Gene Set Enrichment Analysis with R and Bioconductor

Zuguang Gu

2022-09-01

Contents

Chapter 1

About

This is a book on gene set enrichment analysis.

Chapter 2

Introduction

...

Chapter 3

Gene Set Databases

3.1 Overview

GSEA test the significance of a list of genes of interest against a collection of pre-defined gene sets, where each gene set represent a certain type of biological attributes. Once a gene set is signfiicantly enriched, normally we made the cocludsion that the corresponding biological functions are significantly affected. Thus choose a proper type of gene sets is important to the studies. The construction of gene sets a very flexible. In most cases, they are from public databases where xx efforts have already been to categorize genes into sets. Nevertheless, the gene sets can also be self-defineds from personal studies, such as a set of genes in a network module, or xxx. In this chapter, we will introduce several major gene set databases and how to access the data in R.

3.2 Gene Ontology

Gene Ontology (GO) is perhaps the most used gene sets database in current studies. The GO project was started in 1999. It aims to construct biological functions in a well-organized form, which is both effcient for human-readable and machine-readable. It also provides annotations between well-defined biological terms to multiple species, which gives a comprehensive knowledge for functional studies.

GO has three namespaces, which are 1. Biological process, 2. cellular componet and 3. molecular function, which describe a gene or its product protein from the three aspects. biological process describes what a gene's function, which is the major namespace, cellular components describes where a gene's product has function in the cell and molecular function. Under each namespace, GO terms are organized in a hierarchical way, in a form of a directed acyclic graph (DAG). In this structure, a parent term can have multiple child terms and a child term

can have multiple parent terms, but there is no loop in the xx. Also more on the top of the DAG, more general the corresponding term is.

GO has a rich resource that in current version (2022-07-01), there are 43558 GO terms, with more tham 7 million annotations to more than 5000 species. This provides a powerful resouces for studying not only model species, but a huge number of other species, which normally other gene set databases do not support.

GO has two major parts. The first is the GO structure itself, which how to orgianize differnet biological terms, which is species-independent. The second is the GO annotation (GOA) which provides mappings between GO terms to genes or proteins for a specific species. Thus, a GO term can be linked to a set of genes, and this is called GO gene sets. One thing that should be noted is due to the DAG strucutre, a child term xx to its parent term, thus all genes annotated to a GO terms are also annotated it its parent term.

The GO DAG structure also provides useful information to xxx. In most cases when we do gene set enrichmetn analysis, we normally do not consider its GO structure, however, In Chapter xx we will discuss to visualize GO enrichment results with DAG and in chapter xx we will discuss to simplify GO terms by taking the inforamtion of DAG structures.

Bioconductor provides a nice of GO-related and reformatted resources as core packages, which is convinient to read, to process and to integrated into downstream analysis. More importantly, these annotation packages are always updated by bioconductor core teams and it is always make sure the resouces are up-to-date in each BIoconductor release.

3.2.1 The GO.db package

The first package I will introduce is the **GO.db** package. It is maintained by the Bioconductor core team and it is frequently updated in every bioc release. Thus, it always stores up-to-date data for GO and it can be treated as the standard source to read and process GO data in R. **GO.db** contains detailed information of GO terms as well as the hierarchical structure of the GO tree. The data in **GO.db** is only focused on GO terms and their relations, and it is independent to specific organisms.

GO.db is constructed by the low-level package **AnnotationDbi**, which defines a general database interface for storing and precessing biological annotation data. Internally the GO data is represented as an SQLite database with multiple tables. A large number of low-level functions defined in **AnnotationDbi** can be applied to the objects in **GO.db** to get customized filtering on GO data. Expericenced readers may go to the documentation of **AnnotationDbi** for more information.

First let's load the **GO.db** package.

```
library(GO.db)
```

Simply printing the object `GO.db` shows the basic information of the package. The two most important fields are the source file (`GOSOURCEURL`) and the date (`GOSOURCEDATE`) to build the package.

```
GO.db
# GODb object:
# | GOSOURCENAME: Gene Ontology
# | GOSOURCEURL: http://current.geneontology.org/ontology/go-basic.obo
# | GOSOURCEDATE: 2022-03-10
# | Db type: GODb
# | package: AnnotationDbi
# | DBSCHEMA: GO_DB
# | GOEGSOURCEDATE: 2022-Mar17
# | GOEGSOURCENAME: Entrez Gene
# | GOEGSOURCEURL: ftp://ftp.ncbi.nlm.nih.gov/gene/DATA
# | DBSCHEMAVERSION: 2.1
```

`GO.db` is a database object created by **AnnotationDbi**, thus, the general method `select()` can be applied to retrieve the data by specifying a vector of "keys" which is a list of GO IDs, and a group of "columns" which are the fields where the values are retrieved. In the following code, we extracted the values of `"ONTOLOGY"` and `"TERM"` for two GO IDs. The valid columns names can be obtained by `columns(GO.db)`.

```
select(GO.db, keys = c("GO:0000001", "GO:0000002"),
    columns = c("ONTOLOGY", "TERM"))
#         GOID ONTOLOGY                              TERM
# 1 GO:0000001       BP        mitochondrion inheritance
# 2 GO:0000002       BP mitochondrial genome maintenance
```

However, we don't really need to use the low-level function `select()` to retrieve the data. In **GO.db**, there are several tables that have already been compiled and can be used directly. These tables are represented as objects in the package. The following command prints all the variables that are exported in **GO.db**. Note, in the interactive R terminal, users can also see the variables by typing `GO.db::` with two continuous tabs.

```
ls(envir = asNamespace("GO.db"))
#  [1] "GO"             "GO.db"          "GOBPANCESTOR"   "GOBPCHILDREN"
#  [5] "GOBPOFFSPRING" "GOBPPARENTS"    "GOCCANCESTOR"   "GOCCCHILDREN"
#  [9] "GOCCOFFSPRING" "GOCCPARENTS"    "GOMAPCOUNTS"    "GOMFANCESTOR"
# [13] "GOMFCHILDREN"  "GOMFOFFSPRING" "GOMFPARENTS"    "GOOBSOLETE"
# [17] "GOSYNONYM"     "GOTERM"         "GO_dbInfo"      "GO_dbconn"
# [21] "GO_dbfile"     "GO_dbschema"
```

Before I introduce the GO-related objects, I first briefly introduce the following

four functions `GO_dbInfo()`, `GO_dbconn()`, `GO_dbfile()` and `GO_dbschema()` in **GO.db**. These four functions can be executed without argument. They provides information of the database, e.g., `GO_dbfile()` returns the path of the database file.

```
GO_dbfile()
# [1] "/Users/guz/Library/R/x86_64/4.2/library/GO.db/extdata/GO.sqlite"
```

If readers are interested, they can use external tools to view the `GO.sqlite` file to see the internal representation of the database tables. However, for other readers, the internal representation of data is less important. **GO.db** provides an easy interface for processing the GO data, as data frames or lists.

First let's look at the variable `GOTERM`. It contains basic information of every individual GO term, such as GO ID, the namespace and the definition.

```
GOTERM
# TERM map for GO (object of class "GOTermsAnnDbBimap")
```

`GOTERM` is an object of class `GOTermsAnnDbBimap`, which is a child class of a more general class `Bimap` defined in `AnnotationDbi`. Thus, many low-level functions defined in **AnnotationDbi** can be applied to `GOTERM` (see the documentation of `Bimap`). However, as suggested in the documentation of `GOTERM`, we do not need to directly work with these low-level classes, while we just simply convert it to a list.

```
gl = as.list(GOTERM)
```

Now `gl` is a normal list of GO terms. We can check the total number of GO terms in current version of **GO.db**.

```
length(gl)
# [1] 43705
```

We can get a single GO term by specifying the index.

```
term = gl[[1]]  ## also gl[["GO:0000001"]]
term
# GOID: GO:0000001
# Term: mitochondrion inheritance
# Ontology: BP
# Definition: The distribution of mitochondria, including the
#     mitochondrial genome, into daughter cells after mitosis or meiosis,
#     mediated by interactions between mitochondria and the cytoskeleton.
# Synonym: mitochondrial inheritance
```

The output after printing `term` includes several fields and the corresponding values. Although `gl` is a normal list, its elements are still in a special class `GOTerms` (try typing `class(term)`). There are several "getter" functions to extract values in the corresponding fields. The most important functions are

`GOID()`, `Term()`, `Ontology()`, and `Definition()`.

```
GOID(term)
# [1] "GO:0000001"
Term(term)
# [1] "mitochondrion inheritance"
Ontology(term)
# [1] "BP"
Definition(term)
# [1] "The distribution of mitochondria, including the mitochondrial genome, into daughter cells
```

So, to get the namespaces of all GO terms, we can do:

```
sapply(gl, Ontology)
```

Besides being applied to the single GO terms, these "getter" functions can be directly applied to the global object `GOTERM` to extract information of all GO terms simultaneously.

```
head(GOID(GOTERM))
#   GO:0000001   GO:0000002   GO:0000003   GO:0000006
# "GO:0000001" "GO:0000002" "GO:0000003" "GO:0000006"
```

Now let's go back to the object `GOTERM`. It stores data for all GO terms, so essentially it is a list of elements in a certain format. `GOTERM` or its class `GOTermsAnnDbBimap` allows subsetting to obtain a subset of GO terms. There are two types of subset operators: single bracket `[` and double brackets `[[` and they have different behaviors.

Similar as the subset operators for list, the single bracket `[` returns a subset of data but still keeps the original format. Both numeric and character indices are allowed, but more often, character indices as GO IDs are used.

```
# note you can also use numeric indices
GOTERM[c("GO:0000001", "GO:0000002", "GO:0000003")]
# TERM submap for GO (object of class "GOTermsAnnDbBimap")
```

The double-bracket operator `[[` is different. It degenerates the original format and directly extracts the element in it. Note here only a single character index is allows:

```
# note numeric index is not allowed
GOTERM[["GO:0000001"]]
# GOID: GO:0000001
# Term: mitochondrion inheritance
# Ontology: BP
# Definition: The distribution of mitochondria, including the
#     mitochondrial genome, into daughter cells after mitosis or meiosis,
#     mediated by interactions between mitochondria and the cytoskeleton.
# Synonym: mitochondrial inheritance
```

Directly applying `Ontology()` on `GOTERM`, it is easy to count number of GO terms in each namespace.

```
table(Ontology(GOTERM))
#
#        BP        CC        MF universal
#     28336      4183     11185         1
```

Interestingly, besides the three namespaces `"BP"`, `"CC"` and `"MF"`, there is an additionally namespace `"universal"` that only contains one term. As mentioned before, the three GO namespaces are isolateted. However, some tools may require all GO terms are connected if the relations are represented as a graph. Thus, one pseudo "universal root term" is added as the parent of root nodes in the three namespaces. In **GO.db**, this special term has an ID `"all"`.

```
which(Ontology(GOTERM) == "universal")
#   all
# 43705
GOTERM[["all"]]
# GOID: all
# Term: all
# Ontology: universal
# Definition: NA
```

GO.db also provides variables that contains the relations between GO terms. Take biological process namespace as an example, there are the following four variables (similar for other two namespaces, but with `GOCC` and `GOMF` prefix).

- `GOBPCHILDREN`
- `GOBPPARENTS`
- `GOBPOFFSPRING`
- `GOBPANCESTOR`

`GOBPCHILDREN` and `GOBPPARENTS` contain parent-child relations. `GOBPOFFSPRING` contains all offspring terms of GO terms (i.e., all downstream terms of a term in the GO tree) and `GOBPANCESTOR` contais all ancestor terms of a GO term (i.e. all upstream terms of a term). The information in the four variables are actually redudant, e.g., all the other three objects can be constructed from `GOBPCHILDREN`. However, these pre-computated objects and it will save time in downstream analysis.

The four variables are in the same format (objects of the `AnnDbBimap` class). Taking `GOBPCHILDREN` as an example, we directly convert it to a list.

```
lt = as.list(GOBPCHILDREN)
head(lt)
# $`GO:0000001`
# [1] NA
#
```

```
# $`GO:0000002`
#      part of
# "GO:0032042"
#
# $`GO:0000003`
#          isa          isa      part of          isa          isa          isa
# "GO:0019953" "GO:0019954" "GO:0022414" "GO:0032504" "GO:0032505" "GO:0075325"
#          isa
# "GO:1990777"
#
# $`GO:0000011`
# [1] NA
```

`lt` is a simple list of vectors where each vector are child terms of a specific GO term, e.g., `GO:0000002` has a child term `GO:0032042`. The element vectors in `lt` are also named and the names represent the relation of the child term to the parent term. When the element vector has a value `NA`, e.g. `GO::0000001`, this means the GO term is a leaf in the GO tree, and it has no child term.

Some downstream analysis, e.g., network analysis, may expect the relations to be represented as two columns. In this case, we can use the general function `toTable()` defined in **AnnotationDbi** to convert `GOBPCHILDREN` to a data frame.

```
tb = toTable(GOBPCHILDREN)
head(tb)
#        go_id      go_id RelationshipType
# 1 GO:0032042 GO:0000002          part of
# 2 GO:0019953 GO:0000003              isa
# 3 GO:0019954 GO:0000003              isa
# 4 GO:0022414 GO:0000003          part of
```

Unfortunately, the first two columns in `tb` have the same name. A good idea is to add meaningful column names to it.

```
colnames(tb)[1:2] = c("child", "parent")
```

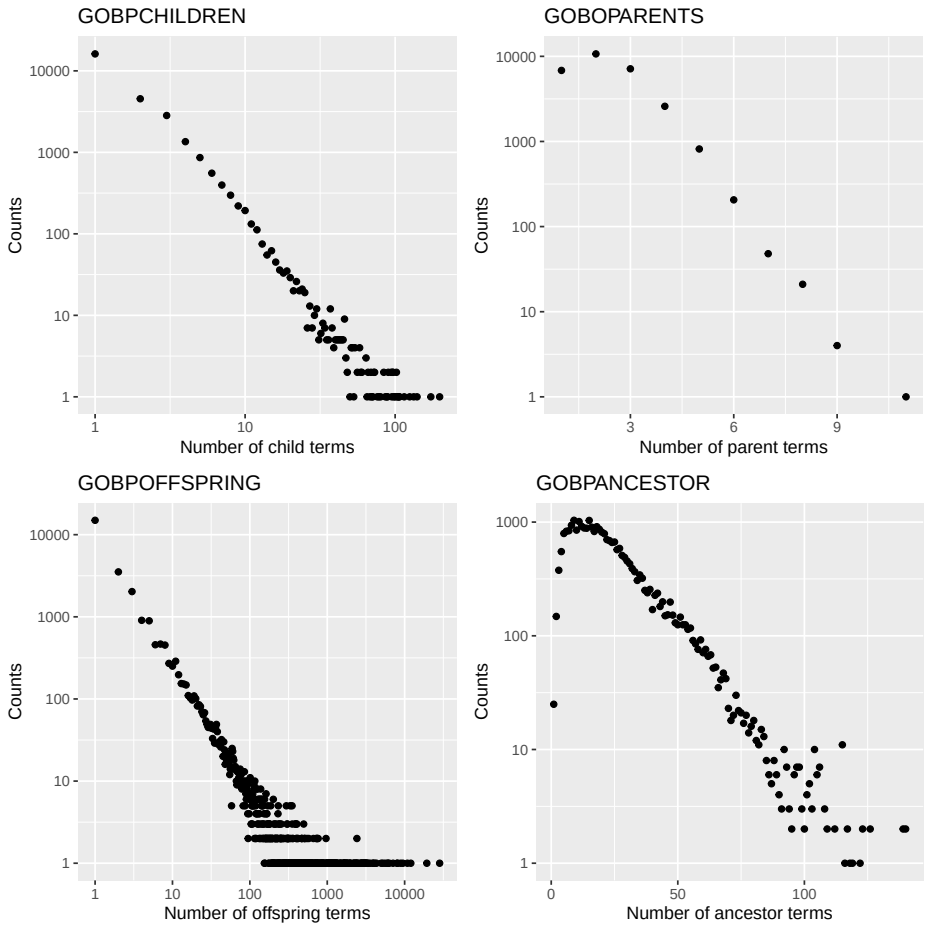
With `tb`, we can calculate the fraction of different relations of GO terms.

```
tb = toTable(GOBPCHILDREN)
table(tb[, 3])
#
#                  isa negatively regulates              part of
#                52089                 2753                 5048
# positively regulates            regulates
#                 2742                 3197
```

Note, only `GOBPPARENTS` and `GOBPANCESTOR` contain the universal root term

"all".

```
lt = as.list(GOBPPARENTS)
lt[["GO:0008150"]]  # GO:0008150 is "biological process"
#   isa
# "all"
```



Relations in `GOBPCHILDREN` or `GOBPPARENTS` can be used to construct the GO graph. In the remaining part of this section, I will demonstrate some explorary analysis to show interesting attributes of the GO graph.

Remember `toTable()` returns the relations between GO terms as a data frame, thus, it can be used as "edge list" (or adjacency list). In the following code, we use **igraph** package for network analysis. The function `graph.edgelist()` construct a graph object from a two-column matrix where the first column is the source of the link and the second column is the target of the link.

```
library(igraph)
tb = toTable(GOBPCHILDREN)
g = graph.edgelist(as.matrix(tb[, 2:1]))
```

We can extract GO term with the highest in-degree. This is the term with the largest number of parents. This value can also be get from `GOBPPARENTS`. Please note `which.max()` only returns one index of the max value, but it does not mean it is the only max value.

```
d = degree(g, mode = "in")
d_in = d[which.max(d)]
d_in
# GO:0106110
#         11
GOTERM[[ names(d_in) ]]
# GOID: GO:0106110
# Term: vomitoxin biosynthetic process
# Ontology: BP
# Definition: The chemical reactions and pathways resulting in the
#     formation of type B trichothecene vomitoxin, also known as
#     deoxynivalenol, a poisonous substance produced by some species of
#     fungi and predominantly occurs in grains such as wheat, barley and
#     oats.
# Synonym: deoxynivalenol biosynthetic process
# Synonym: DON biosynthetic process
# Synonym: vomitoxin anabolism
# Synonym: vomitoxin biosynthesis
# Synonym: vomitoxin formation
# Synonym: vomitoxin synthesis
```

We can calculate GO term with the highest out-degree. This is the term with the largest number of children. This value can also be get from `GOBPCHILDREN`.

```
d = degree(g, mode = "out")
d_out = d[which.max(d)]
d_out
# GO:0048856
#        198
GOTERM[[ names(d_out) ]]
# GOID: GO:0048856
# Term: anatomical structure development
# Ontology: BP
# Definition: The biological process whose specific outcome is the
#     progression of an anatomical structure from an initial condition to
#     its mature state. This process begins with the formation of the
#     structure and ends with the mature structure, whatever form that
```

```
#     may be including its natural destruction. An anatomical structure
#     is any biological entity that occupies space and is distinguished
#     from its surroundings. Anatomical structures can be macroscopic
#     such as a carpel, or microscopic such as an acrosome.
# Synonym: development of an anatomical structure
```

We can explore some long-disatnce attributes, such as the distance from the root term to every term in the namespace. The distance can be thought as the depth of a term in the GO tree.

```
dist = distances(g, v = "GO:0008150", mode = "out")
table(dist)
# dist
#    0    1    2    3    4    5    6    7    8    9   10   11
#    1   30  564 3601 7498 7992 4962 2414 1021  208   40    5
```

We can also extract the longest path from the root term to the leaf terms.

```
d = degree(g, mode = "out")
leave = names(d[d == 0])
sp = shortest_paths(g, from = "GO:0008150", to = leave, mode = "out")
i = which.max(sapply(sp$vpath, length))
path = sp$vpath[[i]]
Term(GOTERM[names(path)])
#
#                                                                                "biolo
#
#                                                                                 "meta
#
#                                                                  "positive regulation of meta
#
#                                                              "positive regulation of cellular meta
#
#                                                     "positive regulation of cellular protein meta
#
#                                                                        "positive regulation o
#
#                                                                  "positive regulation of pepti
#
#                                                              "positive regulation of endopepti
#
#                                                     "positive regulation of cysteine-type endopepti
#
#        "positive regulation of cysteine-type endopeptidase activity involved in apop
#
#                  "activation of cysteine-type endopeptidase activity involved in apop
```

```
#                                                                                                    GO:0008
# "activation of cysteine-type endopeptidase activity involved in apoptotic process by cytochrome
```

3.2.2 Link GO terms to genes

As introduced in the previous section, **GO.db** only contains data for GO terms. GO also provides gene annotated to GO terms, by manual curation or computation prediction. Such annotations are represented as mapping between GO IDs and gene IDs from external databases, which are usually synchronized between major public databases such NCBI. To obtains genes in each GO term in R, Bioconductor provides a family of packages with name **org.*.db**. Let's take human for example, the corresponding package is **org.Hs.eg.db**. **org.Hs.eg.db** which is a standard way to contains mappings from Entrez gene IDs to a variaty of other databases.

```
library(org.Hs.eg.db)
```

In this package, there are two database table objects for mapping between GO IDs and genes:

- `org.Hs.egGO2EG`
- `org.Hs.egGO2ALLEGS`

The difference between the two objects is `org.Hs.egGO2EG` contains genes that are *directly annotated* to every GO term, while `org.Hs.egGO2ALLEGS` contains genes that directly assigned to the GO term, *as well as* genes assigned to all its offspring GO terms.

Again, `org.Hs.egGO2ALLEGS` is a database object. There are two ways to obtain gene annotations to GO terms. The first is to convert `org.Hs.egGO2ALLEGS` to a list of gene vectors.

```
lt = as.list(org.Hs.egGO2ALLEGS)
lt[3:4]
# $`GO:0000012`
#         IGI         IDA         IDA         IDA         NAS         IDA
#       "142"      "1161"      "2074"      "3981"      "7014"      "7141"
#         IEA         IGI         IMP         IMP         IBA         IDA
#      "7515"      "7515"      "7515"     "23411"     "54840"     "54840"
#         IBA         IDA         IMP         IMP         IEA
#     "55775"     "55775"     "55775"    "200558" "100133315"
#
# $`GO:0000017`
#    IDA    IMP    ISS    IDA
# "6523" "6523" "6523" "6524"
```

In this case, each element vector actually is a GO gene set. Note here the genes are in Entrez IDs, which are digits but in character mode. Node this is important

to save the Gene IDs explicitely as characters to get rid of potential error due to indexing. We will emphasize it again in later text.

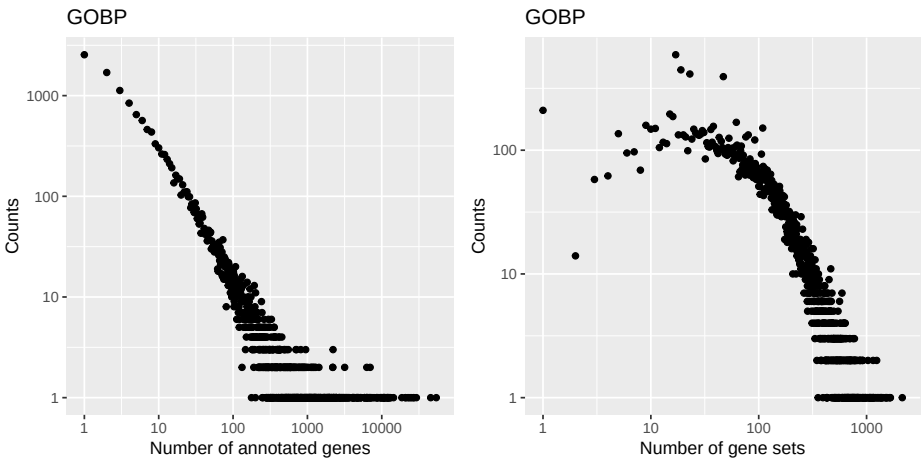
Also the gene vector has names which are teh evidence to

`org.Hs.egGO2ALLEGS` is a database object, thus `toTable()` can be directly applied to convert it as a table.

```
tb = toTable(org.Hs.egGO2ALLEGS)
head(tb)
#   gene_id      go_id Evidence Ontology
# 1       1 GO:0008150       ND       BP
# 2       2 GO:0000003      IEA       BP
# 3       2 GO:0001553      IEA       BP
# 4       2 GO:0001867      IDA       BP
```

Now there is an additional column `"Ontology"`.

With `tb`, we can look at the distribution of number of genes in GO gene sets.



Biocondutor core team maintaines **org.*.db** for 18 organisms

Package	Organism	Package	Organism
'org.Hs.eg.db'	Human	'org.Ss.eg.db'	Pig
'org.Mm.eg.db'	Mouse	'org.Gg.eg.db'	Chicken
'org.Rn.eg.db'	Rat	'org.Mmu.eg.db'	Rhesus monkey
'org.Dm.eg.db'	Fruit fly	'org.Cf.eg.db'	Canine
'org.At.tair.db'	Arabidopsis	'org.EcK12.eg.db'	E coli strain K12
'org.Sc.sgd.db'	Yeast	'org.Xl.eg.db'	African clawed frog
'org.Dr.eg.db'	Zebrafish	'org.Ag.eg.db'	Malaria mosquito
'org.Ce.eg.db'	Nematode	'org.Pt.eg.db'	Chimpanzee
'org.Bt.eg.db'	Bovine	'org.EcSakai.eg.db'	E coli strain Sakai

The **org.*.db** provided by Bioconductor should be enough for most of the analysis. However, there may be users who mainly work on non-model organism or microbomens where there is no such pre-compiled package on Bioconductor. In this case, they may use the **biomaRt** package to ...

The BioMart (https://www.ensembl.org/info/data/biomart) is a web-based sevice from Ensembl database which can be used to extract a huge umber of xxx. The companion R package **biomaRt** provides a programatical interface to directly access BioMart purely in R. Using **biomaRt** to extract GO gene sets is complex and I will demonstrate step by step.

1. Connect to a mart (database).

```
library(biomaRt)
ensembl = useEnsembl(biomart = "genes")
```

By default, it uses the newest mart... To use

```
listEnsemblGenomes()
```

```
ensembl_fungi = useEnsemblGenomes("fungi_mart")
```

2. select a dataset for a specific organism. To find a valid value of dataset, `listDatasets(ensembl)`. The dataset is in the first column and should have suffix of `"_eg_gene"` or `"_gene_ensembl"`. In the following example, we choose the dataset `"amelanoleuca_gene_ensembl"` which is giant panda.

```
dataset = "amelanoleuca_gene_ensembl"
ensembl = useDataset(dataset = dataset, mart = ensembl)
```

3. get all genes for the organism. Having configured the xxx, we use the core function `getBM()` to get teh xxx Again, we need a proper value for the `attributes` argument. `listAttributes(ensembl)`. It returns a long table, users can need to filter xxx.

```
tb_go = getBM(attributes = c("ensembl_gene_id", "go_id", "namespace_1003"),
    mart = ensembl)
```

4. merge xxx

```
tb = tb_go[tb_go$namespace_1003 == "biological_process", , drop = FALSE]
gs = split(tb$ensembl_gene_id, tb$go_id)

bp_terms = keys(GOBPOFFSPRING)
gs2 = lapply(bp_terms, function(nm) {
    go_id = c(nm, GOBPOFFSPRING[[nm]])
    unique(unlist(gs[go_id]))
})
names(gs2) = bp_terms
```

```
gs2 = gs2[sapply(gs2, length) > 0]
```

For some organisms, the table might be huge. Since the data is retrieved from Ensembl server via internet, large table might be corrupted during data transfering. A safer way is to split genes into blocks and downloaded the annotation xxx. Readers can take it as an excise .

```
library(rGREAT)
lt = getGeneSetsFromBioMart("amelanoleuca_gene_ensembl", "bp")
```

3.3 KEGG pathways

Kyoto Encyclopedia of Genes and Genomes (KEGG) is a comprehensive database of genomic and molecular data for variaty of organisms. Its sub-database the pathway database is a widely used gene set database used in current studies. In KEGG, pathways are manually curated and number of genes in pathways are intermediate

KEGG provides its data via a REST API (https://rest.kegg.jp/). There are several commands that can be used to retrieve specific types of data. For example, to get the mapping between human genes and KEGG pathways, users can use the `link` command:

```
df1 = read.table(url("https://rest.kegg.jp/link/pathway/hsa"),
    sep = "\t")
head(df1)
#          V1             V2
# 1 hsa:10327 path:hsa00010
# 2   hsa:124 path:hsa00010
# 3   hsa:125 path:hsa00010
# 4   hsa:126 path:hsa00010
```

In the example, `url()` construct a connection object that directly transfer data from the remote URL. In output, the first column contains Entrez ID (users may remove the hsa: prefix for downstream analysis) and the second columncontains KEGG pathways IDs (users may remove the "path:" previx).

To get the full name of pathways, use the `list` command:

```
df2 = read.table(url("https://rest.kegg.jp/list/pathway/hsa"),
    sep = "\t")
head(df2)
#              V1                                                          V2
# 1 path:hsa00010              Glycolysis / Gluconeogenesis - Homo sapiens (human)
# 2 path:hsa00020                 Citrate cycle (TCA cycle) - Homo sapiens (human)
# 3 path:hsa00030                 Pentose phosphate pathway - Homo sapiens (human)
# 4 path:hsa00040 Pentose and glucuronate interconversions - Homo sapiens (human)
```

There are two Bioconductor packages for retrieving pathway data from KEGG. Both of them are based on KEGG REST API. The first one is the package **KEGGREST** which implements a full interface to access KEGG data in R. All the API from KEGG REST service are suppoted in **KEGGREST**. For example, to get the mapping between genes and pathways, the function `keggLink()` can be used.

```
library(KEGGREST)
pathway2gene = keggLink("pathway", "hsa")
head(pathway2gene)
#       hsa:10327         hsa:124         hsa:125         hsa:126
# "path:hsa00010" "path:hsa00010" "path:hsa00010" "path:hsa00010"
```

The returned object `pathway2gene` is a named vector, where the names corresponding to the source and the values correspond to the target. Readers can try to execute `keggLink("hsa", "pathway")` to compare the results.

The named vectors are not common for downstream gene set analysis. A more used format is a data frame. We can simply converted them as:

```
p2g_df = data.frame(gene_id = gsub("hsa:", "", names(pathway2gene)),
    pathway_id = gsub("path:", "", pathway2gene))
head(p2g_df)
#   gene_id pathway_id
# 1   10327   hsa00010
# 2     124   hsa00010
# 3     125   hsa00010
# 4     126   hsa00010
```

In the pathway ID, the prefix in letters corresponds to the organism, e.g. `hsa` for human. Users can go to the KEGG website to find the prefix for their organisms. Programmitically, `keggList("organims")` can be used which lists all supported organisms and their prefix in KEGG.

Last but not the least, another useful function in **KEGGREST** is `keggGet()` which implements the `get` command from REST API. With this function users can download images and KGML of pathways.

```
keggGet("hsa", "image")
keggGet("hsa", "kgml")
```

However, the conf file is not supported by `keggGet()`. Users need to directly read from the URL

```
read.table(url("https://rest.kegg.jp/get/hsa05130/conf"),
    sep = "\t")
```

The conf file contains coordinate of genes or nodes in the image. It is useful if users want to highlght genes in the image.

The second Bioconductor pacakge **clusterProfiler** has a simple function `download_KEGG()` which accepts the prefix of a organism and returns a list of two data frames, one for the mapping between genes and pathways and the other for the full name of pathways.

```
lt = clusterProfiler::download_KEGG("hsa")
head(lt$KEGGPATHID2EXTID)
#       from    to
# 1 hsa00010 10327
# 2 hsa00010   124
# 3 hsa00010   125
# 4 hsa00010   126
head(lt$KEGGPATHID2NAME)
#       from                                      to
# 1 hsa00010             Glycolysis / Gluconeogenesis
# 2 hsa00020                Citrate cycle (TCA cycle)
# 3 hsa00030                Pentose phosphate pathway
# 4 hsa00040 Pentose and glucuronate interconversions
```

The two packages mentioned above do not provide data for teh network representation of pathways. In Chapter I will demonstrate how to read and process pathways as networks. Here we simply treat pathways as lists of genes and we ignore the relations of genes .

3.4 Reactome pathways

Reactome is another popular pathway database. It organise pathways in an hierarchical category, which contains pathways and sub pathways or pathway components. Currently there are xx pathways. The up-to-date pathway data can be direclty found at https://reactome.org/download-data.

There is a **reactome.db** on Bioconductor. Similar as other annotation packages. Users can type `reactome.db::` with two continuous tabs to see the objects supported in the package. In it, the important objects are

- `reactomePATHID2EXTID` contains mappings between reacotme pathway IDs and gene entrez IDs
- `reactomePATHID2NAME` contains pathway names

```
library(reactome.db)
tb = toTable(reactomePATHID2EXTID)
head(tb)
#          DB_ID gene_id
# 1 R-HSA-109582       1
# 2 R-HSA-114608       1
# 3 R-HSA-168249       1
# 4 R-HSA-168256       1
```

```
p2n = toTable(reactomePATHID2NAME)
head(p2n)
#           DB_ID
# 1   R-BTA-73843
# 2 R-BTA-1971475
# 3 R-BTA-1369062
# 4  R-BTA-382556
#                                                                 path_name
# 1                                          1-diphosphate: 5-Phosphoribose
# 2 Bos taurus: A tetrasaccharide linker sequence is required for GAG synthesis
# 3                          Bos taurus: ABC transporters in lipid homeostasis
# 4                         Bos taurus: ABC-family proteins mediated transport
```

In the previous code, we use the function `toTable()` to retrieve the data as data frames. Readers may try `as.list()` on the two objects and compare the output.

Reactome also contains pathway for multiple organisms. In the reactome ID, teh second section contains the organism, e.g. in previous output `HSA`.

```
table( gsub("^R-(\\w+)-\\d+$", "\\1", p2n[, 1]) )
#
#  BTA  CEL  CFA  DDI  DME  DRE  GGA  HSA  MMU  MTU  PFA  RNO  SCE  SPO  SSC  XTR
# 1691 1299 1653  979 1473 1671 1703 2585 1710   13  595 1697  809  816 1656 1576
```

Again, **reactome.db** only contains pathways as list of genes.

3.5 MSigDB

Molecular signature xxx is a manually

```
library(msigdbr)
```

Which species are supported. Please note MSigDB only provides gene sets for human, while **msigdbr** supports more species by annotating the homologous genes.

```
msigdbr_species()
# # A tibble: 20 x 2
#    species_name                species_common_name
#    <chr>                       <chr>
#  1 Anolis carolinensis         Carolina anole, green anole
#  2 Bos taurus                  bovine, cattle, cow, dairy cow, domestic cat~
#  3 Caenorhabditis elegans      <NA>
#  4 Canis lupus familiaris      dog, dogs
#  5 Danio rerio                 leopard danio, zebra danio, zebra fish, zebr~
#  6 Drosophila melanogaster     fruit fly
#  7 Equus caballus              domestic horse, equine, horse
```

```
#  8 Felis catus                     cat, cats, domestic cat
#  9 Gallus gallus                   bantam, chicken, chickens, Gallus domesticus
# 10 Homo sapiens                    human
# 11 Macaca mulatta                  rhesus macaque, rhesus macaques, Rhesus monk~
# 12 Monodelphis domestica           gray short-tailed opossum
# 13 Mus musculus                    house mouse, mouse
# 14 Ornithorhynchus anatinus        duck-billed platypus, duckbill platypus, pla~
# 15 Pan troglodytes                 chimpanzee
# 16 Rattus norvegicus               brown rat, Norway rat, rat, rats
# 17 Saccharomyces cerevisiae        baker's yeast, brewer's yeast, S. cerevisiae
# 18 Schizosaccharomyces pombe 972h- <NA>
# 19 Sus scrofa                      pig, pigs, swine, wild boar
# 20 Xenopus tropicalis              tropical clawed frog, western clawed frog
```

To obtain all gene sets:

```
all_gene_sets = msigdbr() # by default it is human
dim(all_gene_sets)
# [1] 4331807      15
```

```
head(all_gene_sets)
# # A tibble: 4 x 15
#   gs_cat gs_subcat gs_name gene_~1 entre~2 ensem~3 human~4 human~5 human~6 gs_id
#   <chr>  <chr>     <chr>   <chr>     <int> <chr>   <chr>     <int> <chr>   <chr>
# 1 C3     MIR:MIR_~ AAACCA~ ABCC4     10257 ENSG00~ ABCC4     10257 ENSG00~ M126~
# 2 C3     MIR:MIR_~ AAACCA~ ABRAXA~   23172 ENSG00~ ABRAXA~   23172 ENSG00~ M126~
# 3 C3     MIR:MIR_~ AAACCA~ ACTN4        81 ENSG00~ ACTN4        81 ENSG00~ M126~
# 4 C3     MIR:MIR_~ AAACCA~ ACTN4        81 ENSG00~ ACTN4        81 ENSG00~ M126~
# # ... with 5 more variables: gs_pmid <chr>, gs_geoid <chr>,
# #   gs_exact_source <chr>, gs_url <chr>, gs_description <chr>, and abbreviated
# #   variable names 1: gene_symbol, 2: entrez_gene, 3: ensembl_gene,
# #   4: human_gene_symbol, 5: human_entrez_gene, 6: human_ensembl_gene
# # i Use `colnames()` to see all variable names
```

```
as.data.frame(head(all_gene_sets))
#   gs_cat      gs_subcat        gs_name gene_symbol entrez_gene    ensembl_gene
# 1     C3 MIR:MIR_Legacy AAACCAC_MIR140       ABCC4       10257 ENSG00000125257
# 2     C3 MIR:MIR_Legacy AAACCAC_MIR140    ABRAXAS2       23172 ENSG00000165660
# 3     C3 MIR:MIR_Legacy AAACCAC_MIR140       ACTN4          81 ENSG00000130402
# 4     C3 MIR:MIR_Legacy AAACCAC_MIR140       ACTN4          81 ENSG00000282844
#   human_gene_symbol human_entrez_gene human_ensembl_gene  gs_id gs_pmid
# 1             ABCC4             10257    ENSG00000125257 M12609
# 2          ABRAXAS2             23172    ENSG00000165660 M12609
# 3             ACTN4                81    ENSG00000130402 M12609
# 4             ACTN4                81    ENSG00000282844 M12609
#   gs_geoid gs_exact_source gs_url
```

```
# 1
# 2
# 3
# 4
#                                                                                gs_description
# 1 Genes having at least one occurence of the motif AAACCAC in their 3' untransla..
# 2 Genes having at least one occurence of the motif AAACCAC in their 3' untransla..
# 3 Genes having at least one occurence of the motif AAACCAC in their 3' untransla..
# 4 Genes having at least one occurence of the motif AAACCAC in their 3' untransla..
```

All categories of gene sets:

```
as.data.frame(msigdbr_collections())
#    gs_cat       gs_subcat num_genesets
# 1      C1                         299
# 2      C2             CGP        3384
# 3      C2              CP          29
# 4      C2     CP:BIOCARTA         292
# 5      C2         CP:KEGG         186
# 6      C2          CP:PID         196
# 7      C2     CP:REACTOME        1615
# 8      C2 CP:WIKIPATHWAYS         664
# 9      C3       MIR:MIRDB        2377
# 10     C3  MIR:MIR_Legacy         221
# 11     C3        TFT:GTRD         518
# 12     C3  TFT:TFT_Legacy         610
# 13     C4             CGN         427
# 14     C4              CM         431
# 15     C5           GO:BP        7658
# 16     C5           GO:CC        1006
# 17     C5           GO:MF        1738
# 18     C5             HPO        5071
# 19     C6                         189
# 20     C7     IMMUNESIGDB        4872
# 21     C7             VAX         347
# 22     C8                         700
# 23      H                          50
```

E.g., we want to extract genesets in C2 category and CP:KEGG sub-category:

```
gene_sets = msigdbr(category = "C2", subcategory = "CP:KEGG")
gene_sets
# # A tibble: 16,283 x 15
#    gs_cat gs_sub~1 gs_name gene_~2 entre~3 ensem~4 human~5 human~6 human~7 gs_id
#    <chr>  <chr>    <chr>   <chr>     <int> <chr>   <chr>     <int> <chr>   <chr>
#  1 C2     CP:KEGG  KEGG_A~ ABCA1        19 ENSG00~ ABCA1        19 ENSG00~ M119~
#  2 C2     CP:KEGG  KEGG_A~ ABCA10    10349 ENSG00~ ABCA10    10349 ENSG00~ M119~
```

```
#  3 C2     CP:KEGG  KEGG_A~ ABCA12    26154 ENSG00~ ABCA12    26154 ENSG00~ M119~
#  4 C2     CP:KEGG  KEGG_A~ ABCA13   154664 ENSG00~ ABCA13   154664 ENSG00~ M119~
#  5 C2     CP:KEGG  KEGG_A~ ABCA2        20 ENSG00~ ABCA2        20 ENSG00~ M119~
#  6 C2     CP:KEGG  KEGG_A~ ABCA3        21 ENSG00~ ABCA3        21 ENSG00~ M119~
#  7 C2     CP:KEGG  KEGG_A~ ABCA4        24 ENSG00~ ABCA4        24 ENSG00~ M119~
#  8 C2     CP:KEGG  KEGG_A~ ABCA5     23461 ENSG00~ ABCA5     23461 ENSG00~ M119~
#  9 C2     CP:KEGG  KEGG_A~ ABCA6     23460 ENSG00~ ABCA6     23460 ENSG00~ M119~
# 10 C2     CP:KEGG  KEGG_A~ ABCA7     10347 ENSG00~ ABCA7     10347 ENSG00~ M119~
# # ... with 16,273 more rows, 5 more variables: gs_pmid <chr>, gs_geoid <chr>,
# #   gs_exact_source <chr>, gs_url <chr>, gs_description <chr>, and abbreviated
# #   variable names 1: gs_subcat, 2: gene_symbol, 3: entrez_gene,
# #   4: ensembl_gene, 5: human_gene_symbol, 6: human_entrez_gene,
# #   7: human_ensembl_gene
# # i Use `print(n = ...)` to see more rows, and `colnames()` to see all variable name
```

3.6 UniProt keywords

```
library(UniProtKeywords)
gl = load_keyword_genesets(9606)
```

3.7 Other databases

Other pathway databases, ...

- DO
- Mesh
- pathway common
- other ...

3.8 Gene ID map

To perform gene set enrichment analysis, there are always two sources of data, one from genes and one from gene sets. Thus, the gene IDs in the two sources may not be the same. Thus

Thanks to the Bioconductor annotation ecosystem, the **org.*.eg.db** family packages provide annotation sources of genes from various databases, take **org.Hs.eg.db** package as an example

The second widely used package for gene ID coversion is **biomart** package which uses the service from ensemble biomart service.

There are also helper functions implemented in packages using xxx. `bitr()` uses org.db packages for gene ID coversion. it returns a two-column data from for the

mapping. However, users need to set the type of , the values can be obtained by `columns(org.Hs.eg.db)`

In the comparion package of this book, we have implemented a function `convert_to_entrez_id()`, which automatically recognize the input gene ID type and convert to Entrez ID since almost all gene set database provide genes in Entrez IDs. The function will be used through this book. It can recognize the following input types:

1. a character vector
2. a numeric vector with gene names as names
3. a matrix with gene names as row names.

For the last two scenarios, if multple genes are mapped to a single Entrez ID, the corresponding elements rows are averaged.

3.9 Data structure

Currently there is no widely-accepted standard format for general gene set representation. As has been demonstrated, gene sets can be represetnted as a two-column data frame where each row contains a gene in a gene set. ALternatively, which can be easily converted from teh two-column data frame is as a list where each element in the list is a vector of genes.

In the early days of bioconductor when dealling with microarray data, there is a **GSEAbase** package from bioconductor core team, which xxx.

BiocSet

Chapter 4

Over-Representation Analysis

4.1 Overview

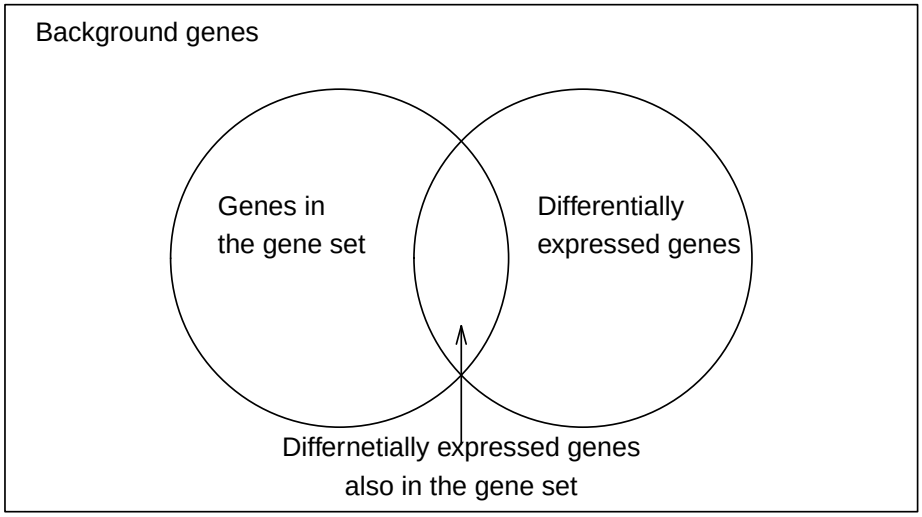
Over-representation analysis (ORA) uses a simplified model for gene set enrichment anlaysis. It directly works on the counts of genes in different categories, i.e., whether genes are in the list of interest and whether genes are in the gene set. Because of its simplicity, ORA is the mostly used method for gene set enrichment analysis. In this chapter, we will introduce different tests for ORA and the implementations in R. We will also point out the limitations of ORA of which users need to be careful when they apply ORA on their datasets.

4.2 What is over-representation?

In many cases, ORA is applied to a list of differentially expressed genes, thus, to make readers read this chapter more naturally, we use "differentially expressed genes" to represent the list of genes of interest. but please keep in mind that the differential genes is just a special case of the gene list. In applications, it can any kind of gene lists.

For a gene denoted as G, it has two attributes: whether it is differentially expressed (DE) and whether it belongs to a gene set. The two relations can be represented in a Venn Diagram (Figure @ref(fig:ora_venn)). We denote p_1 as the probability of G being differentially expressed, and p_2 as the probability of G being in the gene set, then we have

$$p_1 = n_{\text{diff}}/n$$
$$p_2 = n_{\text{set}}/n$$



where n_{diff} is the number of DE genes, n_{set} is the number of genes in the gene set, and n is the total number of genes in the study. Assume the two attributes of G are independent, i.e., whether genes are DE has no preference on whether genes are in the gene set, then the expected number of DE genes also in the gene set is np_1p_2. Denote the observed number of DE genes in the gene set as m, we can calculate a ratio $r = m/np_1p_2$. If there are more observed genes in the gene set than expected, i.e., $r > 1$, we say DE genes are over-represented in the gene set, or identically, we can say genes in the gene set are over-represented in DE genes. And when $r < 1$, we say DE genes are down-represented in the gene set.

A quick calculation shows

$$r = \frac{m}{np_1p_2} = \frac{mn}{n_{\text{diff}}n_{\text{set}}}$$

We can look at the problem from a slightly different aspect. We treat p_1 which was previously defined as the "background probability" of a gene being DE. We also calculate the probability of a genes being DE but only in the gene set, which we term it as "foreground probability" and denote it as p_1^{fore}:

$$p_1^{\text{fore}} = m/n_{\text{set}}$$

Then if the foreground probability is higher than background probability, i.e., $p_1^{\text{fore}} > p_1$ or $p_1^{\text{fore}}/p_1 > 1$, we can say DE genes are over-represented in the gene set. It is easy to see

$$\frac{P_1'}{P_1} = \frac{mn}{n_{\text{diff}}n_{\text{set}}}$$

.

Similarly, we can treat p_2 as the "background probability" of a genes being in the gene set, and we calculate the "foreground probability" of a gene being in the gene set, but only in the DE genes, denoted as p_2^{fore}:

$$p_2^{\text{fore}} = m/n_{\text{diff}}$$

It is easy to see p_2^{fore}/p_2 is identical to p_1^{fore}/p_1

The score r can be used to measures whether DE genes are over-represented in a gene set, where a higher value of r implies there is stronger over-representation. Under the statistical procedures, we need to statistical test to calculate a p-value for the over-representation to assign a "signficance level" for the enrichment. Although r is able to measure the over-representation, its distribution with analytical form is hard to obtain. In the next section, we introduce the specific distributions or statistical tests for calculating p-values.

4.3 Statistical tests

4.3.1 Hypergeometric distribution

Hypergeometric distribution is a xxx for discrete events. We will first briefly introduce the form of hypergenometric and then we map xxx to the ORA analysis.

The problem can be formulated as follows. Assume there are N balls in a bag, where there are K red balls, and $N - K$ blue balls. If randomly picking n balls without replacement from there (once a ball is picked, it will be put back), what is the probability of having k red balls out of n balls?

Also we assume it is independent to pick any ball, then, we can have the following numbers:

- Total number of combinations of picking n balls from the bag: $\binom{N}{n}$.
- Number of combinations of pick k red balls from K red balls: $\binom{K}{k}$.
- Number of combinations of picking $n-k$ blue balls from $N-K$ blue balls: $\binom{N-K}{n-k}$.

Since picking red balls and picking blue balls are independent, the number of combinations of picking n balls which contain k red and $n-k$ blue balls is $\binom{K}{k}\binom{N-K}{n-k}$, and the probability denoted as P_{hyper} is calculated as:

$$P_{\text{hyper}} = \frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{n}}$$

.

Let's denote the number of red balls as a random variable X, then X follows the hypergenometric distribution with parameters N, K and n, written as $X \sim \text{Hyper}(N, K, n)$.

Now we can map to ORA analysis. Before that, we use commonly used denotations in literatures.

	In the gene set	Not in the gene set	Total
DE genes	n_{11}	n_{12}	n_{1+}
Non-DE genes	n_{21}	n_{22}	n_{2+}
Total	n_{+1}	n_{+2}	n

ORA basically has the same form as the ball problem. We just need to change the terms while the statistical model is unchanged. E.g. we change red balls to DE genes and blue balls to non-DE genes. balls to pick is the genes in the gene set. The full map is in Table x.

Total balls	N	Total genes	n
Red balls	K	DE genes	n_{1+}
Blue balls	$N-K$	non-DE genes	n_{2+}
Balls to pick	n	genes in gene set	n_{+1}
Red balls that are picked	k	DE genes in the gene set	n_{11}
Blue balls that are picked	$n-k$	non-DE genes in the gene set	n_{21}

Specifically for ORA, denote number of DE genes in a gene set as a random variable X, then

$$X \sim \text{Hyper}(n, n_{1+}, n_{+1})$$

If n_{11} is big, whether a gene id DE and whether a gene is in the gene set is highly dependent, or in other words, DE genes are over-represented in the gene set. For the random variable X, the p-value is calculated as $Pr(X \geq n_{11})$ which is the probability of X no less than n_{11}. If the p-value is very small, and we observed $X = n_{11}$, we can say, a rare event happens and the p-value is signficant.

$$Pr(X >= n_1 1) = 1 - Pr(X < n_1 1) = 1 - \sum_{x \in 0,..,n_1 1-1} p(x, n_{+1}, n_{1+}, n)$$

4.3.2 Binomial distribution

When n_{1+} or n_{+1} or n is large, the hypergenometric distribution can be approximated to a Binomial test. Now we need to change the question a little bit. Now we only look at the genes in the gene set. In a gene set with n_{+1} genes, each gene can be a DE genes with probabilty p_{diff}. What is the probability of observing n_{11} DE genes. First p_{diff} is estimated as

$$p_{\text{diff}} = n_{1+}/n$$

The probelm can be thought as pick ball n_{+1} times, the number of picking n_{11} balls is $\binom{n_{+1}}{n_{11}}$. The probability of all n_{11} balls are all DE is $p_{\text{diff}}^{n_{11}}$ and the other $n_{+1} - n_{11}$ balls are not DE is $(1 - p_{\text{diff}})^{n_{+1}-n_{11}}$, then the final probability is

$$p = \binom{n_{+1}}{n_{11}} p_{\text{diff}}^{n_{11}} (1 - p_{\text{diff}})^{n_{+1}-n_{11}}$$

If again, denote the number of DE genes in the gene set as a random variable X, then $X \sim \text{Binom}(n_{+1}, p_{\text{diff}})$.

And p-value is calculated as $Pr(X \geq n_{11})$.

We can also do in other other direction, we look at DE genes and each gene has a probability being in the gene set.

$$p_{\text{set}} = n_{+1}/n$$

And approximatedly, $X \sim \text{Binom}(n_{1+}, p_{\text{set}})$. But note, the two p-values are not identical.

4.3.3 z-test

Let's look back Table xx, if the two events "genes are DE" and "gene is in the gene set" are independent, then the probability of a gene being DE in the gene set should be identical to the probability of a gene being DE not in the gene set, which are the following two probabilities:

$$p_1 = n_{11}/n_{+1}$$
$$P_2 = n_{12}/n_{+2}$$

For genes in the gene set, number of DE genes actually follow a Binomilal distribution $\text{Binom}(n_{+1}, p_1)$, and for genes not in the gene set, number of DE genes also follow a Binomial distribution: $\text{Binom}(n_{+2}, p_2)$. Now the problem is to test whether the two Binomial distribution is idential. The null hypothesis is $p_1 = p_2$, then when n_{+1} and n_{+2} are large, the following z-score:

$$z = \frac{p_1 - p_2}{\sqrt{p(1-p)}\sqrt{\frac{1}{n_{1+}} + \frac{1}{n_{2+}}}}$$

where $p = \frac{n_{11}+n_{12}}{n_{+1}+n_{+2}} = \frac{n_{1+}}{n}$. z follows a standard normal distribution $N(0, 1)$.

It easy to see the test is idential is p_1 and p_2 are calculated as the probabilities of a genes being in the gene set, for DE genes and non-DE genes.

4.3.4 Fisher's exact test

Fisher's exact test can be used to

4.3.5 Chi-square test

Pearson's Chi-square test can also be applied to test whether there are dependencies on categorized data. The Chi-square statistic measures the relative sum of squares of the difference between observed values and expected in categories:

$$\chi^2 = \sum_i \frac{(O_i - E_i)^2}{E_i}$$

where O_i is the observed value in category i and E_i is the expected value in category i.

If we apply it to the 2x2 contigency table, actually the data is splt into four non-intersected categories, DE/set, DE/not in set, non-DE/in set and non-DE/not in set. Let's take the first category, e.g. genes are DE and also in the set, the observed value is simply n_{11}. The expected value is $np_{1+}p_{+1}$ which assumes whether a gene is DE and whether a gene is in the set is independent, where $p_{1+} = n_{1+}/n$ and $p_{+1} = n_{+1}/n$. And if we write all four categories, we have

$$\chi^2 = \sum_{i=1}^{2}\sum_{j=1}^{2} \frac{(n_{ij} - np_{i+}p_{+j})^2}{np_{i+}p_{+j}}$$

where $p_{i+} = n_{i+}/n$ and $p_{+j} = n_{+j}/n$.

If n is large, the χ^2 statistic can be approximated as followinng χ^2 distribution with degree of freedom of 1.

Simply calcus reveals that the value of the χ^2 is the square of the z-statistic.

4.4 Calculate in R

To demonstrate the different distributions and tests, we use a data fro EBI Altas database with accession ID E-GEOD-101794. In the diffferential expression analysis, there are 968 egnes that are differential under cutoff FDR $<$ 0.05, and total number of genes are 38592, we use the gene set named "HALLMARK_KRAS_SIGNALING_DN" fro MsigDb database which contains 200 genes. The 2x2 contigency table is as follows

	In the gene set	Not in the gene set	Total
DE	14	954	968
Noe DE	186	37438	37624

	In the gene set	Not in the gene set	Total
Total	200	38392	38592

The function `phyper()` calculates the p-value from hypergeometric distribution. The usage of `phyer()` is:

```
phyper(q, m, n, k)
```

In ORA, `q` is the number of differnetial genes in the gene set, `m` is the size of the gene set, `n` is the number of genes not in the gene set, `k` is the number of differnetial genes. since the hypergeometric can also be calcualted from the other dimension, `m` can be the number of differneital genes, `n` is the number of non-diff genes and `k` is the number of genes in the gene set.

By default `phyper()` calculates the probabilty of Pr(X <= q)

```
1 - phyper(14 - 1, 200, 38392, 968)
# [1] 0.0005686084
```

Note it is the same as

```
phyper(14 - 1, 200, 38392, 968, lower.tail = FALSE)
# [1] 0.0005686084
```

```
1 - phyper(14 - 1, 968, 37624, 200)
# [1] 0.0005686084
```

P-value from Binominal distribution can be calcualted with the function `pbinom()`. The usage is

```
pbinom(q, size, prob)
```

```
1 - pbinom(14 - 1, 200, 968/38592)
# [1] 0.0005919725
```

```
1 - pbinom(14 - 1, 968, 200/38592)
# [1] 0.0006924557
```

To calculate the z-test, we first calculate p1 and p2

```
p1 = 14/200
p2 = 954/38392
p = 968/38592

z = abs(p1 - p2)/sqrt(p * (1-p))/sqrt(1/200 + 1/38392)
```

Since z follows a standard normal distribution, we can use `pnorm()` to calcualte p-value:

```
2*pnorm(z, lower.tail = FALSE)
# [1] 4.647219e-05
```

We can also try to calculate the p-valeu from other other direction. similarly,

```
p1 = 14/968
p2 = 186/37624
p = 200/38592

z = abs(p1 - p2)/sqrt(p * (1-p))/sqrt(1/968 + 1/37624)
2*pnorm(z, lower.tail = FALSE)
# [1] 4.647219e-05
```

The two p-values are identical.

Fisher's exact test can be directly performed by the function `fisher.test()`. The input is the 2x2 contigency table without the margins.

```
cm = matrix(c(14, 186, 954, 37438), nrow = 2)
cm
#      [,1]  [,2]
# [1,]   14   954
# [2,]  186 37438
fisher.test(cm)
#
#   Fisher's Exact Test for Count Data
#
# data:  cm
# p-value = 0.0005686
# alternative hypothesis: true odds ratio is not equal to 1
# 95 percent confidence interval:
#  1.577790 5.106564
# sample estimates:
# odds ratio
#   2.953612
```

`fisher.test()` generates many other results, here the odd ratio is the statistic of fisher exact test. The odd ratio is defined as

$$\text{oddratio} = \frac{n_{11}}{n_{21}} / \frac{n_{11}}{n_{22}} = \frac{n_{11}n_{22}}{n_{12}n_{21}}$$

which is the ratio of fraction of DE in the gene set and not in the gene set. If odd ratio is larger than 1, over-representation.

Last, the Chi-square test can be applied with the function `chisq.test()`. Similarly the input is the 2x2 contigency table without the margins. Note here we also set `correct = FALSE` so that ...

```
chisq.test(cm, correct = FALSE)
#
#   Pearson's Chi-squared test
#
# data:  cm
# X-squared = 16.587, df = 1, p-value = 4.647e-05
```

```
z^2
# [1] 16.58685
```

Since there are several ways to perform the test, it would be interesting to test which method runs faster.

```
library(microbenchmark)

microbenchmark(
    hyper = 1 - phyper(13, 200, 38392, 968),
    fisher = fisher.test(cm),
    binom = 1 - pbinom(13, 968, 200/38592),
    chisq = chisq.test(cm, correct = FALSE),
    ztest = {
        p1 = 14/200
        p2 = 954/38392
        p = 968/38592

        z = abs(p1 - p2)/sqrt(p*(1-p))/sqrt(1/200 + 1/38392)
        2*pnorm(z, lower.tail = FALSE)
    },
    times = 1000
)
# Unit: nanoseconds
#    expr    min       lq       mean   median       uq     max neval
#   hyper   1048   1570.0   2420.791   2304.0   2820.0   29427  1000
#  fisher 581880 670109.0 804108.552 714968.0 810881.5 7828993  1000
#   binom    932   1467.0   2675.919   2339.5   3083.5   21100  1000
#   chisq  43040  57800.5  71985.373  66121.5  75873.0  416321  1000
#   ztest   2148   3167.0   5320.409   4840.5   5987.5   23785  1000
```

We can see from the benchmark result, hypergeometric and bionimal distribution-based method are the fastest. As a comparison, Chi-square test and fisher's method run slow, especially for fisher's exact method. The reason is the latter two also include many other calculations besides p-values. This benchmark results actually tells us, if in the future readers want to implement ORA analysis by their own, hypergeometric and bimonial methods should be considered firstly.

Method	p-value
Hypergeometric distribution	0.0005686 (exact p-value)
Fisher's exact test	0.0005686 (exact p-value)
Binomial distribution	0.0005920 / 0.0006924
z-test	0.0000465
Chi-square test	0.0000465

4.5 Current tools

There are many tools that implement ORA analysis compared to other gene set enrichment analysis tools which will introduced in later chapters, mainly because it runs fast, the method is simply to understand. All most all the ORA web-based tools are in a two-step analysis. 1. upload the gene lists and setting parameters and 2. see the results. In this section, we will go through three web-based ORA tools, as well as one Bioconductor package.

```
load("data/demo_ora.RData")
```

```
library(clusterProfiler)
```

You need to make sure the gene IDs are Entrez IDs. The following function helps to automatically convert gene IDs to Entrez IDs. The input can be a vector of genes or a gene expression matrix.

We convert the diff genes `diff_gene` to Entrez IDs. Note some genes are lost due to the conversion.

```
head(diff_gene)
# [1] "FGR"    "NIPAL3" "LAP3"   "CASP10"
diff_gene = convert_to_entrez_id(diff_gene)
# testing org.Hs.egENSEMBL...
# testing org.Hs.egREFSEQ...
# testing org.Hs.egSYMBOL...
#   gene id might be SYMBOL (p = 0.990)
head(diff_gene)
# [1] "2268"  "57185" "51056" "843"
length(diff_gene)
# [1] 963
```

Next we perform ORA on different gene sets

1. GO enrichment

```
library(org.Hs.eg.db)
tb = enrichGO(gene = diff_gene, ont = "BP", OrgDb = org.Hs.eg.db)
head(tb)
#                    ID                  Description GeneRatio   BgRatio
```

```
# GO:0006959 GO:0006959       humoral immune response    70/848 317/18800
# GO:0002443 GO:0002443   leukocyte mediated immunity    80/848 457/18800
# GO:0006909 GO:0006909                   phagocytosis    65/848 310/18800
# GO:0002253 GO:0002253 activation of immune response    72/848 386/18800
#                  pvalue     p.adjust       qvalue
# GO:0006959 3.056579e-29 1.548157e-25 1.234858e-25
# GO:0002443 4.501981e-26 1.140127e-22 9.094002e-23
# GO:0006909 7.358671e-26 1.242389e-22 9.909678e-23
# GO:0002253 2.898616e-25 3.670373e-22 2.927602e-22
#                                                                                                  geneID
# GO:0006959 6556/729/2920/2219/4069/54209/730249/6347/3458/5967/729230/722/725/5266/6590/5..
# GO:0002443 2268/4843/7305/57823/729/50943/7037/3383/54209/50487/51311/10312/57379/10384/8..
# GO:0006909 2268/23221/7305/6556/2219/54209/10326/3055/1089/6347/3458/4688/729230/722/725/..
# GO:0002253 2268/7305/11119/729/50943/2219/58484/54209/3055/80381/10384/2633/8013/722/725/..
#            Count
# GO:0006959    70
# GO:0002443    80
# GO:0006909    65
# GO:0002253    72
```

2. KEGG enrichment

```
tb = enrichKEGG(gene = diff_gene, organism = "hsa")
head(tb)
#                ID                                                 Description
# hsa04060 hsa04060                         Cytokine-cytokine receptor interaction
# hsa04061 hsa04061 Viral protein interaction with cytokine and cytokine receptor
# hsa04657 hsa04657                                      IL-17 signaling pathway
# hsa04380 hsa04380                                  Osteoclast differentiation
#          GeneRatio  BgRatio       pvalue     p.adjust       qvalue
# hsa04060    50/443 295/8163 2.498917e-13 7.721653e-11 6.760227e-11
# hsa04061    27/443 100/8163 1.504786e-12 2.324895e-10 2.035421e-10
# hsa04657    25/443  94/8163 1.489344e-11 1.534025e-09 1.343023e-09
# hsa04380    28/443 128/8163 1.387173e-10 1.071591e-08 9.381668e-09
#                                                                                              geneID
# hsa04060 53832/608/2920/3595/3589/5008/1440/6354/6347/55801/3458/3552/7850/9173/8809/88..
# hsa04061 53832/2920/6354/6347/8809/8807/729230/6372/51554/3569/4283/56477/3577/2921/637..
# hsa04657 5743/2920/1440/6354/6347/3458/727897/6372/3553/2354/3569/4322/6279/3934/4314/6..
# hsa04380 7305/54209/10326/54/11024/3458/3552/6772/2274/4688/8503/3553/2354/10288/2212/2..
#          Count
# hsa04060    50
# hsa04061    27
# hsa04657    25
# hsa04380    28
```

3. Reactome enrichment

```
library(ReactomePA)
tb = enrichPathway(gene = diff_gene)
head(tb)
#                          ID                          Description GeneRatio
# R-HSA-6798695 R-HSA-6798695            Neutrophil degranulation    72/590
# R-HSA-6783783 R-HSA-6783783            Interleukin-10 signaling    21/590
# R-HSA-380108   R-HSA-380108 Chemokine receptors bind chemokines    22/590
# R-HSA-449147   R-HSA-449147           Signaling by Interleukins    63/590
#                BgRatio       pvalue     p.adjust       qvalue
# R-HSA-6798695 482/10891 1.508088e-15 1.666437e-12 1.585873e-12
# R-HSA-6783783  47/10891 6.047144e-15 3.341047e-12 3.179524e-12
# R-HSA-380108   59/10891 1.285611e-13 4.735333e-11 4.506404e-11
# R-HSA-449147  473/10891 1.882086e-11 5.199262e-09 4.947905e-09
#
# R-HSA-6798695 2268/64386/5329/7305/55/6556/2219/4680/4069/5836/10326/10562/3614/1049
# R-HSA-6783783 5743/2920/3383/7076/1440/6347/942/3552/7850/729230/3553/3569/3557/2919/
# R-HSA-380108  2920/6354/6347/729230/6372/51554/4283/56477/58191/3577/2921/6374/5196/
# R-HSA-449147  4843/53832/3082/6196/5743/2920/3595/3383/3589/5008/3055/7076/1440/6347/
#               Count
# R-HSA-6798695    72
# R-HSA-6783783    21
# R-HSA-380108     22
# R-HSA-449147     63
```

4. DO enrichment

```
library(DOSE)
tb = enrichDO(gene = diff_gene, ont = "DO")
head(tb)
#                  ID         Description GeneRatio  BgRatio       pvalue
# DOID:403   DOID:403       mouth disease    52/503 188/8007 8.786533e-21
# DOID:3388 DOID:3388 periodontal disease    44/503 139/8007 2.928521e-20
# DOID:1091 DOID:1091       tooth disease    47/503 162/8007 8.272464e-20
# DOID:850   DOID:850        lung disease    85/503 499/8007 3.005721e-18
#               p.adjust       qvalue
# DOID:403  6.915002e-18 3.394376e-18
# DOID:3388 1.152373e-17 5.656669e-18
# DOID:1091 2.170143e-17 1.065261e-17
# DOID:850  5.913756e-16 2.902894e-16
#                                                                                 
# DOID:403  4843/6401/3082/50943/5743/3595/5243/4069/3589/5008/7076/4210/5054/9126/635
# DOID:3388 4843/6401/3082/50943/5743/3595/5243/4069/3589/5008/7076/5054/9126/6354/634
# DOID:1091 4843/6401/3082/50943/5743/3595/5243/4069/3589/5008/7076/5054/9126/6354/634
# DOID:850  4843/5329/6556/3082/50943/5743/3383/3589/3055/5054/6347/374/595/7450/59341
#           Count
# DOID:403     52
```

```
# DOID:3388    44
# DOID:1091    47
# DOID:850     85
```

5. MSigDB enrichment

There is no built-in function specific for MSigDB gene sets, but there is a universal function `enrichr()` which accepts manually-specified gene sets. The gene sets object is simply a two-column data frame:

- the first column is the gene set ID
- the second column is the gene ID

```
library(msigdbr)
gene_sets = msigdbr(category = "H")
map = gene_sets[, c("gs_name", "entrez_gene")]

tb = enricher(gene = diff_gene, TERM2GENE = map)
head(tb)
#                                                                    ID
# HALLMARK_INTERFERON_GAMMA_RESPONSE HALLMARK_INTERFERON_GAMMA_RESPONSE
# HALLMARK_INFLAMMATORY_RESPONSE         HALLMARK_INFLAMMATORY_RESPONSE
# HALLMARK_INTERFERON_ALPHA_RESPONSE HALLMARK_INTERFERON_ALPHA_RESPONSE
# HALLMARK_TNFA_SIGNALING_VIA_NFKB     HALLMARK_TNFA_SIGNALING_VIA_NFKB
#                                                          Description GeneRatio
# HALLMARK_INTERFERON_GAMMA_RESPONSE HALLMARK_INTERFERON_GAMMA_RESPONSE    55/348
# HALLMARK_INFLAMMATORY_RESPONSE         HALLMARK_INFLAMMATORY_RESPONSE    54/348
# HALLMARK_INTERFERON_ALPHA_RESPONSE HALLMARK_INTERFERON_ALPHA_RESPONSE    33/348
# HALLMARK_TNFA_SIGNALING_VIA_NFKB     HALLMARK_TNFA_SIGNALING_VIA_NFKB    49/348
#                                     BgRatio       pvalue     p.adjust
# HALLMARK_INTERFERON_GAMMA_RESPONSE 200/4383 1.471098e-17 6.914161e-16
# HALLMARK_INFLAMMATORY_RESPONSE     200/4383 7.383369e-17 1.735092e-15
# HALLMARK_INTERFERON_ALPHA_RESPONSE  97/4383 9.355985e-14 1.465771e-12
# HALLMARK_TNFA_SIGNALING_VIA_NFKB   200/4383 1.524722e-13 1.791549e-12
#                                         qvalue
# HALLMARK_INTERFERON_GAMMA_RESPONSE 5.264982e-16
# HALLMARK_INFLAMMATORY_RESPONSE     1.321234e-15
# HALLMARK_INTERFERON_ALPHA_RESPONSE 1.116153e-12
# HALLMARK_TNFA_SIGNALING_VIA_NFKB   1.364225e-12
#
# HALLMARK_INTERFERON_GAMMA_RESPONSE 51056/57823/10797/3430/5743/3383/10135/6354/6347/6648/942/67
# HALLMARK_INFLAMMATORY_RESPONSE     6401/5329/2769/490/3383/5008/7076/4210/366/3249/6324/10135/3
# HALLMARK_INTERFERON_ALPHA_RESPONSE 51056/3430/135112/3659/6737/94240/8638/10906/2766/834/10561/
# HALLMARK_TNFA_SIGNALING_VIA_NFKB   5329/490/5743/2920/3383/10135/5054/6347/374/595/6648/1839/35
#                                    Count
# HALLMARK_INTERFERON_GAMMA_RESPONSE    55
# HALLMARK_INFLAMMATORY_RESPONSE        54
```

```
# HALLMARK_INTERFERON_ALPHA_RESPONSE    33
# HALLMARK_TNFA_SIGNALING_VIA_NFKB      49
```

4.6 Limitations of ORA

ORA analysis is simply and it runs fast. Thus, currently, there are many online tools support it. However, there are many limiations.

4.6.1 different tools generate inconsistent results

There are a lot of tools, but using the same gene set database, results from tools normally do not agree very well. The reasons are:

1. Different versions of annotation databases.
2. How they process redundant terms
3. Different default cutoffs
4. Different methodd for control p-values
5. differnet background genes.

4.6.2 sensitive to the selection of background genes.

Genome / all protein-coding gene / all genes measured? The selection mainly affects the value in the blue cells. Note the ORA actually compares diff gene in the set to the 'other gene,' and we actually assume 'other genes' are not diff nor in the set, thus, "other genes" are not useless. Select a "large" background may increase false positives. Select a "small" background may miss some positives, but it has low false positives. Also some arguments: If genes are not measured, they should not be put into the analysis.

```
1 - phyper(13, 200, 38392, 968)
# [1] 0.0005686084
1 - phyper(13, 200, 38392 - 10000, 968)
# [1] 0.008444096
1 - phyper(13, 200, 38392 - 20000, 968)
# [1] 0.1606062
1 - phyper(13, 200, 38392 + 10000, 968)
# [1] 5.473547e-05
1 - phyper(13, 200, 38392 + 20000, 968)
# [1] 7.140953e-06
```

In general, with larger background set, the p-value becomes significant. ORA

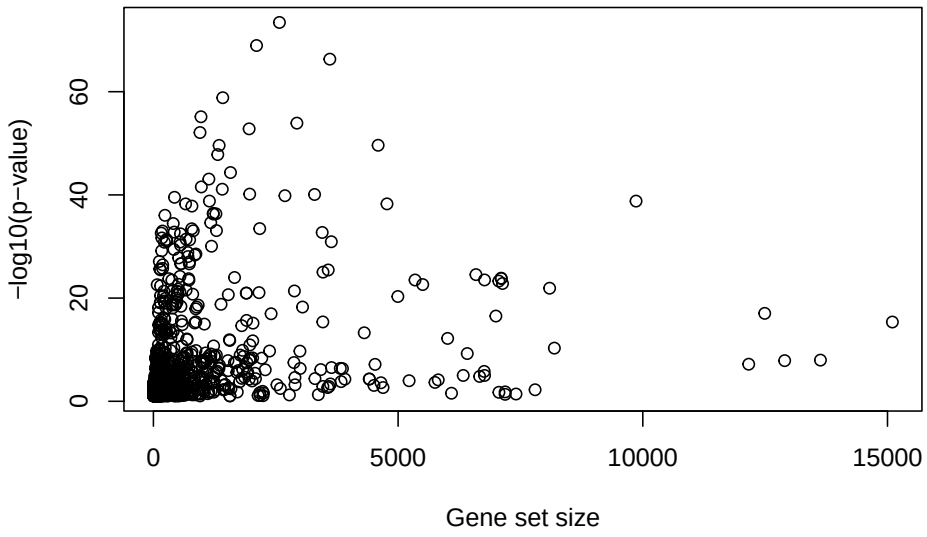
4.6.3 Preference of larger gene sets

There is a trend that large gene sets may have more significant p-values. This is actually expected because as the sample size increases, the test power also

increases. Under the context of hypergeometric test or Binomial test, number of genes is the "sample size."

```
1 - pbinom(52, 100, 0.5)
# [1] 0.3086497
1 - pbinom(52, 1000, 0.5)
# [1] 1
1 - pbinom(52, 10000, 0.5)
# [1] 1
```

```
tb = read.table("data/david_result.txt", sep = "\t", header = TRUE)
plot(tb$Pop.Hits, -log10(tb$PValue), xlab = "Gene set size", ylab = "-log10(p-value)")
```

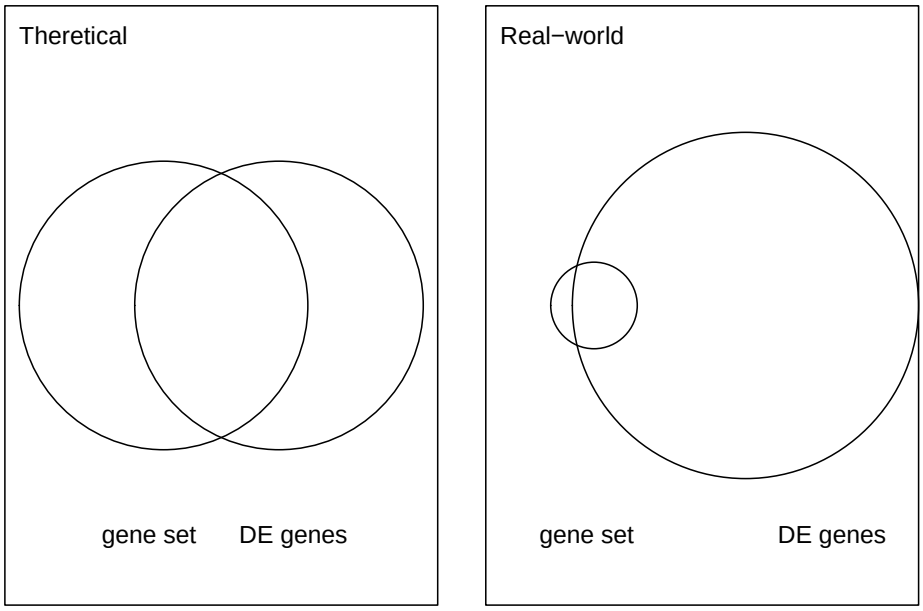


4.6.4 Imbalanced contigency table

In real-world scenario, as show in xx. There are many gene sets with small number of genes, which makes the 2x2 contigency imbalanced.

```
library(grid)
grid.newpage()
pushViewport(viewport(x = 0, width = 0.5, just = "left"))
pushViewport(viewport(xscale = c(-1.5, 1.5), yscale = c(-1.5, 1.5), width = 0.9, height = 0.9))
grid.rect()
grid.circle(x = -0.4, y = 0, r = 1, default.units = "native")
grid.circle(x = 0.4, y = 0, r = 1, default.units = "native")
grid.text("gene set", x = -0.5, y = -1.1, just = "top", default.units = "native")
grid.text("DE genes", x = 0.5, y = -1.1, just = "top", default.units = "native")
grid.text("Theretical", x = -1.4, y = 1.4, default.units = "native", just = c("left", "top"))
popViewport()
```

```
popViewport()

pushViewport(viewport(x = 0.5, width = 0.5, just = "left"))
pushViewport(viewport(xscale = c(-1.5, 1.5), yscale = c(-1.5, 1.5), width = 0.9, height
grid.rect()
grid.circle(x = -0.75, y = 0, r = 0.3, default.units = "native")
grid.circle(x = 0.3, y = 0, r = 1.2, default.units = "native")
grid.text("gene set", x = -0.8, y = -1.1, just = "top", default.units = "native")
grid.text("DE genes", x = 0.9, y = -1.1, just = "top", default.units = "native")
grid.text("Real-world", x = -1.4, y = 1.4, default.units = "native", just = c("left",
popViewport()
popViewport()
```



In many cases when the gene set is small, the enrichment anlaysis will be sensitive to the value of number of DE genes in the gene set, i.e., the valeu of n_{11}.

```
1 - phyper(13, 200, 38392, 968)
# [1] 0.0005686084
1 - phyper(13 - 3, 200, 38392, 968)
# [1] 0.01263274
1 - phyper(13 + 3, 200, 38392, 968)
# [1] 1.35041e-05
```

In DE analysis, normally we set a cutoff of adjusted p-values for filter the significant DE genes, since the p-values are sensitive to n_{11}, actually xxx

4.6.5 Theoretical reasons

1. the assumption

Chapter 5

The GSEA method

5.1 Overview

The tool GSEA is the mostly used for gene set enrichment analysis. In a study, genes are very moderate change, that after filter by p-values from DE anlaysis, no signficant genes are left. Thus ORA anlaysis cannot be applied to it. Mootha developed a new method which takes all genes in to consideration and test teh significance of gene sets by evaluating the accumulated effect of genes' diffenetial expression. It successfully xxx . In this chapter xxx

5.2 The GSEA method, version one

ORA analysis actually applies a binary conversion on genes where genes pass the cutoff are set as 1 and others are set as 0. This binary transformation over-simplifies the problem and a lot of information are lost, e.g. the differential expression, and genes around the cutoff can be optionally set to 1 or 0 by object choise of cutoffs.

The GSEA method was first proposed in 2003. Authors worked on a dataset where all genes only had intermediate expression change, where with normal differential gene expression methods, there is no significant gene left under normal cutoffs, thus ORA analysis is impossible to perform. Mootha developed a new method named GSEA which convert genes to the rank of the differential expression. Compared to ORA, GSEA v1 can distingush the differnet expression changes.

```
ORA, differenital expression -> 1111100000
GSEA   -> 1 2 3 4 5 6 ...
```

5.3 GSEA v1, step 1

To make is simply to discuss, we assume the expression data is from a two-condition comparison, e.g. tumor vs normal or treatment vs control. We assume there are repeated samples in both conditions. The first step of GSEA is to reduce the original matrix to gene-level scores,, i.e. for a row with m1+m2 values, reduce it to a single value which measures the differnetial expression in the conditions. In GSEA, it proposed to use the signial to noise ratio $((\mu_1 - \mu_2)/(\sigma_1 + \sigma_2)$ which actually does not taken account of the sample size. However, larger S2N absolute value, the more differential the gene is expressed.

After obtaining the gene-level score, a rank transformation is applied which is applied to the gene-level scores. all genes are sorted from the highest to the lowest to form a sorted gene list.

5.4 GSAE v1, step 2

With the sorted gene list, for a specific gene set, an enrichment score can be calculated. The genes in the ranked list are put into two groups, in the gene set and not in the gene set. Denote there are total n genes, n_k is the number of genes in the gene set and p is a certain position in the ranked list. For genes in the gene set, we calcualte the fraction denoted as F_{1p} at the position p:

$$F_{1p} = \frac{1}{n_k} \sum_{i=1}^{p} I(g_i \in G)$$

where $I()$ is a identify function, g_i is the gene at position i and G is the set of genes in the gene set. Similarly, for genes not in the gene set, we calculate a second fraction denoted as F_{2p} at position p:

$$F_{2p} = \frac{1}{n - n_k} \sum_{i=1}^{p} I(g_i \notin G)$$

As can be seen, F_{1p} and F_{2p} are actually the cummulative probability of genes in the two sets. Then at position p, the difference at the two cummulative probalibyy is calculated, and the maximal difference is defined as the enrichmetn score (ES):

$$ES = max_{p \in 1..n}(F_{1p} - F2p)$$

According to the definition, actaully the ES is exactly teh Kolmogorov-Smirnov statistic and use for test whether two CDFs are the same. Till now, since ES is a statistic with known sources, the p-value can be directly from the KS test. However, KS test is not a powerful test and in the orignal paper, which will also

be demonstrated later, the null hypothesis is constructed via permuation-based test.

According to the definition, ES is actaully directional, which is always non-negative. When ES score is large, the gene sets are more enriched on the top of the ranked gene list, which shows higher expression in group 1, if we assume positive different means high expression in group 1. To obtain the down-regulation, the denifition of ES can be changed a little bit:

$$ES_{down} = min_{p \in 1..n}(F_{1p} - F2p)$$

In this case, ES_{down} is non-positive and a larger absolute value mean s...

To capture both the two directional change,

$$k = argmax_p(|F_{1p} - F_{2p}|)$$
$$ES_{bidirectional} = |F_{1k} - F_{2k}|or$$

5.5 Permutation-based test

Once we have the definition of the ES statistic which measures differential expression for genes in a gene set, we need a null hypothesis distribution of ES to calculate p-values. KS-test can be directly used to calcualte p-values, but here GSEA uses a more robust way to construct the null distribution of ES by sample permutation.

In the two-contidional data, if genes in the gene set are differneital expressed to the conditions, we can say the gene expression relates to the condition. A randomly case, is the gene expression is independent to the conditions and with the observed value we want to reject the null hypothesis. Since the input matris is normally complex, and unlike the the t-test which can have a anatical form with the normality assumption, to obtain a analytical form of ES is difficult. However, to destroy the relation of gene expression and conditions, a simple but power way is to random sample the sample labels or the condition labels. If fixed the matrix, but random assign the xx of which same is group 1 and which samle is group 2. In this rando setting, it actally means the gene expression has no relation to the condition because the conditions are randomly assigned. The permuation is a random assignment process which randomly shuffle the xxx of labels, but keeps the number of samples in the two groups unchanged.

sample permutation can be simply done with the `sample()` function. Assume we have a condition desigh with 10 samples of 5 group A and 5 group B:

```
cmp = rep(c("A", "B"), each = 5)
sample(cmp, length(cmp))
#  [1] "A" "A" "B" "A" "A" "B" "B" "B" "A" "B"
```

Each sample permutation generates a new experimental design where the condition is independent to gene expression, thus a ES can be calculated with step 1 and 2. teh sample permutation will be executed to a large number of times to gain enough "random" ES for form the null distribution. The number of permulations normally is large and it is recommended to not less than 1000. The null distribution of ES is from the random ES scores and p-value is defined as the probability of ES being euqla to or larger than the observed one:

The p-value can be easily calculated as:

$$p = \frac{1}{K} \sum I(ES_r \geq ES)$$

Sample permutation is a powerful and a general way in statistics. In Chapter xx, we will demonstrate it is used to construct a non-parametric null distribution for any kind of gene set level statistics. Also in Chapter x, we will also introduce permuation data by genes.

5.6 The GSEA method, version 2

The first version of GSEA successfully discovered affected biological functions that show moderate expression chagnes. However, in a study, reseachers pointed several limitations of the algorithm. The main argument is under the GSEA algorithm, although it can distinguish the different levels of differential expression, genes have equal distance to neighbouring genes in the ranked list, and there are the following two consiquences:

1. genes with higher differnetial expression are more important
2. genes in the middle of hte gene list are normally have very small change or even no change. However, an assumulation in the middle can also cause an enrichment.

Due to the gene-level score is not an uniform distribution while xxx, In 2005, Subrammian imporved the GSEA algorithm by taking the gene-level score as the weight, in this case, the differential genes are enhanced and genes with weak differneital expression are suppressed. And it forms the current standard GSEA algorithm.

Recall how we define F_{1p} and F_{2p}. The new GSEA changes the definition of F_{1p} as:

$$F_{1p} = \frac{1}{n_r} \sum_{i=1}^{p} I(g_i \in G) \cdot |r_j|^a$$

$$where n_r = \sum_{i=1}^{n} I(g_i \in G) \cdot |r_j|^a$$

Note F_{2p} for the genes not in the genes is unchanged. The power a reflects the importance of the differential expression. By default, GSEA set $a = 1$. When $a = 0$, it reduces to the original version of GSEA.

Similarlly the ES score is the same as in the origial version, for the one-directional or two-directinoal.

Now the form of ES is changed, thus, it becomes the so-called "weighted KS statistics." It is difficult to perform the test. Again, we perform the sample-permutation that in every permutations, the new ES score is calclated .

5.7 Compare the two GSEA

It would be interesting to compare the two versions of GSEA to see how the weight helps to xxx and also the power of the test.

5.8 Permutations

To construct the null distribution, we introduced to randonly suffle samples to break the relation of xxx. There is actually a second way to construct "random dataset" by gene permutation, which randomly shuffle the genes or identially, randoly shuffle whether genes is in the gene set or not. This is a way to break the relation of differnetial expression and the xx of genes in the gene sets. If assume most of the genes are not differneitally expressed, the the randomized gene sets will mostly not show differneital expression. In sample permutation, the gene-level scores need to be calculated in every permutatin, as a comparison, in gene permutation, the gene-level scores can only be calculated once and can be repeated used. In some of current tools which implements GSEA algorithm, teh null distribution is generated by gene permutation. Without going to deep, we wold like to mention, the two permutation methods, although they all produce ramdom dataset, they correspond to two complete different things. and with some specific datasets, the two xxx may give complete different p-values (or conclusions)

5.9 Other aspects of GSEA

5.9.1 The direction of GSEA

5.9.2 Leading edge genes

5.9.3 Normalized enrichment score

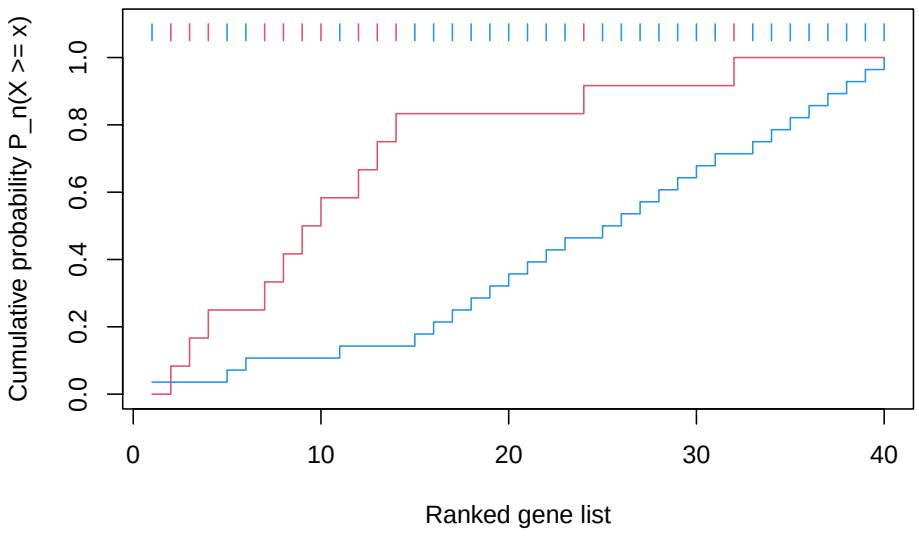
Because teh enrichment score is averaged from all genes in a gene set, in general, large significant gene sets may only have a small ES scores. Thus, the ES cannot reflect the degree of the enrichment and cannot be compariable between gene sets. The normalized

5.10 Compare ORA and GSEA

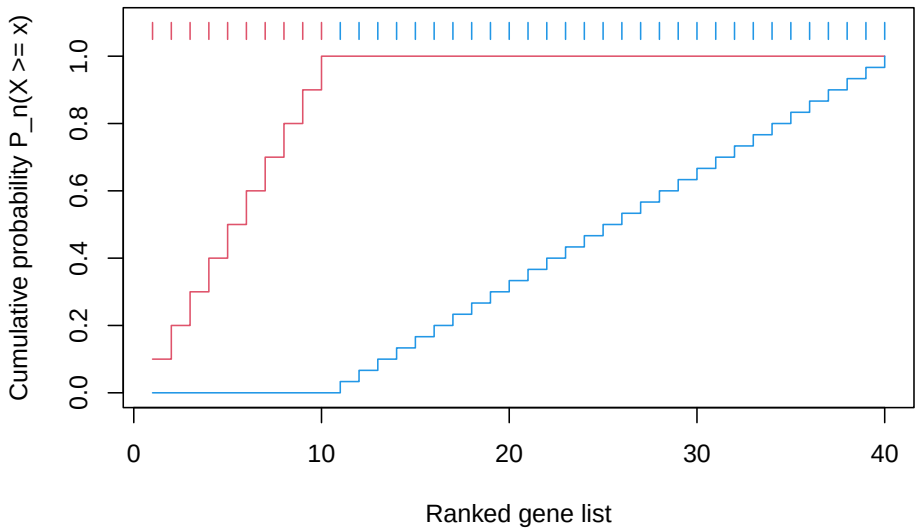
Since ORA and GSEA are the mostly used methods for gene set enrichmetn anlaysis, it would be interesting to apply the two methods on a same dataset to compare xxx

```
x = 1:40
gs = c(F, T, T, T, F, F, T, T, T, T, F, T, T, T, F, F, F, F, F, F, F, F, F, T, F, F, F
```

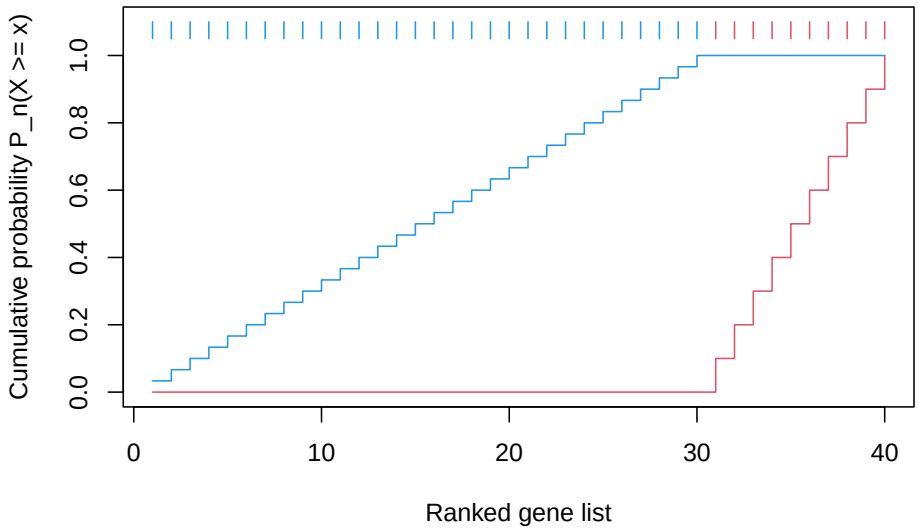
```
gsea_cdf = function(x, gs, ...) {
    f1 = cumsum(gs)/sum(gs)

    f2 = cumsum(!gs)/sum(!gs)

    n = length(x)
    plot(1:n, f1, type = "s", col = 2, ylim = c(0, 1.1),
        xlab = "Ranked gene list", ylab = "Cumulative probability P_n(X >= x)", ...)
    lines(1:n, f2, type = "s", col = 4)

    segments(1:n, 1.05, 1:n, 1.1, col = ifelse(gs, 2, 4))
}

gsea_cdf(x, gs)
```



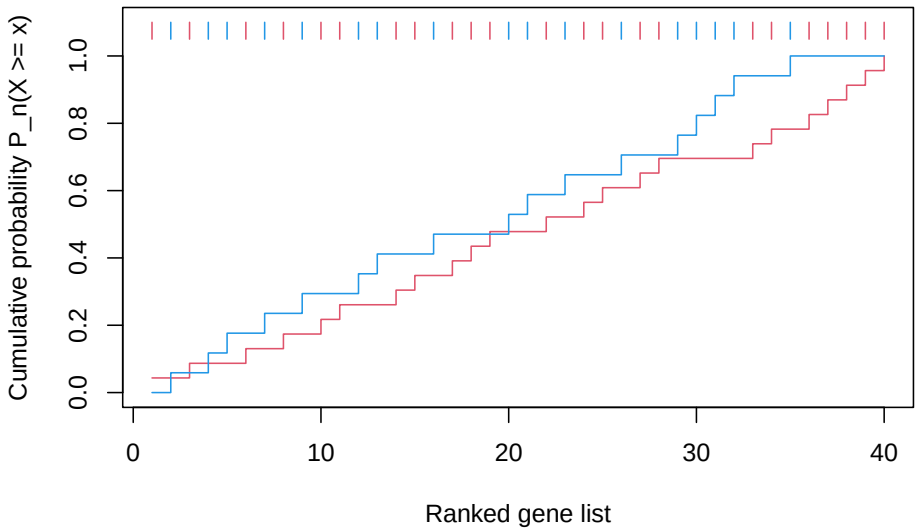

```
gs = c(rep(T, 10), rep(F, 30))
gsea_cdf(x, gs)
```



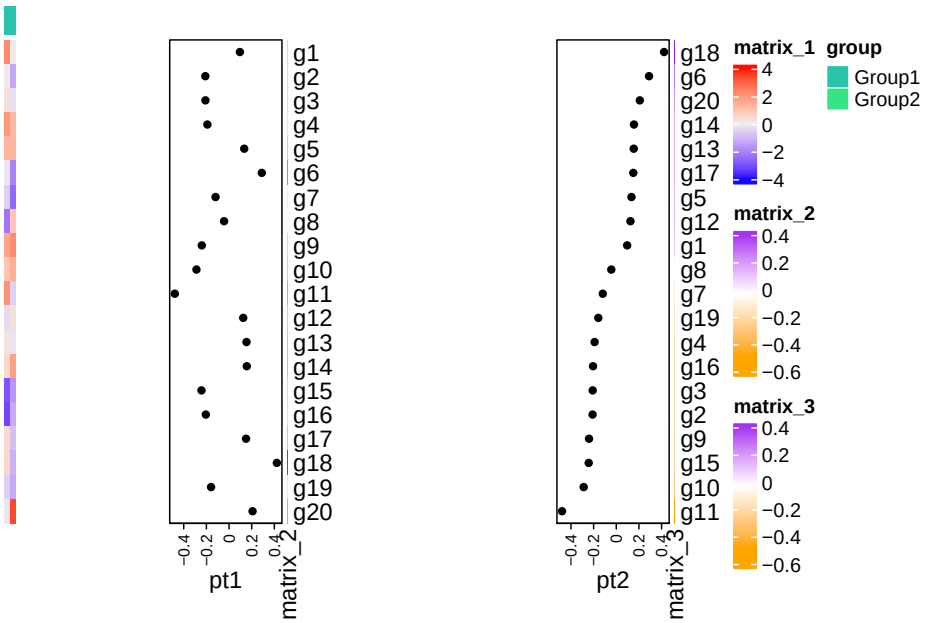

```
gs = c(rep(F, 30), rep(T, 10))
gsea_cdf(x, gs)
```




```
gs = sample(c(T, F), 40, replace = TRUE)
gsea_cdf(x, gs)
```




```
set.seed(54)
mean_diff1 = rnorm(20)

m1 = do.call(rbind, lapply(1:20, function(i) {
    c(rnorm(10, mean = mean_diff1[i]), rnorm(10, mean = -mean_diff1[i]))
}))
rownames(m1) = paste0("g", 1:20)

x1 = rowMeans(m1)

x2 = sort(x1, decreasing = TRUE)

fa = c(rep("Group1", 10), rep("Group2", 10))
library(ComplexHeatmap)
ht_list = Heatmap(m1, top_annotation = HeatmapAnnotation(group = fa), cluster_rows = FA
    Heatmap(x1, col = c("orange", "white", "purple"), left_annotation = rowAnnotation(
    Heatmap(x2, col = c("orange", "white", "purple"), left_annotation = rowAnnotation(

draw(ht_list, ht_gap = unit(4, "cm"), auto_adjust = FALSE)
```



Prepare the data. Here we read three types of data:

1. the phenotype/condition data
2. the gene expression matrix
3. the gene set

The gene expression data is saved in `.gct` format and experimental condition/phenotype is saves in `.cls` format. Both formats are very simple, and you can try to write your own code to parse them.

Gene sets are saved in "`.gmt` format. It is also a simple format. You can try to parse it by your own.

```
library(CePa)
condition = read.cls("data/P53.cls", treatment = "MUT", control = "WT")$label
expr = read.gct("data/P53_collapsed_symbols.gct")

ln = strsplit(readLines("data/c2.symbols.gmt"), "\t")
gs = lapply(ln, function(x) x[-(1:2)])
names(gs) = sapply(ln, function(x) x[1])

geneset = gs[["p53hypoxiaPathway"]]
```

- `expr`
- `condition`
- `geneset`

This gene set is very small:

```
length(geneset)
# [1] 20
```

Note here gene IDs in the expression matrix and in the gene sets are all gene symbols, thus no more adjustment needs to be done here.

The gene-level difference score is set as signal-to-noise ratios, which is:

- mean in group 1
- mean in group 2
- sd in group 1
- sd in group 2

$$\frac{\mu_1 - \mu_2}{\sigma_1 + \sigma_2}$$

We calculate the gene-level difference score in `s`:

```
s = apply(expr, 1, function(x) {
    x1 = x[condition == "WT"]
    x2 = x[condition == "MUT"]
    (mean(x1) - mean(x2))/(sd(x1) + sd(x2))
})
```

- s: gene-level difference score

Sort the gene scores from the highest to the lowest (to make it into a ranked list):

```
s = sort(s, decreasing = TRUE)
```

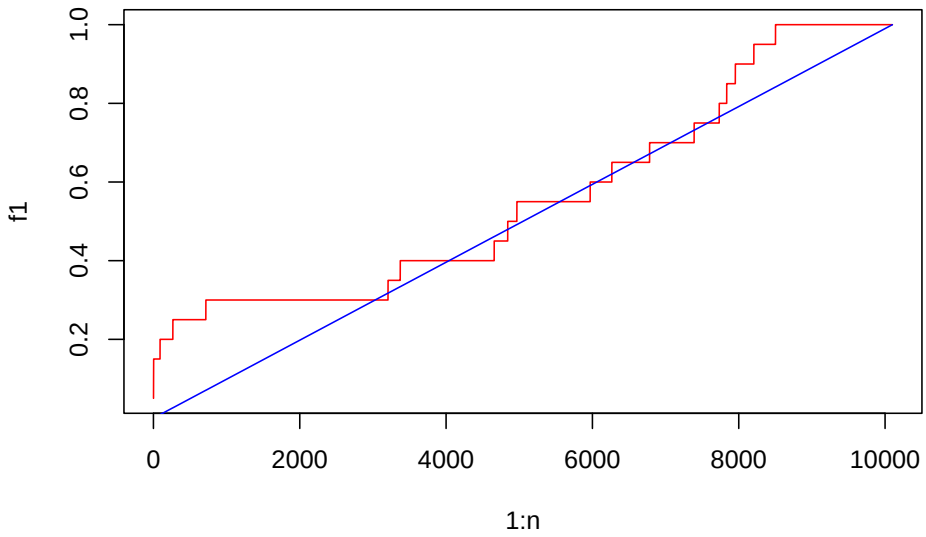
Next we first implement the original GSEA method, which was proposed in Mootha et al., 2003.

```
## original GSEA
l_set = names(s) %in% geneset
f1 = cumsum(l_set)/sum(l_set)

l_other = !names(s) %in% geneset
f2 = cumsum(l_other)/sum(l_other)
```

Here `f1` is the cumulative probability function of genes in the set and `f2` is the cumulative probability function of genes not in the set.

We first plot the CDF of two distributions.

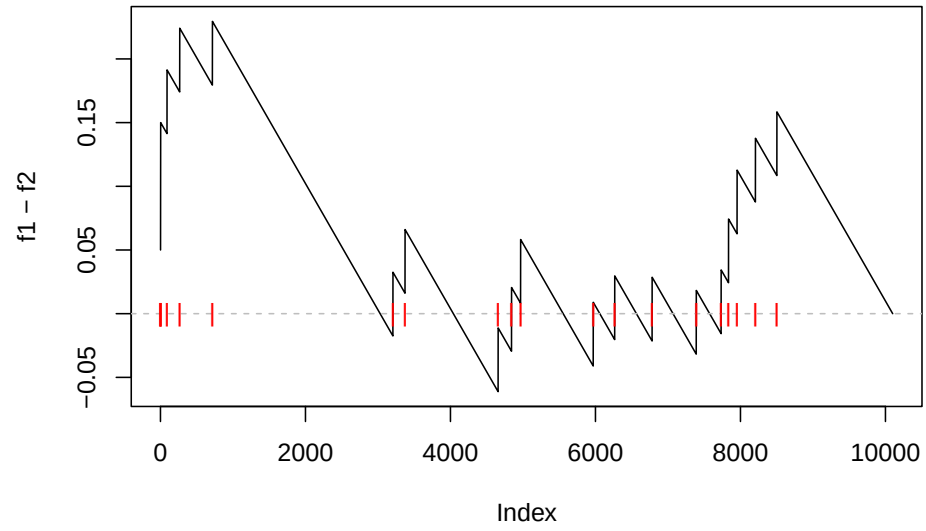
```
n = length(s)
plot(1:n, f1, type = "l", col = "red")
lines(1:n, f2, col = "blue")
```



The reason why the blue locates almost on the diagonal is the gene set is very small.

Next the difference of cumulative probability (`f1 - f2`) at each position of the ranked gene list. Let's call it "the GSEA plot."

```
plot(f1 - f2, type = "l")
abline(h = 0, lty = 2, col = "grey")
points(which(l_set), rep(0, sum(l_set)), pch = "|", col = "red")
```

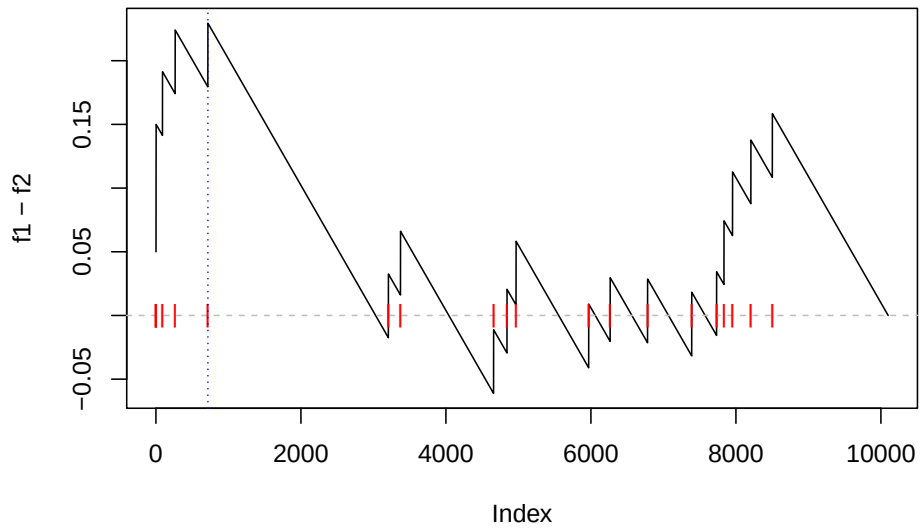


The enrichment score (ES) defined as `max(f1 - f2)` is:

```
es = max(f1 - f2)
es
# [1] 0.2294643
```

And the position in the "GSEA plot":

```
plot(f1 - f2, type = "l")
abline(h = 0, lty = 2, col = "grey")
points(which(l_set), rep(0, sum(l_set)), pch = "|", col = "red")
abline(v = which.max(f1 - f2), lty = 3, col = "blue")
```



The statistic `es` actually is the Kolmogorov-Smirnov statistics, thus, we can directly apply the KS test:

```
ks.test(which(l_set), which(l_other))
#
#   Asymptotic two-sample Kolmogorov-Smirnov test
#
# data:  which(l_set) and which(l_other)
# D = 0.22946, p-value = 0.244
# alternative hypothesis: two-sided
```

However, we can see the p-value is not significant, this is because KS test is not a powerful test. Next we construct the null distribution by sample permutation.

In the next code chunk, the calculation of ES score is wrapped into a function, also we use `rowMeans()` and `rowSds()` to speed up the calculation of gene-level scores.

```
library(matrixStats)
# expr: the complete expression matrix
# condition: the condition labels of samples
# cmp: a vector of two, cmp[1] - cmp[2] > 0 means up-regulation
# geneset: A vector of genes
calculate_es = function(expr, condition, cmp, geneset) {

    m1 = expr[, condition == cmp[1]]  # only samples in group 1
    m2 = expr[, condition == cmp[2]]  # only samples in group 2

    s = (rowMeans(m1) - rowMeans(m2))/(rowSds(m1) + rowSds(m2)) # a gene-level difference socre

    s = sort(s, decreasing = TRUE)  # ranked gene list

    l_set = names(s) %in% geneset
    f1 = cumsum(l_set)/sum(l_set)   # CDF for genes in the set

    l_other = !l_set
    f2 = cumsum(l_other)/sum(l_other)  # CDF for genes not in the set

    max(f1 - f2)
}
```

The ES score calculated by `calculate_es()`:

```
es = calculate_es(expr, condition, cmp = c("WT", "MUT"), geneset = geneset)
es
# [1] 0.2294643
```

We randomly permute sample labels or we randomly permute `condition`. We do it 1000 times. The ES scores in null distributions are saved in `es_rand`.

```
set.seed(123)
es_rand = numeric(1000)
for(i in 1:1000) {
    es_rand[i] = calculate_es(expr, sample(condition),
        cmp = c("WT", "MUT"), geneset = geneset)
}
```

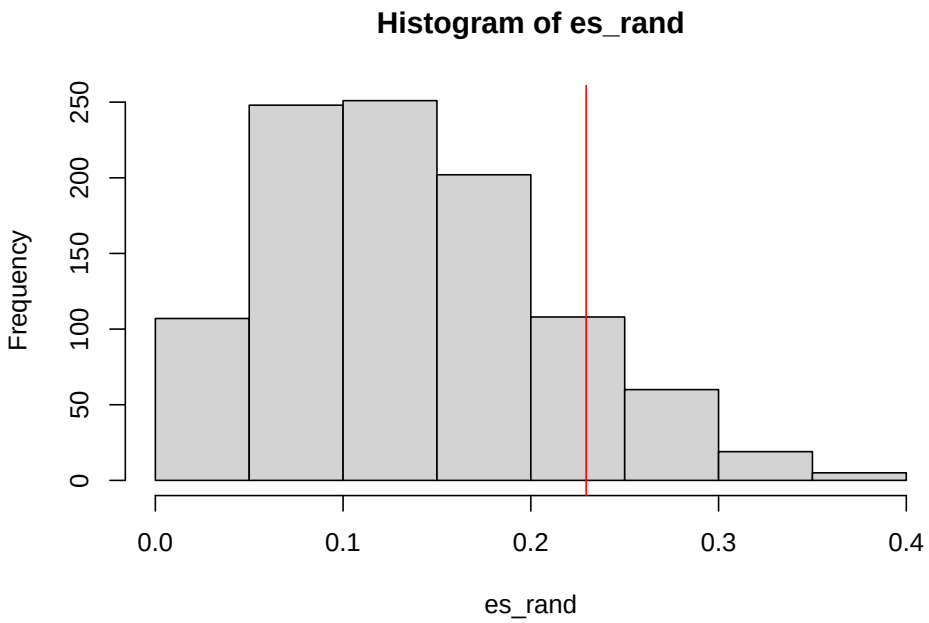
p-value is calculated as the proportion of `es` being equal to or larger than in values in `es_rand`.

```
sum(es_rand >= es)/1000
# [1] 0.129
```

0.129

The null distribution of ES:

```
hist(es_rand)
abline(v = es, col = "red")
```



Next we implement the improved GSEA (Subramanian et al., PNAS, 2005) where gene-level scores are taken as the weight.

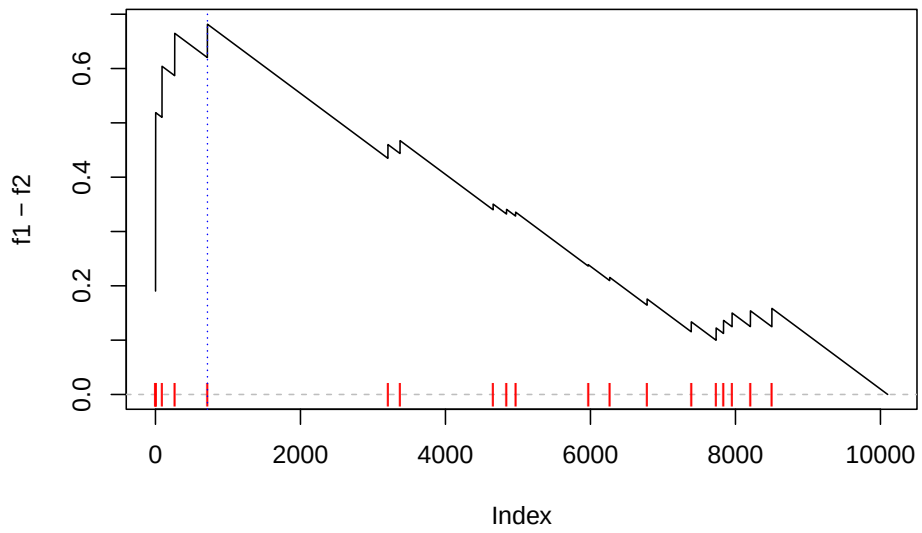
We directly modify `calculate_es()` to `calculate_es_v2()` where there is only two lines new, which we highlight in the code chunk:

```
calculate_es_v2 = function(expr, condition, cmp, geneset, plot = FALSE, power = 1) {

    m1 = expr[, condition == cmp[1]]
    m2 = expr[, condition == cmp[2]]

    s = (rowMeans(m1) - rowMeans(m2))/(rowSds(m1) + rowSds(m2))

    s = sort(s, decreasing = TRUE)

    l_set = names(s) %in% geneset
    # f1 = cumsum(l_set)/sum(l_set)  # <<-- the original line
    s_set = abs(s)^power   # <<-- here
    s_set[!l_set] = 0
    f1 = cumsum(s_set)/sum(s_set)  ## <<- here

    l_other = !l_set
    f2 = cumsum(l_other)/sum(l_other)
```

```
    if(plot) {
        plot(f1 - f2, type = "l")
        abline(h = 0, lty = 2, col = "grey")
        points(which(l_set), rep(0, sum(l_set)), pch = "|", col = "red")
        abline(v = which.max(f1 - f2), lty = 3, col = "blue")
    }

    max(f1 - f2)
}
```

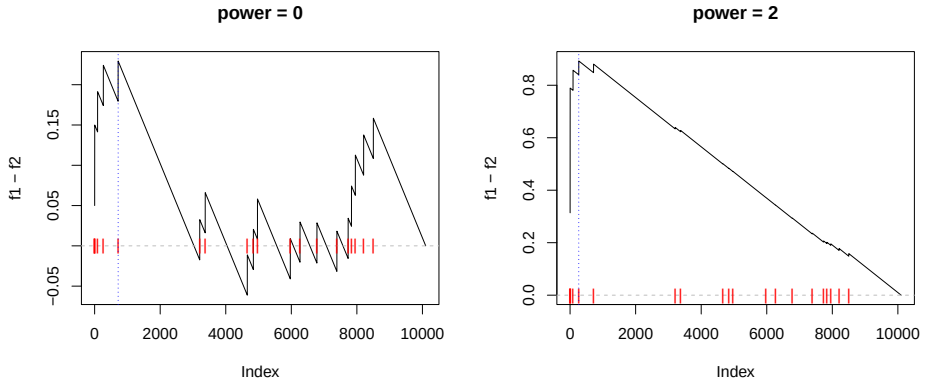
Now we calculate the new ES score and make the GSEA plot:

```
es = calculate_es_v2(expr, condition, cmp = c("WT", "MUT"), plot = TRUE,
    geneset = geneset)
```



We can also check when `power = 0` and `power = 2`:

```
par(mfrow = c(1, 2))
calculate_es_v2(expr, condition, cmp = c("WT", "MUT"), plot = TRUE, power = 0,
    geneset = geneset)  # same as the original GSEA
# [1] 0.2294643
title("power = 0")
calculate_es_v2(expr, condition, cmp = c("WT", "MUT"), plot = TRUE, power = 2,
    geneset = geneset)
# [1] 0.8925371
title("power = 2")
```




```
par(mfrow = c(1, 1))
```

Similarly, we randomly permute samples to obtain the null distribution of ES:

```
es_rand = numeric(1000)
for(i in 1:1000) {
    es_rand[i] = calculate_es_v2(expr, sample(condition),
        cmp = c("WT", "MUT"), geneset = geneset)
}
```

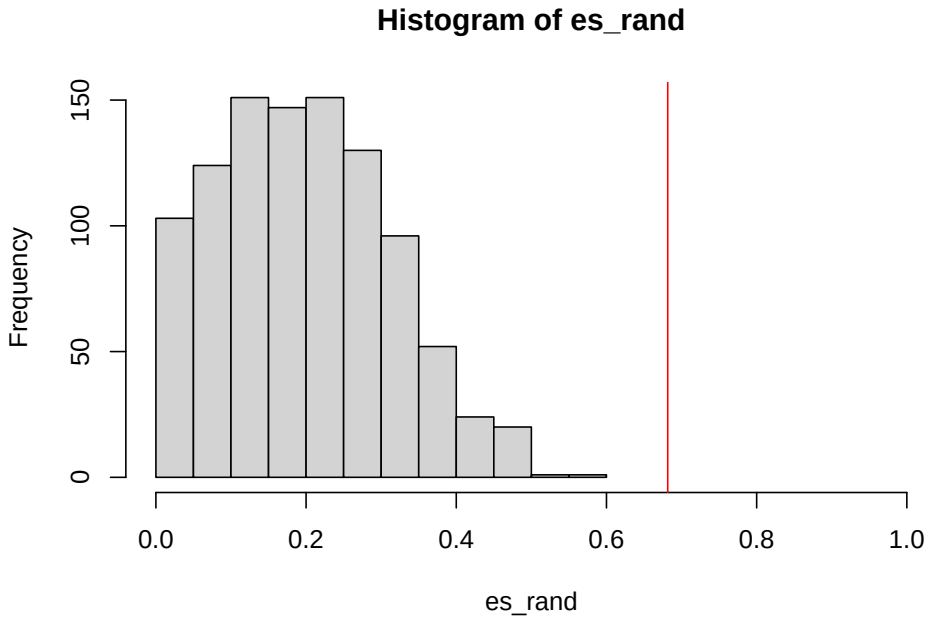
The new p-value:

```
sum(es_rand >= es)/1000
# [1] 0
```

$1/1000, < 0.001$

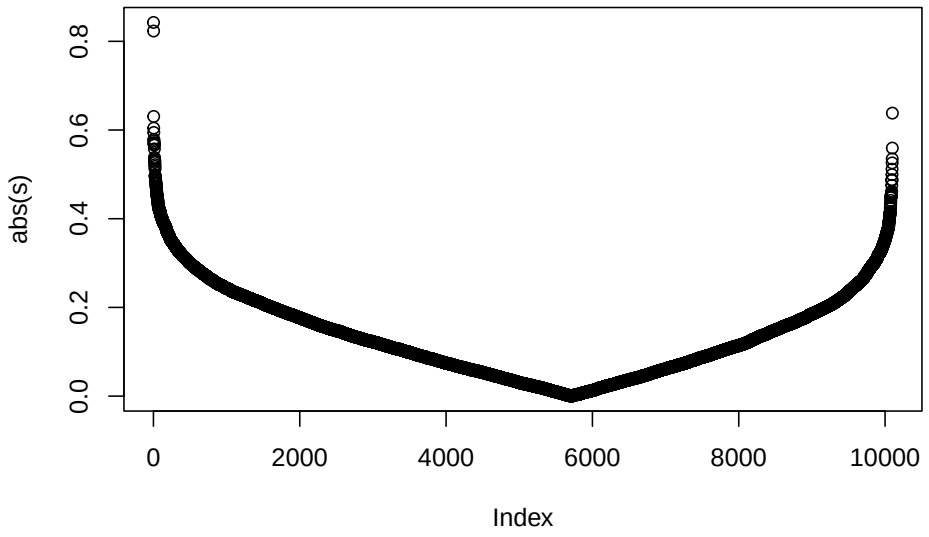
And the null distribution of ES:

```
hist(es_rand, xlim = c(0, 1))
abline(v = es, col = "red")
```



We can see the improved GSEA is more powerful than the original GSEA, because the original GSEA equally weights genes and the improved GSEA weights genes based on their differential expression, which increases the effect of diff genes. Let's plot the weight of genes:

```
plot(abs(s))
```



Null distribution can also be constructed by gene permutation. It is very easy to implement:

```
# s: a vector of pre-calcualted gene-level scores
# s should be sorted
calculate_es_v2_gene_perm = function(s, perm = FALSE, power = 1) {

    if(perm) {
        # s is still sorted, but the gene labels are randomly shuffled
        names(s) = sample(names(s))  ## <<- here
    }

    l_set = names(s) %in% geneset
    s_set = abs(s)^power
    s_set[!l_set] = 0
    f1 = cumsum(s_set)/sum(s_set)

    l_other = !l_set
    f2 = cumsum(l_other)/sum(l_other)

    max(f1 - f2)
}
```

Good thing of gene permutation is the gene-level scores only need to be calculated once and can be repeatedly used.

```
# pre-calculate gene-level scores
m1 = expr[, condition == "WT"]
m2 = expr[, condition == "MUT"]

s = (rowMeans(m1) - rowMeans(m2))/(rowSds(m1) + rowSds(m2))
s = sort(s, decreasing = TRUE)  # must be pre-sorted
```

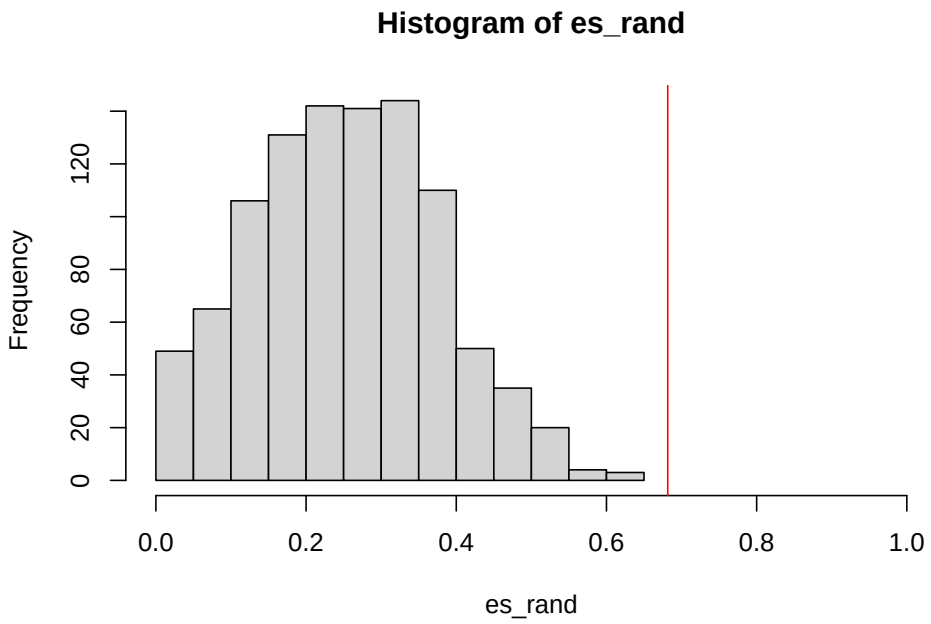
We calculate the null distribution of ES from gene permutation:

```
es = calculate_es_v2_gene_perm(s)
es_rand = numeric(1000)
for(i in 1:1000) {
    es_rand[i] = calculate_es_v2_gene_perm(s, perm = TRUE)
}

sum(es_rand >= es)/1000
# [1] 0
```

also < 0.001

The null distribution of ES from gene permutation:

```
hist(es_rand, xlim = c(0, 1))
abline(v = es, col = "red")
```



Chapter 6

GSEA framework

6.1 Overview

Recall in Chapter xx when we introduce the GSEA algorithm, the first step is to calcualte gene-level scores, then the gene-levels scores in a gene set is aggregated into a gene set-level score for testing. This procedure can be generalized into a general framework which includes calculation of gene level scores , gene set level scores and constructin of null distributions. According to how set-level statistic is designed, there are two main methologies: 1. univariate methods and 2. multivariate methods. where the first one is more used in current studies and the second one considered gene-gene correlation structures...

There are seveal reivews on the xxx.

6.2 The univariate methods

THe univariate methods is a two-step methods where the expression matrix is first merged into a gene-level scores which measures the gene-level differential expression. Later for each gene set, the member genes are merged into a single statistic where the permutation test is applied to construct the null distribution.

In two steps, the gene-gene correlation strucutre is actually not taken into consideration, thus it follows the univriate procedure where no co-variants are considered.

6.2.1 Gene-level methods

The first step of the univarirate procedures is to calculate the gene-level statistic, again, to make the discussion simple, we assume the data is from a two-condition comparison. and later we will demonstrate more complicated experimental designs. Let's denote $\mathbf{x_1}$ as the vector of gene expression for gene i in group 1,

and $\mathbf{x_2}$ as the vector of gene expression for gene i in group 2. The gene-level transformation is a function $f()$ which applies on $\mathbf{x_1}$ and $\mathbf{x_2}$ togenerate a single value:

$$f(\mathbf{x_1}, \mathbf{x_1}) -> asinglevalue$$

There are variaty of methods that can be used to calculate gene-level scores. The only thing is that the score measures the degree of differnetial expression, thus high the absolute value of the score, more differnetial the gene is expressed. Commonly used methods are:

1. t-value, defined as .
2. log2 fold change
3. signal-to-noise ratio
4. SAM regularized t-values

In xxx, lists more gene-level statistics. However, the authors demonstarted the selection of gene-level statistic is less important for GSEA. The main reason is the method is always applied to the two vectors and without extra information, they performs similar. but still there are difference between various methods.

– scater plot

More generally, the relation between gene expression and the condition design can be modeled via linear regression. With linear regression, we can deal with more complicated experimental designs, such as 1. conditions with more than 2, 2. with continouse conditions e.g. age or dose treatment, 3. time series data, 4 multiple condition variates and 5. it allows a full model vs a reduced model to test the partial effect of a variate. For univariet,

$$y = \mu + \beta x + \epsilon$$

where x is the condition variable, which can be categorical variable, continuous variable or time series variable. Or with multiple variates:

$$y = \mu + \beta_1 x_1 + \beta_2 x_2 + ... + \epsilon$$

The gene-level statistic can use the regression coeffcient to other statistics which measures the goodness of the liear fitting. However, users need to be careful that more complex models make the results less explainable.

6.2.2 Transformation of gene-level statistics

This step is optional and can be merged in to the previous step. With obtaining a vector of gene-level scores, we can apply proper transformation to it. The transformation is also a function which takes the original gene-level scores in

and it outputs a new gene-level scores. There are the followng four widel y used transformations

- absolute
- square or power
- binary
- rank

The absolute or square transformation helps to capture the bi-directional changes and the square transformation can addtionally increase the weight of more differetiaally expressed genes. Binary transformation corresponds to ORA analysis and rank transformaton is used in GSEA.

Please note, till now we haven't used any information of gene sets. The calculation of gene-level statistics is independen of gene sets. It is just a processing of the original matrix, In other words, it is a transformation of the original nxm matrix to a n vector. and in the next step, genes for gene sets are extracted from this gene-level vectors to calcualte into a gene set level statistic.

6.2.3 Set-level methods

With gene-level scores calculated already, we can check whether genes in a gene sets are in general differnetial expressed. Regarding what to compare, ie. the background, there are the following two scenarios:

1. only consider genes in the gene set, in this case, the "background" is that all genes are not differnetially expressed. If there is a function for calculating gene-set level scores, it only takes a vector as input, which is the gene-level scores for genes in the current gene set.
2. COmpare genes in the gene set and genes not in the gene set. Now the background is other genes. In this case, the gene set level function takes two vectors as input.

The two scenarions actaully measures different things, of which users need to keep in mide when they use corresponding methods.

For the first case, where the gene set level statistic is only calcualted in the gene set, it basically measures the overall difference of genes in the gene set. It is a aggregation of gene-level statstis into a singl set0-level statistic. There are the following widely used methods:

- Sum / Mean
- Median
- Maxmean

Note genes in a gene set may also show bi-directional pattern, which is some genes are up-regulated and some are down-regulated. Sum/Mean/Median are good at measure one-direcitonal change, but bi-direcitional change wil be cancled, thus get a very small value. This problem can be fixed by first apply a absolute transformation on gene-level scores.

Maxmean method was proposed in xxx. It basically takes the max mean of positive vlaues and negative values.

For the second scenario where also compare to genes not in the gene set. Denote the gene-level statistis for genes in the set as r_1 and not in the set as r_0, teh set-level statistic measures the difference between r_1 and r_0. There are the following three methods:

1. (weighted) KS statistic
2. Wilcoxon-rank statistic
3. 2x2 contigency table

wiht the two vectors of r_1 and r_0, normally non-paremetric statistics are used, but for some methods which assume the normality, parametric test such as t-test can be used. But ...

The 2x2 contigen table for ORA analysis actually also compare genes in the set and genes not in the set.

It works better when the differential expression for genes not in the set are weak. However, when a data has huge number of genes that are differnetial expressed e.g. cancer vs normal. this method may give wrong conclusions. e.g. when a gene set is significant, reseachers may make the conclusion that the biological function for this gnee set is significantly altered, but actually it xxx

Finally, it is very flexible to design a set-level statistic. There is only one principle for that: higher the value, more differnetial the genes in the gene set are.

6.2.4 Current tools

Many current tools follows the univarate frameworks.

6.2.5 Implement ORA under univariate framework

The ORA can be implemented under teh univate framework. In the 2x2 contigency table, the numer of genes in the gene set n_{11} actually can be used as the set-level statistic becuase higher the n_{11}, the more enriched xxx. Then to translate into the univariate framework, we have:

- gene-level statistics: p-value/FDR from the differential expression test
- gene-level transformation: binary by setting a cutoff where e.g. if FDR $<$ 0.05, 1, else 0)
- set-level statistics: sum, which is n_{11}

Note here n_{11} only used genes in the gene set.

We also introduced using chi-square test on the 2x2 contigency table, Thus, the Chi-suqare statistic can be used as teh set-level statistic. being differnet fro the first ORA translation, the chi-square test acautally compares genes in the gene set and not in the gene set.

The third way is, in the begnining when we introduce what is the overpresentation, we simply defined a value which is the fraction of two ratios. That value can also be a set-level statitic. Note this statistics alsoused information of genes in the set and not in the set.

For xxx, the p-values is calculatd as the probability of being larger than or equal to the observed values. When the set-level statists is only for one-directional change, and to see the power of down-regulation. Users may need to perform a second aalyais which looks at the other tail of hte null distribution.

6.2.6 Null distribution of set-level statistics

Once we have the set-level statistic, we need to construct the null distribution of it for calculting p-values. depends on the methods, there are teh following three ways:

1. exact distribution
2. parametric distribution
3. permutation-based distributino

hypergeometric distribution is the exact distribution for ORA, however, it is normally impossbiel to obtain exact distibutions. Also users recall the hypergeometrix distribution has strong conditions that genes are independenly to be differnetially expressed, which may not fit the real cases.

Parametrix distribution needs assumptions of the distribution whihc is normally normal distributions for univaraite framework, but xxx. THe most used is the permuation-based distribution which construct the distribution directly from the data.

6.2.7 Permutation-based distribution

The permutaion generates null distribution from data, by permuting the origial data set to destroy the dependencies between xx and xx. It is non-parametrix and has no assumption of proir distribion. ALso there are three permutation methods:

1. permute by samples
2. permute by genes
3. permute by both dimensions

In general, permutation is a powerful method to generate null distribution and to calculate p-values. But sinde the distribution

The two permutations, although they all randomize the data, they actually correspond to two very different null hypotheses:

for sample permutation, It only looks at the genes in the geneset and the null hypothesis is:

gene expression are not related to the condition settings.

In xxx, it is termed as self-contained.

For the gene permutation, it compares genes in the gene set and genes not in the gene set. The null hypotheses is :

differential gene expression is the gene set are the same as genes outside of the set.

Users can imagien one is horizontal comparison and the other is vertical comparison.

The two permutations are all widely used in current tools, each one has its advantages and disadvanges. For sample permutation, the main advantages ae:

- the gene-gene correlation is kept
- THe permutation has a clear and real biological meaning

THe distadvantages are

- Each time, gene-level statistics need to be recalculated
- more sensitive, assume only one genes in the set are differnetially expressed, the whole gene set can be assessed as significant.

For the gene permutation. the advantage is

- the gene-level statistics can be repeated used. Or in other words, the calculation of gene-level scores can be separated from GSEA where the input can just be a gene-level score vector.

The disadvanges are:

- the permutaiton assumes independency of genes and gene-gene correlation in the gene set are broken
- null distribution has no clear biological meanings
- It may work better when gene set is small and most of genes have no diffenritla expression in thie whote dataset.

It would be interesting to compare the sample permutation and gene permutation with a realworld data. Here the P53 dataset which was used in the original GSEA paper where the two groups are P53 wild type and P53 mutant. We take the “P53 pathway” gene set" which are expected to be significant.

to compare ... assume s is the set-level score calclated from the real data and s_{null} is a vector of length 1000 calculated from permutation by sample or by gene, to compare between different methods, we calculated a relative value:

$$\frac{s - mean_null}{sd_null}$$

which is a relative distance fro s to the center of null distribution.

In general, sample permutation is more statistically pwoerful than gene-permutation.

6.3 The multivariate methods

A second framework is the multivariate methods which takes in matrix as a singl input and returns the xx. They basically consider the co-variance in the data.

Normally, multivaraite methods are complex. They use complicated statistic computations trying to etablish a statsitcal framwork and ...

- globaltest
- GlobalANCOVA
- Hotelling' T2 test

Since their parametric nature, they need to assumptions, someti

The basically idea of multivariate methods is to follow either the following two linear models:

$$A = bC + e$$
$$C = bA + e$$

where A and C are both matrices. to solve the problem, we can use pricinpal componet regression, partially lease square regression or regulazeid linear regression.

6.4 Implementation of GSEA framework

```
library(CePa)
condition = read.cls("data/P53.cls", treatment = "MUT", control = "WT")$label
condition = factor(condition, levels = c("WT", "MUT"))

expr = read.gct("data/P53_collapsed_symbols.gct")
```

The process of (univariate) GSEA analysis:

So basically, the calculation of gene-level statistics and set-level statistics can be separated.

We will implement the three independent parts: $f()$, $f'()$ and $g()$.

We first implement the function that calculates gene-level statistics. To make things simple, we assume the matrix is from a two-condition comparison.

- input: an expresion matrix (with condition labels)
- output: a vector of gene-level scores

- method: (t-value, log2fc, ...)

```
# implement t-values as gene-level stat
gene_level = function(mat, condition) {

  tdf = genefilter:rowttests(mat, factor(condition))
  stat = tdf$statistic
  return(stat)
}
```

```
library(matrixStats)
library(genefilter)

# -condition to be a factor
gene_level = function(mat, condition, method = "tvalue") {

    le = levels(condition)
    l_group1 = condition == le[1]
    l_group2 = !l_group1

    mat1 = mat[, l_group1, drop = FALSE]  # sub-matrix for condition 1
    mat2 = mat[, l_group2, drop = FALSE]  # sub-matrix for condition 2

    if(method == "log2fc") {
        stat = log2(rowMeans(mat1)/rowMeans(mat2))
    } else if(method == "s2n") {
        stat = (rowMeans(mat1) - rowMeans(mat2))/(rowSds(mat1) + rowSds(mat2))
    } else if(method == "tvalue") {
        stat = (rowMeans(mat1) - rowMeans(mat2))/sqrt(rowVars(mat1)/ncol(mat1) + rowVa
    } else if(method == "sam") {
        s = sqrt(rowVars(mat1)/ncol(mat1) + rowVars((mat2)/ncol(mat2)))
        stat = (rowMeans(mat1) - rowMeans(mat2))/(s + quantile(s, 0.1))
    } else if(method == "ttest") {
        stat = rowttests(mat, factor(condition))$p.value
    } else {
        stop("method is not supported.")
    }

    return(stat)
}
```

```
s = gene_level(expr, condition, method = "s2n")
```

The transformation on gene-level values can actually be integrated as a part of the calculation of gene-level values, i.e. $f'(f())$ can also be thought as a gene-level statistic.

if the gene-level stat is p-values we need to set a cutoff of p to convert to 1/0

binarize() input: is the origial gene-level stat (e.g. p-value) ouput: the values are 1/0

binarize = function(x) ifelse(x < 0.05, 1, 0)

if the gene-level stat is log2fc

binarize = function(x) ifelse(abs(x) > 1, 1, 0)

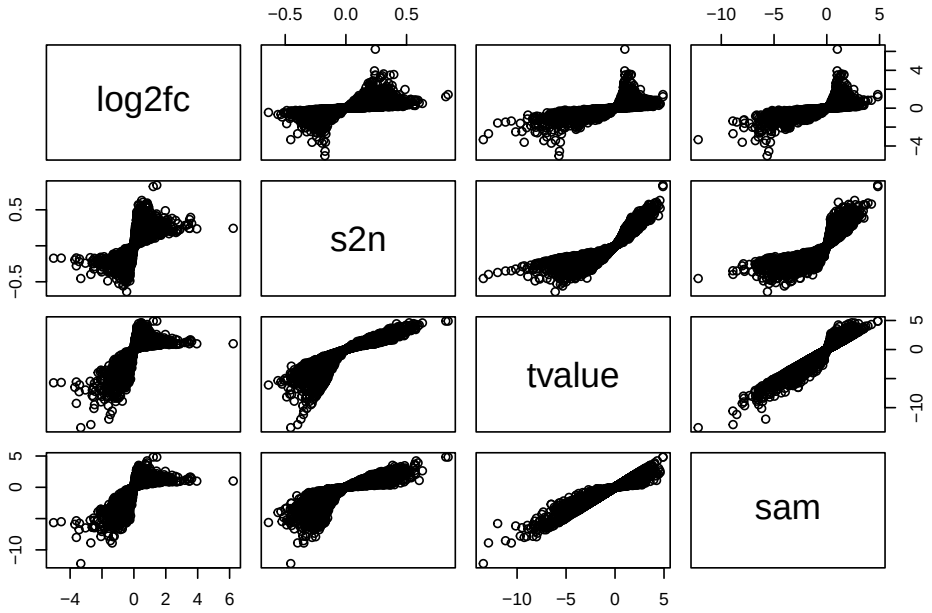
```
gene_level = function(mat, condition, method = "tvalue", transform = "none",
  binarize = function(x) x) {

    le = levels(condition)
    l_group1 = condition == le[1]
    l_group2 = !l_group1

    mat1 = mat[, l_group1, drop = FALSE]
    mat2 = mat[, l_group2, drop = FALSE]

    if(method == "log2fc") {
        stat = log2(rowMeans(mat1)/rowMeans(mat2))
    } else if(method == "s2n") {
        stat = (rowMeans(mat1) - rowMeans(mat2))/(rowSds(mat1) + rowSds(mat2))
    } else if(method == "tvalue") {
        stat = (rowMeans(mat1) - rowMeans(mat2))/sqrt(rowVars(mat1)/ncol(mat1) + rowVars((mat2)/r
    } else if(method == "sam") {
        s = sqrt(rowVars(mat1)/ncol(mat1) + rowVars((mat2)/ncol(mat2)))
        stat = (rowMeans(mat1) - rowMeans(mat2))/(s + quantile(s, 0.1))
    } else if(method == "ttest") {
        stat = rowttests(mat, factor(condition))$p.value
    } else {
        stop("method is not supported.")
    }

    if(transform == "none") {

    } else if(transform == "abs") {
        stat = abs(stat)
    } else if(transform == "square") {
        stat = stat^2
    } else if(transform == "binary") {
        stat = binarize(stat)
    } else {
        stop("method is not supported.")
    }

    return(stat)
```

```
}
```

Let's test `gene_level()`. Here we still use the p53 dataset which is from the GSEA original paper.

Let's check the gene-level values:

```
methods = c("log2fc", "s2n", "tvalue", "sam")
lt = lapply(methods, function(x) gene_level(expr, condition, method = x))
names(lt) = methods
pairs(lt)
```



Also we can check number of differential genes (gene level: `ttest` + transform: `binary`). Note a better way is to filter by FDR, but for simplicity, we use p-values directly.

```
s = gene_level(expr, condition, method = "ttest", transform = "binary",
    binarize = function(x) ifelse(x < 0.05, 1, 0))
table(s)
# s
#    0    1
# 9394  706
```

If method is set to `log2fc`, then the differential genes can be selected by setting a cutoff for log2 fold change.

```
s = gene_level(expr, condition, method = "log2fc", transform = "binary",
    binarize = function(x) ifelse(abs(x) > 1, 1, 0))
table(s)
# s
#    0    1
# 9717  383
```

Implementing `gene_level()` is actually simply.

Next we implement the calculation of set-level statistics. A nature design for the set-level function is to let it accept a vector of gene-level statistics and a gene set represented as a vector of genes, like follows:

```
set_fun = function(gene_stat, geneset) {
    s = gene_stat[geneset]
    mean(s)
}
```

However, we need to make sure all genes in `geneset` are also in `gene_stat`. A safer way is to test which genes in `gene_stat` are also in `geneset`:

```
set_fun = function(gene_stat, geneset) {
    s = gene_stat[ names(gene_stat) %in% geneset ]
    mean(s)
}
```

However, recall the set-level can also be calculated based on genes outside of the gene set. Thus the two arguments in `set_level()` are a vector of gene-level statistics for all genes and a logical vector which shows whether genes in the current gene set. In this setting, we can know both which genes are in the set and which genes are not in the set.

before `geneset` is a vector of gene IDs now `l_set`: a logical vector, which has the same length of `gene_stat`, if the value is TRUE, it means the gene is in the set, if it is FALSE, the gene is NOT in the set

```
set_level = function(gene_stat, l_set, method = "mean") {
    if(!any(l_set)) {
        return(NA)
    }

    if(method == "mean") {
        stat = mean(gene_stat[l_set])
    } else if(method == "sum") {
        stat = sum(gene_stat[l_set])
    } else if(method == "median") {
        stat = median(gene_stat[l_set])
    } else if(method == "maxmean") {
        s = gene_stat[l_set]
```

```
        s1 = mean(s[s > 0]) # s1 is positive
        s2 = mean(s[s < 0])  # s2 is negative
        stat = ifelse(s1 > abs(s2), s1, s2)
    } else if(method == "ks") {
        # order gene_stat
        od = order(gene_stat, decreasing = TRUE)
        gene_stat = gene_stat[od]
        l_set = l_set[od]

        s_set = abs(gene_stat)
        s_set[!l_set] = 0
        f1 = cumsum(s_set)/sum(s_set)

        l_other = !l_set
        f2 = cumsum(l_other)/sum(l_other)

        stat = max(f1 - f2)
    } else if(method == "wilcox") {
        stat = wilcox.test(gene_stat[l_set], gene_stat[!l_set])$statistic
    } else if(method == "chisq") {
        # should on work with binary gene-level statistics
        stat = chisq.test(factor(gene_stat), factor(as.numeric(l_set)))$statistic
    } else {
        stop("method is not supported.")
    }

    return(stat)
}
```

Let's check `set_level()`:

input of `set_level()`:

1. a gene-level scores
2. a logical vector which shows whether the genes are in a set

```
gene_stat = gene_level(expr, condition)

ln = strsplit(readLines("data/c2.symbols.gmt"), "\t")
gs = lapply(ln, function(x) x[-(1:2)])
names(gs) = sapply(ln, function(x) x[1])

geneset = gs[["p53hypoxiaPathway"]]
l_set = rownames(expr) %in% geneset
set_level(gene_stat, l_set)
# [1] 0.8476668
set_level(gene_stat, l_set, method = "ks")
```

```
# [1] 0.6380268
```

Now we can wrap `gene_level()` and `set_level()` into a single function `gsea_tiny()` which accepts the expression and one gene set as input, and it returns the set-level score.

gsea_tiny():

- expression matrix (condition labels)
- a gene set

output: a set-level statistic

gsea_tiny() [gene_level() + set_level()]

```
gsea_tiny = function(mat, condition,
    gene_level_method = "tvalue", transform = "none", binarize = function(x) x,
    set_level_method = "mean", geneset) {

    gene_stat = gene_level(mat, condition, method = gene_level_method,
        transform = transform, binarize = binarize)

    l_set = rownames(mat) %in% geneset

    set_stat = set_level(gene_stat, l_set, method = set_level_method)

    return(set_stat)
}
```

We apply `gsea_tiny()` to the p53 dataset.

```
gsea_tiny(expr, condition, geneset = geneset)
# [1] 0.8476668
```

We use `wilcox.test()` to calculate the Wilcoxon statistic. Note this function also does a lot of extra calculations. We can implement a function which "just" calculates the Wilcoxon statistic but do nothing else:

The formula is from Wikipedia (https://en.wikipedia.org/wiki/Mann%E2%80%93Whitney_U_test).

Note, to make `wilcox_stat()` faster, we only use maximal 100 data points. It is only for demonstration purpose, you should not use it in real applications.

"outer" calculation

x, y every value in x to every value in y

if length(x) is n, length(y) is m

n*m

`outer()`

m = outer(x, y, ">")

```
wilcox_stat = function(x1, x2) {
  if(length(x1) > 100) {
    x1 = sample(x1, 100)
  }
  if(length(x2) > 100) {
    x2 = sample(x2, 100)
  }
  sum(outer(x1, x2, ">"))
}
```

Similarly, we implement a new function which only calculates chi-square statistic:

```
# x1: a logical vector or a binary vector
# x2: a logical vector or a binary vector
chisq_stat = function(x1, x2) {
    n11 = sum(x1 & x2)
    n10 = sum(x1)
    n20 = sum(!x1)
    n01 = sum(x2)
    n02 = sum(!x2)
    n = length(x1)

    n12 = n10 - n11
    n21 = n01 - n11
    n22 = n20 - n21

    p10 = n10/n
    p20 = n20/n
    p01 = n01/n
    p02 = n02/n

    e11 = n*p10*p01
    e12 = n*p10*p02
    e21 = n*p20*p01
    e22 = n*p20*p02

    stat = (n11 - e11)^2/e11 +
           (n12 - e12)^2/e12 +
           (n21 - e21)^2/e21 +
           (n22 - e22)^2/e22
    return(stat)
}
```

and we change `set_level()` accordingly:

```
set_level = function(gene_stat, l_set, method = "mean") {
    if(!any(l_set)) {
        return(NA)
    }

    if(method == "mean") {
        stat = mean(gene_stat[l_set])
    } else if(method == "sum") {
        stat = sum(gene_stat[l_set])
    } else if(method == "median") {
        stat = median(gene_stat[l_set])
    } else if(method == "maxmean") {
        s = gene_stat[l_set]
        s1 = mean(s[s > 0])
        s2 = mean(s[s < 0])
        stat = ifelse(s1 > abs(s2), s1, s2)
    } else if(method == "ks") {
        # order gene_stat
        od = order(gene_stat, decreasing = TRUE)
        gene_stat = gene_stat[od]
        l_set = l_set[od]

        s_set = abs(gene_stat)
        s_set[!l_set] = 0
        f1 = cumsum(s_set)/sum(s_set)

        l_other = !l_set
        f2 = cumsum(l_other)/sum(l_other)

        stat = max(f1 - f2)
    } else if(method == "wilcox") {
        stat = wilcox_stat(gene_stat[l_set], gene_stat[!l_set])
    } else if(method == "chisq") {
        # should on work with binary gene-level statistics
        stat = chisq_stat(gene_stat, l_set)
    } else {
        stop("method is not supported.")
    }

    return(stat)
}
```

Next we will adjust `gsea_tiny()` to let it work for multiple gene sets and support random permutation for p-value calculation.

To let is support a list of gene sets, simply change the format of `geneset` variable.

6.5 geneset to be a list of gene sets

```
# geneset: a list of vectors (gene IDs)
gsea_tiny = function(mat, condition,
    gene_level_method = "tvalue", transform = "none", binarize = function(x) x,
    gene_stat, set_level_method = "mean", geneset) {

    gene_stat = gene_level(mat, condition, method = gene_level_method,
        transform = transform, binarize = binarize)

    set_stat = sapply(geneset, function(set) {
        l_set = rownames(mat) %in% set

        set_level(gene_stat, l_set, set_level_method)
    })

    return(set_stat)
}
```

Check the new version of `gsea_tiny()`:

```
ss = gsea_tiny(expr, condition, geneset = gs)
head(ss)
#          41bbPathway          ace2Pathway acetaminophenPathway
#           -0.3125486            1.2710342            0.3441254
#           achPathway
#           -0.2944815
```

Now with `gsea_tiny()`, we can also generate the null distribution of the set-level statistics, just by generating random matrices.

```
# sample permutation
ss_random = list()
for(i in 1:1000) {
    ss_random[[i]] = gsea_tiny(mat, sample(condition), geneset = gs)
}

# or gene permutation
for(i in 1:1000) {
    mat2 = mat
    rownames(mat2) = sample(rownames(mat))
    ss_random[[i]] = gsea_tiny(mat2, condition, geneset = gs)
}
```

A better design is to integrate permutation procedures inside `gsea_tiny()`. We first integrate sample permutation:

```
gsea_tiny = function(mat, condition,
    gene_level_method = "tvalue", transform = "none", binarize = function(x) x,
    gene_stat, set_level_method = "mean", geneset,
    nperm = 1000) {

    gene_stat = gene_level(mat, condition, method = gene_level_method,
        transform = transform, binarize = binarize)

    set_stat = sapply(geneset, function(set) {
        l_set = rownames(mat) %in% set

        set_level(gene_stat, l_set, set_level_method)
    })

    ## null distribution
    set_stat_random = list()

    for(i in seq_len(nperm)) {
        condition2 = sample(condition)
        gene_stat = gene_level(mat, condition2, method = gene_level_method,
            transform = transform, binarize = binarize)

        set_stat_random[[i]] = sapply(geneset, function(set) {
            l_set = rownames(mat) %in% set

            set_level(gene_stat, l_set, set_level_method)
        })

        if(i %% 100 == 0) {
            message(i, " permutations done.")
        }
    }

    set_stat_random = do.call(cbind, set_stat_random)

    n_set = length(geneset)
    p = numeric(n_set)
    for(i in seq_len(n_set)) {
        p[i] = sum(set_stat_random[i, ] >= set_stat[i])/nperm
    }

    # the function returns a data frame
    df = data.frame(stat = set_stat,
                    size = sapply(geneset, length),
                    p.value = p)
```

```
    df$fdr = p.adjust(p, "BH")

    return(df)
}
```

Let's have a try. It is actually quite slow to run 1000 permutations.

```
df = gsea_tiny(expr, condition, geneset = gs)
```

This is the basic procedures of developing new R functions. First we make sure the functions are working, next we optimize the functions to let them running faster or use less memory.

`gsea_tiny()` running with 100 permutations only needs several seconds.

```
df = gsea_tiny(expr, condition, geneset = gs, nperm = 100)
```

The package **profvis** provides an easy to for profiling.

```
library(profvis)
profvis(gsea_tiny(expr, condition, geneset = gs, nperm = 100))
```

We can see the process of `%in%` uses quite a lot of running time.

we can first calculate the relations of genes and sets and later they can be repeatedly used.

```
gsea_tiny = function(mat, condition,
    gene_level_method = "tvalue", transform = "none", binarize = function(x) x,
    gene_stat, set_level_method = "mean", geneset,
    nperm = 1000) {

    gene_stat = gene_level(mat, condition, method = gene_level_method,
        transform = transform, binarize = binarize)

    # now this only needs to be calculated once
    l_set_list = lapply(geneset, function(set) {
        rownames(mat) %in% set
    })

    set_stat = sapply(l_set_list, function(l_set) {
        set_level(gene_stat, l_set, set_level_method)
    })

    ## null distribution
    set_stat_random = list()

    for(i in seq_len(nperm)) {
        condition2 = sample(condition)
```

```
        gene_stat_random = gene_level(mat, condition2, method = gene_level_method,
            transform = transform, binarize = binarize)

        # here we directly use l_set_list
        set_stat_random[[i]] = sapply(l_set_list, function(l_set) {
            set_level(gene_stat_random, l_set, set_level_method)
        })

        if(i %% 100 == 0) {
            message(i, " permutations done.")
        }
    }

    set_stat_random = do.call(cbind, set_stat_random)

    n_set = length(geneset)
    p = numeric(n_set)
    for(i in seq_len(n_set)) {
        p[i] = sum(set_stat_random[i, ] >= set_stat[i])/nperm
    }

    df = data.frame(stat = set_stat,
                    size = sapply(geneset, length),
                    p.value = p)
    df$fdr = p.adjust(p, "BH")

    return(df)
}
```

Now it is faster for 1000 permutations:

```
df = gsea_tiny(expr, condition, geneset = gs)
```

To support gene permutation, we only need to permute the gene-level statistics calculated from the original matrix. Note we also move position of `geneset` argument to the start of the argument list because it is a must-set argument.

```
gsea_tiny = function(mat, condition, geneset,
    gene_level_method = "tvalue", transform = "none", binarize = function(x) x,
    gene_stat, set_level_method = "mean",
    nperm = 1000, perm_type = "sample") {

    gene_stat = gene_level(mat, condition, method = gene_level_method,
        transform = transform, binarize = binarize)
    l_set_list = lapply(geneset, function(set) {
        rownames(mat) %in% set
    })
```

```
set_stat = sapply(l_set_list, function(l_set) {
    set_level(gene_stat, l_set, set_level_method)
})

## null distribution
set_stat_random = list()

for(i in seq_len(nperm)) {

    if(perm_type == "sample") {
        condition2 = sample(condition)
        gene_stat_random = gene_level(mat, condition2, method = gene_level_method,
            transform = transform, binarize = binarize)

        set_stat_random[[i]] = sapply(l_set_list, function(l_set) {
            set_level(gene_stat_random, l_set, set_level_method)
        })
    } else if(perm_type == "gene") {
        gene_stat_random = sample(gene_stat)

        set_stat_random[[i]] = sapply(l_set_list, function(l_set) {
            set_level(gene_stat_random, l_set, set_level_method)
        })
    } else {
        stop("wrong permutation type.")
    }

    if(i %% 100 == 0) {
        message(i, " permutations done.")
    }
}

set_stat_random = do.call(cbind, set_stat_random)

n_set = length(geneset)
p = numeric(n_set)
for(i in seq_len(n_set)) {
    p[i] = sum(set_stat_random[i, ] >= set_stat[i])/nperm
}

df = data.frame(stat = set_stat,
                size = sapply(geneset, length),
                p.value = p)
df$fdr = p.adjust(p, "BH")
```

```
    return(df)
}
```

Let's check:

```
set.seed(123)
df1 = gsea_tiny(expr, condition, geneset = gs, perm_type = "sample")
df2 = gsea_tiny(expr, condition, geneset = gs, perm_type = "gene")
```

```
df1 = df1[order(df1$p.value), ]
df2 = df2[order(df2$p.value), ]
head(df1)
#                         stat size p.value fdr
# hsp27Pathway       1.1077565   16       0   0
# p53hypoxiaPathway 0.8476668   20       0   0
# p53Pathway         1.1781147   16       0   0
# HTERT_DOWN         0.3002211   67       0   0
head(df2)
#                     stat size p.value fdr
# ace2Pathway    1.2710342   12       0   0
# inflamPathway 0.7852193   29       0   0
# p53Pathway     1.1781147   16       0   0
# P53_UP         0.9864715   40       0   0
```

Note, above settings can only detect the up-regulated gene sets.

Great! If we think each combination of gene-level method, gene-level transformation and set-level method is *a GSEA method*, then our `gsea_tiny()` actually already support many GSEA methods! The whole functionality only contains 180 lines of code (https://gist.github.com/jokergoo/e8fff4a57ec59efc694b9e730da22b9f).

6.6 current tools for GSEA framework

Package **EnrichmentBrowser** integrates a lot of GSEA methods. The integrated methods are:

```
library(EnrichmentBrowser)
sbeaMethods()
#  [1] "ora"        "safe"       "gsea"       "gsa"        "padog"
#  [6] "globaltest" "roast"      "camera"     "gsva"       "samgs"
# [11] "ebm"        "mgsa"
```

EnrichmentBrowser needs a special format (in `SummarizedExperiment`) as input. Condition labels should be stored in a column "GROUP." Log2 fold change and adjusted p-values should be saved in "FC" and "ADJ.PVAL" columns.

```
library(CePa)
condition = read.cls("data/P53.cls", treatment = "MUT", control = "WT")$label
expr = read.gct("data/P53_collapsed_symbols.gct")


library(SummarizedExperiment)
se = SummarizedExperiment(assays = SimpleList(expr = expr))
colData(se) = DataFrame(GROUP = ifelse(condition == "WT", 1, 0))

l = condition == "WT"

library(genefilter)
tdf = rowttests(expr, factor(condition))
rowData(se) = DataFrame(FC = log2(rowMeans(expr[, l])/rowMeans(expr[, !l])),
                        ADJ.PVAL = p.adjust(tdf$p.value))
se
# class: SummarizedExperiment
# dim: 10100 50
# metadata(0):
# assays(1): expr
# rownames(10100): TACC2 C14orf132 ... AMACR LDLR
# rowData names(2): FC ADJ.PVAL
# colnames: NULL
# colData names(1): GROUP
```

Note, to run `eaBrowse()`, you need to explicitly convert to ENTREZID.

```
se = idMap(se, org = "hsa", from = "SYMBOL", to = "ENTREZID")  # !! Gene ID must be co
```

We load the hallmark gene sets by package **msigdbr**.

When using the Entrez ID, make sure the "numbers" are converted to "characters."

```
library(msigdbr)
gs = msigdbr(category = "H")
gs = split(gs$human_entrez_gene, gs$gs_name)  # Entrez ID must be used
gs = lapply(gs, as.character)                 # be careful Entrez ID might be wrongly
```

Simply call `sbea()` function with a specific method:

Note now you can use `eaBowse(res)` to create the tiny website for detailed results.

```
res = sbea(method = "gsea", se = se, gs = gs)
tb = gsRanking(res, signif.only = FALSE)
```

Next we run all supported GSEA methods in **EnrichmentBrowser**.

```
all_gsea_methods = sbeaMethods()
all_gsea_methods
#  [1] "ora"        "safe"       "gsea"       "gsa"        "padog"
#  [6] "globaltest" "roast"      "camera"     "gsva"       "samgs"
# [11] "ebm"        "mgsa"
```

```
all_gsea_methods = setdiff(all_gsea_methods, "padog")
res_list = lapply(all_gsea_methods, function(method) {
    sbea(method = method, se = se, gs = gs)
})
names(res_list) = all_gsea_methods
```

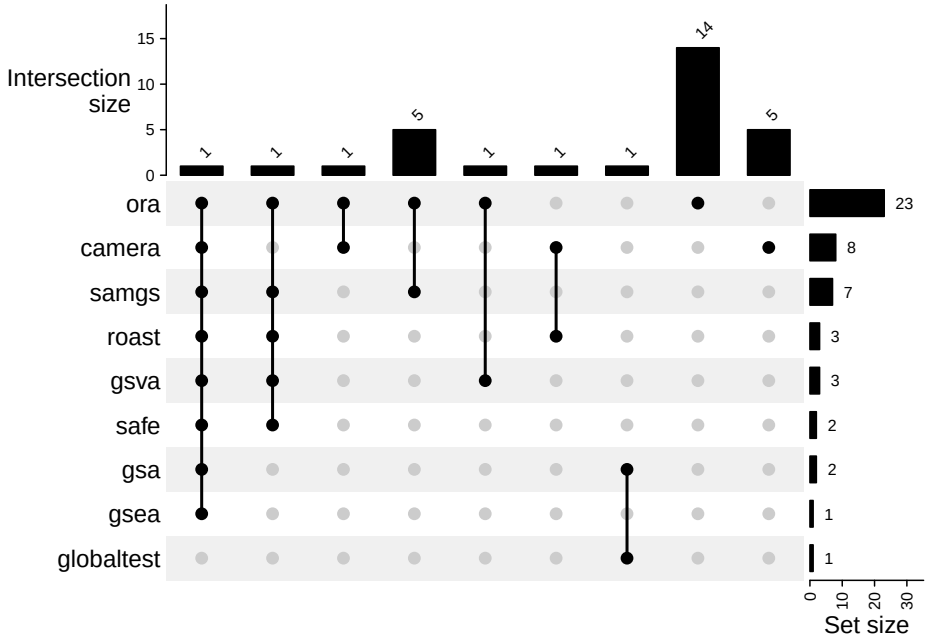
We compare the significant gene sets from different methods.

```
tb_list = lapply(res_list, gsRanking)
tb_list
# $ora
# DataFrame with 23 rows and 4 columns
#                    GENE.SET GLOB.STAT NGLOB.STAT      PVAL
#                 <character> <numeric>  <numeric> <numeric>
# 1     HALLMARK_P53_PATHWAY         2    0.01480     0.001
# 2       HALLMARK_APOPTOSIS         2    0.01420     0.001
# 3   HALLMARK_PI3K_AKT_MT..         1    0.01250     0.001
# 4   HALLMARK_INTERFERON_..         1    0.00730     0.001
# 5      HALLMARK_MYOGENESIS         1    0.00592     0.002
# ...                    ...       ...        ...       ...
# 19  HALLMARK_REACTIVE_OX..         0          0     0.023
# 20  HALLMARK_INTERFERON_..         0          0     0.026
# 21  HALLMARK_TGF_BETA_SI..         0          0     0.029
# 22  HALLMARK_ANDROGEN_RE..         0          0     0.039
# 23  HALLMARK_CHOLESTEROL..         0          0     0.044
#
# $safe
# DataFrame with 2 rows and 4 columns
#                  GENE.SET GLOB.STAT NGLOB.STAT      PVAL
#               <character> <numeric>  <numeric> <numeric>
# 1   HALLMARK_P53_PATHWAY    225000       1660     0.021
# 2 HALLMARK_HEDGEHOG_SI..     51300       1830     0.033
#
# $gsea
# DataFrame with 1 row and 4 columns
#                GENE.SET        ES       NES      PVAL
#             <character> <numeric> <numeric> <numeric>
# 1 HALLMARK_P53_PATHWAY     0.355       1.5    0.0421
#
# $gsa
```

```
# DataFrame with 2 rows and 3 columns
#                 GENE.SET     SCORE      PVAL
#              <character> <numeric> <numeric>
# 1   HALLMARK_P53_PATHWAY     0.388     0.012
# 2 HALLMARK_IL6_JAK_STA..     0.455     0.020
#
# $globaltest
# DataFrame with 1 row and 3 columns
#                 GENE.SET      STAT      PVAL
#              <character> <numeric> <numeric>
# 1 HALLMARK_IL6_JAK_STA..      4.41     0.008
#
# $roast
# DataFrame with 3 rows and 4 columns
#                 GENE.SET  NR.GENES       DIR      PVAL
#              <character> <numeric> <numeric> <numeric>
# 1   HALLMARK_P53_PATHWAY       135         1  0.000999
# 2 HALLMARK_ALLOGRAFT_R..       168         1  0.038000
# 3 HALLMARK_HEDGEHOG_SI..        28         1  0.040000
#
# $camera
# DataFrame with 8 rows and 4 columns
#                 GENE.SET  NR.GENES       DIR      PVAL
#              <character> <numeric> <numeric> <numeric>
# 1 HALLMARK_G2M_CHECKPO..       142        -1  0.000412
# 2 HALLMARK_ALLOGRAFT_R..       168         1  0.000824
# 3   HALLMARK_E2F_TARGETS       129        -1  0.001960
# 4 HALLMARK_MITOTIC_SPI..       116        -1  0.011800
# 5   HALLMARK_P53_PATHWAY       135         1  0.014700
# 6 HALLMARK_PROTEIN_SEC..        80        -1  0.022000
# 7 HALLMARK_KRAS_SIGNAL..       124         1  0.022900
# 8 HALLMARK_UV_RESPONSE..       122        -1  0.028800
#
# $gsva
# DataFrame with 3 rows and 3 columns
#                 GENE.SET   t.SCORE      PVAL
#              <character> <numeric> <numeric>
# 1 HALLMARK_WNT_BETA_CA..     -2.12    0.0389
# 2 HALLMARK_HEDGEHOG_SI..      2.12    0.0390
# 3   HALLMARK_P53_PATHWAY      2.06    0.0445
#
# $samgs
# DataFrame with 7 rows and 4 columns
#                 GENE.SET SUMSQ.STAT NSUMSQ.STAT      PVAL
#              <character>  <numeric>   <numeric> <numeric>
```

```
# 1   HALLMARK_P53_PATHWAY        322        2.38     0.000
# 2     HALLMARK_APOPTOSIS        257        1.83     0.003
# 3 HALLMARK_HEDGEHOG_SI..         50        1.78     0.012
# 4    HALLMARK_MYOGENESIS        245        1.45     0.017
# 5 HALLMARK_INTERFERON_..        209        1.52     0.024
# 6 HALLMARK_INFLAMMATOR..        218        1.36     0.025
# 7 HALLMARK_TNFA_SIGNAL..        226        1.44     0.041
#
# $ebm
# NULL
#
# $mgsa
# NULL
```

```
tb_list = tb_list[sapply(tb_list, length) > 0]

library(ComplexHeatmap)
cm = make_comb_mat(lapply(tb_list, function(x) x[[1]]))
UpSet(cm,
    top_annotation = upset_top_annotation(cm, add_numbers = TRUE),
    right_annotation = upset_right_annotation(cm, add_numbers = TRUE)
)
```



6.7 Important aspects of GSEA methogology

Sample size is a general factor of stastics where more samples give more pwoer for the statistical tests.

Gene sets have differnet sizes. it will affect teh null distribution (mostly the standard deviation) of set-level statistic

In ORA, it assumes genes are independent, also in some parametric methods, after xxx, which also assume ... However, ignore the gene correlation will, moslty estimate the wrong varaince. Nevertheless, we suggest to use sample permutation which keeps the gene correlation structure during the xxx

6.8 recommandations of methods

Due to the test is a information reduction from a matrix to a sigle value,

- use GSEA method
- use sample permutation
- use self-contained setlevel
- use simply model

Additionally, which is also recommneded for general data ananlysis, explorrary data anlaysi (EDA) should be first applied to the , eg.. to check the global direrential expression, which helps to find a proper method.

If you only have a list of genes, then use ORA with hypergeometric distribution and set whole protaincoding genes as background

If you have a vector of pre-computed gene-level scores, use GSEA tool with gene-permutation version. ANd it suggested to first check the global distribution of gene level scores...

If yo uhave a complete matrix, then use GSEA with sample permutations

Chapter 7

Gene Set Enrichment Analysis in Genomics

7.1 Overview

Given a list of genomic regions, what are the biological functions they relate. A nature solution is to first annotate genomic regions to the nearest TSS, then only use genes with a certain distance for ORA analysis. In this chapter, we will discuss the problems of directly apply ORA on genomic data and the solutions.

7.2 The GREAT method

7.3 The rGREAT package

7.4 The Lola package

7.5 RNASeq and methylation-adjusted ORA

7.6 SNP-based GSEA

7.7 general comments

E.g. large sample size

compare between different sets of genomic regions

Chapter 8

Topology-based pathway enrichmetn

8.1 Overview

Gene sets are represented as a vector of

8.2 Use topology informatino

8.3 Pathway common structure

8.4 General process of utilizing topology information

8.5 Centrality measures

8.6 centrality-based pathway enrichment

8.7 SPIA: pathway impact analysis

8.8 R packages for topology-based GSEA

Chapter 9

Extensions of GSEA

9.1 Single sample-based GSEA

9.2 Ensembles of multiple GSEA methods

9.3 TBA

Chapter 10

Visualization

10.1 general xxx

10.2 Visualize by statistical plots

10.3 network visualization

10.4 Enrichment map

10.5

Chapter 11

Clustering and simplifying GSEA results

11.1 measures of similarities