

Análisis de microarrays *Clariom D Human* en **limma**

Adam Casas

Compilado: 25 de julio del 2021

Contents

1	Introducción	1
2	Carga de datos y librerías	1
3	Métricas de calidad	5
3.1	Efecto lote	5
3.2	Análisis QC	5
4	Preprocesado de los microarrays: RMA	6
4.1	Comprobación del normalizado: Boxplots	7
5	Análisis DEG	8
5.1	Separación de las muestras: PCA, t-SNE y UMAP	8
5.2	Análisis cualitativo: MA Plots	12
5.3	limma: Análisis de 2 grupos de datos no pareados	13
6	Resultados análisis DEG	16
7	Bibliografía	18
8	sessionInfo()	18

1 Introducción

En este documento se muestra cómo analizar los archivos con extensión .CEL resultantes de secuenciar muestras biológicas en los microchips *Clariom D Human* de Affymetrix. Llevaremos a cabo este análisis en R 3.6 con el paquete de Bioconductor **oligo** versión 1.50.0. La versión de Biocondutor empleada es la 3.10.

2 Carga de datos y librerías

Primero debemos instalar **oligo**. Téngase en cuenta que en R 3.6 se instala la versión 1.50.0 de **oligo**, el cual depende de los paquetes **ff** versión 2.2.0 y **RSQLite** versión 2.1.4. Instalar versiones posteriores ocasionará fallos en la carga de **oligo** y en los análisis posteriores:

```
# Instalación oligo en R 3.6:  
BiocManager::install("oligo")  
  
# Paquete ff versión 2.2.0: (en el portatil la 2.2.14 funciona, en el sobremesa no)  
remove.packages("ff")
```



Figure 1: Microchip *Clariom D Human* y sus correspondientes reactivos.

```
install.packages("https://cran.r-project.org/src/contrib/Archive/ff/ff_2.2-0.tar.gz",
                 repos=NULL, type = "source")

installed.packages()

# Paquete RSQLite versión 2.1.4:
install.packages("https://cran.r-project.org/src/contrib/Archive/RSQLite/RSQLite_2.1.4.tar.gz",
                 repos=NULL, type = "source")
```

Comenzamos cargando `oligo` y paquetes accesorios:

```
library("oligo")
library("ggplot2")
library("dplyr")
library("FactoMineR") # PCA
library("rgl") # 3D plots
library("Rtsne") # t-SNE
library("uwot") # UMAP
library("limma") # DEG Analysis
```

Acto seguido definimos el directorio de trabajo en la carpeta donde se encuentran los archivos de secuenciado.

```
directorio_trabajo <- "../Archivos_secuenciado/"
```

Listamos y cargamos los archivos de secuenciado en el objeto de tipo `HTAFeatureSet` denominado `datos_crudos_microarrays`. La carga de los archivos `.CEL` se hace mediante la función `read.celfiles()`, la cual lee y carga los archivos por orden alfabético de sus nombres (*i.e.* primero carga el archivo `52CNT_(Clariom_D_Human).CEL`, luego `52GEN_(Clariom_D_Human).CEL`, luego `53CNT_(Clariom_D_Human).CEL`, ...etc):

```
# Obtenemos la ruta completa de los archivos con el argumento `full.names = T`
ruta_completa_archivos_secuenciado <- list.files(path = directorio_trabajo,
                                                 pattern = "*.CEL", full.names = T)
```

```

# Cargamos datos
datos_crudos_microarrays <- read.celfiles(filenames = ruta_completa_archivos_secuenciado)

## Reading in : ../Archivos secuenciado/52CNT_(Clariom_D_Human).CEL
## Reading in : ../Archivos secuenciado/52GEN_(Clariom_D_Human).CEL
## Reading in : ../Archivos secuenciado/53CNT_(Clariom_D_Human).CEL
## Reading in : ../Archivos secuenciado/53GEN_(Clariom_D_Human).CEL
## Reading in : ../Archivos secuenciado/54CNT_(Clariom_D_Human).CEL
## Reading in : ../Archivos secuenciado/54GEN_(Clariom_D_Human).CEL
## Reading in : ../Archivos secuenciado/55CNT_(Clariom_D_Human).CEL
## Reading in : ../Archivos secuenciado/55GEN_(Clariom_D_Human).CEL

# Observamos el tipo de objeto que hemos cargado
summary(datos_crudos_microarrays)

##          Length     Class      Mode
## 1 HTAFeatureSet     S4

# Observamos las sondas y muestras que tenemos
dim(datos_crudos_microarrays)

## Features   Samples
## 6892960        8

```

Podemos ver el nombre de los microchips con el comando `sampleNames()`.

```
sampleNames(datos_crudos_microarrays)
```

```

## [1] "52CNT_(Clariom_D_Human).CEL" "52GEN_(Clariom_D_Human).CEL"
## [3] "53CNT_(Clariom_D_Human).CEL" "53GEN_(Clariom_D_Human).CEL"
## [5] "54CNT_(Clariom_D_Human).CEL" "54GEN_(Clariom_D_Human).CEL"
## [7] "55CNT_(Clariom_D_Human).CEL" "55GEN_(Clariom_D_Human).CEL"

```

Vamos a cambiar sus nombres por otros más claros:

```

sampleNames(datos_crudos_microarrays) <- c("Control_1", "Genisteina_1",
                                             "Control_2", "Genisteina_2",
                                             "Control_3", "Genisteina_3",
                                             "Control_4", "Genisteina_4")

sampleNames(datos_crudos_microarrays)

## [1] "Control_1"    "Genisteina_1" "Control_2"    "Genisteina_2" "Control_3"
## [6] "Genisteina_3" "Control_4"    "Genisteina_4"

```

Al cargar los archivos .CEL con `read.celfiles()`, se almacena en el bolsillo `datos_crudos_microarrays@assayData$exprs` la matriz de intensidades crudas de los fluoróforos Cy3 y Cy5 que hibridan con las distintas sondas de cada microarray. Esta matriz tiene tantas filas como sondas tiene el microarray, y tantas columnas como muestras hayamos secuenciado. Las funciones `exprs()` e `intensity()` del paquete `oligo` simplifican el acceso a la misma (ambas funciones devuelven exactamente el mismo *output*).

- Las filas corresponden a cada sonda individual del microarray, identificada por un nº que va desde el 1 hasta el nº máximo de sondas presentes en el mismo (en el caso de los microarrays *Clariom D Human*, estos contienen un máximo de 6.892.960 sondas).
- Las columnas son las muestras, en nuestro caso Control_1, Genisteina_1, Control_2... etc

```

# Obtenemos la matriz de intensidades cruda
intensidades <- oligo::exprs(datos_crudos_microarrays)

```

```
# Identificamos el nº de sondas presentes en cada microchip
dim(intensidades)[1]

## [1] 6892960
```

De acuerdo a la documentación de Affymetrix, las sondas de los microarrays de tipo Clariom D Human (Gene Arrays) están agrupadas en *transcription clusters* en vez de *probesets*.

Con el comando `probeNames()` obtenemos los *transcription clusters* a los que pertenecen las sondas de nuestro microchip, de manera que la primera sonda del microarray pertenece al *transcription cluster* TC1300007722.hg.1:

```
# Obtenemos y mostramos los transcription clusters a los que pertenece cada sonda
transcription_clusters <- probeNames(datos_crudos_microarrays)
head(transcription_clusters)

## [1] "TC1300007722.hg.1" "TC0300011139.hg.1" "TC1600006939.hg.1"
## [4] "TC0100018028.hg.1" "TC0500010071.hg.1" "TC1900010290.hg.1"

# Cada microarray de tipo Clariom D Human contiene 1.220.891 transcription
# clusters
length(transcription_clusters)

## [1] 1220891

# Obtenemos las sondas que pertenecen al transcription cluster TC1300007722.hg.1
which(transcription_clusters == "TC1300007722.hg.1")

## [1] 206732 334298 840938 990710 1101378
```

Si tenemos en cuenta la página web de ThermoFisher, podemos usar el centro de análisis NetAffx para obtener más información sobre dichos clusters. Por ejemplo, si nos fijamos en el segundo *transcription cluster*, TC0300011139.hg.1, vemos que las sondas de este cluster detectan la presencia/ausencia de expresión del pseudogen RNA5SP132 (identificador Ensembl:ENSG00000201595).

Además, el paquete `Biobase` incluye también numerosas funciones que se pueden ejecutar sobre los objetos de tipo `HTAFeatureSet` para obtener información contenida en el mismo, tales como abstracts, anotaciones, ... etc.

```
# Anotación usada para nuestro objeto `HTAFeatureSet`
Biobase::annotation(datos_crudos_microarrays)

## [1] "pd.clariom.d.human"

# Acceder a los datos del experimento del objeto `HTAFeatureSet` (en nuestro
# caso no está anotado)
Biobase::experimentData(datos_crudos_microarrays)

## Experiment data
##   Experimenter name:
##   Laboratory:
##   Contact information:
##   Title:
##   URL:
##   PMIDs:
##   No abstract available.
```

En relación a lo dicho en el párrafo anterior, las gráficas que generemos a lo largo del protocolo de análisis pueden usar información ubicada en el bolsillo `datos_crudos_microarrays@phenoData@data`, por lo que podemos añadir metadatos de interés con el comando `pData()` de Biobase tal que así:

```
# Renombramos la columna "index" a "muestra"
colnames(pData(datos_crudos_microarrays)) <- "muestra"

# Añadimos el factor "Condición" a los metadatos de nuestro estudio
pData(datos_crudos_microarrays)$condicion <- as.factor(rep(c("Control", "Genisteina"),
                                                       times = 4))

# Visualizamos los metadatos
pData(datos_crudos_microarrays)

##           muestra condicion
## Control_1          1     Control
## Genisteina_1        2   Genisteina
## Control_2          3     Control
## Genisteina_2        4   Genisteina
## Control_3          5     Control
## Genisteina_3        6   Genisteina
## Control_4          7     Control
## Genisteina_4        8   Genisteina
```

3 Métricas de calidad

3.1 Efecto lote

Para analizar el efecto lote, podemos usar las fechas en las que se secuenciaron los microarrays. El comando `get.celfile.dates()` del paquete `affyio` devuelve la fecha de dicho proceso.

```
affyio::get.celfile.dates(ruta_completa_archivos_secuenciado)

## [1] "2018-03-13" "2018-03-13" "2018-03-13" "2018-03-13" "2018-03-13"
## [6] "2018-03-13" "2018-03-13" "2018-03-13"
```

Vemos que los microarrays se secuencian el mismo día. Adicionalmente, el programa GeneChip Command Console 6.0 de ThermoFisher (compañía propietaria de Affymetrix) aportó la hora del secuenciado, el cual duró 6 horas en total (11AM-5PM).

En vista de que el secuenciado se realizó en un sólo lote (el mismo día y en horas consecutivas), concluimos que la probabilidad de observar varianza debida al ruido técnico del efecto lote es mínima.

3.2 Análisis QC

El programa TAC 4.0.2, propiedad también de ThermoFisher, muestra un resumen muy claro de las métricas de calidad de los microarrays.

Según la documentación del programa, un valor de `pos vs neg auc > 0.7` es un buen primer cribado de calidad. Además de presentar un `AUC > 0.7`, el resto de controles han salido bien, por lo que podemos confiar en la calidad del secuenciado de los presentes microarrays.

En R, el paquete `oligo` parece no poder sacar dichas métricas en microarrays de tipo *Clariom*, puesto que al ser nuevos, no llevan asociados archivos `.CDF`. Quizás `limma` o `DESeq2` puedan tratar dichos microchips.

File Name count: 8	Labeling Controls Threshold	Hybridization Controls Threshold	Pos vs Neg AUC Threshold	Condition	pos vs neg auc
52CNT_(Clariom_D_Human).sst-r...	Pass	Pass	Pass	Control	0,96
52GEN_(Clariom_D_Human).sst-r...	Pass	Pass	Pass	Genisteina	0,96
53CNT_(Clariom_D_Human).sst-r...	Pass	Pass	Pass	Control	0,95
53GEN_(Clariom_D_Human).sst-r...	Pass	Pass	Pass	Genisteina	0,96
54CNT_(Clariom_D_Human).sst-r...	Pass	Pass	Pass	Control	0,96
54GEN_(Clariom_D_Human).sst-r...	Pass	Pass	Pass	Genisteina	0,96
55CNT_(Clariom_D_Human).sst-r...	Pass	Pass	Pass	Control	0,95
55GEN_(Clariom_D_Human).sst-r...	Pass	Pass	Pass	Genisteina	0,96

Figure 2: Las muestras pasan todos los controles de calidad.

4 Preprocesado de los microarrays: RMA

Antes de analizar los microarrays, necesitamos preprocesar las señales de las sondas para corregir el ruido técnico (problema común en los microarrays) y agrupar la información de las sondas que mapeen para un mismo gen (*summarization*).

Para preprocesar y normalizar datos de microarrays, se suelen emplear el algoritmo RMA o derivados (*i.e.* GCRMA o SST-RMA). En nuestro caso vamos a usar el algoritmo RMA, implementado en `oligo`. Este algoritmo consta de 3 pasos:

- **Background subtraction.** De acuerdo a la documentación de `oligo`, el método de *background subtraction* implementado en RMA trata las sondas PM (*Perfect Match*, más informacion en Flight *et al.*, 2012) como una convolución de ruido y señal verdadera.
- **Quantile normalization.** Paso necesario para dotar de igual valor a los genes estudiados, independientemente de su nivel de expresión constitutivo (este paso evita, por ejemplo, que genes expresados de manera constitutiva pero poco interesantes para el estudio estén sobrerepresentados en análisis posteriores con respecto de genes más relevantes, pero de menor expresión basal como son los genes codificantes de factores de transcripción). El método de normalizado implementado en `oligo` es quantile.
- **Summarization.** Pasamos de analizar intensidades de sondas a niveles de expresión de genes. `oligo` realiza este paso mediante el método medianpolish.

Para más información sobre la implementación de estos pasos en `oligo`, por favor consulte la documentación de los comandos `rma()`, `backgroundCorrectionMethods()` y sucedáneos.

En `oligo`, el comando `rma()` nos devuelve un objeto de tipo `ExpressionSet` (similar al objeto `datos_crudos_microarrays`) con los datos preprocesados en el bolsillo `eSet_normalizado@assayData$exprs`. Procedemos al RMA:

```
eSet_normalizado <- oligo::rma(object = datos_crudos_microarrays,
                                background = T, normalize = T)
```

```
## Background correcting
## Normalizing
## Calculating Expression

head(eSet_normalizado@assayData$exprs) [,1:4]

##          Control_1 Genisteina_1 Control_2 Genisteina_2
## AFFX-BkGr-GC03_st 1.790273    2.420966  2.160333   1.985337
## AFFX-BkGr-GC04_st 1.799491    1.960165  2.075830   1.984062
## AFFX-BkGr-GC05_st 1.727440    1.866754  1.896221   1.768564
```

```

## AFFX-BkGr-GC06_st 1.813971      1.986778  2.054344      1.935531
## AFFX-BkGr-GC07_st 1.869190      2.018058  2.090683      1.985589
## AFFX-BkGr-GC08_st 1.929733      2.107665  2.187569      2.076767

```

4.1 Comprobación del normalizado: Boxplots

Podemos comparar los datos antes y después del normalizado y control de ruido técnico. Observamos que las intensidades medianas y los cuartiles 25-75 son idénticos entre todos los microarrays tras el normalizado, cosa que no era cierta en los datos crudos:

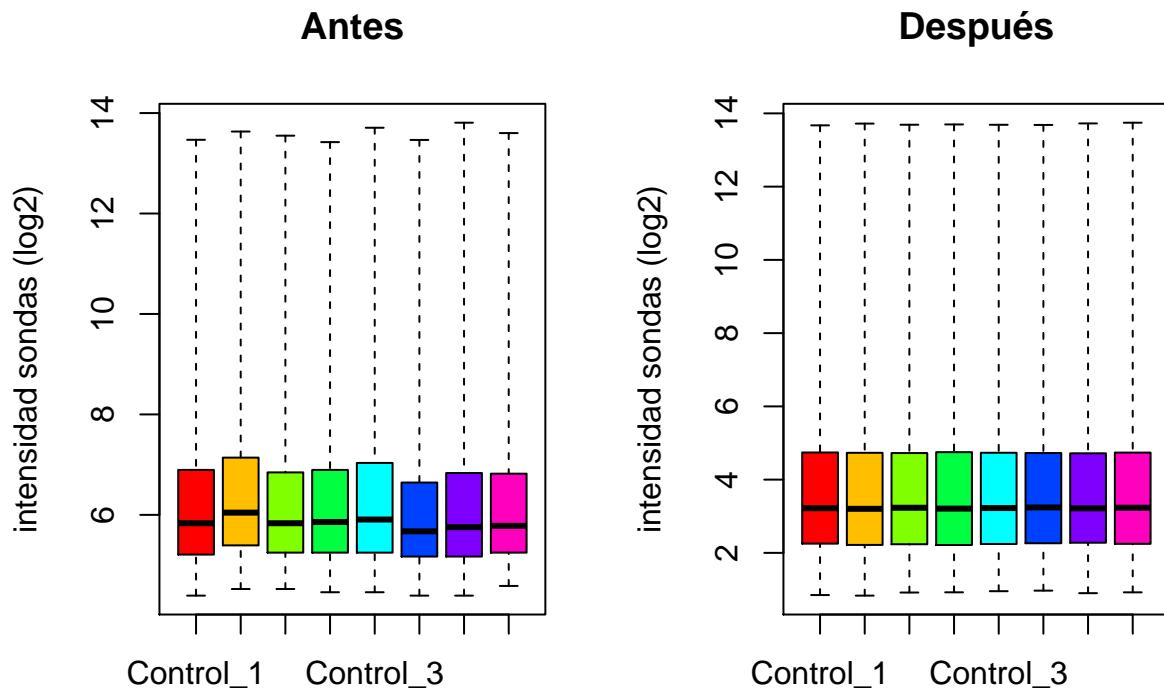
```

par(mfcol = c(1,2))

boxplot(datos_crudos_microarrays, "all",
         main = "Antes", col = rainbow(8),
         ylab = "intensidad sondas (log2)")

boxplot(eSet_normalizado, main = "Después",
        col = rainbow(8), ylab = "intensidad sondas (log2)")

```



5 Análisis DEG

5.1 Separación de las muestras: PCA, t-SNE y UMAP

Con los datos ya normalizados, podemos graficar los microarrays estudiados en un espacio de dimensionalidad reducida. La separación de los microarrays en dicho espacio nos dará una idea general de si existen genes diferencialmente expresados entre las condiciones estudiadas.

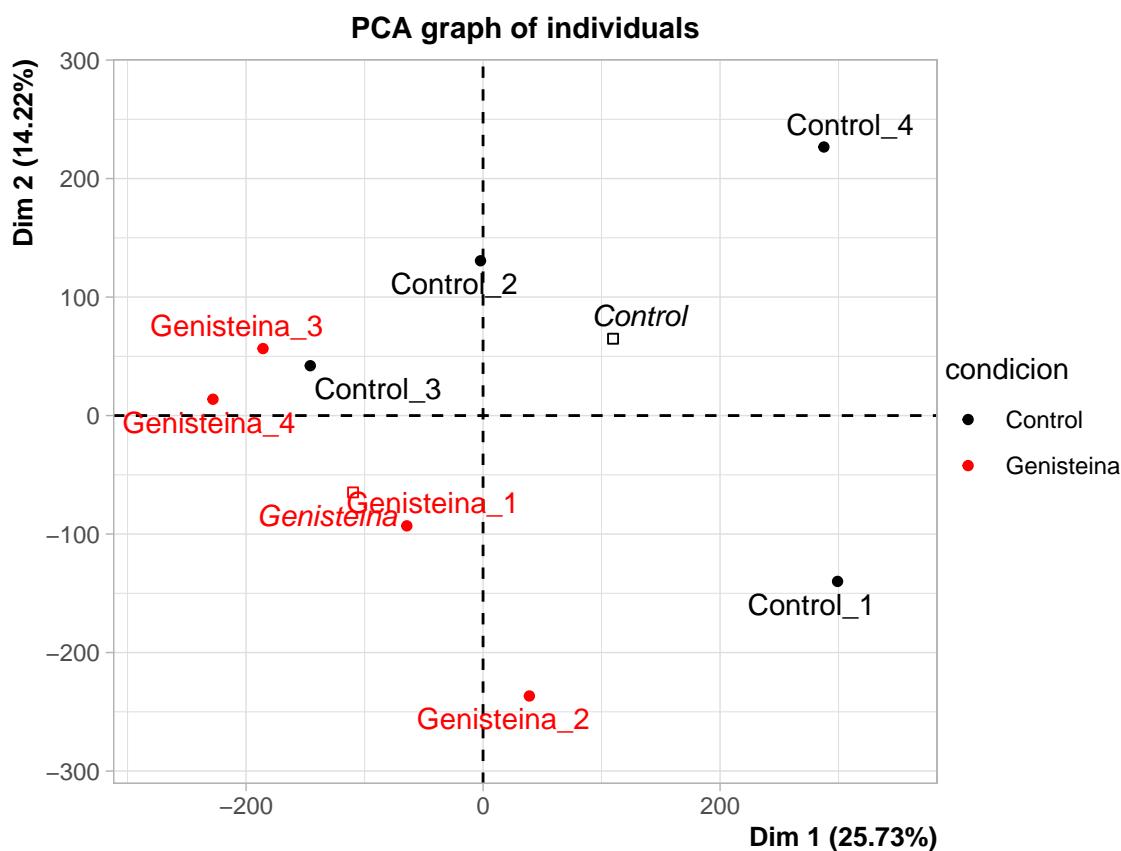
Para el PCA, podemos usar las utilidades del paquete FactoMineR tal que así:

```
# Creamos un dataframe de intensidades procesable por el PCA de
# dimensiones instancias x variables
intensidades_normalizadas <- t(exprs(eSet_normalizado))
intensidades_normalizadas_df <- as.data.frame(intensidades_normalizadas)

# Añadimos al dataframe el factor "condicion"
intensidades_normalizadas_df$condicion <- pData(eSet_normalizado)$condicion

# Computamos las 3 primeras componentes principales
pca_microarrays <- PCA(intensidades_normalizadas_df, graph = F,
                        axes = c(1:3), quali.sup = 138746)

# Graficamos las 2 primeras componentes principales
plot.PCA(pca_microarrays, choix = "ind", habillage = 138746)
```



```
# Graficamos las 3 primeras componentes principales
plot3d(pca_microarrays$ind$coord[,1:3], type = "s",
        col = c(rep(c("black", "red"), times = 4)))
```

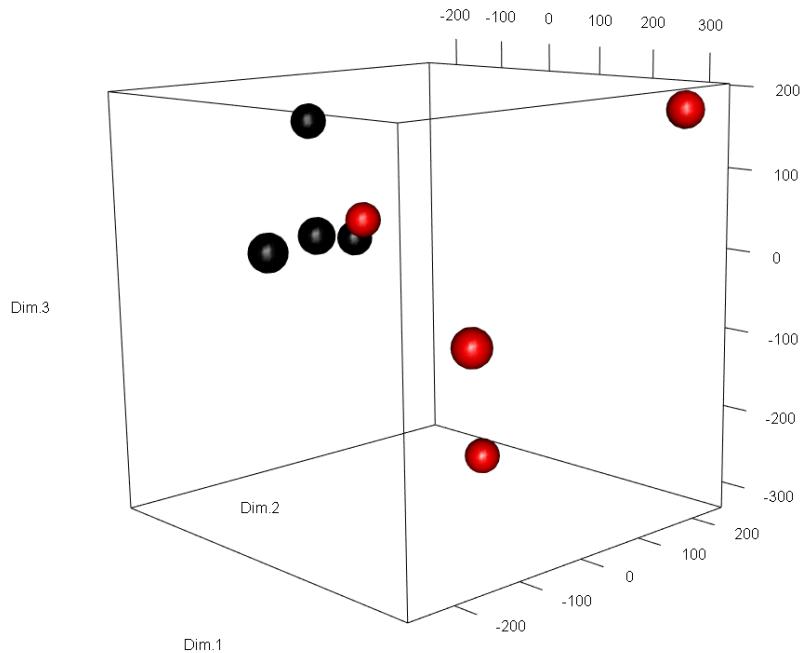


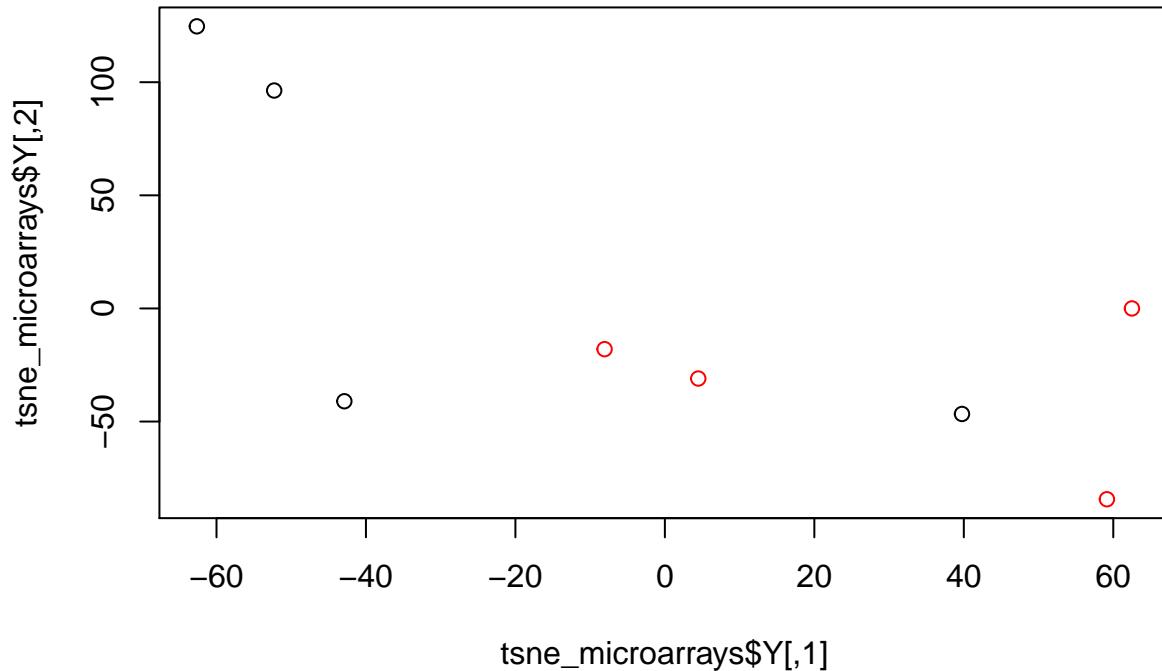
Figure 3: PCA tridimensional. Los puntos negros representan las muestras control, mientras que los puntos rojos representan las muestras tratadas con genisteína.

y para el t-SNE:

```
normalizado_tsne <- normalize_input(intensidades_normalizadas)
tsne_microarrays <- Rtsne(normalizado_tsne, dims = 3, perplexity = 2)

# Leyenda: Control negro, Genisteína rojo
plot(tsne_microarrays$Y, col = c("black", "red"), main = "t-SNE plot of individuals")
```

t-SNE plot of individuals



```
plot3d(tsne_microarrays$Y[,1:3], type = "s",
       col = c(rep(c("black", "red"), times = 4)))
```

y el UMAP:

```
umap_microarrays <- uwot::umap(intensidades_normalizadas, n_neighbors = 3,
                                 n_components = 3)

# Leyenda: Control negro, Genisteína rojo
plot(umap_microarrays, col = c("black", "red"), main = "UMAP plot of individuals")
```

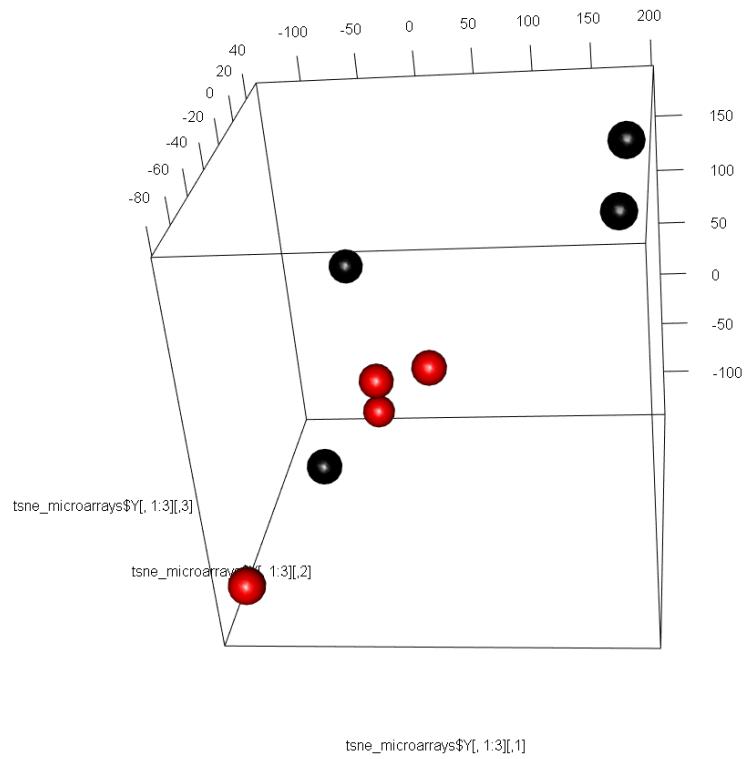
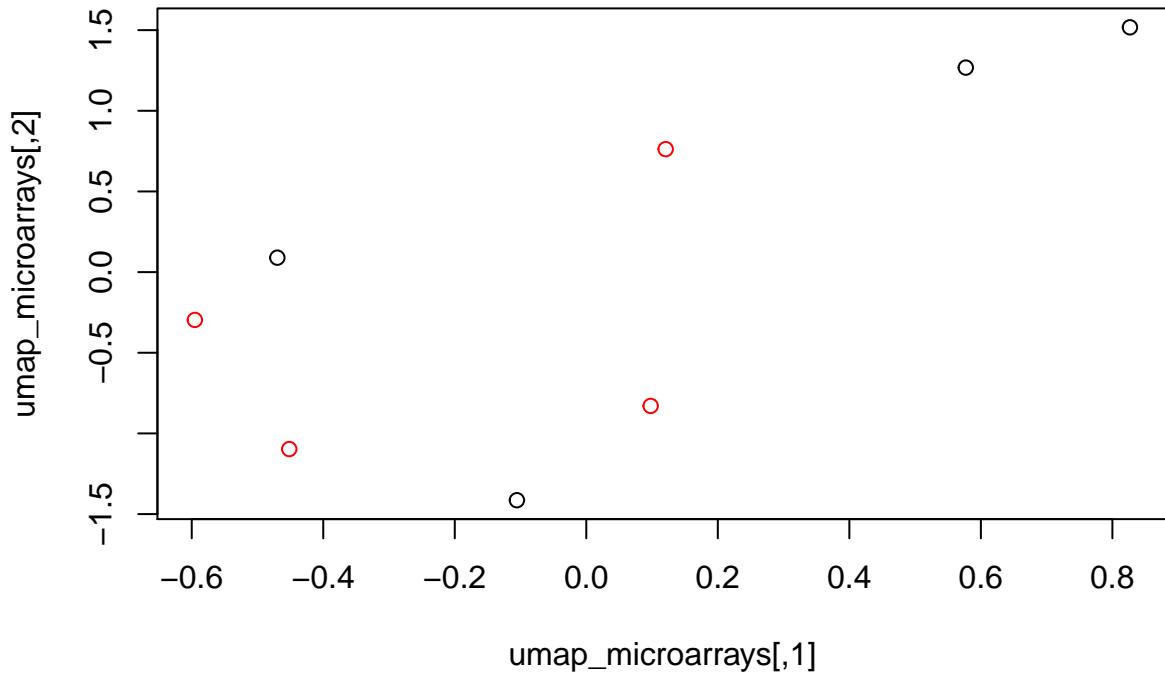


Figure 4: Reducción dimensional no lineal t-SNE de 3 dimensiones. Los puntos negros representan las muestras control, mientras que los puntos rojos representan las muestras tratadas con genisteína.

UMAP plot of individuals



```
# Graficamos el UMAP en 3D
plot3d(umap_microarrays, type = "s",
       col = c(rep(c("black", "red"), times = 4)))
```

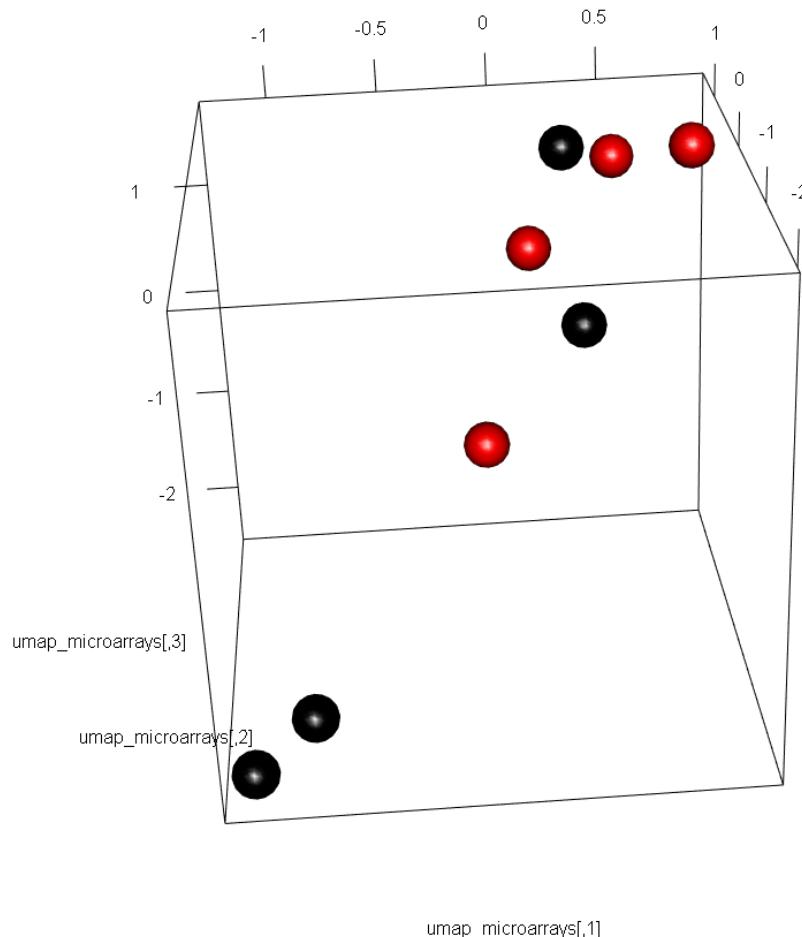


Figure 5: Reducción dimensional no lineal UMAP de 3 dimensiones. Los puntos negros representan las muestras control, mientras que los puntos rojos representan las muestras tratadas con genisteína.

En resumen, el PCA de 2 dimensiones es la reducción dimensional más informativa de la posible distribución de las muestras. El t-SNE y el UMAP en 3D no son concluyentes (los datos parecen mezclarse).

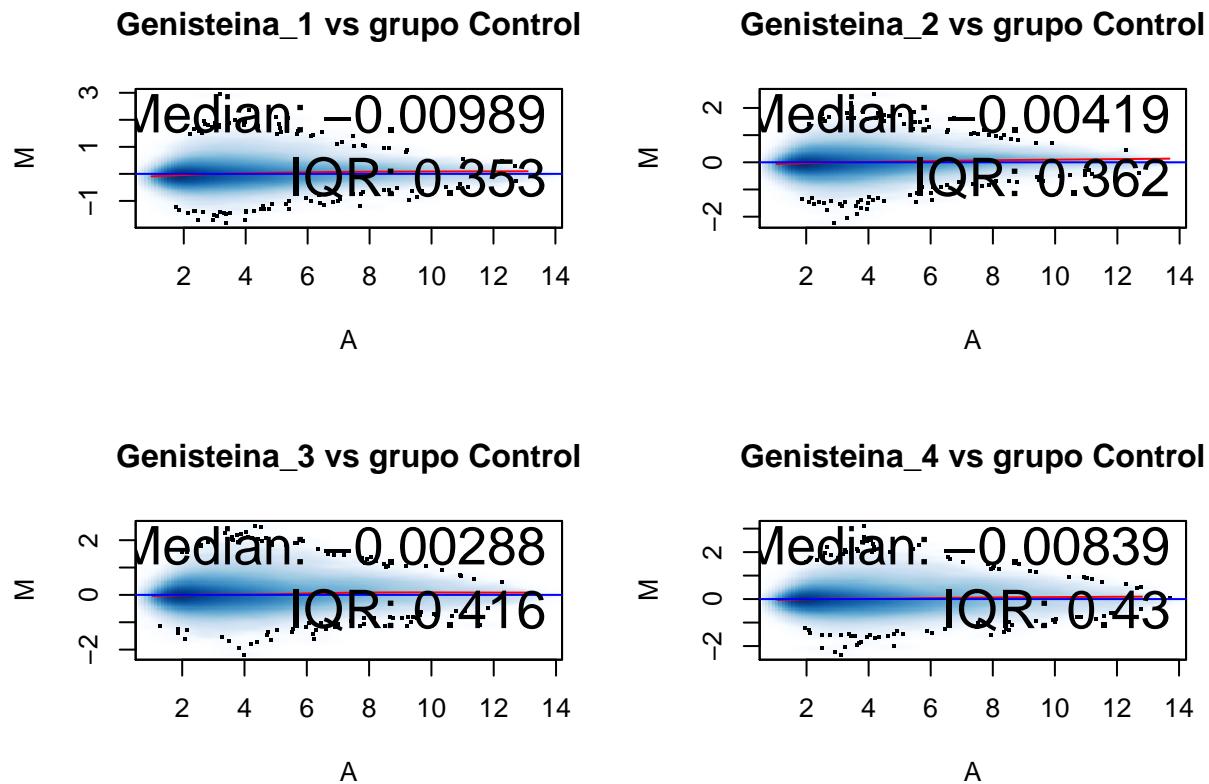
5.2 Análisis cualitativo: MA Plots

Para visualizar si hay genes diferencialmente expresados, podemos emplear el comando `MApplot()` del paquete `oligo`. Los gráficos MA muestran en el eje X la intensidad media de una sonda dada a lo largo de los microarrays estudiados, y en el eje Y se observa el $\log_2(\text{fold change})$ o logaritmo en base dos del ratio de sobre/infraexpresión del gen al que mapea dicha sonda.

Los genes diferencialmente expresados se encontrarán lejos de la nube de puntos ($\log_2(\text{FC}) > |1|$), mientras que los genes que no varían entre condiciones presentarán valores de $\log_2(\text{FC})$ cercanos a 0. No obstante, el MA plot no devuelve p-valores, por lo que para asegurarnos de que los resultados son significativos, deberíamos recurrir a un Volcano plot, que combina el $\log_2(\text{FC})$ y los p-valores de los genes estudiados.

```
par(mfrow=c(2,2))

oligo::MAnplot(eSet_normalizado, refSamples = c(1,3,5,7), which = c(2,4,6,8), main = "vs grupo Control")
```



5.3 limma: Análisis de 2 grupos de datos no pareados

Procedemos a detectar los genes diferencialmente expresados mediante las utilidades del paquete `limma`, ya que `oligo` carece de las funciones necesarias para ello. `limma` usa como *input* el objeto que devuelve el comando `rma()`.

Nuestros datos no son pareados, por lo que usaremos un t-test normal en lugar de uno para datos pareados (si comparásemos 3 grupos, usaríamos un ANOVA para medidas repetidas en lugar de un ANOVA de 1 vía).

`limma` contiene funciones para hacer t-tests o ANOVAs para identificar DEGs en microarrays. Estas funciones pueden ser usadas en datos provenientes de cualquier tipo de microarray y funcionan incluso para microarrays con diseños complejos o múltiples muestras. La idea principal es ajustar un modelo lineal a los datos de expresión de cada gen. Los datos de expresión pueden ser log-ratios o log-intensidades. Los métodos empleados obtienen información sobre todos los genes para hacer que los análisis sean estables incluso en experimentos con un bajo tamaño muestral.

Antes de nada tenemos que indicarle al susodicho paquete mediante el hueco `phenoData` a qué grupo (en este caso, factor `condicion`) pertenece cada muestra. Podemos ver que ya hemos hecho eso en pasos anteriores:

```
pData(eSet_normalizado)
```

```
##           muestra condicion
## Control_1      1     Control
```

```

## Genisteina_1      2 Genisteina
## Control_2        3   Control
## Genisteina_2     4 Genisteina
## Control_3        5   Control
## Genisteina_3     6 Genisteina
## Control_4        7   Control
## Genisteina_4     8 Genisteina

```

Ahora tenemos que crear una **matriz de diseño**, o sea una matriz con los niveles del factor que agrupa los datos. Los t-tests, ANOVAs y GLMs necesitan dicha matriz para saber qué muestras pertenecen a qué grupo. Puesto que **limma** usa un t-test o un ANOVA, necesita dicha matriz de diseño. puedes crearla con el método **model.matrix()**. Esta función toma como argumento un modelo de fórmula (*model formula*). La virgulilla (~) especifica el lado derecho de la fórmula. Si usas como fórmula ~0, **limma** calculará sólo la expresión génica media de cada grupo.

```

# Creamos el diseño del t-test no pareado
matriz_diseno = model.matrix(~ 0 + eSet_normalizado$condicion)
colnames(matriz_diseno) <- c("Control", "Genisteina")

```

Nuestra matriz de diseño tiene esta forma:

```

##   Control Genisteina
## 1      1      0
## 2      0      1
## 3      1      0
## 4      0      1
## 5      1      0
## 6      0      1

```

En esencia, **limma** va a comparar, para cada gen, su expresión media en las muestras control y su expresión media en las muestras tratadas con genisteína. Lo primero, debemos calcular la expresión media de cada grupo usando el comando **lmFit()**. Este comando ajustará un modelo lineal (definido previamente con el comando **model.matrix()**) a los datos para calcular la expresión génica media en las muestras control y en las muestras tratadas:

```

# Generamos el modelo lineal
modelo_lineal <- lmFit(t(intensidades_normalizadas), matriz_diseno)

# Vemos los bolsillos que posee el objeto `modelo_lineal`
names(modelo_lineal)

## [1] "coefficients"    "rank"           "assign"         "qr"
## [5] "df.residual"     "sigma"          "cov.coefficients" "stdev.unscaled"
## [9] "pivot"           "Amean"          "method"         "design"

```

Podemos ver la expresión génica media de cada grupo en formato \log_2 en el bolsillo **\$coefficients** del modelo lineal:

```

modelo_lineal$coefficients[5000:5005,]

##                               Control Genisteina
## TC0100008471.hg.1 2.149597  2.016560
## TC0100008472.hg.1 1.934316  1.975520
## TC0100008474.hg.1 3.331085  3.412404
## TC0100008475.hg.1 1.869798  1.788975

```

```
## TC0100008476.hg.1 1.750567 1.543073
## TC0100008477.hg.1 1.698808 1.536160
```

Ahora debes indicarle al paquete qué grupos quieras comparar. Para ello debes definir una **matriz de contrastes** en la cual se especifiquen las comparaciones a realizar mediante el comando `makeContrasts()`. Usando los nombres de las columnas de la matriz de diseño, puedes especificar el grupo control respecto al cual quieras comparar el grupo de interés (tratado con genisteína).

```
# creamos matriz de contraste
matriz_contrastes <- makeContrasts(Genisteina - Control, levels = matriz_diseno)
```

Nuestra matriz de contrastes tiene esta forma:

```
matriz_contrastes
```

```
##           Contrasts
## Levels      Genisteina - Control
##   Control              -1
##   Genisteina             1
```

Ahora calculamos las comparaciones de interés con el comando `contrasts.fit()`. En este caso, calculamos la diferencia de la expresión de cada gen entre las muestras tratadas con genisteína y las muestras control. Nótese que el output del susodicho comando es similar al del comando `lmFit()`:

```
diferencia_genisteina_control <- contrasts.fit(fit = modelo_lineal,
                                                 contrasts = matriz_contrastes)
```

Con las diferencias calculadas, estamos listos para realizar los tests estadísticos para comparar los grupos. Ya que compararemos dos grupos, usaremos un t-test para datos no pareados. No obstante, en el caso de los microarrays se debe usar un t-test moderado ya que al tener pocas muestras, aumenta la varianza de los datos y los resultados se vuelven poco fiables. El comando `eBayes()` estima la varianza media de todo el genoma y converge la varianza de cada gen hacia dicho valor global estimado (*i.e.* aumenta la varianza de genes poco variables y disminuye la varianza de genes muy variables). Este t-test moderado por el método de Bayes empírico es a consecuencia más potente que el t-test estándar para estudios de microarrays.

```
t_test_bayes = eBayes(diferencia_genisteina_control)
```

El comando `eBayes()` devuelve un dataframe con múltiples huecos, similar al output del comando `contrasts.fit()`. Nótese que a diferencia de `contrasts.fit()` (que sólo calcula la diferencia entre la expresión génica de los dos grupos), `eBayes()` calcula los p-valores de dichas diferencias:

```
names(t_test_bayes)
```

```
## [1] "coefficients"      "rank"          "assign"        "qr"
## [5] "df.residual"       "sigma"         "cov.coefficients" "stdev.unscaled"
## [9] "Amean"            "method"        "design"        "contrasts"
## [13] "df.prior"         "s2.prior"      "var.prior"     "proportion"
## [17] "s2.post"          "t"             "df.total"     "p.value"
## [21] "lods"             "F"             "F.p.value"
```

Puedes obtener el fold change en formato \log_2 de la expresión de cada gen a través del hueco `$coefficients` del t-test moderado:

```
# Log2FC de genes:
head(t_test_bayes$coefficients)

##           Contrasts
##           Genisteina - Control
## AFFX-BkGr-GC03_st      0.17265919
## AFFX-BkGr-GC04_st      0.03927341
## AFFX-BkGr-GC05_st      0.01800755
## AFFX-BkGr-GC06_st      0.03202369
## AFFX-BkGr-GC07_st      0.01623366
## AFFX-BkGr-GC08_st      0.04744642
```

Puedes encontrar los valores del estadístico t y el p-valor de cada gen en sus respectivos huecos:

```
head(t_test_bayes$p.value)

##           Contrasts
##           Genisteina - Control
## AFFX-BkGr-GC03_st      0.2672743
## AFFX-BkGr-GC04_st      0.7518232
## AFFX-BkGr-GC05_st      0.8830088
## AFFX-BkGr-GC06_st      0.7957134
## AFFX-BkGr-GC07_st      0.8947237
## AFFX-BkGr-GC08_st      0.7085777

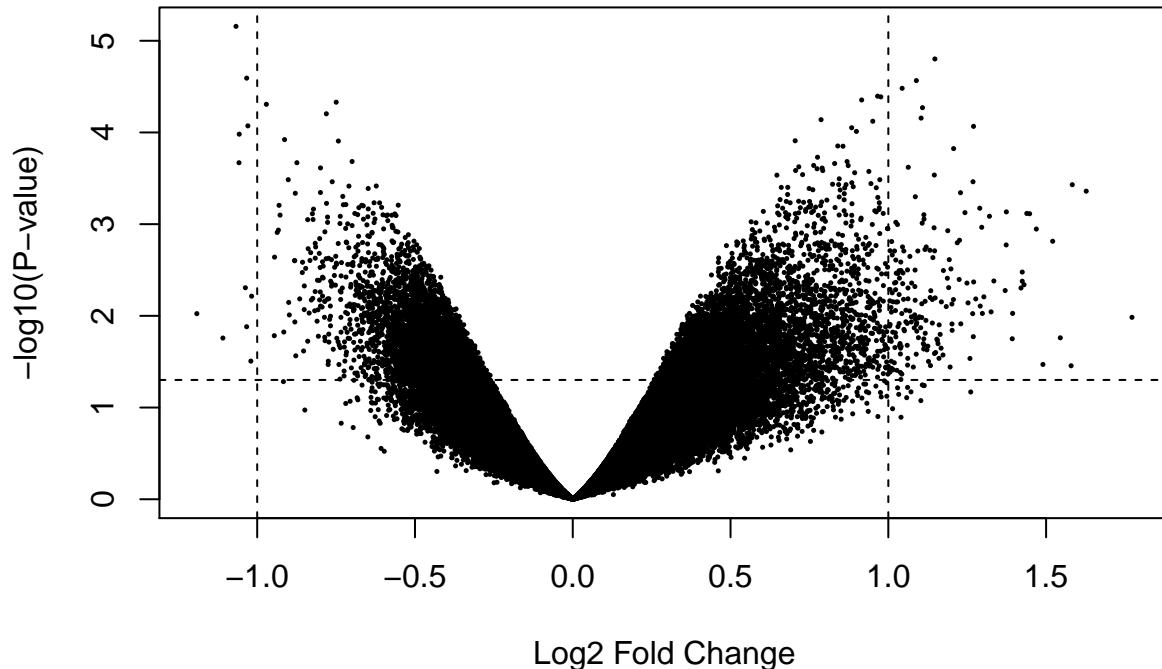
head(t_test_bayes$t)

##           Contrasts
##           Genisteina - Control
## AFFX-BkGr-GC03_st      1.1653006
## AFFX-BkGr-GC04_st      0.3238311
## AFFX-BkGr-GC05_st      0.1504270
## AFFX-BkGr-GC06_st      0.2649215
## AFFX-BkGr-GC07_st      0.1352582
## AFFX-BkGr-GC08_st      0.3830832
```

6 Resultados análisis DEG

Tras realizar el t-test moderado, se grafican los resultados en un volcano plot. El volcano plot muestra en el eje X el log₂FC entre dos grupos (*i.e.* impacto biológico del cambio) y en el eje Y mide el p-valor del cambio (*i.e.* la evidencia estadística de los resultados). En nuestro caso resaltaremos las regiones del gráfico cuyos genes tengan simultáneamente un p-valor < 0.05 y un FC > |2| (cuadrantes superiores derecho e izquierdo):

```
volcanoplot(t_test_bayes, coef = 1, highlight = 0); abline(h = 1.301, v = c(-1,1), lty = 2)
```



Nótese que el parámetro `coef` del comando `volcanoplot()` te permite seleccionar la comparación a graficar (disponible en el bolsillo `t_test_bayes$coefficients`), pero en nuestro caso sólo hemos hecho una comparación (Genisteína vs Control), por lo que lo dejamos con su valor por defecto.

Tras analizar las regiones de interés del volcano plot, vemos que hay genes candidatos a estar diferencialmente expresados. Ahora procederemos a plasmarlos en una lista.

```
tab = topTable(t_test_bayes ,coef = 1 ,number = 138745, adjust.method ="BH", sort.by = "p")
head(tab)
```

```
##          logFC AveExpr      t     P.Value adj.P.Val
## TC1900009657.hg.1 -1.067277 4.600619 -7.677840 6.963882e-06 0.6527128
## TC1100007696.hg.1  1.147524 5.124595  7.057113 1.578805e-05 0.6527128
## TC0700010496.hg.1 -1.033416 3.181483 -6.707667 2.556252e-05 0.6527128
## TC0700008020.hg.1  1.088852 3.466509  6.663894 2.718324e-05 0.6527128
## TC0100014464.hg.1  1.043817 3.929857  6.526972 3.300025e-05 0.6527128
## TC0200011114.hg.1  0.965998 5.156211  6.389172 4.021292e-05 0.6527128
##          B
## TC1900009657.hg.1  0.40901771
## TC1100007696.hg.1  0.14872590
## TC0700010496.hg.1 -0.01626027
## TC0700008020.hg.1 -0.03794775
## TC0100014464.hg.1 -0.10731770
## TC0200011114.hg.1 -0.17953818
```

```
# Elegimos los genes que caen en las regiones del volcano plot de interés
genes_diferencialmente_expresados <- which(tab$P.Value <= 0.05 & abs(tab$logFC) >= 1)
```

```

# Encontramos 204 candidatos. Recordemos que algunos de ellos puede ser ruido,
# otros serán sondas de QC (HTA-pos/neg-3371052_st...)
length(genes_diferencialmente_expresados)

## [1] 204

lista_DGE_limma <- rownames(tab[genes_diferencialmente_expresados,])

# lista_DGE_limma <- as.vector(lista_DGE_limma)
write.table(x = lista_DGE_limma, file = "lista_DGE_limma.txt", quote = F,
            row.names = F, col.names = F)

```

Tras guardar los nombres de los genes diferencialmente expresados en el archivo de texto `lista_DGE_limma.txt`, podemos realizar los análisis posteriores pertinentes (*i.e.* GEA, GSEA, dianas de miRNA...).

7 Bibliografía

- https://wiki.bits.vib.be/index.php/Analyze_your_own_microarray_data_in_R/Bioconductor
- <https://www.thermofisher.com/es/es/home/life-science/microarray-analysis/microarray-data-analysis.html>
- https://www.affymetrix.com/support/help/exon_glossary/index.jsp#clusteroverview
- <https://www.affymetrix.com/analysis/netaffx>
- <http://bioconductor.org/packages/release/bioc/vignettes/oligo/inst/doc/oug.pdf>
- R. M. Flight, A. M. Eteleeb and E. C. Rouchka, “Affymetrix® Mismatch (MM) Probes: Useful after All,” 2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom), 2012, pp. 6-13, doi: 10.1109/BioMedCom.2012.8.

8 sessionInfo()

```

## R version 3.6.3 (2020-02-29)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 19043)
##
## Matrix products: default
##
## locale:
## [1] LC_COLLATE=Spanish_Spain.1252  LC_CTYPE=Spanish_Spain.1252
## [3] LC_MONETARY=Spanish_Spain.1252 LC_NUMERIC=C
## [5] LC_TIME=Spanish_Spain.1252
##
## attached base packages:
## [1] stats4     parallel   stats      graphics   grDevices utils      datasets
## [8] methods    base
##
## other attached packages:
## [1] pd.clariom.d.human_3.14.1 DBI_1.1.1
## [3] RSQLite_2.1.4              limma_3.42.2

```

```

## [5] uwot_0.1.10                  Matrix_1.3-4
## [7] Rtsne_0.15                   rgl_0.106.8
## [9] FactoMineR_2.4               dplyr_1.0.7
## [11] ggplot2_3.3.4                oligo_1.50.0
## [13] Biostrings_2.54.0            XVector_0.26.0
## [15] IRanges_2.20.2              S4Vectors_0.24.4
## [17] Biobase_2.46.0              oligoClasses_1.48.0
## [19] BiocGenerics_0.32.0

##
## loaded via a namespace (and not attached):
## [1] bitops_1.0-7                 matrixStats_0.59.0
## [3] bit64_4.0.5                 GenomeInfoDb_1.22.1
## [5] tools_3.6.3                  utf8_1.2.1
## [7] R6_2.5.0                     DT_0.18
## [9] affyio_1.56.0                KernSmooth_2.23-20
## [11] colorspace_2.0-1            manipulateWidget_0.11.0
## [13] withr_2.4.2                 tidyselect_1.1.1
## [15] bit_4.0.4                   compiler_3.6.3
## [17] preprocessCore_1.48.0       flashClust_1.01-2
## [19] DelayedArray_0.12.3         labeling_0.4.2
## [21] scales_1.1.1                stringr_1.4.0
## [23] digest_0.6.27              rmarkdown_2.9
## [25] pkgconfig_2.0.3             htmtools_0.5.1.1
## [27] highr_0.9                   fastmap_1.1.0
## [29] htmlwidgets_1.5.3           rlang_0.4.11
## [31] FNN_1.1.3                  shiny_1.6.0
## [33] farver_2.1.0                generics_0.1.0
## [35] jsonlite_1.7.2             crosstalk_1.1.1
## [37] BiocParallel_1.20.1         RCurl_1.98-1.3
## [39] magrittr_2.0.1              GenomeInfoDbData_1.2.2
## [41] leaps_3.1                   Rcpp_1.0.6
## [43] munsell_0.5.0              fansi_0.5.0
## [45] lifecycle_1.0.0             scatterplot3d_0.3-41
## [47] stringi_1.6.2              yaml_2.2.1
## [49] MASS_7.3-54                 SummarizedExperiment_1.16.1
## [51] zlibbioc_1.32.0            grid_3.6.3
## [53] affxparser_1.58.0           blob_1.2.1
## [55] promises_1.2.0.1           ggrepel_0.9.1
## [57] crayon_1.4.1                miniUI_0.1.1.1
## [59] lattice_0.20-44             splines_3.6.3
## [61] knitr_1.33                  pillar_1.6.1
## [63] GenomicRanges_1.38.0        codetools_0.2-18
## [65] glue_1.4.2                  evaluate_0.14
## [67] BiocManager_1.30.16          vctrs_0.3.8
## [69] httpuv_1.6.1                foreach_1.5.1
## [71] gtable_0.3.0                 purrr_0.3.4
## [73] cachem_1.0.5                xfun_0.24
## [75] mime_0.10                   xtable_1.8-4
## [77] ff_2.2-0                    RSpectra_0.16-0
## [79] later_1.2.0                 tibble_3.1.2
## [81] iterators_1.0.13            memoise_2.0.0
## [83] cluster_2.1.2              ellipsis_0.3.2

```