The Principle of R Package GSClassifier

Weibin Huang

2023-08-04

Contents

Welcome

About

The Priciple of GSClassifier is a book for users of the R package **GSClassifier** who want to know the most details. If you're looking for the PDF edition, you can find it here.

GSClassifier is an R-based comprehensive classification tool for subtypes modeling and personalized calling based on pure transcriptomics. It could be used for precision medicine, such as cancer diagnosis. The inspiration for **GSClassifier** comes from ImmuneSubtype-Classifier, an R package for classification of PanCancer immune subtypes based on the work of Gibbs et al [1,2].

Lots of surprising features in **GSClassifier** are as follows:

- Optimized for just `one sample`
- Available for modeling and calling of brand-new `GEPs-based subtypes` in any diseases (cancers)
- No limitation of the number of `gene signatures`(1) or `subtypes`(2)
- `Insensitive normalization` due to the use of the individual `gene rank matrix`
- More ensemble and repeatable modeling process
- More optimizations in the parallel computing
- New useful functions as supplements

ATTENTION! In the future, there might be third-party contributors in `GSClassifier` platform, with some useful models for specific usages. If you use models provided by these people, **you had better know more details as possible**, including **designs, data sources, destinations, training scripts, and limitations** of models, especially those from studies under peer review.

License

- **GSClassifier** is released under the Apache-2.0 license. See LICENSE for details.
- The technical documentation, as a whole, is licensed under a Creative Commons Attribution- NonCommercial-ShareAlike 4.0 International License. The code contained in this book is simultaneously available under the MIT license; this means that you are free to use it in your packages, as long as you cite the source.

Installation

RStudio/Posit is one of the best Integrated Development Environments (IDE) in R programming. If you're struggling in R-GUI, it is recommended to turn to RStudio/Posit.

For installation of **GSClassifier**, please run these commands in an R environment:

```
# Install "devtools" package
if (!requireNamespace("devtools", quietly = TRUE))
    install.packages("devtools")

# Install dependencies
if (!requireNamespace("luckyBase", quietly = TRUE))
    devtools::install_github("huangwb8/luckyBase")

# Install the "GSClassifier" package
if (!requireNamespace("GSClassifier", quietly = TRUE))
```

```
    devtools::install_github("huangwb8/GSClassifier")
```

In the future, a stable `GSClassifier` version might be sent to CRAN. Still beta.

Mirror

For some special countries or regions, users could also try:

```
# Install dependencies
install.packages("https://gitee.com/huangwb8/luckyBase/repository
   /archive/Primary?format=tar.gz", repos=NULL, method="libcurl")

# Install the "GSClassifier" package
install.packages("https://gitee.com/huangwb8/GSClassifier/
   repository/archive/Primary?format=tar.gz", repos=NULL, method=
   "libcurl")
```

Change log

- Version 0.1.27
  - Enhaned `geneMatch` function
  - Repair some bugs
- Version 0.1.9
  - Optimize function verbose
  - Optimize for a routine scenario: one gene set and two subtypes
  - Optimize the strategy of automatic parameters selection for modeling training with R package `caret`
  - Interact with external models from the luckyModel package

- Version 0.1.8
  - Primary public version of `GSClassifier`
  - Apache License, Version 2.0
  - Friendly wiki-based tutorial
  - Platform for developers

TODO

- More medical fields included, such as in the pan-cancer utility
- Advanced methods (such as artificial intelligence) for enhanced robustness
- Unsupervised learning for de-novo classification based on intrinsic frames of omics instead of human knowledge
- Multi-omics exploration and support
- More friendly characteristics for developers and contributors
- Web application for newbies to R programming

Other Projects

You may also be interested in:

- "**luckyBase**" The base functions of lucky series.
- "**luckyModel**" Model ensemble for third-party lucky series, such GSClassifier.

Chapter 1

The Principle of GSClassifier

Leave some introductions

1.1 Packages

```
# Install "devtools" package
if (!requireNamespace("devtools", quietly = TRUE))
  install.packages("devtools")

# Install dependencies
if (!requireNamespace("luckyBase", quietly = TRUE))
  devtools::install_github("huangwb8/luckyBase")

# Install the "GSClassifier" package
if (!requireNamespace("GSClassifier", quietly = TRUE))
  devtools::install_github("huangwb8/GSClassifier")

# Install the "pacman" package
if (!requireNamespace("pacman", quietly = TRUE)){
  install.packages("pacman")
```

```
  library(pacman)
} else {
  library(pacman)
}

# Load needed packages
packages_needed <- c(
  "readxl",
  "ComplexHeatmap",
  "GSClassifier",
  "rpart",
  "tidyr",
  "reshape2",
  "ggplot2")
for(i in packages_needed){p_load(char=i)}
```

Here is the environment of R programming:

```
# R version 4.2.2 (2022-10-31 ucrt)
# Platform: x86_64-w64-mingw32/x64 (64-bit)
# Running under: Windows 10 x64 (build 19042)
#
# Matrix products: default
#
# locale:
# [1] LC_COLLATE=Chinese (Simplified)_China.utf8
# [2] LC_CTYPE=Chinese (Simplified)_China.utf8
# [3] LC_MONETARY=Chinese (Simplified)_China.utf8
# [4] LC_NUMERIC=C
# [5] LC_TIME=Chinese (Simplified)_China.utf8
#
```

```
# attached base packages:
# [1] grid      stats     graphics  grDevices utils     datasets
    methods
# [8] base
#
# other attached packages:
# [1] ggplot2_3.4.1         reshape2_1.4.4        tidyr_1.3.0
# [4] rpart_4.1.19          GSClassifier_0.1.27   luckyBase_0.1.4
# [7] ComplexHeatmap_2.14.0 readxl_1.4.2          pacman_0.5.1
#
# loaded via a namespace (and not attached):
#   [1] colorspace_2.1-0     ggsignif_0.6.4       rjson_0.2.21
#   [4] ellipsis_0.3.2       class_7.3-20         circlize_0.4.15
#   [7] GlobalOptions_0.1.2  fs_1.6.1             clue_0.3-64
#  [10] rstudioapi_0.14      listenv_0.9.0        ggpubr_0.6.0
#  [13] remotes_2.4.2        lubridate_1.9.2      prodlim_
   2019.11.13
#  [16] fansi_1.0.4          codetools_0.2-18     splines_4.2.2
#  [19] doParallel_1.0.17    cachem_1.0.7         knitr_1.42
#  [22] pkgload_1.3.2        jsonlite_1.8.4       pROC_1.18.0
#  [25] caret_6.0-93         broom_1.0.3          cluster_2.1.4
#  [28] png_0.1-8            shiny_1.7.4          compiler_4.2.2
#  [31] backports_1.4.1      Matrix_1.5-1         fastmap_1.1.1
#  [34] cli_3.6.0            later_1.3.0          htmltools_0.5.4
#  [37] prettyunits_1.1.1    tools_4.2.2          gtable_0.3.1
#  [40] glue_1.6.2           dplyr_1.1.0          Rcpp_1.0.10
#  [43] carData_3.0-5        cellranger_1.1.0     vctrs_0.5.2
#  [46] nlme_3.1-160         iterators_1.0.14     timeDate_
   4022.108
#  [49] xfun_0.39            gower_1.0.1          stringr_1.5.0
```

```
#  [52] globals_0.16.2       ps_1.7.2              timechange_
   0.2.0
#  [55] mime_0.12            miniUI_0.1.1.1        lifecycle_1.0.3
#  [58] devtools_2.4.5       rstatix_0.7.2         future_1.31.0
#  [61] MASS_7.3-58.1        scales_1.2.1          ipred_0.9-13
#  [64] promises_1.2.0.1     parallel_4.2.2        RColorBrewer_
   1.1-3
#  [67] yaml_2.3.7           memoise_2.0.1         stringi_1.7.12
#  [70] S4Vectors_0.36.2     randomForest_4.7-1.1  foreach_1.5.2
#  [73] BiocGenerics_0.44.0  hardhat_1.2.0         pkgbuild_1.4.0
#  [76] lava_1.7.2.1         shape_1.4.6           tuneR_1.4.2
#  [79] rlang_1.0.6          pkgconfig_2.0.3       matrixStats_
   0.63.0
#  [82] evaluate_0.20        lattice_0.20-45       purrr_1.0.1
#  [85] recipes_1.0.5        htmlwidgets_1.6.1     processx_3.8.0
#  [88] tidyselect_1.2.0     parallelly_1.34.0     plyr_1.8.8
#  [91] magrittr_2.0.3       bookdown_0.34         R6_2.5.1
#  [94] IRanges_2.32.0       generics_0.1.3        profvis_0.3.7
#  [97] withr_2.5.0          pillar_1.8.1          survival_3.4-0
# [100] abind_1.4-5          nnet_7.3-18           future.apply_
   1.10.0
# [103] tibble_3.1.8         crayon_1.5.2          car_3.1-1
# [106] xgboost_1.7.3.1      utf8_1.2.3            rmarkdown_2.20
# [109] urlchecker_1.0.1     GetoptLong_1.0.5      usethis_2.1.6
# [112] data.table_1.14.8    callr_3.7.3           ModelMetrics_
   1.2.2.2
# [115] digest_0.6.31        xtable_1.8-4          httpuv_1.6.9
# [118] signal_0.7-7         stats4_4.2.2          munsell_0.5.0
# [121] sessioninfo_1.2.2
```

1.2 Flowchart

The flowchart of **GSClassifier** is showed in Figure 1.1.

1.2.1 Data Processing

For each dataset, the RNA expression matrix would be normalized internally (**Raw Matrix**) so that the expression data of the samples in the dataset were comparable and suitable for subtype identification. As demonstrated in Figure 1.1, the **Subtype Vector** is identified based on independent cohorts instead of a merged matrix with batch effect control technologies. More details about batch effect control are discussed in 2.4.

There is no standard method to figure out subtype vectors. It depends on the Gene Expression Profiles (GEPs), the biological significance, or the ideas of researchers. For **Pan-immune Activation and Dysfunction (PAD)** subtypes, the GEPs, **Pan-Immune Activation Module (PIAM)** and **Pan-Immune Dysfunction Genes (PIDG)**, are biologically associated and suitable for calling four subtypes ($PIAM^{high}PIDG^{high}$, $PIAM^{high}PIDG^{low}$, $PIAM^{low}PIDG^{high}$, and $PIAM^{low}PIDG^{low}$). Theoretically, we can also use a category strategy like low/medium/high, but more evidence or motivations are lacking for chasing such a complex model.

With subtype vectors and raw matrices, **Top Scoring Pairs (TSP)**, the core data format for model training and application in GSClassifier, would be calculated for the following process. The details of TSP normalization are summarized in 1.3.

1.2.2 Model Establishment and Validation

The TSP matrix would be divided into the training cohort and the internal validation cohort. In the PAD project, the rate of samples (training vs. test) is **7:3**. Next, each **SubSet** (70% of the training cohort) would be further selected randomly to build a **SubModel** via cross-validation Extreme Gradient Boosting algorithm (**xgboost::xgb.cv** function) [3]. The number of submodels is suggested over 20 (more details in 4.5).

The internal validation cohort and external validation cohort (if any) would be used to

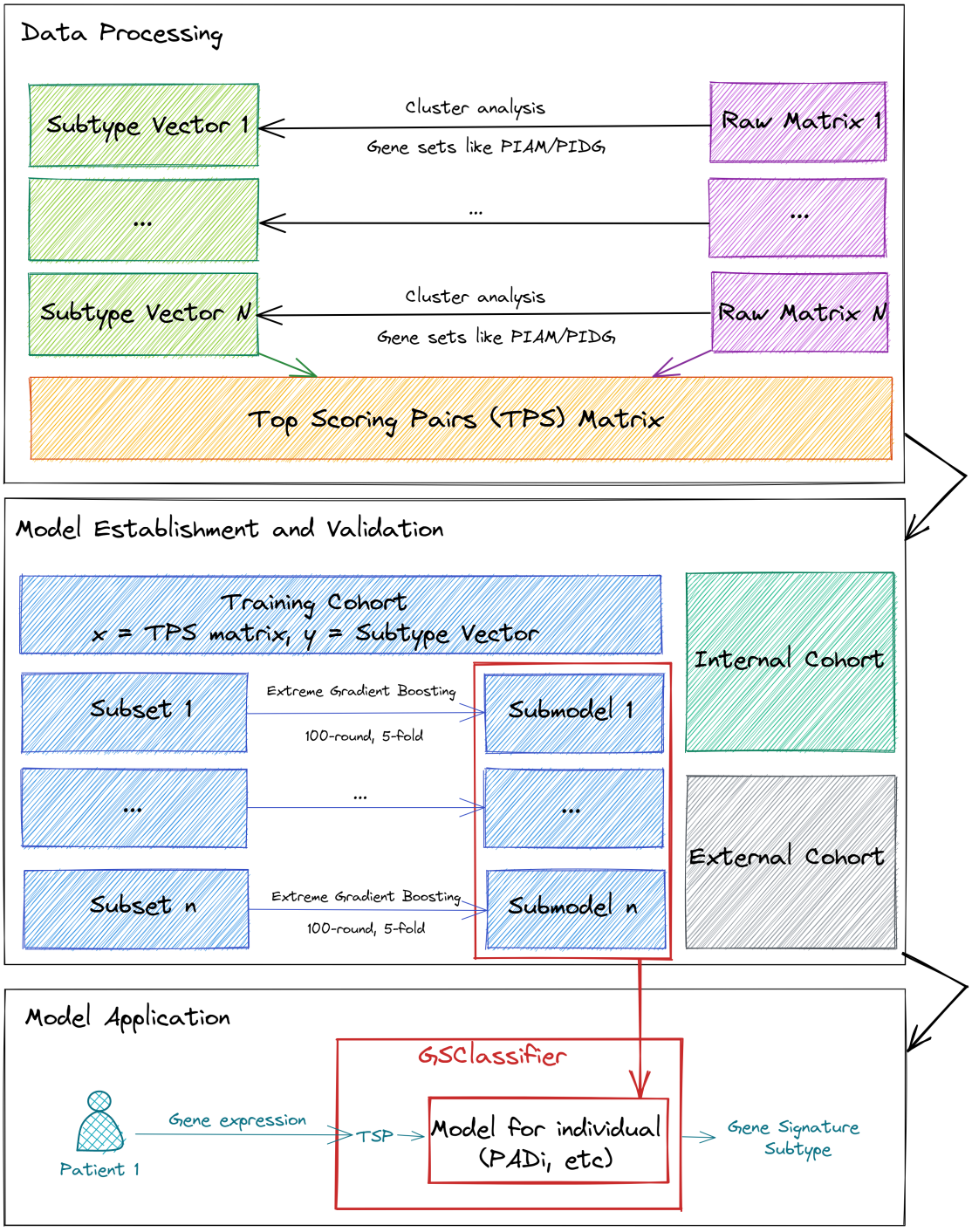


Figure 1.1: The flow chart of GSClassifier

test the performance of the trained model. By the way, **the data of both internal and external validation cohorts would not be used during model training** to avoid over-fitting.

1.2.3 Model Application

In the PAD project, **Model for individual**, the ensemble of submodels, is called "PAD for individual" (**PADi**). Supposed raw RNA expression of a sample was given. As shown in 1.1 and 1.2, **GSClassifier** would turn raw RNA expression into a TSP vector, which would be as an input to **Model for individual**. Then, **GSClassifier** would output the possibility matrix and the subtype for this sample. No extra data (RNA expression of others, follow-up data, etc) would be needed but RNA expression of the patient for subtype identification, so we suggest **Model for individual** (**PADi**, etc) as personalized model.

1.3 Top scoring pairs

1.3.1 Introduction

Genes expression of an individual is normalized during the model training and the subtype identification via **Top Scoring Pairs** (**TSP**, also called **Relative Expression Orderings** (**REOs**)) algorithm, which was previously described by Geman et al [4]. **TSP** normalization for an individual depends on its transcript data, implying that subtype calling would not be perturbed by data from other individuals or other extra information like follow-up data. **TSP** had been used in cancer research and effectively predicts cancer progression and ICIs response [5–7].

As shown in Figure 1.2, The TSP data in GSClassifier consists of three parts: **binned expression**, **pair difference**, and **set difference**. In this section, we would conduct some experiments to demonstrate the potential of TSP normalization for development of cross-dataset/platform GEP-based models.

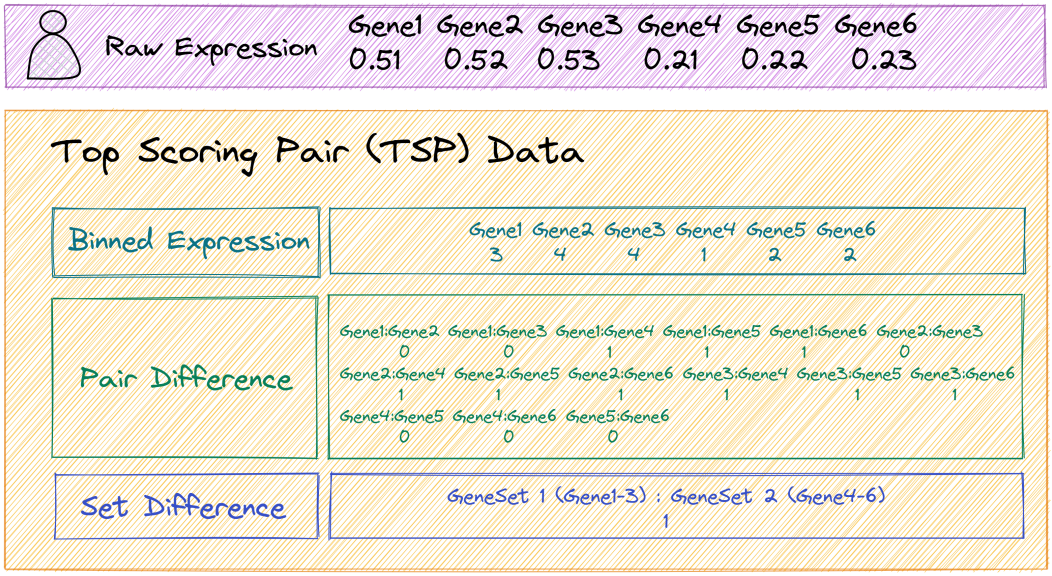


Figure 1.2: The components of TSP (2 gene sets)

1.3.2 Simulated Dataset

We simulated a dataset to demonstrate TSP normalization in GSClassifier:

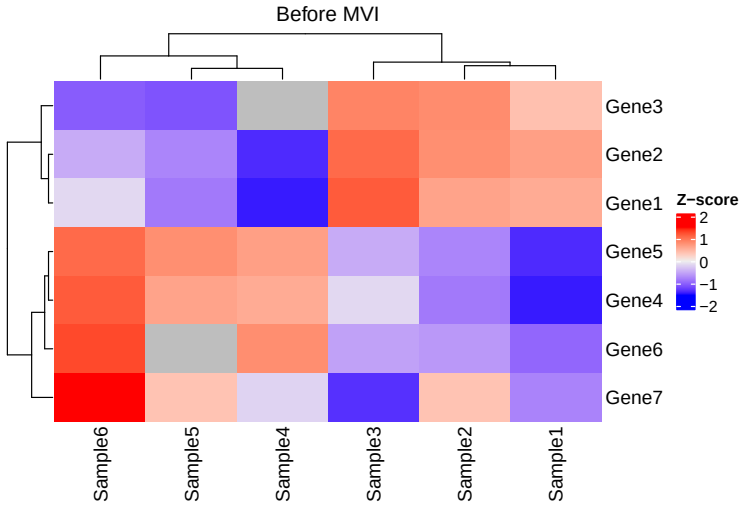
```
# Geneset
geneSet <- list(
  Set1 = paste('Gene',1:3,sep = ''),
  Set2 = paste('Gene',4:6,sep = '')
)

# RNA expression
x <- read_xlsx('./data/simulated-data.xlsx', sheet = 'RNA')
expr0 <- as.matrix(x[,-1])
rownames(expr0) <- as.character(as.matrix(x[,1])); rm(x)

# Missing value imputation (MVI)
expr <- na_fill(expr0, method = "quantile", seed = 447)
# 2023-08-04 23:19:14 | Missing value imputation with quantile
   algorithm!
```

```
# Subtype information
# It depends on the application scenarios of GEPs
subtype_vector <- c(1, 1, 1, 2, 2, 2)
# Binned data for subtype 1
Ybin <- ifelse(subtype_vector == 1, 1, 0)

# Parameters
breakVec = c(0, 0.25, 0.5, 0.75, 1.0)

# Report
cat(c('\n', 'Gene sets:', '\n'))
print(geneSet)
cat('RNA expression:', '\n')
print(expr0); cat('\n')
cat('RNA expression after MVI:', '\n')
print(expr)
#
#  Gene sets:
# $Set1
# [1] "Gene1" "Gene2" "Gene3"
#
# $Set2
# [1] "Gene4" "Gene5" "Gene6"
#
# RNA expression:
#       Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
# Gene1    0.51    0.52    0.60    0.21    0.30    0.40
# Gene2    0.52    0.54    0.58    0.22    0.31    0.35
# Gene3    0.53    0.60    0.61      NA    0.29    0.30
```

```
# Gene4    0.21    0.30    0.40    0.51    0.52    0.60
# Gene5    0.22    0.31    0.35    0.52    0.54    0.58
# Gene6    0.23    0.29    0.30    0.53      NA    0.61
# Gene7    0.10    0.12    0.09    0.11    0.12    0.14
#
# RNA expression after MVI:
#       Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
# Gene1    0.51    0.52    0.60  0.2100 0.30000    0.40
# Gene2    0.52    0.54    0.58  0.2200 0.31000    0.35
# Gene3    0.53    0.60    0.61  0.2486 0.29000    0.30
# Gene4    0.21    0.30    0.40  0.5100 0.52000    0.60
# Gene5    0.22    0.31    0.35  0.5200 0.54000    0.58
# Gene6    0.23    0.29    0.30  0.5300 0.51774    0.61
# Gene7    0.10    0.12    0.09  0.1100 0.12000    0.14
```

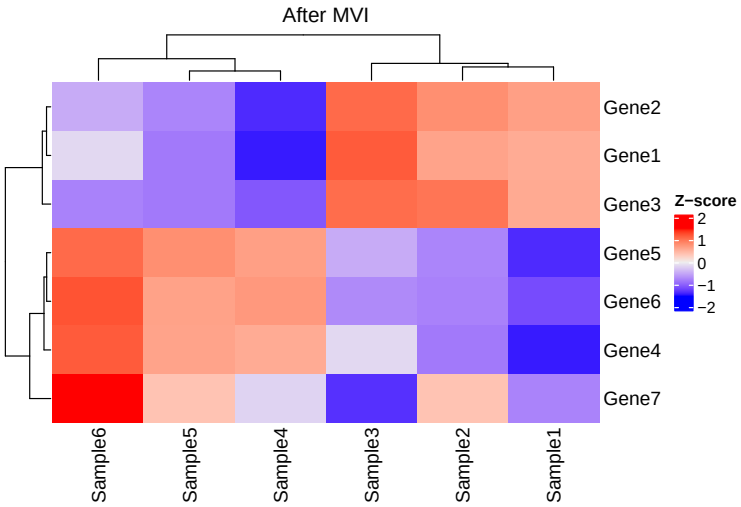
Look at the matrix via heatmap:

```
Heatmap(t(scale(t(expr0))), name = "Z-score", column_title = "
   Before MVI")
```




```
Heatmap(t(scale(t(expr))), name = "Z-score", column_title = "
```

```
  After MVI")
```



This is an interesting dataset with features as follows:

- **Distinguished gene sets**: The expression profile between **Gene 1-3** and **Gene 4-6** is different across samples. Thus, these gene sets might represent different biological significance.
- **Stable gene**: The expression level and rank of **Gene 7** seemed to be similar across samples. Thus, **Gene 7** might not be a robust marker for subtype modeling. Thus, it could help us to understand how the filtering of **GSClassifier** works.
- **Expression heterogeneity & rank homogeneity**: Take **Sample1** and **Sample3** as examples. The expression of **Gene 1-6** in **Sample3** seemed to be higher than those of **Sample1**. However, the expression of **Gene 1-3** is higher than **Gene 4-6** in both **Sample1** and **Sample3**, indicating similar bioprocess in these samples exists so that they should be classified as the same subtype.

1.3.3 Binned expression

First, we binned genes with different quantile intervals so that the distribution of rank information could be more consistent across samples.

Take **Sample4** as an example:

```
# Data of Sample4
x <- expr[,4]

# Create quantiles
brks <- quantile(as.numeric(x),
                 probs=breakVec,
                 na.rm = T)

# Get interval orders
xbin <- .bincode(x = x,
                 breaks = brks,
                 include.lowest = T)
xbin <- as.numeric(xbin)
names(xbin) <- names(x)

# Report
cat('Quantiles:', '\n'); print(brks)
cat('\n')
cat('Raw expression:', '\n');print(x)
cat('\n')
cat('Binned expression:', '\n'); print(xbin)
# Quantiles:
#     0%    25%    50%    75%   100%
# 0.1100 0.2150 0.2486 0.5150 0.5300
#
# Raw expression:
#  Gene1  Gene2  Gene3  Gene4  Gene5  Gene6  Gene7
# 0.2100 0.2200 0.2486 0.5100 0.5200 0.5300 0.1100
#
```

```
# Binned expression:
# Gene1 Gene2 Gene3 Gene4 Gene5 Gene6 Gene7
#     1     2     2     3     4     4     1
```

For example, **0.110** is the minimun of the raw expression vector, so its binned expression is **1**. Similarly, the binned expression of maximum **0.530** is **4**.

Generally, we calculate binned expression via function **breakBin** of **GSClassifier**:

```
expr_binned <- apply(
  expr, 2,
  GSClassifier:::breakBin,
  breakVec)
rownames(expr_binned) <- rownames(expr)
print(expr_binned)
#       Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
# Gene1       3       3       4       1       2       2
# Gene2       4       4       3       2       2       2
# Gene3       4       4       4       2       1       1
# Gene4       1       2       2       3       4       4
# Gene5       2       2       2       4       4       3
# Gene6       2       1       1       4       3       4
# Gene7       1       1       1       1       1       1
```

In this simulated dataset, **Gene7** is a gene whose expression is always the lowest across all samples. In other words, the rank of **Gene7** is stable or invariable across samples so it's not robust for the identification of differential subtypes.

Except for binned expression, we also calculated pair difference later. Because the number of gene pairs is C_n^2, the exclusion of genes like **Gene7** before modeling could reduce the complexity and save computing resources. In all, genes with low-rank differences should be dropped out to some extent in **GSClassifier**.

First, We use **base::rank** to return the sample ranks of the values in a vector:

```
expr_binned_rank <- apply(
  expr_binned, 2,
  function(x)rank(x, na.last = TRUE)
)
print(expr_binned_rank)
#       Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
# Gene1     5.0     5.0     6.5     1.5     3.5     3.5
# Gene2     6.5     6.5     5.0     3.5     3.5     3.5
# Gene3     6.5     6.5     6.5     3.5     1.5     1.5
# Gene4     1.5     3.5     3.5     5.0     6.5     6.5
# Gene5     3.5     3.5     3.5     6.5     6.5     5.0
# Gene6     3.5     1.5     1.5     6.5     5.0     6.5
# Gene7     1.5     1.5     1.5     1.5     1.5     1.5
```

Then, get weighted average rank difference of each gene based on specified subtype distribution (**Ybin**):

```
testRes <- sapply(
  1:nrow(expr_binned_rank),
  function(gi){

    # Rank vector of each gene
    rankg = expr_binned_rank[gi,];

    # Weighted average rank difference of a gene for specified
       subtype
    # Here is subtype 1 vs. others
    (sum(rankg[Ybin == 0], na.rm = T) / sum(Ybin == 0, na.rm = T)
       ) -
    (sum(rankg[Ybin == 1], na.rm = T) / sum(Ybin == 1, na.rm = T)
```

```
      )
  }
)
names(testRes) <- rownames(expr_binned_rank)
print(testRes)
#      Gene1      Gene2      Gene3      Gene4      Gene5      Gene6
       Gene7
# -2.666667 -2.500000 -4.333333  3.166667  2.500000  3.833333
   0.000000
```

Gene7 is the one with the lowest absolute value (0) of rank diffrence. By the way, **Gene 1-3** have the same direction (<0), and so does **Gene 4-6** (>0), which indicates the nature of clustering based on these two gene sets.

In practice, we use **ptail** to select differential genes based on rank diffrences. **Smaller ptail is, less gene kept**. Here, we just set **ptail=0.4**:

```
# ptail is a numeber ranging (0,0.5].
ptail = 0.4

# Index of target genes with big rank differences
idx <- which((testRes < quantile(testRes, ptail, na.rm = T)) |
             (testRes > quantile(testRes, 1.0-ptail, na.rm = T)))

# Target genes
gene_bigRank <- names(testRes)[idx]

# Report
cat('Index of target genes: ','\n');print(idx); cat('\n')
cat('Target genes:','\n');print(gene_bigRank)
# Index of target genes:
```

```
# Gene1 Gene2 Gene3 Gene4 Gene5 Gene6
#     1     2     3     4     5     6
#
# Target genes:
# [1] "Gene1" "Gene2" "Gene3" "Gene4" "Gene5" "Gene6"
```

Hence, **Gene7** was filtered and excluded in the following analysis. By the way, both **ptail** and **breakVec** are hyperparameters in GSClassifier modeling.

1.3.4 Pair difference

In GSClassifier, we use an ensemble function **featureSelection** to select data for pair difference scoring.

```
expr_feat <- featureSelection(expr, Ybin,
                              testRes = testRes,
                              ptail = 0.4)

expr_sub <- expr_feat$Xsub
gene_bigRank <- expr_feat$Genes

# Report
cat('Raw xpression without NA:', '\n')
print(expr_sub)
cat('\n')
cat('Genes with large rank diff:', '\n')
print(gene_bigRank)
# Raw xpression without NA:
#       Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
# Gene1    0.51    0.52    0.60  0.2100 0.30000    0.40
# Gene2    0.52    0.54    0.58  0.2200 0.31000    0.35
```

```
# Gene3    0.53    0.60    0.61  0.2486 0.29000    0.30
# Gene4    0.21    0.30    0.40  0.5100 0.52000    0.60
# Gene5    0.22    0.31    0.35  0.5200 0.54000    0.58
# Gene6    0.23    0.29    0.30  0.5300 0.51774    0.61
#
# Genes with large rank diff:
# [1] "Gene1" "Gene2" "Gene3" "Gene4" "Gene5" "Gene6"
```

In GSClassifier, we use function **makeGenePairs** to calculate pair differences:

```
gene_bigRank_pairs <- GSClassifier:::makeGenePairs(
  gene_bigRank,
  expr[gene_bigRank,])
print(gene_bigRank_pairs)
#             Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
# Gene1:Gene2       0       0       1       0       0       1
# Gene1:Gene3       0       0       0       0       1       1
# Gene1:Gene4       1       1       1       0       0       0
# Gene1:Gene5       1       1       1       0       0       0
# Gene1:Gene6       1       1       1       0       0       0
# Gene2:Gene3       0       0       0       0       1       1
# Gene2:Gene4       1       1       1       0       0       0
# Gene2:Gene5       1       1       1       0       0       0
# Gene2:Gene6       1       1       1       0       0       0
# Gene3:Gene4       1       1       1       0       0       0
# Gene3:Gene5       1       1       1       0       0       0
# Gene3:Gene6       1       1       1       0       0       0
# Gene4:Gene5       0       0       1       0       0       1
# Gene4:Gene6       0       1       1       0       1       0
# Gene5:Gene6       0       1       1       0       1       0
```

Take **Gene1:Gene4** of **Sample1** as an example. $Expression_{Gene1} - Expression_{Gene4} = 0.51 - 0.21 = 0.3 > 0$, so the pair score is 1. If the difference is less than or equal to 0, the pair score is 0. In addition, the scoring differences of gene pairs between **Sample 1-3** and **Sample 4-6** are obvious, revealing the robustness of pair difference for subtype identification.

1.3.5 Set difference

In **GSClassifier**, **set difference** is defined as a weight average of gene-geneset rank difference.

```
# No. of gene sets
nGS = 2

# Name of gene set comparision, which is like s1s2, s1s3 and so
   on.
featureNames <- 's1s2'

# Gene set difference across samples
resultList <- list()
for (i in 1:ncol(expr_sub)) { # i=1
  res0 <- numeric(length=length(featureNames))
  idx <- 1
  for (j1 in 1:(nGS-1)) { # j1=1
    for (j2 in (j1+1):nGS) { # j2=2

      # If j1=1 and j2=2, gene sets s1/s2 would be selected

      # Genes of different gene sets
      set1 <- geneSet[[j1]] # "Gene1" "Gene2" "Gene3"
      set2 <- geneSet[[j2]] # "Gene4" "Gene5" "Gene6"
```

```
      # RNA expression of Genes by different gene sets
      vals1 <- expr_sub[rownames(expr_sub) %in% set1,i]
      # Gene1 Gene2 Gene3
      # 0.51  0.52  0.53
      vals2 <- expr_sub[rownames(expr_sub) %in% set2,i]
      # Gene4 Gene5 Gene6
      # 0.21  0.22  0.23

      # Differences between one gene and gene sets
      # Compare expression of each gene in Set1 with all genes in
          Set2.
      # For example, 0.51>0.21/0.22/0.23, so the value of Gene1:
         s2 is 3.
      res1 <- sapply(vals1, function(v1) sum(v1 > vals2, na.rm=T)
         )
      # Gene1:s2   Gene2:s2   Gene3:s2
      # 3          3           3

      # Weight average of gene-geneset rank difference
      res0[idx] <- sum(res1, na.rm = T) / (length(vals1) * length
         (vals2))

      # Next gene set pair
      idx <- idx + 1
    }
  }
  resultList[[i]] <- as.numeric(res0)
}
resMat <- do.call(cbind, resultList)
```

```
colnames(resMat) <- colnames(expr_sub)
rownames(resMat) <- featureNames

# Report
cat('Set difference across samples: ', '\n')
print(resMat)
# Set difference across samples:
#      Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
# s1s2       1       1       1       0       0       0
```

In **GSClassifier**, we established **makeSetData** to evaluate set difference across samples:

```
# Gene set difference across samples
geneset_interaction <- GSClassifier:::makeSetData(expr_sub,
   geneSet)

# Report
cat('Set difference across samples: ', '\n')
print(resMat)
# Set difference across samples:
#      Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
# s1s2       1       1       1       0       0       0
```

We have known that the subtype of **Sample 1-3** differs from that of **Sample 4-6**, which revealed the robustness of set differences for subtype identification.

Based on the structure of TSP in Figure 1.2, the TSP matrix of the simulated dataset should be :

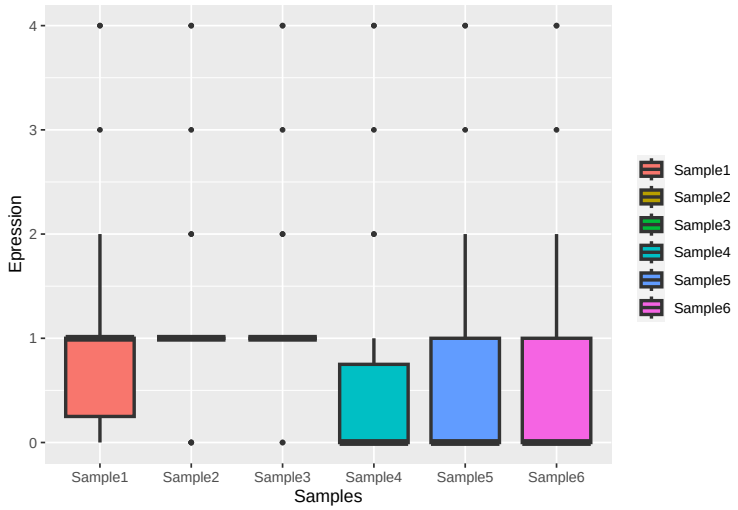
```
# TSP matrix
tsp <- rbind(
```

```
  # Binned expression
  expr_binned[gene_bigRank,],

  # Pair difference
  gene_bigRank_pairs,

  # Set difference
  resMat
)

# Report
cat('TSP matrix: ', '\n')
print(tsp)
# TSP matrix:
#             Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
# Gene1             3       3       4       1       2       2
# Gene2             4       4       3       2       2       2
# Gene3             4       4       4       2       1       1
# Gene4             1       2       2       3       4       4
# Gene5             2       2       2       4       4       3
# Gene6             2       1       1       4       3       4
# Gene1:Gene2       0       0       1       0       0       1
# Gene1:Gene3       0       0       0       0       1       1
# Gene1:Gene4       1       1       1       0       0       0
# Gene1:Gene5       1       1       1       0       0       0
# Gene1:Gene6       1       1       1       0       0       0
# Gene2:Gene3       0       0       0       0       1       1
# Gene2:Gene4       1       1       1       0       0       0
# Gene2:Gene5       1       1       1       0       0       0
# Gene2:Gene6       1       1       1       0       0       0
```

```
# Gene3:Gene4      1      1      1      0      0      0
# Gene3:Gene5      1      1      1      0      0      0
# Gene3:Gene6      1      1      1      0      0      0
# Gene4:Gene5      0      0      1      0      0      1
# Gene4:Gene6      0      1      1      0      1      0
# Gene5:Gene6      0      1      1      0      1      0
# s1s2             1      1      1      0      0      0
```

Have a look at the distribution:

```
# Data
tsp_df <- reshape2::melt(tsp)

ggplot(tsp_df,aes(x=Var2,y=value,fill=Var2)) +
      geom_boxplot(outlier.size = 1, size = 1) +
      labs(x = 'Samples',
           y = 'Epression',
           fill = NULL)
```



Chapter 2

Discussion

In this section, we would discuss some key topics about **GSClassifier**, including **Missing value imputation (MVI)**, **Batch effect**, **hyperparameters**, and so on.

2.1 Packages

```
# Install "devtools" package
if (!requireNamespace("devtools", quietly = TRUE))
  install.packages("devtools")

# Install dependencies
if (!requireNamespace("luckyBase", quietly = TRUE))
  devtools::install_github("huangwb8/luckyBase")

# Install the "**GSClassifier**" package
if (!requireNamespace("GSClassifier", quietly = TRUE))
  devtools::install_github("huangwb8/GSClassifier")

# Install the "pacman" package
if (!requireNamespace("pacman", quietly = TRUE)){
```

```
  install.packages("pacman")
  library(pacman)
} else {
  library(pacman)
}

# Load needed packages
packages_needed <- c(
  "readxl",
  "ComplexHeatmap",
  "GSClassifier",
  "rpart",
  "tidyr",
  "reshape2",
  "ggplot2")
for(i in packages_needed){p_load(char=i)}
```

Here is the environment of R programming:

```
# R version 4.2.2 (2022-10-31 ucrt)
# Platform: x86_64-w64-mingw32/x64 (64-bit)
# Running under: Windows 10 x64 (build 19042)
#
# Matrix products: default
#
# locale:
# [1] LC_COLLATE=Chinese (Simplified)_China.utf8
# [2] LC_CTYPE=Chinese (Simplified)_China.utf8
# [3] LC_MONETARY=Chinese (Simplified)_China.utf8
# [4] LC_NUMERIC=C
# [5] LC_TIME=Chinese (Simplified)_China.utf8
```

```
#
# attached base packages:
# [1] grid      stats     graphics  grDevices utils     datasets
    methods
# [8] base
#
# other attached packages:
# [1] ggplot2_3.4.1         reshape2_1.4.4        tidyr_1.3.0
# [4] rpart_4.1.19          GSClassifier_0.1.27   luckyBase_0.1.4
# [7] ComplexHeatmap_2.14.0 readxl_1.4.2          pacman_0.5.1
#
# loaded via a namespace (and not attached):
#   [1] colorspace_2.1-0     ggsignif_0.6.4       rjson_0.2.21
#   [4] ellipsis_0.3.2       class_7.3-20         circlize_0.4.15
#   [7] GlobalOptions_0.1.2  fs_1.6.1             clue_0.3-64
#  [10] rstudioapi_0.14      listenv_0.9.0        ggpubr_0.6.0
#  [13] remotes_2.4.2        lubridate_1.9.2      prodlim_
   2019.11.13
#  [16] fansi_1.0.4          codetools_0.2-18     splines_4.2.2
#  [19] doParallel_1.0.17    cachem_1.0.7         knitr_1.42
#  [22] pkgload_1.3.2        jsonlite_1.8.4       pROC_1.18.0
#  [25] caret_6.0-93         broom_1.0.3          cluster_2.1.4
#  [28] png_0.1-8            shiny_1.7.4          compiler_4.2.2
#  [31] backports_1.4.1      Matrix_1.5-1         fastmap_1.1.1
#  [34] cli_3.6.0            later_1.3.0          htmltools_0.5.4
#  [37] prettyunits_1.1.1    tools_4.2.2          gtable_0.3.1
#  [40] glue_1.6.2           dplyr_1.1.0          Rcpp_1.0.10
#  [43] carData_3.0-5        cellranger_1.1.0     vctrs_0.5.2
#  [46] nlme_3.1-160         iterators_1.0.14     timeDate_
   4022.108
```

```
#  [49] xfun_0.39              gower_1.0.1           stringr_1.5.0
#  [52] globals_0.16.2         ps_1.7.2              timechange_
  0.2.0
#  [55] mime_0.12              miniUI_0.1.1.1        lifecycle_1.0.3
#  [58] devtools_2.4.5         rstatix_0.7.2         future_1.31.0
#  [61] MASS_7.3-58.1          scales_1.2.1          ipred_0.9-13
#  [64] promises_1.2.0.1       parallel_4.2.2        RColorBrewer_
  1.1-3
#  [67] yaml_2.3.7             memoise_2.0.1         stringi_1.7.12
#  [70] S4Vectors_0.36.2       randomForest_4.7-1.1  foreach_1.5.2
#  [73] BiocGenerics_0.44.0    hardhat_1.2.0         pkgbuild_1.4.0
#  [76] lava_1.7.2.1           shape_1.4.6           tuneR_1.4.2
#  [79] rlang_1.0.6            pkgconfig_2.0.3       matrixStats_
  0.63.0
#  [82] evaluate_0.20          lattice_0.20-45       purrr_1.0.1
#  [85] recipes_1.0.5          htmlwidgets_1.6.1     processx_3.8.0
#  [88] tidyselect_1.2.0       parallelly_1.34.0     plyr_1.8.8
#  [91] magrittr_2.0.3         bookdown_0.34         R6_2.5.1
#  [94] IRanges_2.32.0         generics_0.1.3        profvis_0.3.7
#  [97] withr_2.5.0            pillar_1.8.1          survival_3.4-0
# [100] abind_1.4-5            nnet_7.3-18           future.apply_
  1.10.0
# [103] tibble_3.1.8           crayon_1.5.2          car_3.1-1
# [106] xgboost_1.7.3.1        utf8_1.2.3            rmarkdown_2.20
# [109] urlchecker_1.0.1       GetoptLong_1.0.5      usethis_2.1.6
# [112] data.table_1.14.8      callr_3.7.3           ModelMetrics_
  1.2.2.2
# [115] digest_0.6.31          xtable_1.8-4          httpuv_1.6.9
# [118] signal_0.7-7           stats4_4.2.2          munsell_0.5.0
# [121] sessioninfo_1.2.2
```

2.2 Subtype Vector

In the PAD project, the **Subtype Vector** were identified based on independent cohorts under unsupervised hierarchical clustering, instead of an merged expression matrix after batch-effect control of the **sva::ComBat** function.

Here were some considerations:

1. **Batch control would damage the raw rank differences**

Here we just showed how could this happen.

```
# Data
testData <- readRDS(
  system.file("extdata",
              "testData.rds",
              package = "GSClassifier")
  )
expr_pad <- testData$PanSTAD_expr_part

# Missing value imputation
expr_pad <- na_fill(expr_pad,
                    method = 'quantile',
                    seed = 698,
                    verbose = F)

# PADi
padi <- readRDS(system.file("extdata", "PAD.train_20220916.rds",
   package = "GSClassifier"))
```

The raw rank differences were as follows:

```
# Time-consuming
rank_pad <- GSClassifier:::trainDataProc_X(
  expr_pad,
  geneSet = padi$geneSet,
  breakVec=c(0, 0.25, 0.5, 0.75, 1.0)
)
print(rank_pad$dat$Xbin[1:5,1:5])
#             ENSG00000122122 ENSG00000117091 ENSG00000163219
   ENSG00000136167
# GSM2235556               3               2               1
                4
# GSM2235557               3               3               1
                4
# GSM2235558               3               2               1
                4
# GSM2235559               3               4               1
                4
# GSM2235560               3               3               1
                4
#             ENSG00000005844
# GSM2235556               3
# GSM2235557               2
# GSM2235558               3
# GSM2235559               3
# GSM2235560               3
```

Here, the expression matrix after removing batch effect were calculated:

```
# Cleaned and normalized data for sva::Combat
expr_pad2 <- apply(expr_pad, 2, scale)
```

```
rownames(expr_pad2) <- rownames(expr_pad)
```

Remove batch effects:

```
# BatchQC data
library(sva)
# Loading required package: mgcv
# Loading required package: nlme
# This is mgcv 1.8-41. For overview type 'help("mgcv-package")'.
# Loading required package: genefilter
#
# Attaching package: 'genefilter'
# The following object is masked from 'package:ComplexHeatmap':
#
#     dist2
# Loading required package: BiocParallel
batch <- testData$PanSTAD_phenotype_part$Dataset
expr_pad2_rmbat <- ComBat(
  dat=expr_pad2,
  batch=batch,
  mod=NULL)
# Found14batches
# Adjusting for0covariate(s) or covariate level(s)
# Standardizing Data across genes
# Fitting L/S model and finding priors
# Finding parametric adjustments
# Adjusting the Data
```

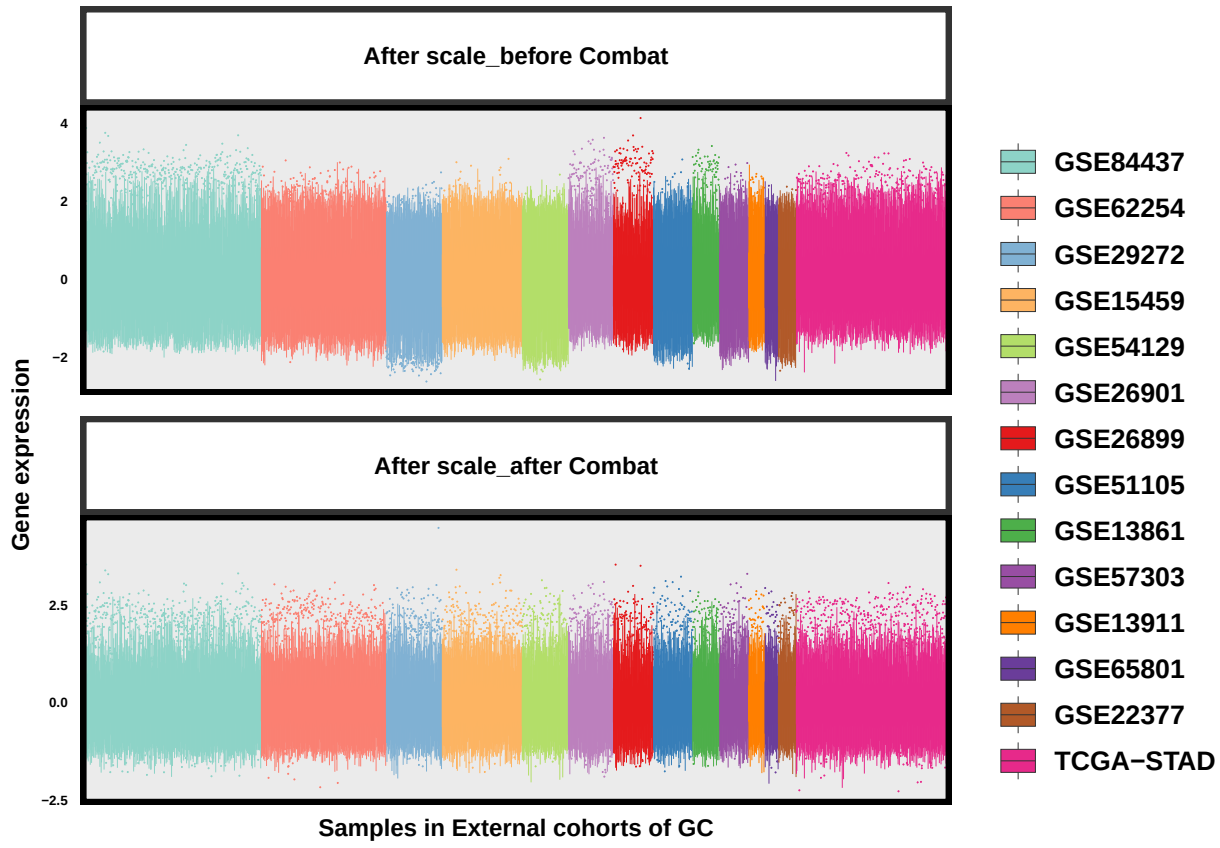
Look at the alteration of batch effects in GC datasets:

```

# Compare
```

```
df1 <- reshape2::melt(expr_pad2)
df2 <- reshape2::melt(expr_pad2_rmbat)
df3 <- rbind(
  cbind(df1, type = 'After scale_before Combat'),
  cbind(df2, type = 'After scale_after Combat')
)
df3$type <- factor(df3$type, levels = c('After scale_before
   Combat', 'After scale_after Combat'))
df3$dataset <- convert(df3$Var2,'ID','Dataset', testData$PanSTAD_
   phenotype_part)
dataset <- unique(df3$dataset)
dataset_color <- mycolor[c(1,4,5,6,7,10,21,22,23,24,25,54,56,60)]

# ggplot:     Combat
if(T){

  size=6

  p <- ggplot(df3,aes(x=Var2,y=value,fill=dataset,color=dataset))
      +
    geom_boxplot(outlier.size = -0.3, size = 0.02) +
    scale_fill_manual(
      breaks = dataset,
      values = dataset_color,
      labels = dataset
    ) +
    scale_color_manual(
      breaks = dataset,
      values = dataset_color,
      labels = dataset
```

```
  ) +
  guides(color = "none",
         fill = guide_legend(override.aes = list(size=1))) +
  facet_wrap(. ~ type,
             nrow = length(unique(df3$type)),
             scales = 'free') +
  labs(x = 'Samples in External cohorts of GC',
       y = 'Gene expression',
       fill = NULL) +
  # coord_flip() +
  theme_bw() +
  theme(
    axis.text.x = element_blank(),
    axis.text.y = element_text(size = size/15*12,colour = "
      black",face = "bold"),
    axis.title.x = element_text(size = size*1.5,colour = "black
      ",face = "bold"),
    axis.title.y = element_text(size = size*1.5,colour = "black
      ",face = "bold"),
    legend.text = element_text(size = size/15*25,colour = "
      black",face = "bold"),
    legend.title = element_text(size = size/15*25,colour = "
      black",face = "bold"),
    legend.position='right',
    strip.background = element_rect(fill="white",size = 2),
    strip.text.x = element_text(size = size*1.5,colour = "black
      ",face = "bold", margin = margin(t = 0.5, r = 0, b =
      0.5, l = 0, unit = "cm")),
    panel.border = element_rect(colour = "black",size=2),
    # panel.grid = element_blank(),
```

```
      # panel.border=element_rect(fill='transparent',color='
         transparent'),
      # axis.ticks = element_line(colour = "black",size = 2.5,
         linetype = 1,lineend = 'square'),
      axis.ticks = element_blank()
      # axis.line = element_line(colour = "black",size = 2.5,
         linetype = 1,lineend = 'square')
    )
  print(p)
}
```



The rank differences after batch-effect control were as follows:

```
# Time-consuming
rank_pad_rmbat <- GSClassifier:::trainDataProc_X(
```

```
  expr_pad2_rmbat,
  geneSet = padi$geneSet,
  breakVec=c(0, 0.25, 0.5, 0.75, 1.0)
)
```

The comparison demonstrated that the adjustment for batch effects would change the rank differences:

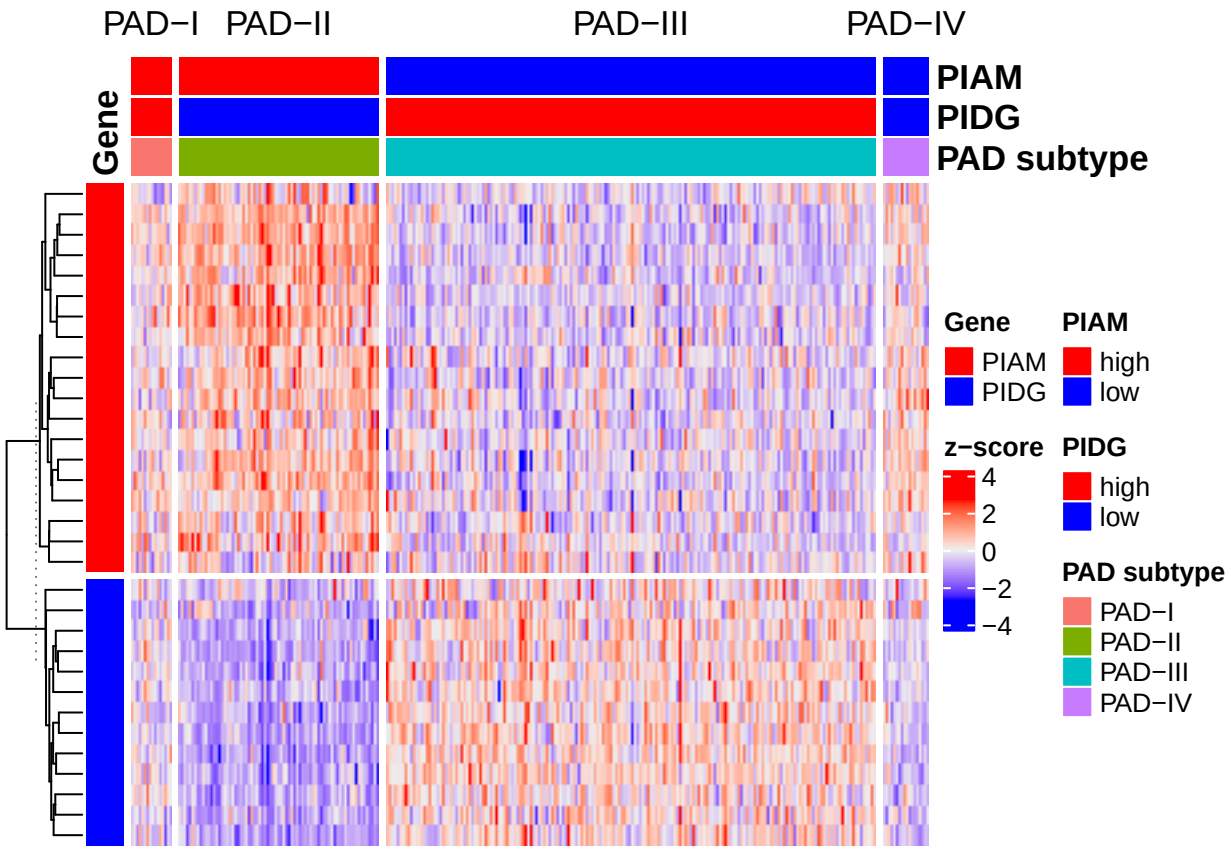
```
mean(rank_pad_rmbat$dat$Xbin == rank_pad$dat$Xbin)
# [1] 0.7788832
```

The rank differences, the base of TSP normalization, were critical for model training and subtype identification, so it's not recommended to adjust for batch effects using **sva::ComBat** before model training in GSClassifier.

2. **Batch control would lead to an unbalanced subtype vectors**.

Here, a heatmap were used to show the self-clustering of the training cohort based on **PIAM** and **PIDG**:

```
res <- PAD(
  expr = expr_pad2_rmbat,
  cluster.method = "ward.D2",
  subtype = "PAD.train_20220916",
  verbose = T
)
```



Look at the percentage of PAD subtypes:

```
table(res$Data$`PAD subtype`)/ncol(expr_pad2_rmbat)
#
#       PAD-I      PAD-II     PAD-III      PAD-IV
# 0.05169938 0.25801819 0.63140258 0.05887985
```

This results demonstrated the unbalanced self-clustering of PAD subtypes based on the expression matrix after **sva::ComBat**, which would damage the performance of trained models.

2.3 Missing value imputation (MVI)

Due to reasons like weak signal, contamination of microarray surfaces, inappropriate manual operations, insufficient resolution, or systematic errors during the laboratory process [8–10],

missing value in high-input genetic data is common. Generally, tiny missing values could be just dealt with case deletion, while the biological discovery might be damaged when the missing rate tops 15% [11,12]. Currently, lots of methods, including statistic-based or machine learning-based methods (Figure 2.1), had been developed for **missing value imputation (MVI)** [12]. Wang et al [13] categorized MVI methods into simple (zeros or average), biology knowledge-, global learning-, local learning-, and hybrid-based methods. In order to satisfy the working conditions of **xgboost** [14] functions (`xgb.train`, `xgboost` , and `xgb.cv`) in GSClassifer, missing values in the expression matrix must be deleted or imputation.

In **PAD** project, several strategies were applied to reduce the impact of missing values as possible. First, both **PIAM** and **PIDG** in **PAD** project were curated GEPs that were not missing in over 80% of gastric cancer datasets. Here we showed the actual distribution of missing values across samples in gastric cancer datasets we used.

```
# Data
testData <- readRDS(
  system.file("extdata",
              "testData.rds",
              package = "GSClassifier")
  )
expr_pad <- testData$PanSTAD_expr_part

# Missing value
expr_pad_na <- apply(expr_pad, 2,
                     function(x) sum(is.na(x))/length(x))
expr_pad_na_df <- data.frame(
  sample = names(expr_pad_na),
  prob = as.numeric(expr_pad_na),
  stringsAsFactors = F
)
```

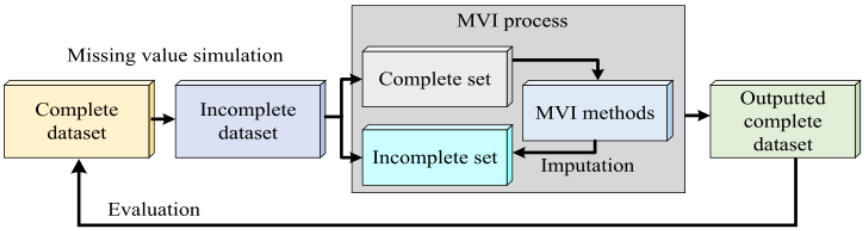


Fig. 3. The typical experimental configuration for MVI procedures to impute the missing values in any attributes [2].

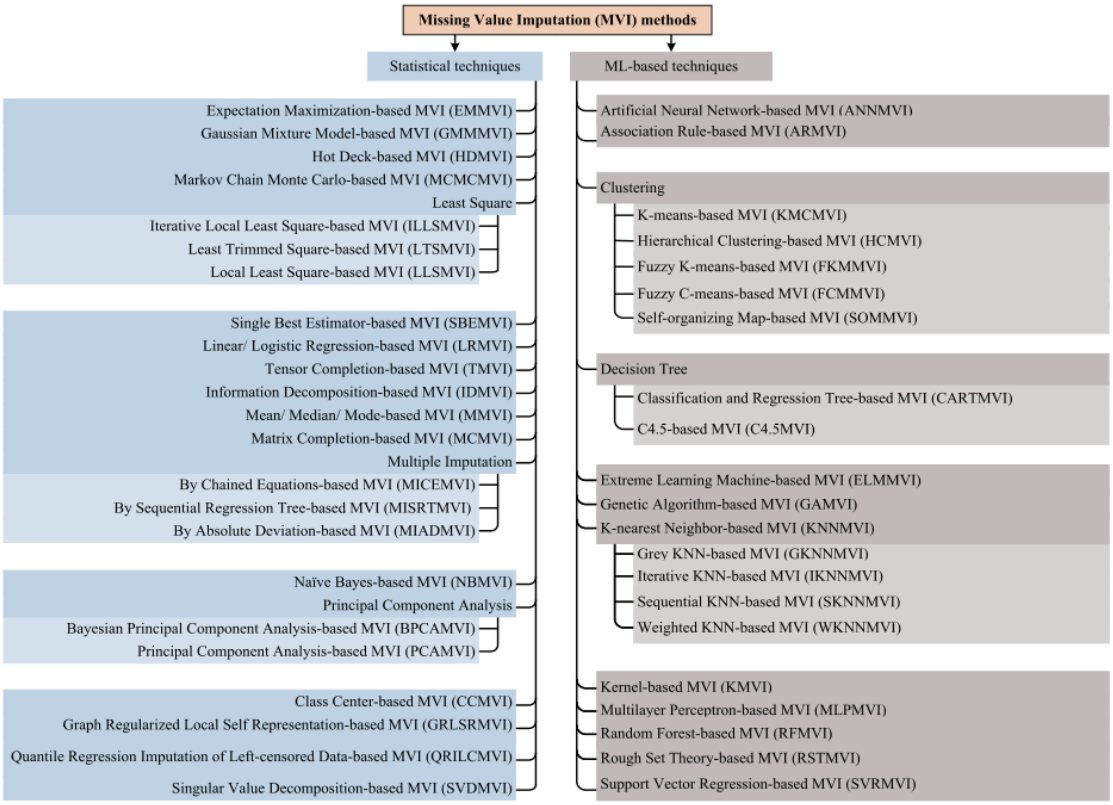


Fig. 4. The categorized tree exhibition of the commonly employed MVI methods, available in the literature.

Figure 2.1: Missing value imputation methods reviewed by Hasan et al.

As shown in Figure 2.2, the percentage of all samples in gastric cancer datasets we used is lower than 8%.

```
# ggplot
p1 <- ggplot(data = expr_pad_na_df,
       aes(x = sample, y = prob)) +
  geom_bar(stat = 'identity', color = mycolor[3]) +
  scale_y_continuous(labels=scales::percent) +
  labs(x = 'Samples in gastric cancer cohorts',
       y = 'Percentage of missing value') +
  theme_bw() +
  theme(
    axis.text.x = element_blank(),
    axis.ticks = element_blank(),
    axis.title = element_text(size = 15),
    axis.text = element_text(size = 12)
  )
print(p1)
```

Second, we did conduct some MVI strategies to deal with data before model training in **GSClassifier**. Due to the low missing rate of our experimental data, we just **set missing values as zero** during model training and subtype identification in the early version of PADi (**PAD.train.v20200110**). The model seemed to be robust in both the internal cohort and external cohorts, and greatly predicted the response to immune checkpoint inhibitors (ICIs) in advanced gastric cancer.

In the latest version of PADi (**PAD.train.v20220916**), we designed the so-called **quantile** algorithm for random MVI during **PADi** model training, which also seemed to work well for PADi model training.

Here, we demonstrated the principle of **quantile** algorithm in the simulated dataset:

```
# Simulated data
```

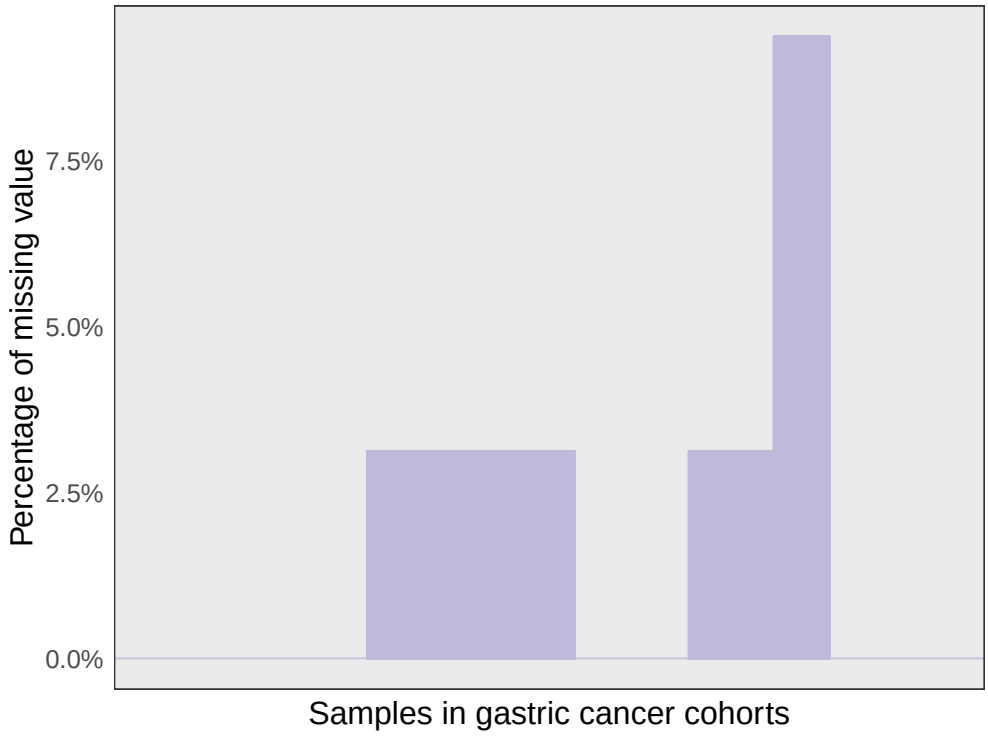


Figure 2.2: The distribution of missing value across gastric cancer samples.

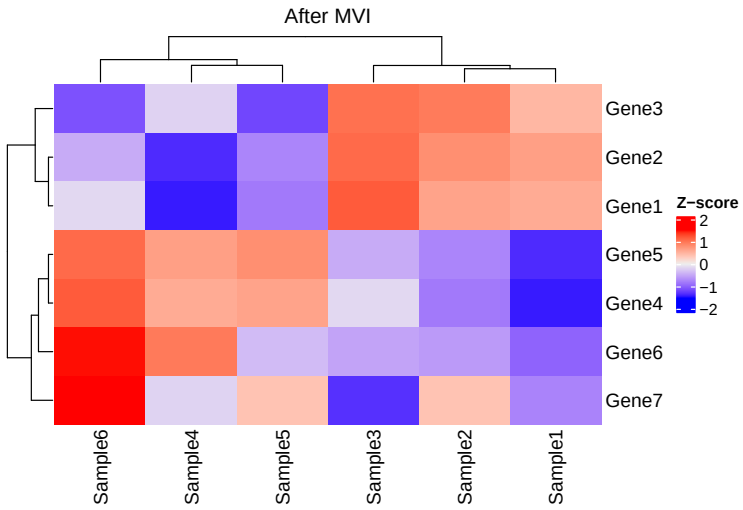
```
x <- read_xlsx('./data/simulated-data.xlsx', sheet = 'RNA')
expr0 <- as.matrix(x[,-1])
rownames(expr0) <- as.character(as.matrix(x[,1])); rm(x)

# MVI with Quantile algorithm
expr <- expr0
na.pos <- apply(expr,2,is.one.na)
set.seed(478); seeds <- sample(1:ncol(expr)*10, sum(na.pos),
   replace = F)
tSample <- names(na.pos)[na.pos]
quantile_vector <- (1:1000)/1000
for(i in 1:length(tSample)){ # i=1

  sample.i <- tSample[i]
  expr.i <- expr[, sample.i]
```

```
  expr.i.max <- max(expr.i, na.rm = T)
  expr.i.min <- min(expr.i, na.rm = T)
  set.seed(seeds[i]);

  # Details of quantile algorithm
  expr.i[is.na(expr.i)] <-
    expr.i.min +
    (expr.i.max-expr.i.min) * sample(quantile_vector,
                                     sum(is.na(expr.i)),
                                     replace = T)
  expr[, sample.i] <- expr.i
}



# Report
cat('RNA expression:', '\n')
print(expr0)
cat('\n')
cat('RNA expression without NA value:', '\n')
print(expr)
# RNA expression:
#       Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
# Gene1    0.51    0.52    0.60    0.21    0.30    0.40
# Gene2    0.52    0.54    0.58    0.22    0.31    0.35
# Gene3    0.53    0.60    0.61      NA    0.29    0.30
# Gene4    0.21    0.30    0.40    0.51    0.52    0.60
# Gene5    0.22    0.31    0.35    0.52    0.54    0.58
# Gene6    0.23    0.29    0.30    0.53      NA    0.61
# Gene7    0.10    0.12    0.09    0.11    0.12    0.14
#
```

```
# RNA expression without NA value:
#       Sample1 Sample2 Sample3 Sample4 Sample5 Sample6
# Gene1    0.51    0.52    0.60 0.21000 0.30000    0.40
# Gene2    0.52    0.54    0.58 0.22000 0.31000    0.35
# Gene3    0.53    0.60    0.61 0.43256 0.29000    0.30
# Gene4    0.21    0.30    0.40 0.51000 0.52000    0.60
# Gene5    0.22    0.31    0.35 0.52000 0.54000    0.58
# Gene6    0.23    0.29    0.30 0.53000 0.32622    0.61
# Gene7    0.10    0.12    0.09 0.11000 0.12000    0.14
```

Look at the new matrix via heatmap, where the clustering result is not significantly disturbed after MVI:

```
Heatmap(t(scale(t(expr))), name = "Z-score", column_title = "
   After MVI")
```



Because missing values might damage the integrity of biological information, we explored **how much the number of missing values in one sample impacts subtype identification via PADi**. The steps are as follows: (i) we used the "quantile" algorithm to do MVI in the internal validation cohort of gastric cancer; (ii) we randomly masked different proportions of genes as zero expression; (iii) we calculated the relative multi-ROC [15] (masked

data vs. MVI data). In **GSClassifier**, we developed a function called **mv_tolerance** to complete the task.

(i) Load the internal validation cohort:

```
# Internal validation cohort
testData <- readRDS(
  system.file("extdata", "testData.rds", package = "GSClassifier"
     )
  )
expr_pad <- testData$PanSTAD_expr_part
modelInfo <- modelData(
  design = testData$PanSTAD_phenotype_part,
  id.col = "ID",
  variable = c("platform", "PAD_subtype"),
  Prop = 0.7,
  seed = 19871
)
validInform <- modelInfo$Data$Valid
expr_pad_innervalid <- expr_pad[,validInform$ID]
```

(ii) Missing value tolerance analysis:

```
# Time-consuming
mvt <- mv_tolerance(
  X = expr_pad_innervalid,
  gene.loss = c(2, 4, 6, 8, 10, 12),
  levels = c(1, 2, 3, 4),
  model = "PAD.train_20220916",
  seed = 487,
  verbose = T
)
```

(iii) multi-ROC analysis:

```
# Data
mvt_auc <- mvt$multiAUC
mvt_auc_df <- data.frame()
for(i in 1:length(mvt_auc)){ # i=1
  df.i <- data.frame(
    x = as.integer(Fastextra(names(mvt_auc)[i], '=', 2)),
    y = as.numeric(mvt_auc[[i]]$auc),
    stringsAsFactors = F
  )
  mvt_auc_df <- rbind(mvt_auc_df, df.i)
}
```

```
# Plot
p2 <- ggplot(mvt_auc_df, aes(x,y)) +
  geom_point() +
  scale_x_continuous(breaks = c(2, 4, 6, 8, 10, 12)) +
  stat_smooth(formula = y ~ x,method = 'glm') +
  labs(x = 'No. of missing value',
       y = 'Relative AUC in multi-ROC analysis') +
  theme(
    axis.title = element_text(size = 15),
    axis.text = element_text(size = 12)
  )
print(p2)
```

As shown in Figure 2.3, there is a linear negative correlation between the number of missing values (missing rate ranges from 6.25% to 37.5%) and the subtype identification performance of **PADi** model. One of the reasons might be that PIAM/PIDG were small GEPs, so little gene loss might significantly impact the performance of **PADi**. By the way, there is no

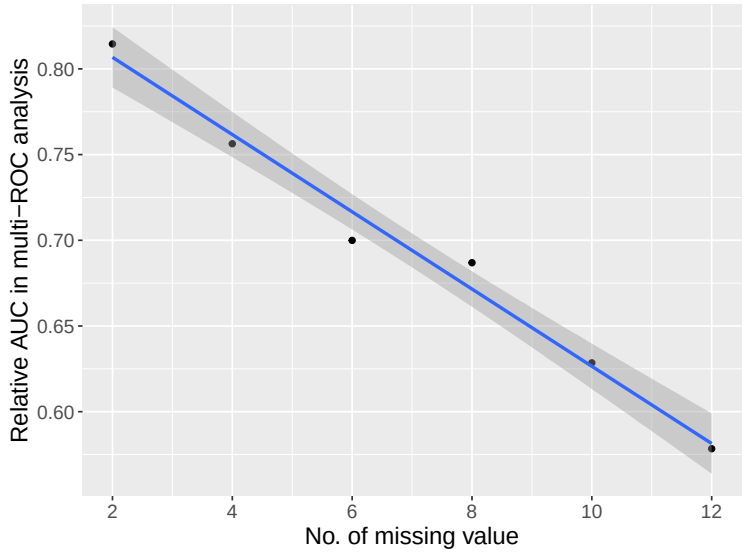


Figure 2.3: The association between the number of missing value and subtype identification performance.

missing value in PIAM/PIDG of the 'Kim2018' cohort, an external validation cohort for ICIs therapy response prediction via **PADi**. Nonetheless, we still used the **zero strategy** during subtype identification of **PADi** if any missing values exist, because randomization might make the result unstable, which is not suitable for clinical decision.

In conclusion, the zero or "quantile" strategy could be applied for MVI before **GSClassifier** model training. However, missing values should be avoided as possible in subtype identification for missing values could damage the performance of **GSClassifier** models. Nonetheless, due to low-input GEPs used in **PADi** model (No. of Gene=32), it's easy to avoid missing value in clinical practice.

2.4 Batch effect

TSP was widely applied to control batch effects in transcriptomic data [16–23]. Still, we tested whether **TSP** is a robust method for batch effect control in real-world data. As demonstrated in Figure 2.4, the obvious batch effects across gastric cancer datasets were significantly reduced after **TSP** normalization.

To confirm the association between **gene counts** in modeling and batch effect control via

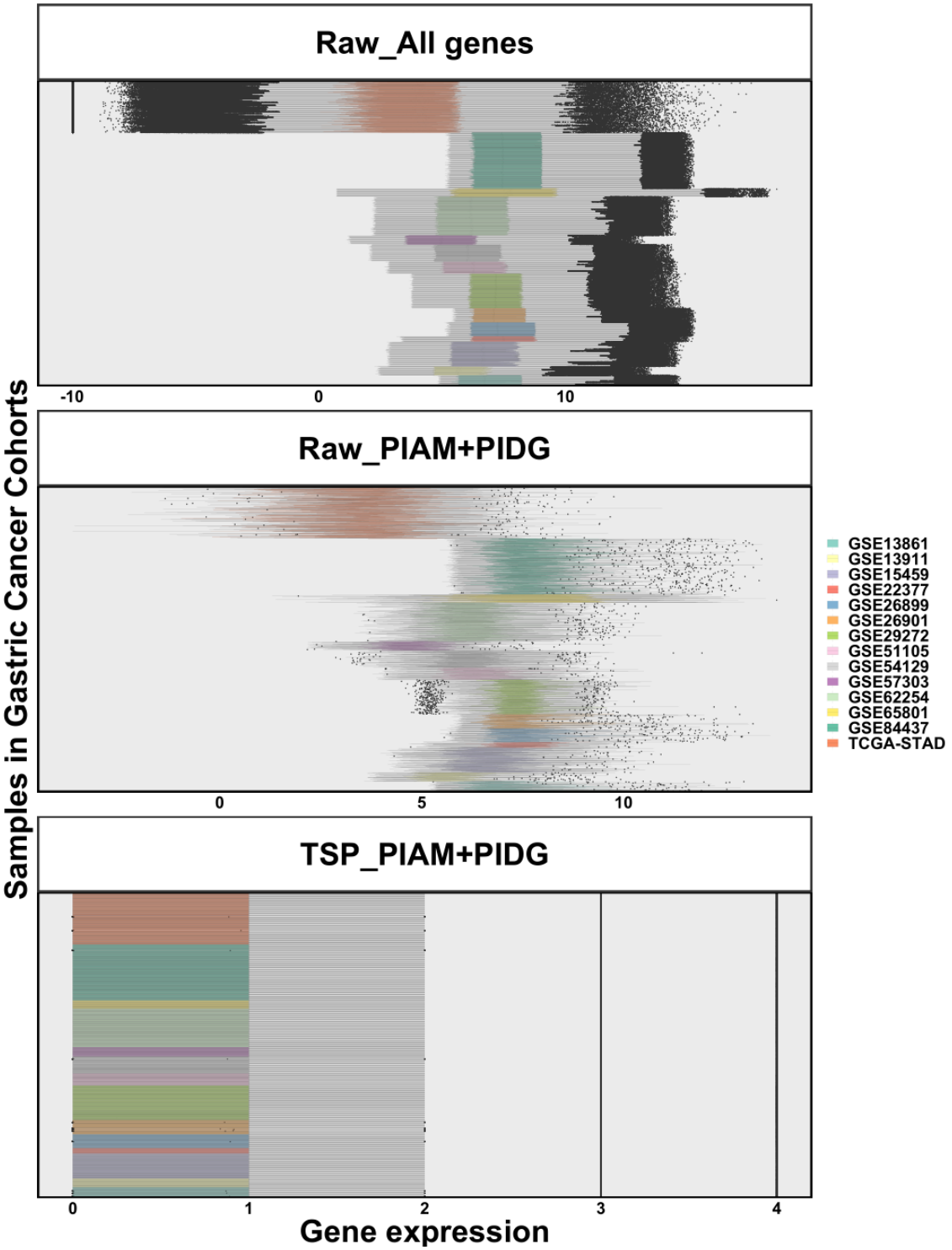


Figure 2.4: Batch effects across gastric cancer cohorts. All gene pairs were used because subtype vectors were not specified. Top: Raw expression of all genes across samples. Middle: Raw expression of PIAM and PIDG across samples. Bottom: TSP of PIAM and PIDG across samples.

TSP normalization, we selected random genes with counts ranging from 4 to 80 for TSP matrix establishment. As shown in Figure 2.5, **TSP** normalization works greatly in different gene counts for batch effect control compared with the raw expression matrix.

2.5 Subtype number

For PAD subtypes, it's easy to determine the subtype number as 4. First, PAD subtypes are identified via 2 simple GEPs. The binary status (high/low expression) of 2 GEPs consists of a 2×2 matrix and hence the subtype number is exactly 4. Second, four PAD subtypes displayed different genetic/epigenetic alterations and clinical features (survival and ICI response), indicating that it's biologically meaningful to distinguish gastric cancer into 4 immune subtypes. Third, clinicians are familiar with four subtypes, for the subtype number of the classical TNM stage in clinical oncology is 4 (Stage I to IV).

Also, there's another more simple situation—determining the subtype number with only one GEP, where 2 (high/low) or 3 (low/moderate/high) deserve to be tried. However, with the number of GEPs increasing, the situation would become more complex. Regrettably, GSClassifier can not help determine what subtype number should be used; GSClassifier only promises a robust model no matter how many GEPs or subtypes. **There's no gold standard for subtype number selection, which is more like art instead of math**.

Nevertheless, there're several suggestions we can follow for best practice. First, the R package "**ConsensusClusterPlus**" [24–27] can help figure out consensus clustering for transcriptomic data, which provides visualization (consensus matrices, consensus cumulative distribution function plot, delta area plot, tracking plot, and so on) for clustering quality control. Second, **paying more attention to the biological problem usually gives extra or even crucial clues for the subtype number decision**.

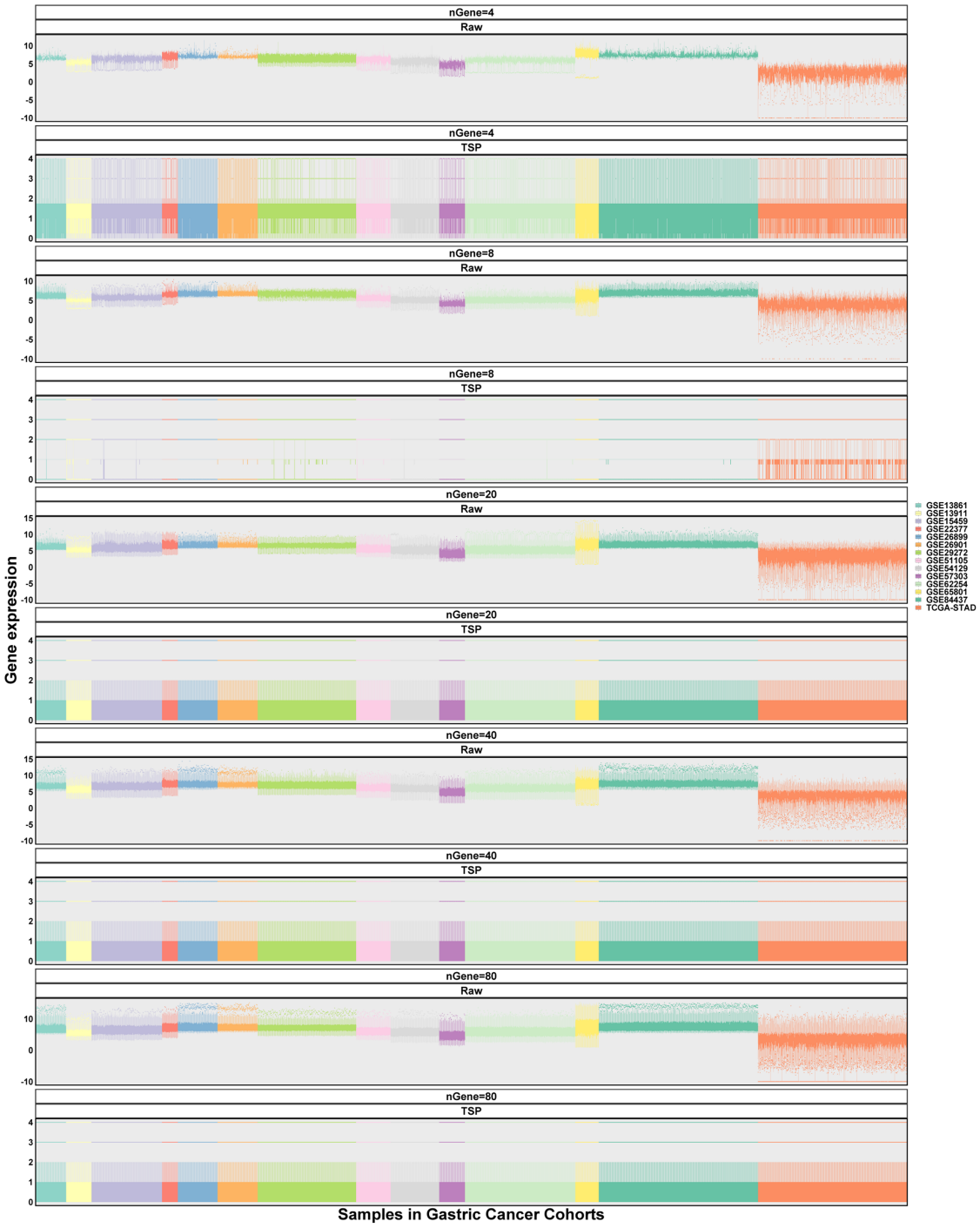


Figure 2.5: Batch effects of random genes across gastric cancer cohorts. All gene pairs were used because subtype vectors were not specified. Gene counts 4, 8, 20, 40, and 80 were detected. Data of set difference were not available because only one gene set were applied.

Chapter 3

Quick start

3.1 About

- Although with bright prospects in Pan-disease analysis, GSClassifier was primarily developed for clinic-friendly immune subtypes of gastric cancer (GC). Currently, only `PAD` subtypes and `PADi` for GC were supported. We would try to support more cancer types in the future as possible. More details in Plans in the future section.
- Gibbs' `PanCancer immune subtypes` based on five gene signatures (485 genes) could also be called in `GSClassifier`, with a pre-trained model from the ImmuneSubtypeClassifier package. If you use their jobs, please cite these papers.
- Particularly, all normal tissues should be eliminated before subtype identificaiton for cancer research.

3.2 Package

```
# Install "devtools" package
if (!requireNamespace("devtools", quietly = TRUE))
  install.packages("devtools")

# Install dependencies
```

```
if (!requireNamespace("luckyBase", quietly = TRUE))
  devtools::install_github("huangwb8/luckyBase")

# Install the "GSClassifier" package
if (!requireNamespace("GSClassifier", quietly = TRUE))
  devtools::install_github("huangwb8/GSClassifier")

# Load needed packages
library(GSClassifier)
# Loading required package: luckyBase
```

3.3 Data

To lower the learning cost of `GSClassifier`, we provide some test data:

```
testData <- readRDS(system.file("extdata", "testData.rds",
   package = "GSClassifier"))
```

Explore the `testData`:

```
names(testData)
# [1] "Kim2018_3"              "PanSTAD_phenotype_part" "PanSTAD_
   expr_part"
```

3.4 PAD

3.4.1 The work flow of PAD exploration

The basic process of PAD exploration was summarized in Figure 3.1

- With WGCNA method and TIMER datasets, Pan-Immune Activation Module (PIAM) was identified as a GEP representing co-infiltration of immune cells in the tumor mi-

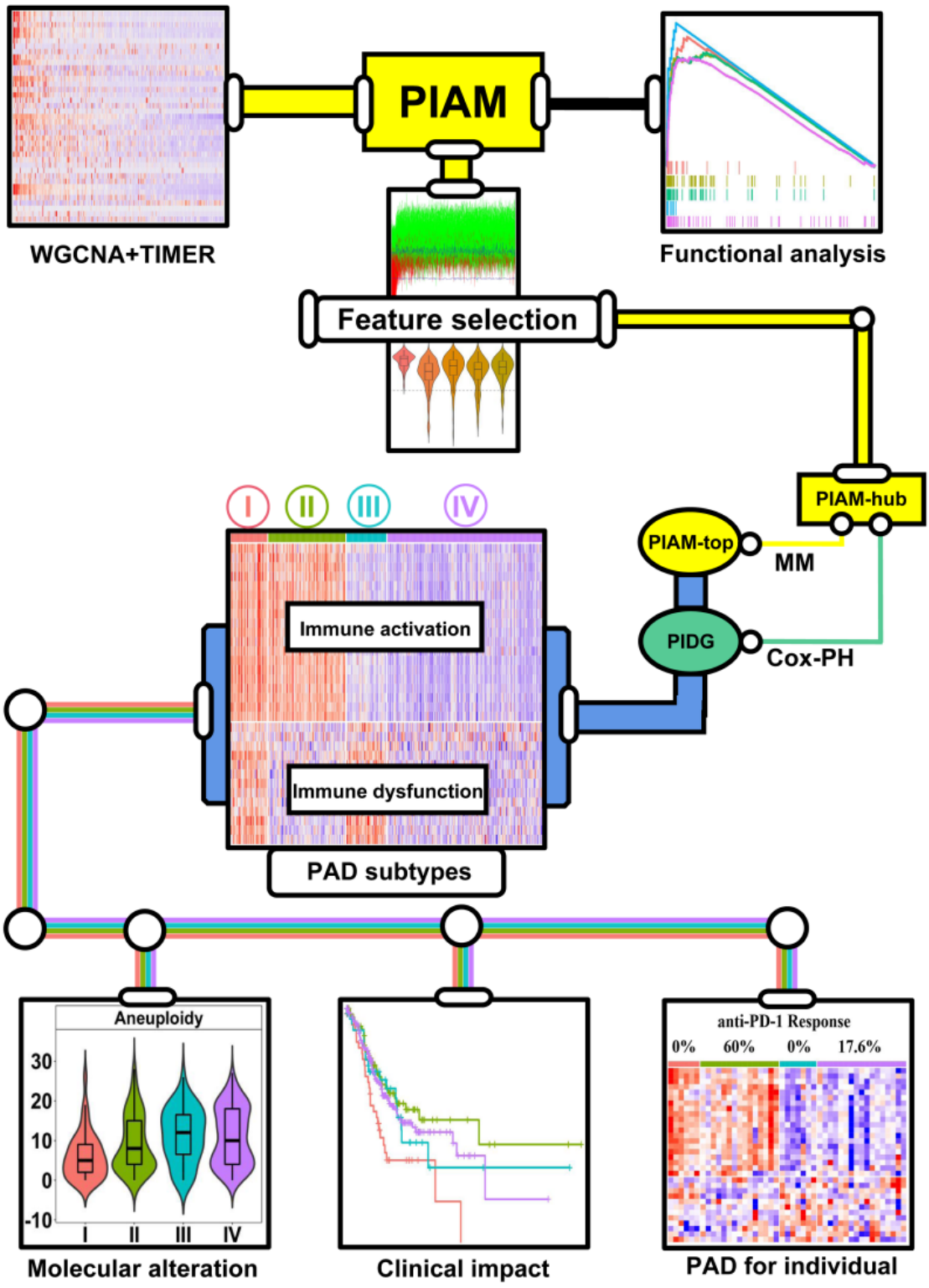


Figure 3.1: The process of PAD subtypes establishment

croenvironment. Functional analysis such as GSEA was done for the exploration of PIAM functions.

- With feature selection based on Boruta algorithm, missing value control (without missing value in over 80% of GC datasets we used), we retained 101 genes and named as "PIAM-hub".
- Pan-Immune Dysfunction Genes (PIDG) were explored based on "PIAM-hub" via a strategy similar to the computational framework of the Tumor Immune Dysfunction and Exclusion (TIDE) database [20]. Finally, 13 PIDGs were selected for downstream analysis as they were further validated in 2 or more external GC cohorts.
- To further reduce PIAM-hub for downstream modeling, genes with Mean of Correlation with Eigengene (MCE) 0.8 were selected and termed as "PIAM-top" subset (n=19).
- "PIAM-top" and PIDG, two curated tiny GEPs, were applied to establish Pan-immune Activation and Dysfunction (PAD) subtypes (PAD-I, PIAMhighPIDGhigh; PAD-II, PIAMhighPIDGlow; PAD-III, PIAMlowPIDGhigh; and PAD-IV, PIAMlowPIDGlow) in independent GC cohorts.
- Molecular alteration and patient survival across PAD subtypes were analyzed to figure out its biological and clinical impact. Also, a GSClassifier model called "PAD for individual" (PADi) was established for personalized subtype identification for immune checkpoint inhibitors response prediction in GC (More details in Online Section/PDF).

3.4.2 Preparation of the test data

Load phenotype data:

```
design <- testData$PanSTAD_phenotype_part
table(design$Dataset)
#
#  GSE13861  GSE13911  GSE15459  GSE22377  GSE26899  GSE26901
   GSE29272  GSE51105
```

```
#         65         39        192         43         96        108
         134         94
#  GSE54129  GSE57303  GSE62254  GSE65801  GSE84437 TCGA-STAD
#       111         70        300         32        433        372
```

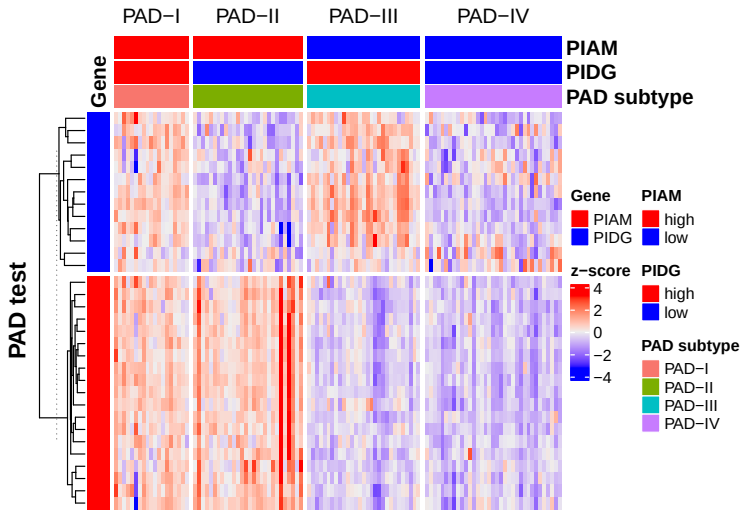
Load target sample IDs in `GSE54129` cohort:

```
target_ID <- design$ID[design$Dataset %in% 'GSE54129']
expr <- testData$PanSTAD_expr_part[,target_ID]
head(expr[,1:10])
#                 GSM1308413 GSM1308414 GSM1308415 GSM1308416
   GSM1308417
# ENSG00000122122   7.888349   7.623663   6.873493   6.961102
   7.150572
# ENSG00000117091   7.051760   6.217445   5.651839   5.830996
   5.908532
# ENSG00000163219   6.056472   5.681844   5.411533   5.652684
   5.555147
# ENSG00000136167   9.322191   8.765794   8.502315   8.838166
   8.845952
# ENSG00000005844   7.119594   6.023631   5.400999   6.172863
   6.059838
# ENSG00000123338   7.204051   6.925328   6.259809   6.610681
   6.595882
#                 GSM1308418 GSM1308419 GSM1308420 GSM1308421
   GSM1308422
# ENSG00000122122   7.871423   6.953329   8.334037   6.764335
   6.522554
# ENSG00000117091   6.526917   5.646446   6.617520   5.637693
   5.742848
# ENSG00000163219   5.962885   5.361763   5.975842   5.330428
```

```
   5.172705
# ENSG00000136167    9.366074    8.675718    9.118517    8.614068
   8.114096
# ENSG00000005844    6.523530    6.129181    7.331588    5.547059
   5.867118
# ENSG00000123338    6.699790    6.935390    7.050288    6.536710
   6.200269
```

3.4.3 Unsupervised clustering

```
res_pad <- PAD(
  expr = expr,
  cluster.method = "ward.D2",
  extra.annot = NULL,
  plot.title = 'PAD test',
  subtype = "PAD.train_20220916",
  verbose = T
)
```




```
# Use default PIAM...
# Use default PIDG...
```

```
# Gene match: 100%.
# Done!
```

3.4.4 Of note

- It's strongly recommended that the gene type of `expr` should be always the same, such as ENSEMBL genes (ENSG00000111640 for GAPDH, for example).
- `PAD` function is only for datasets with lots of samples for its classification depends on population-based unsupervised clustering. `PAD` is population-dependent and non-personalized.
- Beta characteristics: You could try random forest classification based on the `randomForest` package or methods in `stats::hclust`.

3.5 PADi

- In `GSClassifier`, `PADi` is a pre-trained out-of-the-box model for GC personalized PAD subtypes calling.
- During the subtype calling, the gene rank relations based on individuals instead of the relative values across samples would be used. Thus, **you don't have to do batch normalization** even though the data (the `X` input) come from multiple cohorts or platforms.
- More limitations were discussed in our paper that you had better know.

In this section, we would show how to use `PADi` function series: `PADi`, `callEnsemble`, and `parCallEnsemble` functions.

3.5.1 Preparation of the test data

```
X <- testData$Kim2018_3
head(X)
```

```
#                    PB-16-002   PB-16-003   PB-16-004
# ENSG00000121410  0.07075272 -2.08976724 -1.43569557
# ENSG00000148584 -1.49631022 -0.23917056  0.94827471
# ENSG00000175899 -0.77315329  0.52163146  0.91264015
# ENSG00000166535 -0.28860715 -0.45964255 -0.38401295
# ENSG00000256069 -0.25034243  0.06863867  0.14429081
# ENSG00000184389  0.08215945 -0.05966481  0.04937924
```

3.5.2 Use a specific function called `PADi`

Very simple, just:

```
res_padi <- PADi(X = X, verbose = F)
```

Check the result:

```
head(res_padi)
#   SampleIDs BestCall BestCall_Max           1          2
           3         4
# 1 PB-16-002        4            4 0.023372779 0.02631794
   0.04864784 0.3336484
# 2 PB-16-003        4            4 0.007271698 0.08650742
   0.01974812 0.9530730
# 3 PB-16-004        4            4 0.011559768 0.02922151
   0.09018894 0.8649045
```

Actually, `PADi` is exactly based on a general function called `callEnsemble`.

3.5.3 Use the `callEnsemble` function

Also simple, just:

```
res_padi <- callEnsemble(
```

```
    X = X,
    ens = NULL,
    geneAnnotation = NULL,
    geneSet = NULL,
    scaller = NULL,
    geneid = "ensembl",
    matchmode = 'fix',
    subtype = "PAD.train_20220916",
    verbose = F
  )
```

Check the result:

```
head(res_padi)
#   SampleIDs BestCall BestCall_Max          1          2
            3         4
# 1 PB-16-002        4            4 0.01338872 0.01624520
   0.03965218 0.8052567
# 2 PB-16-003        4            4 0.04709511 0.08833681
   0.03879361 0.6244038
# 3 PB-16-004        4            4 0.01389035 0.03638009
   0.05852707 0.6980438
```

3.5.4 Parallel strategy for PADi

- Sometimes, the number of patients for subtype callings could be huge (hundreds or even tens of thousands). Thus, parallel computing (Windows or Linux pass; not tested in Mac or other OS) was also developed in the current version of the `GSClassifier` package.
- The parameter `numCores` was used to control the No. of CPU for computing (which depends on your CPU capacity).

```
# No run for the tiny test data. With errors.

# Method 1:
res_padi <- PADi(X = X, verbose = F, numCores = 4)

# Method 2:
res_padi <- parCallEnsemble(
  X = X,
  ens = NULL,
  geneAnnotation = NULL,
  geneSet = NULL,
  scaller = NULL,
  geneids = 'ensembl',
  matchmode = 'fix',
  subtype = 'PAD.train_20220916',
  verbose = T,
  numCores = 4)
```

3.5.5 Single sample subtype calling

In clinical practice, the single sample subtype calling might be one of the most common scenarios and is also supported by functions of the `PADi` series.

Supposed that there is a GC patient, its information should be:

```
X_ind <- X[,1]; names(X_ind) <- rownames(X)
head(X_ind)
# ENSG00000121410 ENSG00000148584 ENSG00000175899 ENSG00000166535
    ENSG00000256069
#      0.07075272     -1.49631022     -0.77315329     -0.28860715
       -0.25034243
```

```
# ENSG00000184389
#      0.08215945
```

Or it can also be another format:

```
X_ind <- as.matrix(X[,1]); rownames(X_ind) <- rownames(X)
head(X_ind)
#                         [,1]
# ENSG00000121410  0.07075272
# ENSG00000148584 -1.49631022
# ENSG00000175899 -0.77315329
# ENSG00000166535 -0.28860715
# ENSG00000256069 -0.25034243
# ENSG00000184389  0.08215945
```

Similar to multiples sample calling, just:

check the result:

```
head(res_padi)
#   SampleIDs BestCall BestCall_Max          1          2
           3         4
# 1    target        4            4 0.02337278 0.02631794
   0.04864784 0.3336484
```

Similarly, there is alternative choice:

```
res_padi <- callEnsemble(
    X = X_ind,
    ens = NULL,
    geneAnnotation = NULL,
    geneSet = NULL,
    scaller = NULL,
```

```
    geneid = "ensembl",
    matchmode = 'fix',
    subtype = "PAD.train_20220916",
    verbose = F
  )
```

Check the result:

```
head(res_padi)
#   SampleIDs BestCall BestCall_Max          1          2
            3          4
# 1    target        4            4 0.01338872 0.0162452
   0.03965218 0.8052567
```

3.5.6 Of note

- In the results of `PADi`, two types of subtypes (`BestCall` and `BestCall_Max`) were integrated. `BestCall` was predicted via an **xgboost** model based on prior knowledge of `PAD` subtypes and the possibility matrix (columns 4 to 7 of four-subtype calling, for example), while `BestCall_Max` was predicted via maximum strategy. Empirically, `BestCall` seemed to be a better choice.
- `PADi` is individual-dependent and personalized, which means that the result of subtype calling would not be influenced by the data of others.

3.6 Use external models from luckyModel package

In the future, there might be lots of models available as a resource of `GSClassifier`, such as luckyModel. Here we show how `luckyModel` support `GSClassifier`.

First, install and load `luckyModel`:

```
# Install luckyModel
```

```
if (!requireNamespace("luckyModel", quietly = TRUE))
    devtools::install_github("huangwb8/luckyModel")
library(luckyModel)
```

Check projects supported in current `luckyModel`:

```
list_project()
# [1] "GSClassifier"
```

Check available models in the project:

```
list_model(project='GSClassifier')
# Available models in GSClassifier:
#   *Gibbs_PanCancerImmuneSubtype_v20190731
#   *HWB_PAD_v20200110
#   *HWB_PAD_v20220916
```

Here, `HWB_PAD_v20200110` is a standard name of `PADi`. They are the same.

Taking `PADi` as an example, we here show how to use an external model from `luckyModel`. First, load a model:

```
model <- lucky_model(project = 'GSClassifier',
                     developer='HWB',
                     model = 'PAD',
                     version = 'v20200110')
```

Then, check the gene id type:

```
model$geneSet
# $PIAM
#  [1] "ENSG00000122122" "ENSG00000117091" "ENSG00000163219" "
   ENSG00000136167"
```

```
#  [5] "ENSG00000005844" "ENSG00000123338" "ENSG00000102879" "
   ENSG00000010671"
#  [9] "ENSG00000185862" "ENSG00000104814" "ENSG00000134516" "
   ENSG00000100055"
# [13] "ENSG00000082074" "ENSG00000113263" "ENSG00000153283" "
   ENSG00000198821"
# [17] "ENSG00000185811" "ENSG00000117090" "ENSG00000171608"
#
# $PIDG
#  [1] "ENSG00000116667" "ENSG00000107771" "ENSG00000196782" "
   ENSG00000271447"
#  [5] "ENSG00000173517" "ENSG00000134686" "ENSG00000100614" "
   ENSG00000134247"
#  [9] "ENSG00000109686" "ENSG00000197321" "ENSG00000179981" "
   ENSG00000187189"
# [13] "ENSG00000140836"
```

The model should use `ensembl` as the value of `geneid` parameter in `callEnsemble` series.

Next, you can use the model like:

```
res_padi <- callEnsemble(
  X = X,
  ens = model$ens$Model,
  geneAnnotation = model$geneAnnotation,
  geneSet = model$geneSet,
  scaller = model$scaller$Model,
  geneid = "ensembl",
  matchmode = 'fix',
  subtype = NULL,
  verbose = F
```

```
)
```

Or just:

```
res_padi <- callEnsemble(
  X,
  ens = NULL,
  geneAnnotation = NULL,
  geneSet = NULL,
  scaller = NULL,
  geneid = "ensembl",
  matchmode = 'fix',
  subtype = model,
  verbose = F
)
```

They are exactly the same.

Finally, check the result:

```
head(res_padi)
#   SampleIDs BestCall BestCall_Max           1          2
            3         4
# 1 PB-16-002        4            4 0.023372779 0.02631794
   0.04864784 0.3336484
# 2 PB-16-003        4            4 0.007271698 0.08650742
   0.01974812 0.9530730
# 3 PB-16-004        4            4 0.011559768 0.02922151
   0.09018894 0.8649045
```

3.7 PanCancer immune subtypes

`GSClassifier` could also call the `PanCancer immune subtypes` of Gibbs' [2].

First, see data available in current `GSClassifier`:

```
GSClassifier_Data()
# Available data:
# Usage example:
#   ImmuneSubtype.rds
#   PAD.train_20200110.rds
#   PAD.train_20220916.rds
#   PAD <- readRDS(system.file("extdata", "PAD.train_20200110.rds
   ", package = "GSClassifier"))
#   ImmuneSubtype <- readRDS(system.file("extdata", "
   ImmuneSubtype.rds", package = "GSClassifier"))
```

Let's use our test data to do this:

```
X <- testData$Kim2018_3
symbol <- convert(rownames(X))
rownames(X) <- symbol
X <- X[!is.na(symbol),]
dim(X)
# [1] 19118     3
```

Have a check

```
head(X)
#                 PB-16-002   PB-16-003   PB-16-004
# A1BG           0.07075272 -2.08976724 -1.43569557
# A1CF          -1.49631022 -0.23917056  0.94827471
# A2M           -0.77315329  0.52163146  0.91264015
# A2ML1         -0.28860715 -0.45964255 -0.38401295
```

```
# RP11-118B22.6 -0.25034243  0.06863867  0.14429081
# A3GALT2        0.08215945 -0.05966481  0.04937924
```

PanCan Immune Subtype callings:

```
res_pis <- callEnsemble(
    X = X,
    ens = NULL,
    geneAnnotation = NULL,
    geneSet = NULL,
    scaller = NULL,
    geneid = "symbol",
    matchmode = 'free',
    subtype = "ImmuneSubtype",
    verbose = F
  )
```

Check the result:

```
head(res_pis)
#   SampleIDs BestCall BestCall_Max            1           2
              3
# 1 PB-16-002        2            2 1.693409e-03 0.561300665
   6.038118e-06
# 2 PB-16-003        4            4 5.415394e-07 0.018167170
   1.090021e-04
# 3 PB-16-004        3            3 3.600861e-06 0.001126488
   2.180974e-01
#              4           5           6
# 1 0.2842663527 0.006611313 0.005062298
# 2 0.3544297069 0.002884648 0.001734889
# 3 0.0002087706 0.007044669 0.012420599
```

Also, you can try to use `luckyModel`:

```
pci <- lucky_model(
  project = "GSClassifier",
  model = "PanCancerImmuneSubtype",
  developer = "Gibbs",
  version = "v20190731"
)
```

PanCan Immune Subtype callings:

```
res_pis <- callEnsemble(
    X = X,
    ens = NULL,
    geneAnnotation = NULL,
    geneSet = NULL,
    scaller = NULL,
    geneid = "symbol",
    matchmode = 'free',
    subtype = pci,
    verbose = F
  )
```

Finally, we take a look at the `PanCancer immune subtypes` model:

```
ImmuneSubtype <- readRDS(system.file("extdata", "ImmuneSubtype.
   rds", package = "GSClassifier"))
names(ImmuneSubtype)
# [1] "ens"            "scaller"        "geneAnnotation" "geneSet
   "
```

Its gene annotation:

```
head(ImmuneSubtype$geneAnnotation)
#      SYMBOL ENTREZID         ENSEMBL
# 235  ACTL6A       86 ENSG00000136518
# 294   ADAM9     8754 ENSG00000168615
# 305 ADAMTS1     9510 ENSG00000154734
# 322    ADAR      103 ENSG00000160710
# 340   ADCY7      113 ENSG00000121281
# 479   AIMP2     7965 ENSG00000106305
```

Its gene sets:

```
ImmuneSubtype$geneSet
# $LIexpression_score
#  [1] "CCL5"  "CD19"  "CD37"  "CD3D"  "CD3E"  "CD3G"  "CD3Z"  "
   CD79A" "CD79B"
# [10] "CD8A"  "CD8B1" "IGHG3" "IGJ"   "IGLC1" "CD14"  "LCK"   "
   LTB"   "MS4A1"
#
# $CSF1_response
#   [1] "CORO1A"   "MNDA"     "CCRL2"    "SLC7A7"   "HLA-DMA"  "
   FYB"
#   [7] "RNASE6"   "TLR2"     "CTSC"     "LILRB4"   "PSCDBP"   "
   CTSS"
#  [13] "RASSF4"   "MSN"      "CYBB"     "LAPTM5"   "DOCK2"    "
   FCGR1A"
#  [19] "EVI2B"    "ADCY7"    "CD48"     "ARHGAP15" "ARRB2"    "
   SYK"
#  [25] "BTK"      "TNFAIP3"  "FCGR2A"   "VSIG4"    "FPRL2"    "
   IL10RA"
#  [31] "IFI16"    "ITGB2"    "IL7R"     "TBXAS1"   "FMNL1"    "
```

```
   FLI1"
#  [37] "RASSF2"   "LYZ"      "CD163"    "CD97"     "CCL2"     "
   FCGR2B"
#  [43] "MERTK"    "CD84"     "CD53"     "CD86"     "HMHA1"    "
   CTSL"
#  [49] "EVI2A"    "TNFRSF1B" "CXCR4"    "LCP1"     "SAMHD1"   "
   CPVL"
#  [55] "HLA-DRB1" "C13orf18" "GIMAP4"   "SAMSN1"   "PLCG2"    "
   OSBPL3"
#  [61] "CD8A"     "RUNX3"    "FCGR3A"   "AMPD3"    "MYO1F"    "
   CECR1"
#  [67] "LYN"      "MPP1"     "LRMP"     "FGL2"     "NCKAP1L"  "
   HCLS1"
#  [73] "SELL"     "CASP1"    "SELPLG"   "CD33"     "GPNMB"    "
   NCF2"
#  [79] "FNBP1"    "IL18"     "B2M"      "SP140"    "FCER1G"   "
   LCP2"
#  [85] "LY86"     "LAIR1"    "IFI30"    "TNFSF13B" "LST1"     "
   FGR"
#  [91] "NPL"      "PLEK"     "CCL5"     "PTPRC"    "GNPTAB"   "
   SLC1A3"
#  [97] "HCK"      "NPC2"     "C3AR1"    "PIK3CG"   "DAPK1"    "
   ALOX5AP"
# [103] "CSF1R"    "CUGBP2"   "APOE"     "APOC1"    "CD52"     "
   LHFPL2"
# [109] "C1orf54"  "IKZF1"    "SH2B3"    "WIPF1"
#
# $Module3_IFN_score
#  [1] "IFI44"  "IFI44L" "DDX58"  "IFI6"   "IFI27"  "IFIT2"  "
   IFIT1"  "IFIT3"
```

```
#  [9] "CXCL10" "MX1"    "OAS1"   "OAS2"   "OAS3"   "HERC5"  "
   SAMD9"  "HERC6"
# [17] "DDX60"  "RTP4"   "IFIH1"  "STAT1"  "TAP1"   "OASL"   "
   RSAD2"  "ISG15"
#
# $TGFB_score_21050467
#  [1] "MMP3"     "MARCKSL1" "IGF2R"    "LAMB1"    "SPARC"    "
   FN1"
#  [7] "ITGA4"    "SMO"      "MMP19"    "ITGB8"    "ITGA5"    "
   NID1"
# [13] "TIMP1"    "SEMA3F"   "RHOQ"     "CTNNB1"   "MMP2"     "
   SERPINE1"
# [19] "EPHB2"    "COL16A1"  "EPHA2"    "TNC"      "JUP"      "
   ITGA3"
# [25] "TCF7L2"   "COL3A1"   "CDH6"     "WNT2B"    "ADAM9"    "
   DSP"
# [31] "HSPG2"    "ARHGAP1"  "ITGB5"    "IGFBP5"   "ARHGDIA"  "
   LRP1"
# [37] "IGFBP2"   "CTNNA1"   "LRRC17"   "MMP14"    "NEO1"     "
   EFNA5"
# [43] "ITGB3"    "EPHB3"    "CD44"     "IGFBP4"   "TNFRSF1A" "
   RAC1"
# [49] "PXN"      "PLAT"     "COL8A1"   "WNT8B"    "IGFBP3"   "
   RHOA"
# [55] "EPHB4"    "MMP1"     "PAK1"     "MTA1"     "THBS2"    "
   CSPG2"
# [61] "MMP17"    "CD59"     "DVL3"     "RHOB"     "COL6A3"   "
   NOTCH2"
# [67] "BSG"      "MMP11"    "COL1A2"   "ZYX"      "RND3"     "
   THBS1"
```

```
# [73] "RHOG"     "ICAM1"    "LAMA4"    "DVL1"     "PAK2"     "
   ITGB2"
# [79] "COL6A1"   "FGD1"
#
# $CHANG_CORE_SERUM_RESPONSE_UP
#   [1] "CEP78"     "LSM3"      "LRRC40"    "STK17A"    "RPN1"
       "JUNB"
#   [7] "NUP85"     "FLNC"      "HMGN2"     "RPP40"     "UQCR10"
     "AIMP2"
#  [13] "CHEK1"     "VTA1"      "EXOSC8"    "CENPO"     "PNO1"
       "SLC16A1"
#  [19] "WDR77"     "UBE2J1"    "NOP16"     "NUDT1"     "SMC2"
       "SLC25A5"
#  [25] "NUPL1"     "DLEU2"     "PDAP1"     "CCBL2"     "COX17"
      "BCCIP"
#  [31] "PLG"       "RGS8"      "SNRPC"     "PLK4"      "NUTF2"
      "LSM4"
#  [37] "SMS"       "EBNA1BP2"  "C13orf27"  "VDAC1"     "PSMD14"
     "MYCBP"
#  [43] "SMURF2"    "GNG11"     "F3"        "IL7R"      "BRIP1"
      "HNRNPA2B1"
#  [49] "DCK"       "ALKBH7"    "HN1L"      "MSN"       "TPM1"
       "HYLS1"
#  [55] "HAUS1"     "NUP93"     "SNRPE"     "ITGA6"     "CENPN"
      "C11orf24"
#  [61] "GGH"       "PFKP"      "FARSA"     "EIF2AK1"   "CENPW"
      "TUBA4A"
#  [67] "TRA2B"     "UMPS"      "MRTO4"     "NUDT15"    "PGM2"
       "DBNDD1"
#  [73] "SNRPB"     "MNAT1"     "NUP35"     "TCEB1"     "HSPB11"
```

```
      "C19orf48"
#   [79] "ID3"       "IPO4"      "FARSB"     "EIF4G1"    "SKA1"
        "MFSD11"
#   [85] "PLAUR"     "MARVELD2"  "MCM3"      "DHFR"      "RNF41"
       "ID2"
#   [91] "H2AFZ"     "CDK2"      "NCLN"      "ZWILCH"    "DYNLT1"
      "C16orf61"
#   [97] "SLC25A40"  "RHOC"      "CCT5"      "PDIA4"     "SNRPA"
       "RBM14"
# [103] "PDLIM7"    "PITPNC1"   "TPM3"      "CORO1C"    "ERLIN1"
      "PAICS"
# [109] "TPRKB"     "SKA2"      "MYBL1"     "SH3BP5L"   "BRCA2"
       "SAR1A"
# [115] "POLR3K"    "MRPS28"    "NUP107"    "TUBG1"     "PNN"
        "FAM167A"
# [121] "RFC3"      "MYL6"      "MCM7"      "MAGOHB"    "FAM89B"
      "TOMM40"
# [127] "CDCA4"     "MT3"       "MTHFD1"    "PSMD12"    "MYBL2"
       "CKLF"
# [133] "NRIP3"     "EZR"       "C12orf24"  "GPLD1"     "SRM"
        "RAB3B"
# [139] "NLN"       "MT1F"      "TNFRSF12A" "TPI1"      "HAS2"
       "APOO"
# [145] "FBXO41"    "MRPL37"    "GSTCD"     "SDC1"      "WDR54"
       "RNF138"
# [151] "APITD1"    "RMND5B"    "ENO1"      "MAP3K8"    "TMEM130"
      "SNX17"
# [157] "KRR1"      "TAGLN"     "PA2G4"     "RUVBL1"    "SNRPD1"
      "LOXL2"
# [163] "POLE2"     "MAPRE1"    "IMP4"      "EMP2"      "PSMD2"
```

```
       "MET"
# [169] "IFRD2"     "LMNB2"     "PLOD2"     "NCEH1"     "NME1"
       "STRA13"
# [175] "ACTL6A"    "DLEU1"     "SNRPA1"    "CBX1"      "LYAR"
       "PTPLB"
# [181] "PFN1"      "CENPJ"     "COTL1"     "SPRYD7"    "USPL1"
       "MRPL12"
# [187] "ADAMTS1"   "GLRX3"     "WSB2"      "MRPS16"    "DCLRE1B"
      "MKKS"
# [193] "C3orf26"   "CPEB4"     "SPAG17"    "MLF1IP"    "UAP1"
       "COQ2"
# [199] "WDHD1"     "DCBLD2"    "KIAA0090"  "SAR1B"     "PSMA7"
       "PSMC3"
# [205] "COPS6"     "DUT"       "PPIH"      "PHF19"     "TPM2"
       "MCTS1"
# [211] "EIF4EBP1"  "HNRNPR"
```

Enjoy `GSClassifier`!

Chapter 4

Model establishment via GSClassifier

4.1 About

Sometimes, researchers might have their gene signatures and know how many subtypes they want to call before (based on some knowledge). Gratifyingly, comprehensive functions were also provided in `GSClassifier`. In this section, we would show how to build a `GSClassifier` model like `PADi`.

4.2 Data preparation

Load packages:

```
library(GSClassifier)
library(plyr)
library(dplyr)
```

Load data:

```
# The test data is only for the demonstration of the modeling
testData <- readRDS(system.file("extdata", "testData.rds",
   package = "GSClassifier"))
```

```
expr <- testData$PanSTAD_expr_part
design <- testData$PanSTAD_phenotype_part
```

Select training and testing cohorts across different platforms and `PAD subtypes` from `PAD` function:

```
modelInfo <- modelData(
    design,
    id.col = "ID",
    variable = c("platform", "PAD_subtype"),
    Prop = 0.7,
    seed = 145
  )
```

Check the result `modelInfo`:

```
names(modelInfo)
# [1] "Repeat" "Data"
```

Explore the training cohort:

```
head(modelInfo$Data$Train)
#                    ID  Dataset PAD_subtype PIAM_subtype PIDG_
   subtype platform
# GSM1606509 GSM1606509 GSE65801       PAD-I         high
           high GPL14550
# GSM1606517 GSM1606517 GSE65801       PAD-I         high
           high GPL14550
# GSM1606503 GSM1606503 GSE65801       PAD-I         high
           high GPL14550
# GSM1606525 GSM1606525 GSE65801       PAD-I         high
           high GPL14550
```

```
# GSM1606511 GSM1606511 GSE65801       PAD-I         high
           high GPL14550
# GSM1606527 GSM1606527 GSE65801       PAD-I         high
           high GPL14550
```

Explore the internal validation cohort:

```
head(modelInfo$Data$Valid)
#                    ID  Dataset PAD_subtype PIAM_subtype PIDG_
   subtype platform
# GSM2235558 GSM2235558 GSE84437       PAD-I         high
           high  GPL6947
# GSM2235561 GSM2235561 GSE84437      PAD-II         high
            low  GPL6947
# GSM2235562 GSM2235562 GSE84437      PAD-IV          low
            low  GPL6947
# GSM2235563 GSM2235563 GSE84437      PAD-IV          low
            low  GPL6947
# GSM2235564 GSM2235564 GSE84437      PAD-IV          low
            low  GPL6947
# GSM2235567 GSM2235567 GSE84437      PAD-IV          low
            low  GPL6947
```

Get training data `Xs` and `Ys`:

```
# Training data
Xs <- expr[,modelInfo$Data$Train$ID]
y <- modelInfo$Data$Train
y <- y[colnames(Xs),]
Ys <- ifelse(y$PAD_subtype == 'PAD-I',1,ifelse(y$PAD_subtype == '
   PAD-II',2,ifelse(y$PAD_subtype == 'PAD-III',3,ifelse(y$PAD_
   subtype == 'PAD-IV',4,NA)))); table(Ys)/length(Ys)
```

```
# Ys
#         1         2         3         4
# 0.1010169 0.2474576 0.1694915 0.4820339
```

Get the number of subtype:

```
# No. of subtypes
nSubtype <- length(unique(Ys))
print(nSubtype)
# [1] 4
```

Also, you can take a look at the validation data:

```
# Validating data
Xs_valid <- expr[,modelInfo$Data$Valid$ID]
y <- modelInfo$Data$Valid
y <- y[colnames(Xs_valid),]
Ys_valid <- ifelse(y$PAD_subtype == 'PAD-I',1,ifelse(y$PAD_
   subtype == 'PAD-II',2,ifelse(y$PAD_subtype == 'PAD-III',3,
   ifelse(y$PAD_subtype == 'PAD-IV',4,NA))))
table(Ys_valid)/length(Ys_valid)
# Ys_valid
#          1          2          3          4
# 0.09609121 0.24592834 0.16612378 0.49185668
```

Note: When you convert your phenotype into numeric, **You CANNOT USE A ZERO VALUE**, which is not supported by the xGboost.

Other parameteres for modeling:

```
# Build 20 models
n=20
```

```
# In every model, 70% samples in the training cohort would be
   selected.
sampSize=0.7

# Seed for sampling
sampSeed = 2020
na.fill.seed = 2022

# A vector for approximate gene rank estimation
breakVec=c(0, 0.25, 0.5, 0.75, 1.0)

# Use 80% most variable gene & gene-pairs for modeling
ptail=0.8/2

# Automatical selection of parameters for xGboost
auto = F

if(!auto){

      # Self-defined params. Fast.
      params = list(max_depth = 10,
                    eta = 0.5,
                    nrounds = 100,
                    nthread = 10,
                    nfold=5)
      caret.seed = NULL

      # No. of CPU for parallel computing. The optimized value
         depends on your CPU and RAM
      numCores = 4
```

```
  } else {

    # caret::train strategy by GSClassifier:::cvFitOneModel2.
       Time consuming
    params = NULL
    caret.seed = 105

    # Self-defined. For this exmaple training grid, there are 2
       ×1×1×3×2×1×2=24 grids. Make sure that you have a
       computer with a powerfull CPU.
    grid = expand.grid(
      nrounds = c(100, 200),
      colsample_bytree = 1,
      min_child_weight = 1,
      eta = c(0.01, 0.1, 0.3),
      gamma = c(0.5, 0.3),
      subsample = 0.7,
      max_depth = c(5,8)
    )

    # If you don't know how to set, just use the same number of
        your subtypes
    numCores = 4
  }
```

Finaly, you have to provide your gene sets as a `list` object:

```
geneSet = <Your gene sets>
```

Let's take `PAD` as an example:

```
PAD <- readRDS(system.file("extdata", "PAD.train_20220916.rds",
   package = "GSClassifier"))
geneSet <- PAD$geneSet
print(geneSet)
# $PIAM
#  [1] "ENSG00000122122" "ENSG00000117091" "ENSG00000163219" "
   ENSG00000136167"
#  [5] "ENSG00000005844" "ENSG00000123338" "ENSG00000102879" "
   ENSG00000010671"
#  [9] "ENSG00000185862" "ENSG00000104814" "ENSG00000134516" "
   ENSG00000100055"
# [13] "ENSG00000082074" "ENSG00000113263" "ENSG00000153283" "
   ENSG00000198821"
# [17] "ENSG00000185811" "ENSG00000117090" "ENSG00000171608"
#
# $PIDG
#  [1] "ENSG00000116667" "ENSG00000107771" "ENSG00000196782" "
   ENSG00000271447"
#  [5] "ENSG00000173517" "ENSG00000134686" "ENSG00000100614" "
   ENSG00000134247"
#  [9] "ENSG00000109686" "ENSG00000197321" "ENSG00000179981" "
   ENSG00000187189"
# [13] "ENSG00000140836"
```

4.3 Fitting models

4.3.1 GSClassifier model training

Just fit the model like:

```
if(!auto){

  # Self-defined

  system.time(
    res <- fitEnsembleModel(Xs,
                            Ys,
                            geneSet = geneSet,
                            na.fill.method = c('quantile','rpart'
                               ,NULL)[1],
                            na.fill.seed = na.fill.seed,
                            n = n,
                            sampSize = sampSize,
                            sampSeed = sampSeed ,
                            breakVec = breakVec,
                            params = params,
                            ptail = ptail,
                            caret.grid = NULL,
                            caret.seed =  caret.seed,
                            verbose = verbose,
                            numCores = numCores)
  )

  # user   system  elapsed
  # 0.08s  0.18s   92.70s

} else {

  # caret::train-defined and time-consuming
  system.time(
```

```
    res <- fitEnsembleModel(Xs,
                            Ys,
                            geneSet = geneSet,
                            na.fill.method = c('quantile','rpart'
                               ,NULL)[1],
                            na.fill.seed = na.fill.seed,
                            n = n,
                            sampSize = sampSize,
                            sampSeed = sampSeed ,
                            breakVec = breakVec,
                            params = NULL,  # This must be NULL
                            ptail = ptail,
                            caret.grid = grid,
                            caret.seed =  caret.seed,
                            verbose = verbose,
                            numCores = numCores)
  )

  # user   system   elapsed
  # 1.10s  2.60s    2311.55s
}

# Remind me with a music when the process completed
mymusic()
```

You should save it for convenience:

```
saveRDS(res,'<your path>/train_ens.rds')
```

Although an `auto-parameter strategy` (hyperparameter tuning) was provided in`GSClassifier`, it's unknown whether this method could significantly improve your

model performance. You can just try. It's not a prior recommendation. Generally, setting `auto=F` in this script could be more cost-effective. Empirically, the speed of `caret::train` depends on the single-core performance of the CPU instead of the core number.

4.3.2 Scaller for the best call

Next, we model the `scaller` for the training cohort, which would be used for `BestCall` based on the `probability matrix` in `callEnsemble` series. Here, `scaller=NULL` would cause an `NA` value of `BestCall` col. It's not a big deal, because the `probability matrix` is the information we need.

```
# Time-consuming modeling
resTrain <- parCallEnsemble(X = Xs,
                            ens = res$Model,
                            geneAnnotation = res$geneAnnotation,
                            geneSet = geneSet,
                            scaller = NULL,
                            geneids = "ensembl",
                            subtype = NULL,
                            numCores = numCores)

# xgboost via best interation
library(xgboost)
dtrain <- xgb.DMatrix(as.matrix(resTrain[4:(3 + nSubtype)]),
   label = Ys-1)

cvRes <- xgb.cv(data = dtrain,
                nrounds=100,
                nthread=10,
                nfold=5,
                max_depth=5,
```

```
                eta=0.5,
                early_stopping_rounds=100,
                num_class = 4,
                objective = "multi:softmax")

# xgboost via best interation
bst <- xgboost(data = dtrain,
               max_depth=5,
               eta=0.5,
               nrounds = cvRes$best_iteration,
               nthread=10,
               num_class = 4,
               objective = "multi:softmax")
Ys_pred <- predict(bst, as.matrix(resTrain[4:7])) + 1
mean(Ys_pred == Ys) # Prediction rates

# Ensemble results
scaller.train <- list(
      Repeat = list(
        data = dtrain,
        max_depth=5,
        eta=0.5,
        nrounds = cvRes$best_iteration,
        nthread=10,
        num_class = 4,
        objective = "multi:softmax"
      ),
      Model = bst
    )
```

4.3.3 Assemble your model

For more information of `geneAnnotation`, you could see the Suggestions for GSClassifier model developers: Gene Annotation section for assistance.

Here we give an example:

```
l.train <- list()

# bootstrap models based on the training cohort
l.train[['ens']] <- res

# Scaller model
l.train[['scaller']] <- scaller.train

# a data frame contarining gene annotation for IDs convertion
l.train[['geneAnnotation']] <- <Your gene annotation>

# Your gene sets
l.train[['geneSet']] <- geneSet
```

Finally, save it for downstream analysis

```
saveRDS(l.train,'<Your path>/train.rds')
```

About model constributions, you can go `Advanced development` in here or here for more information.

4.3.4 Of note

You can take a look at the `PAD.train_20220916` model (`PADi`). You have to make your model frame similar to the `PAD.train_20220916` model.

```
l.train <- readRDS(system.file("extdata", "PAD.train_20220916.rds
   ", package = "GSClassifier"))
```

```
names(l.train)
# [1] "ens"            "scaller"        "geneAnnotation" "geneSet
   "
```

The time of `GSClassifier` modeling depends on the number of individual models (controlled by `n`)/called subtypes/gene signatures, automatic parameter selection, and your CPU capacity.

4.4 Calling subtypes

Supposed that you had got a `GSClassifier` model, next you want to use it for personalized subtype calling.

Just:

```
# Load your model
l <- readRDS('<Your path>/train.rds')

# subtype calling
res_i = callEnsemble(
  X,
  ens = l$ens$Model,
  geneAnnotation = l$geneAnnotation,
  geneSet = l$geneSet,
  scaller = l$scaller$Model,
  geneid = <ID type of your training data>,
  matchmode = 'fix',
  subtype = NULL,
  verbose = T
)
```

The usage of `parCallEnsemble` (for a huge amount of samples) is similar. Empirically,

`parCallEnsemble` can not perform better than `callEnsemble` in a small cohort for the process of xgboost would take advantage of multiple CPU cores.

4.5 Number of SubModel

In the latest version of **PADi**, we trained 100 **SubModel**s for subtype calling. We also trained some models with different training cohorts (200 models, Figure 4.1) or with different numbers (20, 50, 100, 200, 500 and 1000 for the same training cohort, Figure 4.2) of sub-models. Our results showed that **the performance of these models is similar in the "Kim2018" cohort**.

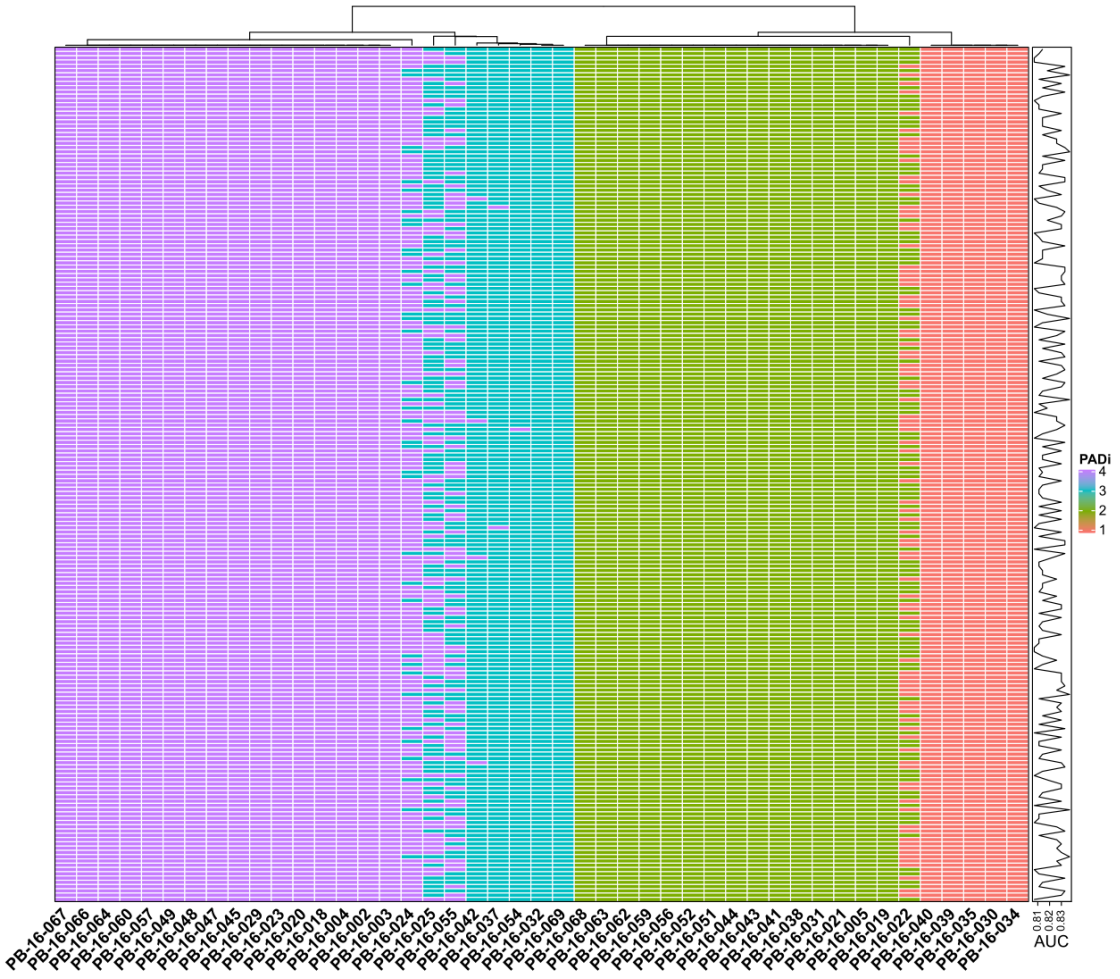


Figure 4.1: Performance of PADi models in "Kim2018" cohort with 200 different training seeds. Each row is the data of a seed. Each column is a sample from the "Kim2018" cohort.

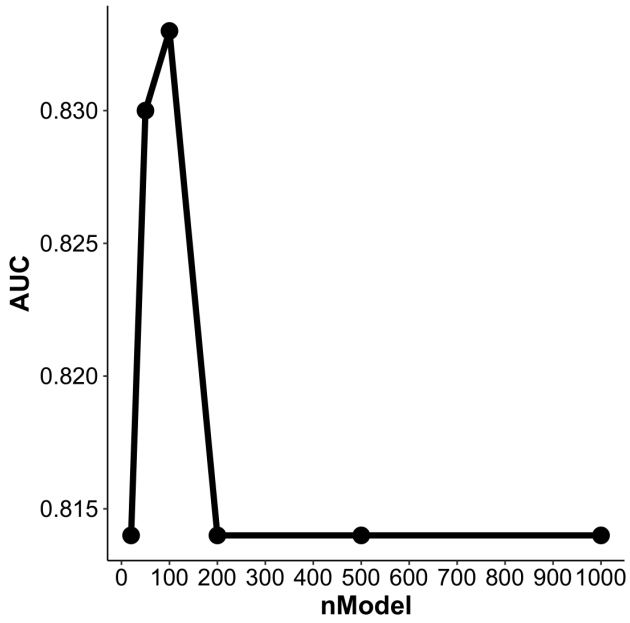


Figure 4.2: Performance of PADi models in "Kim2018" cohort with different numbers of SubModel. The x-axis is the number of SubModels, and the y-axis is the AUC of ROC analysis for ICI response prediction.

4.6 Parameters for PADi training

Here're some key parameters about how the latest PADi (PAD.train_v20220916) was trained.

```
# Build 100 models
n = 100

# In every model, 70% samples in the training cohort would be
   selected.
sampSize = 0.7

# Seed for sampling
sampSeed = 2020
na.fill.seed = 443

# A vector for gene rank estimation
```

```
breakVec = c(0, 0.25, 0.5, 0.75, 1.0)

# Use 80% most variable gene & gene-pairs for modeling
ptail = 0.8/2

# Automatical selection of parameters for xGboost
# self-defined params. Fast.
params = list(max_depth = 10,
              eta = 0.5,
              nrounds = 100,
              nthread = 10,
              nfold=5)
caret.seed = NULL
```

Enjoy **GSClassifier**!

Chapter 5

Suggestions for GSClassifier model developers

5.1 About

- The book R packages is a straightaway and useful reference book for R developers. The free-access website for R packages is https://r-pkgs.org/. As a developer of R, if you haven't heard about it, it's strongly recommended to just read it. Hadley Wickham, the main author of the book, is an active R developer and has led some masterworks like `ggplot2` and `plyr`.

- With `GSClassifier` package, it could be easy for users to build a model only with certain gene sets and transcriptomics data. If you are interested in sharing your model, `GSClassifier` also provides a simple methodology for this vision. In this section, let's see how to achieve it!

First, load the package

```
library(GSClassifier)
# Loading required package: luckyBase
```

5.2 Available models

With `GSClassifier_Data()`, all models supported in the current `GSClassifier` package would be shown.

```
GSClassifier_Data()
# Available data:
# Usage example:
#   ImmuneSubtype.rds
#   PAD.train_20200110.rds
#   PAD.train_20220916.rds
#   PAD <- readRDS(system.file("extdata", "PAD.train_20200110.rds
   ", package = "GSClassifier"))
#   ImmuneSubtype <- readRDS(system.file("extdata", "
   ImmuneSubtype.rds", package = "GSClassifier"))
```

For more details of `GSClassifier_Data()`, just:

```
?GSClassifier_Data()
```

Set `model=F`, all `.rds` data would be showed:

```
GSClassifier_Data(model = F)
# Available data:
# Usage example:
#   general-gene-annotation.rds
#   ImmuneSubtype.rds
#   PAD.train_20200110.rds
#   PAD.train_20220916.rds
#   testData.rds
#   PAD <- readRDS(system.file("extdata", "PAD.train_20200110.rds
   ", package = "GSClassifier"))
```

```
#   ImmuneSubtype <- readRDS(system.file("extdata", "
   ImmuneSubtype.rds", package = "GSClassifier"))
```

5.3 Components of a GSClassifier model

Currently, a GSClassifier model and related product environments are designed as a `list` object. Let's take `PAD.train_20210110`(also called `PADi`) as an example.

```
PADi <- readRDS(system.file("extdata", "PAD.train_20220916.rds",
   package = "GSClassifier"))
```

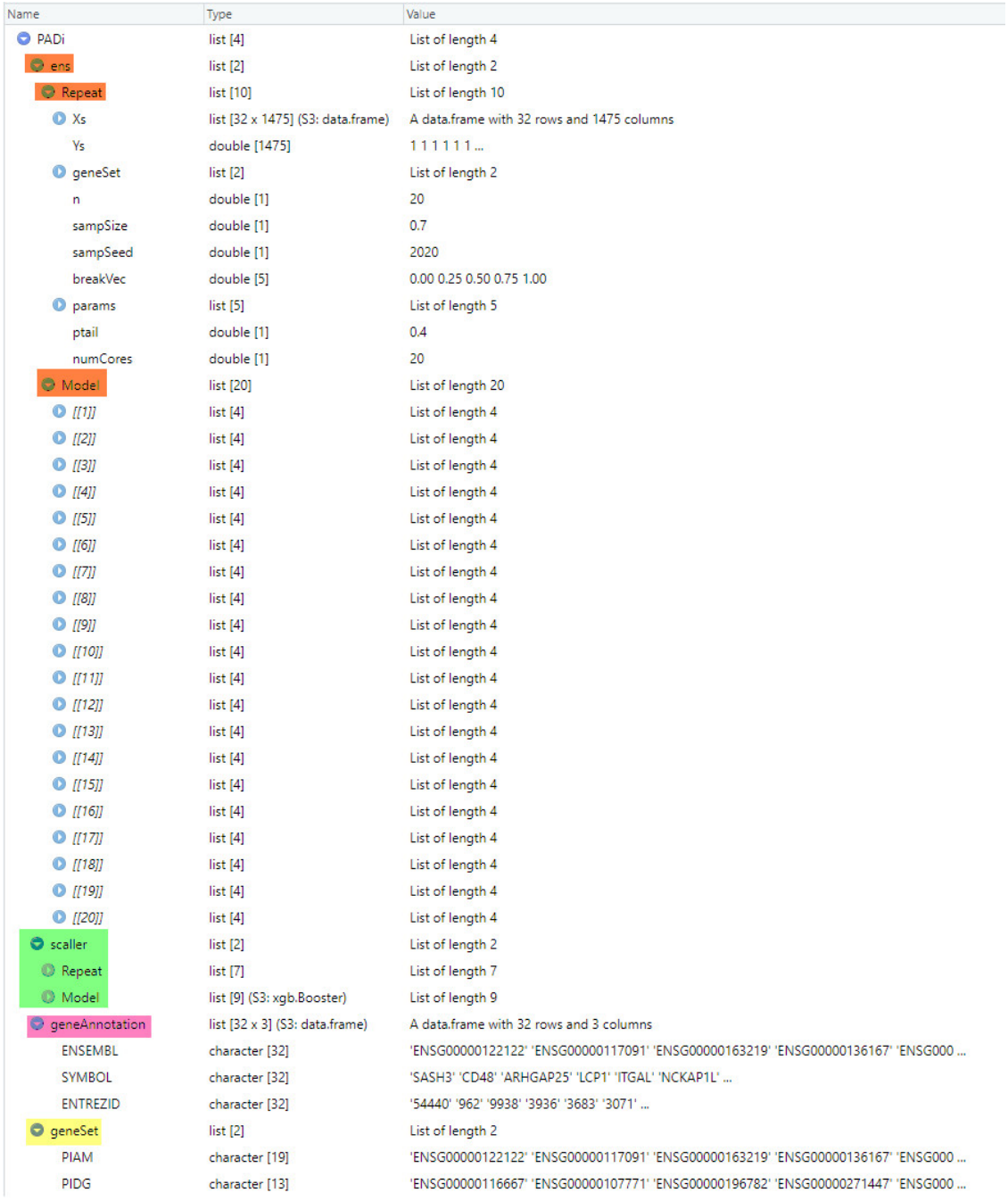
This picture shows the components of `PADi`:

As shown, a typical `GSClassifier` model is consist of four parts (with different colors in the picture):

- `1. ens`:
  - `Repeat`: productive parameters of `GSClassifier` models
  - `Model`: `GSClassifier` models. Here, `PADi` had 20 models from different subs of the training cohorts
- `2. scaller`:
  - `Repeat`: productive parameters of the `scaller` model, which was used for `BestCall` calling
  - `Model`: the `scaller` model
- `3. geneAnnotation`: a data frame containing gene annotation information
- `4. geneSet`: a list contains several gene sets

Thus, you can assemble your model like:

```
model <- list()

# bootstrap models based on the training cohort
model[['ens']] <- <Your model for subtypes calling>
```



Name	Type	Value
PADi	list [4]	List of length 4
ens	list [2]	List of length 2
Repeat	list [10]	List of length 10
Xs	list [32 x 1475] (S3: data.frame)	A data.frame with 32 rows and 1475 columns
Ys	double [1475]	1 1 1 1 1 1 ...
geneSet	list [2]	List of length 2
n	double [1]	20
sampSize	double [1]	0.7
sampSeed	double [1]	2020
breakVec	double [5]	0.00 0.25 0.50 0.75 1.00
params	list [5]	List of length 5
ptail	double [1]	0.4
numCores	double [1]	20
Model	list [20]	List of length 20
[[1]]	list [4]	List of length 4
[[2]]	list [4]	List of length 4
[[3]]	list [4]	List of length 4
[[4]]	list [4]	List of length 4
[[5]]	list [4]	List of length 4
[[6]]	list [4]	List of length 4
[[7]]	list [4]	List of length 4
[[8]]	list [4]	List of length 4
[[9]]	list [4]	List of length 4
[[10]]	list [4]	List of length 4
[[11]]	list [4]	List of length 4
[[12]]	list [4]	List of length 4
[[13]]	list [4]	List of length 4
[[14]]	list [4]	List of length 4
[[15]]	list [4]	List of length 4
[[16]]	list [4]	List of length 4
[[17]]	list [4]	List of length 4
[[18]]	list [4]	List of length 4
[[19]]	list [4]	List of length 4
[[20]]	list [4]	List of length 4
scaller	list [2]	List of length 2
Repeat	list [7]	List of length 7
Model	list [9] (S3: xgb.Booster)	List of length 9
geneAnnotation	list [32 x 3] (S3: data.frame)	A data.frame with 32 rows and 3 columns
ENSEMBL	character [32]	'ENSG00000122122' 'ENSG00000117091' 'ENSG00000163219' 'ENSG00000136167' 'ENSG000 ...
SYMBOL	character [32]	'SASH3' 'CD48' 'ARHGAP25' 'LCP1' 'ITGAL' 'NCKAP1L' ...
ENTREZID	character [32]	'54440' '962' '9938' '3936' '3683' '3071' ...
geneSet	list [2]	List of length 2
PIAM	character [19]	'ENSG00000122122' 'ENSG00000117091' 'ENSG00000163219' 'ENSG00000136167' 'ENSG000 ...
PIDG	character [13]	'ENSG00000116667' 'ENSG00000107771' 'ENSG00000196782' 'ENSG00000271447' 'ENSG000 ...

Figure 5.1: Details of a GSClassifier model

```
# Scaller model
model[['scaller']] <- <Your scaller for BestCall calling>

# a data frame contarining gene annotation for IDs convertion
model[['geneAnnotation']] <- <Your gene annotation>

# Your gene sets
model[['geneSet']] <- <Your gene sets>

saveRDS(model, 'your-model.rds')
```

More tutorials for model establishment, please go to markdown tutorial or html tutorial.

5.4 Submit models to luckyModel package

Considering most users of `GSClassifier` might not need lots of models, We divided the model storage feature into a new ensemble package called **luckyModel**. Don't worry, the usage is very easy!

If you want to submit your model, you should apply for a contributor of `luckyModel` first. Then, just send the model (`.rds`) into the `inst/extdata/<project>` path of `luckyModel`. After an audit, your branch would be accepted and available for the users.

The name of your model must be the format as follows:

```
# <project>
GSClassifier

# <creator>_<model>_v<yyyymmdd>:
HWB_PAD_v20211201.rds
```

5.5 Repeatablility of models

For repeatability, you had better submit a `.zip` or `.tar.gz` file containing the information of your model. Here are some suggestions:

- `<creator>_<model>_v<yyyymmdd>.md`
  - **Destinations**: Why you develop the model
  - **Design**: The evidence for gene sigatures, et al
  - **Data sources**: The data for model training and validating, et al
  - **Applications**: Where to use your model
  - **Limintations**: Limitation or improvement direction of your model
- `<creator>_<model>_v<yyyymmdd>.R`: The code you used for model training and validating.
- `Data-of-<creator>_<model>_v<yyyymmdd>.rds` (Optional): Due to huge size of omics data, it's OK for you not to submit the raw data.

Welcome your contributions!

5.6 Gene Annotation

For convenience, we provided a general gene annotation dataset for different genomics:

```
gga <- readRDS(system.file("extdata", "general-gene-annotation.
   rds", package = "GSClassifier"))
names(gga)
# [1] "hg38" "hg19" "mm10"
```

I believe they're enough for routine medicine studies.

Here, take a look at `hg38`:

```
hg38 <- gga$hg38
```

```
head(hg38)
#           ENSEMBL        SYMBOL   ENTREZID
# 1 ENSG00000223972       DDX11L1  100287102
# 3 ENSG00000227232        WASH7P       <NA>
# 4 ENSG00000278267     MIR6859-1  102466751
# 5 ENSG00000243485 RP11-34P13.3       <NA>
# 6 ENSG00000284332     MIR1302-2  100302278
# 7 ENSG00000237613       FAM138A     645520
```

With this kind of data, it's simple to customize your own gene annotation (take `PADi` as examples):

```
tGene <- as.character(unlist(PADi$geneSet))
geneAnnotation <- hg38[hg38$ENSEMBL %in% tGene, ]
dim(geneAnnotation)
# [1] 32  3
```

Have a check:

```
head(geneAnnotation)
#              ENSEMBL SYMBOL ENTREZID
# 353  ENSG00000171608 PIK3CD     5293
# 1169 ENSG00000134686   PHC2     1912
# 2892 ENSG00000134247 PTGFRN     5738
# 3855 ENSG00000117090 SLAMF1     6504
# 3858 ENSG00000117091   CD48      962
# 4043 ENSG00000198821  CD247      919
```

This `geneAnnotation` could be the `model[['geneAnnotation']]`.

Also, we use a function called `convert` to do gene ID convertion.

```
luckyBase::convert(c('GAPDH','TP53'), 'SYMBOL', 'ENSEMBL', hg38)
# [1] "ENSG00000111640" "ENSG00000141510"
```

Note: the `luckyBase` package integrates lots of useful tiny functions, you could explore it sometimes.

References

1. Thorsson V, Gibbs DL, Brown SD, et al. The immune landscape of cancer. Immunity 2018; 48:812–830 e14

2. Gibbs DL. Robust classification of immune subtypes in cancer. 2020;

3. Chen T, He T, Benesty M, et al. Xgboost: Extreme gradient boosting. 2015; 1:1–4

4. Geman D, d'Avignon C, Naiman DQ, et al. Classifying gene expression profiles from pairwise mRNA comparisons. Stat Appl Genet Mol Biol 2004; 3:Article19

5. Zhao H, Logothetis CJ, Gorlov IP. Usefulness of the top-scoring pairs of genes for prediction of prostate cancer progression. Prostate Cancer Prostatic Dis 2010; 13:252–9

6. Youssef YM, White NM, Grigull J, et al. Accurate molecular classification of kidney cancer subtypes using microRNA signature. Eur Urol 2011; 59:721–30

7. Auslander N, Zhang G, Lee JS, et al. Robust prediction of response to immune checkpoint blockade therapy in metastatic melanoma. Nat Med 2018; 24:1545–1549

8. Troyanskaya O, Cantor M, Sherlock G, et al. Missing value estimation methods for DNA microarrays. Bioinformatics 2001; 17:520–5

9. Arbeitman MN, Furlong EE, Imam F, et al. Gene expression during the life cycle of drosophila melanogaster. Science 2002; 297:2270–5

10. Zhu X, Wang J, Sun B, et al. An efficient ensemble method for missing value imputation in microarray gene expression data. BMC Bioinformatics 2021; 22:188

11. Lin W-C, Tsai C-F. Missing value imputation: A review and analysis of the literature (2006–2017). 2020; 53:1487–1509

12. Hasan MK, Alam MA, Roy S, et al. Missing value imputation affects the performance of machine learning: A review and analysis of the literature (2010–2021). Informatics in

Medicine Unlocked 2021; 27:100799

13. Wang A, Yang J, An N. Regularized sparse modelling for microarray missing value estimation. 2021; 9:16899–16913

14. Chen T, He T, Benesty M, et al. Xgboost: Extreme gradient boosting. 2022;

15. Robin X, Turck N, Hainard A, et al. pROC: An open-source package for r and s+ to analyze and compare ROC curves. BMC Bioinformatics 2011; 12:77

16. Zhu S, Kong W, Zhu J, et al. The genetic algorithm-aided three-stage ensemble learning method identified a robust survival risk score in patients with glioma. Brief Bioinform 2022;

17. Tong M, Zheng W, Li H, et al. Multi-omics landscapes of colorectal cancer subtypes discriminated by an individualized prognostic signature for 5-fluorouracil-based chemotherapy. Oncogenesis 2016; 5:e242

18. Qi L, Li Y, Qin Y, et al. An individualised signature for predicting response with concordant survival benefit for lung adenocarcinoma patients receiving platinum-based chemotherapy. Br J Cancer 2016; 115:1513–1519

19. Huang H, Zou Y, Zhang H, et al. A qualitative transcriptional prognostic signature for patients with stage i-II pancreatic ductal adenocarcinoma. Transl Res 2020; 219:30–44

20. Zheng H, Song K, Fu Y, et al. An absolute human stemness index associated with oncogenic dedifferentiation. Brief Bioinform 2021; 22:2151–2160

21. Kong W, He L, Zhu J, et al. An immunity and pyroptosis gene-pair signature predicts overall survival in acute myeloid leukemia. Leukemia 2022;

22. Liu K, Geng Y, Wang L, et al. Systematic exploration of the underlying mechanism of gemcitabine resistance in pancreatic adenocarcinoma. Mol Oncol 2022; 16:3034–3051

23. Zheng H, Xie J, Song K, et al. StemSC: A cross-dataset human stemness index for single-cell samples. Stem Cell Res Ther 2022; 13:115

24. Monti S, Tamayo P, Mesirov J, et al. Consensus clustering: A resampling-based method for class discovery and visualization of gene expression microarray data. 2003; 52:91–118

25. Wilkerson MD, Hayes DN. ConsensusClusterPlus: A class discovery tool with confidence assessments and item tracking. 2010; 26:1572–1573

26. Hayes DN, Monti S, Parmigiani G, et al. Gene expression profiling reveals reproducible human lung adenocarcinoma subtypes in multiple independent patient cohorts. 2006;

24:5079–5090

27. Verhaak RG, Hoadley KA, Purdom E, et al. Integrated genomic analysis identifies clinically relevant subtypes of glioblastoma characterized by abnormalities in PDGFRA, IDH1, EGFR, and NF1. 2010; 17:98–110