performed upwards of 80.000 t-tests it is important to correct for multiple testing. The q-value method is a similar approach to the Benjamini & Hochberg method of adressing this problem.

This method is to not correct the false psoitive rate to the point that any false positive is unlikely, but instead refer to a more useful and descriptive metric, the false discovery rate (FDR). This gives the proportion of false positives amongst discoveries. Instead of avoiding false positives it guarantees them at a controllable and known rate.This approach has demonstrably more porwer than simply reducing the false positive rate.

What differentiates q-values from the classic FDR is that they retain the same power regardless of the number of tests, have a higher power than the Benjamini Hochberg method and while having the same end point originate in two different mathematical approaches. An important note for the q-value method is that it assumes the p-values used are well behaved, and uses the right hand tail of the distribution to estimate what the FDR is at every p-value. Thus it is important not only that our p-values are properly distributed, but also that we apply the correction now, when the full distribution is present. To use this method at a later point in the analysis could lead to misleading conclusions.

The q-value method was drawn from Storey (Storey, 2002).

```
## First we order our data set so that it's order matches that of the q-value output
g_T <- g_T[order(g_T$p_values, decreasing = TRUE), ]
p_T <- p_T[order(p_T$p_values, decreasing = TRUE), ]

## Second, we apply the package loaded earlier to generate two objects that contain the
## q-values we need
g_q <- qvalue(p = g_T$p_values)
p_q <- qvalue(p = p_T$p_values)

## Finally, we bind the q-values to their respective sequences for later reference. We do
## not remove insignificant sequences yet. in this process we generate new 'Tq' data sets
g_T_q <- cbind(g_T, g_q$qvalue)
colnames(g_T_q)[ncol(g_T_q)] <- c("q_values")

p_T_q <- cbind(p_T, p_q$qvalue)
colnames(p_T_q)[ncol(p_T_q)] <- c("q_values")

## A preview of the new Tq data sets
kable(head(g_T_q, n = 4), row.names = FALSE, align = "c")
```

Mean_AML	SD_AML	N_AML	Mean_Mono	SD_Mono	N_Mono	p_values	q_values
-33.21928	0	10	-33.21928	0	10	1	0.46296
-33.21928	0	8	-33.21928	0	7	1	0.46296
-33.21928	0	8	-33.21928	0	9	1	0.46296
-33.21928	0	9	-33.21928	0	10	1	0.46296

```
## Removing uneccesary data sets
remove(g_q, p_q)
```

We now have q-values associated to every p-value for every sequence. The q-value associated with a p-value indicates the expected FDR if all p-values less than or equal to the one in question were seen as significant.For example, if we set our q-value threshold at 0.50 all rows with a p-value of 1 or lower would be significant (i.e. all our values would be significant and approximately half woulf be false positives).

4.4 Extracting Desirable Sequences with Favorable Traits

Having identified our desirable sequences in the PCA, we will now be generating two ‘T_reduced’ data sets that consist only of said sequences extracted from the Tq data sets.

```
## Extracting sequences deemed favorable after PCA analysis from the Reduced data sets

  g_T_reduced <- g_T_q[which(rownames(g_T_q) %in% g_PC1_names), ]
  p_T_reduced <- p_T_q[which(rownames(p_T_q) %in% p_PC1_names), ]

  print(paste0("Reduced genes data set from ", nrow(g_T_q), " to ", nrow(g_T_reduced), " sequences"))
```

```
## [1] "Reduced genes data set from 48966 to 9792 sequences"
```

```
  print(paste0("Reduced promoters data set from ", nrow(p_T_q), " to ", nrow(p_T_reduced), " sequences"))
```

```
## [1] "Reduced promoters data set from 55270 to 28740 sequences"
```

```
## Removing uneccesary data sets
  remove(g_T_q, g_PC1_names)
  remove(p_T_q, p_PC1_names)
```

4.5 Merging Data Sets

With the previous step complete we will now merge the Resource data sets with the Reduced data sets. This will allow us to select for fold change in the next step, as well as more easily identify research targets later, because the symbol of each gene will be associated with it in the data set. The resulting data set wil be the beginnings of the Finale data sets. At this point we also remove the t-test inputs from the combined data set, as they are not relevant in the presence of p-values.

```
## Merging data sets
  g_Finale <- merge(g_T_reduced, g_Resource, by = 0, all = FALSE)
  rownames(g_Finale) <- g_Finale$Row.names
  g_Finale <- g_Finale[, c(8, 9, 12, 11, 10)]

  p_Finale <- merge(p_T_reduced, p_Resource, by = 0, all = FALSE)
  rownames(p_Finale) <- p_Finale$Row.names
  p_Finale <- p_Finale[, c(8, 9, 12, 11, 10)]

## Removing uneccesary data sets from the environment
  remove(g_T_reduced, p_T_reduced)

## A preview of the new Finale data sets
kable(head(g_Finale, n = 4), row.names = FALSE, align = "c")
```

p_values	q_values	Foldchange_Beta	Symbols	Ensembl_ID
0.0077258	0.0168693	0.0332462	C1orf112	ENSG00000000460
0.0035176	0.0096574	0.1492468	FGR	ENSG00000000938
0.0009690	0.0039246	0.0445028	GCLC	ENSG00000001084
0.0510043	0.0650608	0.0227022	NFYA	ENSG00000001167

As you can see, these data sets willl be ideal for later research. They give mostly only relevant values, such as markers of significance, biological relevance, and references to research these sequences by.

4.6 Selection for Biologically Relevant Sequences

We will now remove all sequences with a q-value above 5% and all those that do not show a minimum of a 2/3 or 3/2 ratio in their fold change. These are the sequences we define as Biologically significant. This means that all of the sequences remaining will be significant with a 5% chance of being false positives, as well as being substantially different in their average expression.

```
## Determining how many genes were originally in the Finale data sets
  nrow_gF <- nrow(g_Finale)
  nrow_pF <- nrow(p_Finale)

## Selection for high fold changes and significance
  g_Finale <- g_Finale[g_Finale$q_values <= 0.05, ]
  g_Finale <- g_Finale[g_Finale$Foldchange_Beta < log2(2/3) |
                         g_Finale$Foldchange_Beta > log2(3/2), ]

  p_Finale <- p_Finale[p_Finale$q_values <= 0.05, ]
  p_Finale <- p_Finale[p_Finale$Foldchange_Beta < log2(2/3) |
                         p_Finale$Foldchange_Beta > log2(3/2), ]

## Sorting the data set for p-values for demonstrative purposes
  g_Finale <- g_Finale[order(g_Finale$p_values, decreasing = TRUE), ]
  p_Finale <- p_Finale[order(p_Finale$p_values, decreasing = TRUE), ]

## Displaying the number of remainin Sequences
  print(paste0("Reduced genes data set from ",
               nrow_gF,
               " to ",
               nrow(g_Finale),
               " sequences, Of which ",
               ceiling(nrow(g_Finale)*0.05),
               " are expected to be false discoveries."))
```

```
## [1] "Reduced genes data set from 9792 to 142 sequences, Of which 8 are expected to be false discoveries."
```

```
  print(paste0("Reduced promoters data set from ",
               nrow_pF,
               " to ",
               nrow(p_Finale),
               " sequences, Of which ",
               ceiling(nrow(p_Finale)*0.05),
               " are expected to be false discoveries."))
```

```
## [1] "Reduced promoters data set from 28740 to 2280 sequences, Of which 114 are expected to be false discoveri
es."
```

```
## Preview of the filtered data sets
  kable(head(g_Finale, n = 1), row.names = FALSE, align = "c")
```

p_values	q_values	Foldchange_Beta	Symbols	Ensembl_ID
0.0272282	0.0415989	0.8476463	BCL2L10	ENSG00000137875

```
  kable(head(p_Finale, n = 1), row.names = FALSE, align = "c")
```

p_values	q_values	Foldchange_Beta	Symbols	Ensembl_ID
0.037842	0.0496143	1.206061	SIX5	ENSG00000177045

```
## Removing uneccesary data from the environment
  remove(nrow_gF, nrow_pF)
```

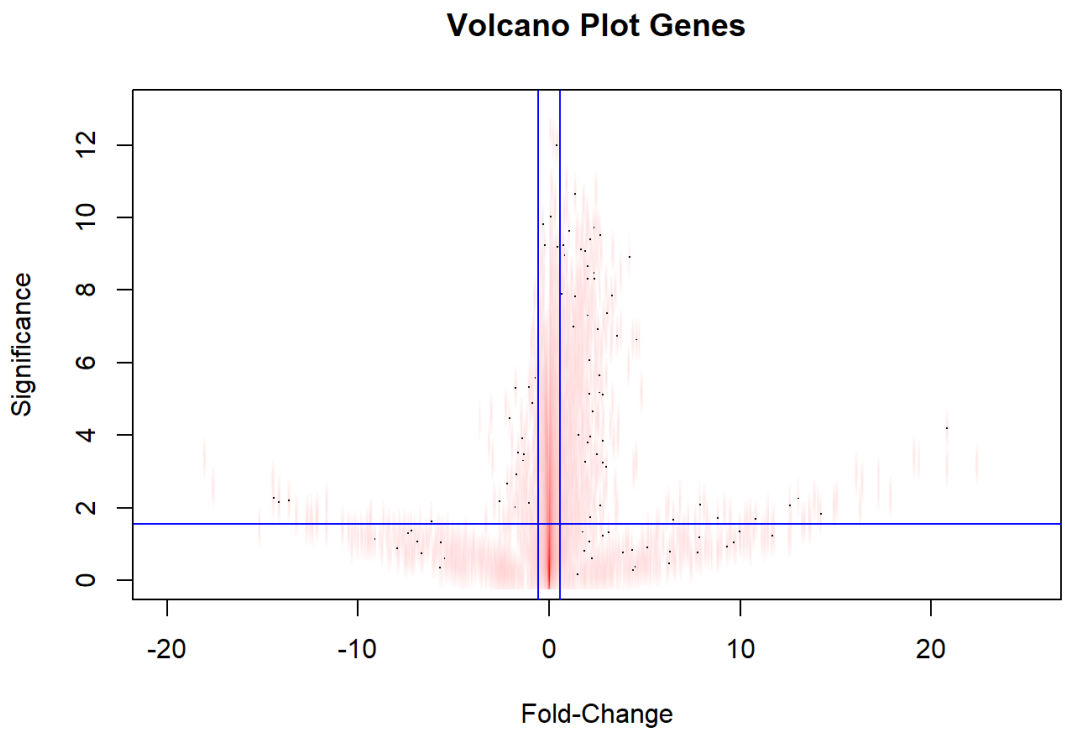
As the output indicates, the genes data frame shrinks proportionally more than the promoters data set, which falls in line with the observation that the promoters data set has had clearer internal structures throughout the entire analysis. Both show a workable sum of of Biologically relevant genes.

In addition, the previews of the ordered data sets show us that the last significant and relevant p-value ended up being between 2.7-3.7% in both cases. This would be equivalent to our mutliple testing corrected p-value, with the consideration that relevance was drawn into account as well.

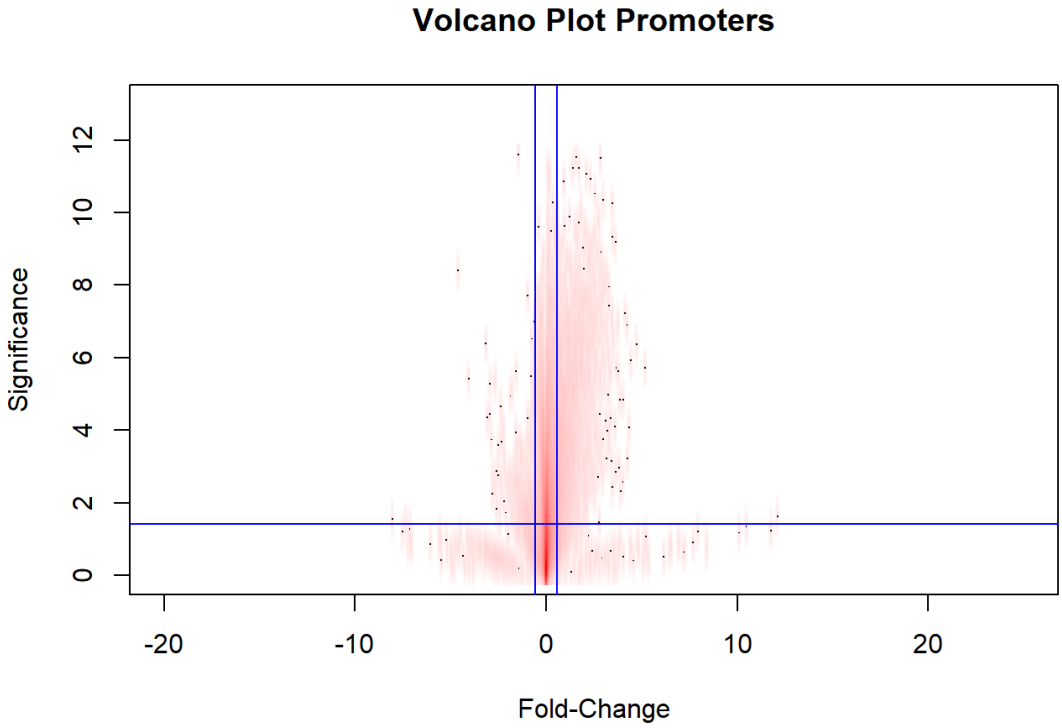
4.7 Plotting of Results

Having completed all steps necessary for the testing portion of this analysis, we will now create a volcano plot of all genes, marking the levels at which relevance/significance was cut. In order to do this we will be referencing the original T data sets after their merging with the Resource data sets (called the Finale Comparison data sets). Each Volcano plot will be for genes and promoters respectively, and the marked significance is equal to the largest p-value included in each respective Finale data set.

```
## Final comparative results and plotting for genes
  g_FinaleComparison <- merge(g_T, g_Resource, by = 0, all = FALSE)
  rownames(g_FinaleComparison) <- g_FinaleComparison$Row.names
  g_FinaleComparison <- g_FinaleComparison[, c(8,11, 10, 9)]

  smoothScatter(g_FinaleComparison$Foldchange_Beta, -log10(g_FinaleComparison$p_values),
                nbin = 1000,
                colramp = colorRampPalette(c("white", "red")),
                main = "Volcano Plot Genes",
                ylab = "Significance",
                xlab = "Fold-Change",
                xlim = c(-20, 25),
                ylim = c(0, 13))

  abline(h = -log10(max(g_Finale$p_values)), col = "blue")
  abline(v = log2(2/3), col = "blue")
  abline(v = log2(3/2), col = "blue")
```



```
## Final comparative results and plotting for promoters
  p_FinaleComparison <- merge(p_T, p_Resource, by = 0, all = FALSE)
  rownames(p_FinaleComparison) <- p_FinaleComparison$Row.names
  p_FinaleComparison <- p_FinaleComparison[, c(8,11, 10, 9)]

  smoothScatter(p_FinaleComparison$Foldchange_Beta, -log10(p_FinaleComparison$p_values),
                nbin = 1000,
                colramp = colorRampPalette(c("white", "red")),
                main = "Volcano Plot Promoters",
                ylab = "Significance",
                xlab = "Fold-Change",
                xlim = c(-20, 25),
                ylim = c(0, 13))

  abline(h = -log10(max(p_Finale$p_values)), col = "blue")
  abline(v = log2(2/3), col = "blue")
  abline(v = log2(3/2), col = "blue")
```




```
## Removing uneccesary data sets

  remove(g_T, g_Resource, g_FinaleComparison)
  remove(p_T, p_Resource, p_FinaleComparison)
```

These outputs show us multiple trends. For one, the promoter results show a trend of higher significance and relevance than genes. Another point is that both genes and promoters show a distinct cloud of data that is significantly hypermethylated branching off of the central non-relevant area. In addition, both graphs show an area of less significant and relevant hypomethylations also taking place. Both graphs show a high density of points in non-significant and non-relevant areas of the graph, which indicates that a high proportion of sequences are not directly impacted in AML. Finally, both graphs show “arms” extending in the lower portion of the graph. These areas of mostly dubious significance and incredibly high Fold-Change likely indicate areas where small means with large standard deviations create this form of distortion. These arms are weaker on the promoter plot, once more in line with the higher quality of data we have come to expect from the promoters data set.

5. Logistic Regression

While having now reached the end of our search for DMR’s, we decided to do a further Logistic regression. This could hopefull show us which sequences amongst our candidates are the most cancer related and important in differentiating phenotypes. In order to perform a Logistic regression with the greatest accuracy possible, we pool our finale data sets. In this manner, the Logistic regression

model can draw from all present markers to describe cancer related sequences, rather than only differentiating within genes or promoters. This allows for the opportunity of a broader scope of the information to be included into the model's calculations.

5.1 Creating a Joint Data Set of Finalists

First we merge each of our final data sets in such a manner that we can still recognize which data set each sequence came from. We then get the Mvalue information for the all the Finalsits and put it all together in a data set along with the assosciated fold change and signficance of each sequence.

```
## Getting M-value data that corresponds to our finalists
lg_Mvalues <- g_Mvalues[which(rownames(g_Mvalues) %in% rownames(g_Finale)), ]
lp_Mvalues <- p_Mvalues[rownames(p_Mvalues) %in% rownames(p_Finale), ]

## We order all four of our data sets alphabetically by row name so that when we cbind the right values are assi
gned to each sequence
g_Finale <- g_Finale[order(rownames(g_Finale)), ]
lg_Mvalues <- lg_Mvalues[order(rownames(lg_Mvalues)), ]

p_Finale <- p_Finale[order(rownames(p_Finale)), ]
lp_Mvalues <- lp_Mvalues[order(rownames(lp_Mvalues)), ]

## Formatting and combining our data sets
lg_Mvalues <- cbind(lg_Mvalues, g_Finale$p_values,
                    g_Finale$Foldchange_Beta)

lp_Mvalues <- cbind(lp_Mvalues, p_Finale$p_values, p_Finale$Foldchange_Beta)

colnames(lg_Mvalues) <- c("AML1", "AML2", "AML3", "AML4", "AML5", "AML6", "AML7", "AML8",
                          "AML9", "AML10","Mon1", "Mon2", "Mon3", "Mon4", "Mon5", "Mon6",
                          "Mon7", "Mon8", "Mon9", "Mon10", "p_values", "Foldchange_Beta")

colnames(lp_Mvalues) <- c("AML1", "AML2", "AML3", "AML4", "AML5", "AML6", "AML7", "AML8",
                          "AML9", "AML10","Mon1", "Mon2", "Mon3", "Mon4", "Mon5", "Mon6",
                          "Mon7", "Mon8", "Mon9", "Mon10", "p_values", "Foldchange_Beta")

row.names(lg_Mvalues) <- paste0(row.names(lg_Mvalues), c("G"))
row.names(lp_Mvalues) <- paste0(row.names(lp_Mvalues), c("P"))

l_Finale_Compound <- rbind(lg_Mvalues, lp_Mvalues)

## Printing the size of new data set
print(paste0("Combined ", nrow(p_Finale), " and ", nrow(g_Finale)," sequences and ended up with a combined size
 of ",nrow(l_Finale_Compound), " sequences."))
```

```
## [1] "Combined 2280 and 142 sequences and ended up with a combined size of 2422 sequences."
```

```
## Previewing our new data set
kable(head(l_Finale_Compound[, c(1:5, 11:15, 21:22)], n = 1), align = "c", digits = 3,)
```

	AML1	AML2	AML3	AML4	AML5	Mon1	Mon2	Mon3	Mon4	Mon5	p_values	Foldchange_Beta
ENSG00000069696G	-0.257	0.157	2.36	-0.432	1.136	-2.425	-2.028	-2.343	-1.939	-2.666	0.001	1.692

```
## Removing uneccesary data sets
remove(lg_Mvalues, g_Mvalues)
remove(lp_Mvalues, p_Mvalues)
```

As we can see in the output, we have properly combined our finalists. We can also see an excerpt from the new data frame, which has the Mvalue information of each sequence assosciated with it's relevance and significance.

5.2 Sorting our Compound Data Set

We will now roughly sort our data set by significance and relevance so that the first vaues to be fed to our Logistic regression model are the most likely to make a significant impact. The fact that we have so many Finalist Sequences that predominantly have positive Fold-Changes and high significances gives an indication that we might have an issue with too many and too highly correlated data, thus we want to give the highest quality sequences to the model first, as these are our most likely research candidates. Since this model is only meant as an indication of how good our predictors are, it is ok for this ordering to be a bit more rough. After all, we are simply asking “do these top finalists really give us the ability to differentiate phenotypes? Are they really cancer assosciated?”.

If we can create an accurate model with them, that question will have been answered in the positive, and we will be creatíng a new ranking anyway with the Logistic regression data.

In the process of this sorting we will also take some formatting steps to make our data set ready for the Logistic regression.

```
## Ranking our Data Set
  ## Generating a ranking of significance
  l_sig <- rank(l_Finale_Compound$p_values)

  ## Generating a ranking for highest absolute value of the fold change
  l_fc <- rank(-abs(l_Finale_Compound$Foldchange_Beta))

  ## Integrating these rankings with one another to generate an overall evaluation of
  ## both relevance and significance
  l_rank <- (l_sig+l_fc)/2
  l_Finale_Compound <- l_Finale_Compound[order(l_rank), 1:20]

  ## Creating a preview of how the ranking systen worked
  l_rank_preview <- cbind(l_fc, l_sig, l_rank)
  l_rank_preview <- l_rank_preview[order(l_rank), ]
  colnames(l_rank_preview) <- c("Relevance", "Significance", "Rank")

  ## Previewing how the ranking worked
  kable(head(l_rank_preview, n = 4), align = "c" )
```

Relevance	Significance	Rank
22	11	16.5
21	29	25.0
57	10	33.5
69	3	36.0

```
  ## removing uneccesary datasets
  remove(l_sig, l_fc, l_rank_preview)

## Data Set Formatting
  ## We now add a top row for phenotype of the patient and set the AML patients to 1 and
  ## healthy ones to 0
  l_Finale_Compound <- rbind(c(rep(0, 20)), l_Finale_Compound)
  rownames(l_Finale_Compound)[1] <- c("Phenotype")
  l_Finale_Compound[1, ] <- c(rep(1, 10), rep(0, 10))

  ## Transposition
  l_Finale_Compound <- data.frame(t(l_Finale_Compound))

## Previewing our new data set
  kable(head(l_Finale_Compound[c(1:5, 11:15), c(1:4, 12:14)], n = 10), align = "c", col.names = c("Phenotype", r
ep("Sequence", 6)), digits = 3)
```

	Phenotype	Sequence	Sequence	Sequence	Sequence	Sequence	Sequence
AML1	1	-1.742	-2.500	-0.670	-1.303	-1.770	-3.114
AML2	1	-3.237	-4.400	-1.650	-2.599	-2.077	-3.475
AML3	1	-1.988	-3.125	-0.377	-0.811	-0.747	-2.241
AML4	1	-2.221	-3.963	-0.783	-0.226	-1.456	-2.987

	Phenotype	Sequence	Sequence	Sequence	Sequence	Sequence	Sequence
AML5	1	-1.971	-3.109	-0.780	-1.106	-2.478	-2.695
Mon1	0	-5.840	-6.545	-4.156	-4.669	-4.536	-5.416
Mon2	0	-6.144	-6.888	-3.948	-5.088	-4.454	-5.877
Mon3	0	-5.876	-6.898	-4.402	-5.128	-4.663	-5.662
Mon4	0	-5.253	-6.778	-2.406	-4.708	-3.061	-6.163
Mon5	0	-6.906	-7.182	-4.756	-5.049	-4.621	-6.060

As we can see from the output, the Compounded data set is now transposed, containing 20 rows of samples and many rows of relevant Sequences. It is also visible that all the samples are marked with a column for phenotype. In cancer the phenotype is “1” and for healthy the phenotype is “0”. In this excerpt, some rows have been omitted and the column names have been generalized.

5.3 Calculation and Assessment of Regression Models

In this stage we will perform 2 steps of an analysis. First we will calculate a specific Logistic regression model. We will assess it based on its number of regression coeffcients, predictive quality, a few qualities presented in the summary and lastly a type 1 anova.

In the second step, in order to find out if the qualities of the distinct model are comparable to to the qualities of any other model that is based on differing permutations of training and test data sets, we will be measuring predictive capabilties of 20 models based on 20 different permutations of test and training data sets. The reader should be aware that only the predictive quality is being measured for repeated permutations, this is because the main interest of the inquiry was whether or not such a model could be created and predict accurately.

The other qualities are not of significant interest, because we are not looking to create an actual model, as an actual model has no real application in our line of inquiry.

5.3.1 Single Model Analysis

```
## Generating a logitical regression model for further analysis
  ## Sampling random rows from our Compound data set, where 70% of a cohort is always in
  ## training
  set.seed(123)

  l_trainseq <- c(sample(seq(1, 10, 1), 7), sample(seq(11, 20, 1), 7))
  l_testseq <- seq(1,20,1)
  l_testseq <- l_testseq[!(l_testseq %in% l_trainseq)]

  l_Finale_Compound_train <- l_Finale_Compound[l_trainseq, ]
  l_Finale_Compound_test <- l_Finale_Compound[l_testseq, ]

  ## Generating the model
  l_Model <- glm(Phenotype ~., family = binomial(link = 'logit'), data = l_Finale_Compound_test)

## Previewing the number of regression coefficients in the Model
  l_Model_coeff <- summary(l_Model)$coefficients
  kable(l_Model_coeff, align = "c")
```

	Estimate	Std. Error	z value	Pr(>|z|)
(Intercept)	-34.330971	4489687.7	-7.60e-06	0.9999939
ENSG00000118946P	-1.614628	827883.2	-2.00e-06	0.9999984
ENSG00000151322P	-18.354034	942273.9	-1.95e-05	0.9999845
ENSG00000152977P	18.978887	1037524.8	1.83e-05	0.9999854
ENSG00000233642P	21.785105	1317338.8	1.65e-05	0.9999868
ENSG00000145147P	-8.676106	576619.7	-1.50e-05	0.9999880

```
## Removing unecessary data sets
  remove(l_Finale_Compound_train, l_Finale_Compound_test, l_Model_coeff, l_Model, l_testseq, l_trainseq)
```

This output shows us that only five regression coefficients ended up being included in the model. The rest of the sequences were deemed non relevant for the futher analysis. These missing predictors were cut away due to high correlation to the other predictors. In order to keep calculation times efficient we will perform this process again and simply shorten the data set to the relevant variables this time around. It is clear that cutting away these additional sequences cannot impact the model.

```
## Reducing the Compound data set
l_Finale_Compound_red <- l_Finale_Compound[, 1:6]

## Generating the Modle again
set.seed(123)

l_trainseq <- c(sample(seq(1, 10, 1), 7), sample(seq(11, 20, 1), 7))
l_testseq <- seq(1,20,1)
l_testseq <- l_testseq[!(l_testseq %in% l_trainseq)]

l_Finale_Compound_train <- l_Finale_Compound_red[l_trainseq, ]
l_Finale_Compound_test <- l_Finale_Compound_red[l_testseq, ]

## Generating the model
l_Model <- glm(Phenotype ~., family = binomial(link = 'logit'), data = l_Finale_Compound_test)

## Analysing the predictive qualities of the model
l_results <- predict(l_Model,newdata=l_Finale_Compound_test,type='response')
l_results <- ifelse(l_results > 0.5, 1, 0)
l_accuracy <- mean(l_results == c(1, 1, 1, 0, 0, 0))
print(paste0("The model assigned the test samples to cohorts with an overall ",100*l_accuracy, " % accuracy."))
```

```
## [1] "The model assigned the test samples to cohorts with an overall 100 % accuracy."
```

```
## Analysing the summary qualities of the model
summary(l_Model)
```

```
##
## Call:
## glm(formula = Phenotype ~ ., family = binomial(link = "logit"),
##     data = l_Finale_Compound_test)
##
## Deviance Residuals:
## [1]  0  0  0  0  0  0
##
## Coefficients:
##                    Estimate Std. Error z value Pr(>|z|)
## (Intercept)      -3.433e+01  4.490e+06       0        1
## ENSG00000118946P -1.615e+00  8.279e+05       0        1
## ENSG00000151322P -1.835e+01  9.423e+05       0        1
## ENSG00000152977P  1.898e+01  1.038e+06       0        1
## ENSG00000233642P  2.179e+01  1.317e+06       0        1
## ENSG00000145147P -8.676e+00  5.766e+05       0        1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 8.3178e+00  on 5  degrees of freedom
## Residual deviance: 2.5720e-10  on 0  degrees of freedom
## AIC: 12
##
## Number of Fisher Scoring iterations: 23
```

```
## Analysing model quality via type 1 anove
anova(l_Model)
```

```
## Analysis of Deviance Table
##
## Model: binomial, link: logit
##
## Response: Phenotype
##
## Terms added sequentially (first to last)
##
##
##                   Df Deviance Resid. Df Resid. Dev
## NULL                                  5     8.3178
## ENSG00000118946P  1   8.3178         4     0.0000
## ENSG00000151322P  1   0.0000         3     0.0000
## ENSG00000152977P  1   0.0000         2     0.0000
## ENSG00000233642P  1   0.0000         1     0.0000
## ENSG00000145147P  1   0.0000         0     0.0000
```

```
## Removing unecessary data sets
remove(l_Finale_Compound_red,l_Finale_Compound_train, l_Finale_Compound_test, l_Model, l_testseq, l_trainseq, l_
results, l_accuracy)
```

These three outputs tell us a number of things. For one, it seems that the predictive accuracy of the model is perfect. This indicates that the five sequences included do answer our original queston in the positive.

In addition, we can see from the summary that our residuals with the full model are nigh on 0, again this answers our original question in the positive. In addition we can see that many iterations were required for the model to converge, something that due to the structure of our data indicates separation in our data set. While this is bad if we are trying to fit our model, it is very good when it comes to answering our original question. This falls in line wth the very large regression coefficients and very large standard deviations for these observed, as this is common when models have trouble converging. It also indicates that if we were interested in creating a model we would need a higher n to our sample sizes.

Lastly the anova gives us two points of information. For one, the warnings show us that again there is separation in our data. in addition, the anova shows us that sequence 1 already gives us 0 deviance. Not only is there separation present, it is already visible with just one sequence in the model.

Given these results it is safe to say that this single Model has fallen victim a very high degree of separation and correlation within our data. This indicates that our first five Finalist sequences are strongly cancer assosciated in our data set and can set the basis of differentiation between the two cohorts.

However, to ensure that this trend holds for the rest of the data and isn't just a byproduct of the starting conditions of the test and training data sets, we will now expand our view.

5.3.2 Multi-Model Analysis

We now analyse the predictive qualities of a multitude of models. We will shorten the compounded data set to always only be the phenotype and one sequence and test the predictive quality of a model relying only on that sequence. We call these types of models "single sequence models". We hope to analyse wether or not single sequence models for all of our Finalist sequences show the same high predictive accuracy due to separation.

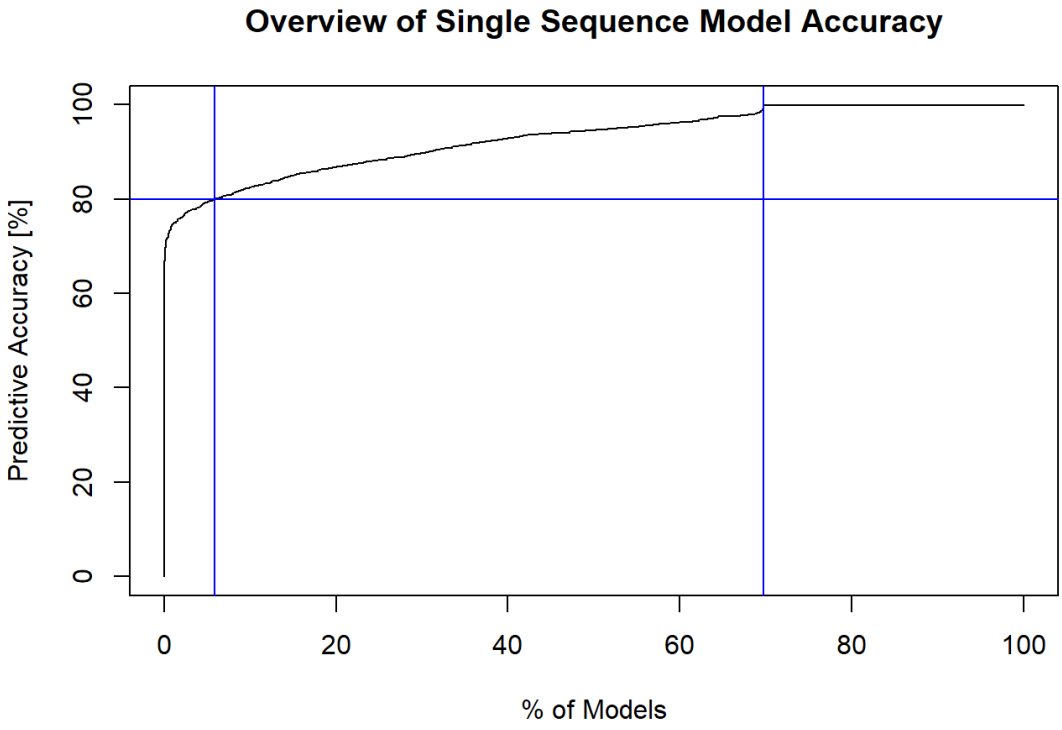
In addition, to avoid bias in the starting conditions (which samples are test and which are training) we will run each single sequence model for 100 different variations of the starting conditions.

The warnings R produces have been supressed in this code because they overwhelm the console and document otherwise. While normally they are useful when building a model, they flood our document in our scenario because so many sequences show high separation.

```
## Defining a function that creates the test and training data sets and calculates the
## accuracy of the resulting model
## What is important to note is that the only changing input when this function is used
## is the seed, which is run through 100 different starting conditions

fl_test <- function(x) {
  set.seed(x)
  l_trainseq <- c(sample(seq(1, 10, 1), 7), sample(seq(11, 20, 1), 7))
  l_testseq <- seq(1,20,1)
  l_testseq <- l_testseq[!(l_testseq %in% l_trainseq)]

  l_Finale_Compound_train <- l_Finale_Compound_red[l_trainseq, ]
  l_Finale_Compound_test <- l_Finale_Compound_red[l_testseq, ]

  l_Model <- glm(Phenotype ~., family = binomial(link = 'logit'),data = l_Finale_Compound_test)

  l_results <- predict(l_Model,newdata=l_Finale_Compound_test,type='response')
  l_results <- ifelse(l_results > 0.5, 1, 0)
  l_accuracy <- mean(l_results == c(1, 1, 1, 0, 0, 0))
  return(l_accuracy)
}

## This for loop will test the predictive accuracy for each sequence for 100 random
## starting conditions (NB: this calculation may take a while)
l_sequence_accuracy <- c()

suppressWarnings(
  for(i in 2:ncol(l_Finale_Compound)) {

    if(i %% floor(ncol(l_Finale_Compound)/100) == 0) {
      cat("|")
    }

    l_Finale_Compound_red <- l_Finale_Compound[, c(1,i)]

    j <- data.frame(seq(1, 100, 1))
    l_accuracy <- apply(j, 1, fl_test)

    l_sequence_accuracy <- c(l_sequence_accuracy, mean(l_accuracy))

  }
)
```

```
## ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

```
## Previewing what the accuracy looks like overall

plot(ecdf(l_sequence_accuracy)(seq(0,1, 0.001))*100,seq(0,1, 0.001)*100,type = "l", main = "Overview of Single S
equence Model Accuracy", ylab = "Predictive Accuracy [%]", xlab = "% of Models")

abline(h = 80, col = "blue")
abline(v = ecdf(l_sequence_accuracy)(0.8)*100, col = "blue")
abline(v = ecdf(l_sequence_accuracy)(0.9999999999999999)*100, col = "blue")
```




```
## Removing uneccesary data sets
remove(i,j, fl_test, l_accuracy, l_Finale_Compound_red)
```

The graph we see here indicates on the y axis a level of accuracy and on the x axis the percentage of single sequence models that are less than or equally accurate to the given level of accuracy. If a line at 50% accuracy intersects the curve at an x value of 0%, then this indicates that 0% of models are less accurate than 50%, and 100% of models are more accurate.

This output shows us that in general our single sequence models perform very well. The incredibly high predictive accuracies show us that most of our sequences show a high degree of separation between the samples. For example, more than 25% of our sequences show 100% accuracy over 100 iterations, and over 90% of our values show accuracy above 80%. While this shows that many of our best sequences are similarly accurate, we can still use this as a ranking method for our Finalists.

Most importantly, it is evident that the Logistic regression shows that even with a hand tied behind their backs the vast majority of our finalist sequences can be said to be strongly cancer assosciated and can be used as predictive indicators for phenotype within our data set.

5.4 Final Output

We will now rank our finalists and produce the final output of this analysis.

```
## Creating the l_Order data set which will order each Sequence by it's corresponding
## Single Sequence model's predictive accuracy

l_rank <- sort(l_rank)
l_Order <- data.frame(cbind(l_sequence_accuracy, l_rank))
rownames(l_Order) <- colnames(l_Finale_Compound[, 2:ncol(l_Finale_Compound)])
colnames(l_Order) <- c("Accuracy", "Rank")
l_Order <- l_Order[order(l_Order$Accuracy, decreasing = TRUE), ]

## Saving the rank and corresponding ensembl ID for each sequence
l_Order <- cbind(rownames(l_Order), seq(1, nrow(l_Order), 1))
l_Order <- data.frame(l_Order)
rownames(l_Order) <- l_Order[, 1]

## Creating a compounded data set from our finalist data sets
row.names(g_Finale) <- paste0(row.names(g_Finale), c("G"))
row.names(p_Finale) <- paste0(row.names(p_Finale), c("P"))

F_comp <- rbind(g_Finale, p_Finale)
F_comp <- F_comp[, c(1,3,4,5)]

## Merging the compound with the ranks by ensemble IDs
F_comp <- merge(F_comp, l_Order, by = "row.names", all = TRUE)

## Formatting
rownames(F_comp) <- F_comp$Row.names
F_comp <- F_comp[, c(2,3,4,5,7)]
F_comp$X2 <- as.numeric(as.character(F_comp$X2))

## Ordering the sequences by their ranks
F_comp <- F_comp[order(F_comp$X2), ]
F_comp <- F_comp[, c(4,3,1,2)]

## Defining our final output
DMRs_AML_Mono <- F_comp

## Previewing our Output
kable(head(DMRs_AML_Mono, n = 10), align = "c")
```

	Ensembl_ID	Symbols	p_values	Foldchange_Beta
ENSG00000118946P	ENSG00000118946	PCDH17	0	3.466262
ENSG00000151322P	ENSG00000151322	NPAS3	0	3.482243
ENSG00000152977P	ENSG00000152977	ZIC1	0	2.981234
ENSG00000233642P	ENSG00000233642	GPR158-AS1	0	2.838737
ENSG00000145147P	ENSG00000145147	SLIT2	0	2.720130
ENSG00000259663P	ENSG00000259663	NA	0	2.761727
ENSG00000257126P	ENSG00000257126	FOXG1-AS1	0	2.878674
ENSG00000223414P	ENSG00000223414	LINC00473	0	2.556296
ENSG00000224825P	ENSG00000224825	RORB-AS1	0	2.766213
ENSG00000276852P	ENSG00000276852	NA	0	2.970254

```
## Removing uneccesary data sets
remove(l_Order, F_comp, l_rank, l_sequence_accuracy, l_Finale_Compound, g_Finale, p_Finale)
```

This final output gives us our “top ten” candidate DMRs produced by the analysis, but obviously goes on to include all 2422 finalist sequences.

6. Bibliography

Du, P., Zhang, X., Huang, C.-C., Jafari, N., Kibbe, W.A., Hou, L., and Lin, S.M. (2010). Comparison of Beta-value and M-value methods for quantifying methylation levels by microarray analysis. BMC Bioinformatics 11, 587.

Fialkow, P., Janssen, J., and Bartram, C. (1991). Clonal remissions in acute nonlymphocytic leukemia: evidence for a multistep pathogenesis of the malignancy. Blood 77, 1415-1417.

Fialkow, P.J. (1976). Clonal origin of human tumors. Biochimica et Biophysica Acta (BBA) - Reviews on Cancer 458, 283-321.

Rakyan, V.K., Down, T.A., Balding, D.J., and Beck, S. (2011). Epigenome-wide association studies for common human diseases. Nature Reviews Genetics 12, 529.

Royston, J.P. (1982). An Extension of Shapiro and Wilk's W Test for Normality to Large Samples. Journal of the Royal Statistical Society Series C (Applied Statistics) 31, 115-124.

Satopaa, V., Albrecht, J., Irwin, D., and Raghavan, B. (2011). Finding a Kneedle in a Haystack: Detecting Knee Points in System Behavior.

Storey, J.D. (2002). A Direct Approach to False Discovery Rates. Journal of the Royal Statistical Society Series B (Statistical Methodology) 64, 479-498.

Ziller, M.J., Hansen, K.D., Meissner, A., and Aryee, M.J. (2015). Coverage recommendations for methylation analysis by whole-genome bisulfite sequencing. Nature methods 12, 230-232.