PINSPlus: Clustering Algorithm for Data Integration and Disease Subtyping

Hung Nguyen, Sangam Shrestha, and Tin Nguyen*
Department of Computer Science and Engineering
University of Nevada, Reno, NV 89557

2018-05-02

Abstract

PINS+ provides a robust approach for data integration and disease subtyping. It allows for unsupervised clustering using multi-omics data. The method automatically determines the optimal number of clusters and then partitions the samples in a way such that the results are robust to noise and data perturbation. PINS+ has been validated on thousands of cancer samples obtained from the Gene Expression Omnibus, the Broad Institute, The Cancer Genome Atlas (TCGA), and the European Genome-Phenome Archive. The approach can accurately identify known subtypes and discover novel groups of patients with significantly different survival profiles. The software is extremely fast and able to cluster hundreds of matched samples in minutes.

Contents

Introduction

In a recent paper published in Genome Research, Nguyen et al. [1] proposed a robust approach for multi-omics data integration and disease subtyping called PINS. The framework was tested upon many datasets obtained from the Gene Expression Omnibus, the Broad Institute, The Cancer Genome Atlas (TCGA), and the European Genome-Phenome Archive. In the analysis, PINS outperforms state-of-the-art clustering methods like Similarity Network Fusion (SNF) [2], Consensus Clustering (CC) [3], and iClusterPlus [4] in identifying known subtypes and in discovering novel groups of patients with significantly different survival profiles. Please consult Nguyen et al. [1], [5] for the mathematical description.

PINS+ offers many improvements of PINS from practical perspectives. One outstanding feature is that the package is extremely fast. For example, it takes PINS+ only five minutes using a single core to analyze the Glioblastoma dataset (273 patients with three data types, mRNA, miRNA, and methylation) while it takes PINS 175 minutes (almost three hours) to analyze the same dataset (see Supplemental Table S14 in Nguyen et al. [1] for running time of PINS). PINS+ also allows for parallelization on any of the three platforms: Windows, Linux, and Mac OS. In addition, PINS+ provides users with more flexibility, including customized basic clustering algorithms, distance metrics, noise levels, subsampling, data perturbation, etc.

This document provides a tutorial on how to use the PINS+ package. PINS+ is designed to be convenient for users and uses two main functions: `PerturbationClustering` and `SubtypingOmicsData`. `PerturbationClustering` allows users to cluster a single data type while `SubtypingOmicsData` allows users to cluster multiple types of data.

PerturbationClustering

The `PerturbationClustering` function automatically determines the optimal number of clusters and the membership of each item (patient or sample) from a single data type in an **unsupervised analysis**.

Preparing data

The input of the function PerturbationClustering is a numerical matrix or data frame in which the rows represent items while the columns represent features.

Load example data `AML2004`

```
library(PINSPlus)
data(AML2004)
```

Run `PerturbationClustering`

Run `PerturbationClustering` with default parameters

```
system.time(result <- PerturbationClustering(data = AML2004$Gene, verbose = FALSE))
```

```
##  [1]     NA 0.9617 0.9767 0.9993 0.9109 0.9972 0.9870 0.9801 0.9643 0.9573
```

```
##    user  system elapsed
##   4.392   0.528   3.111
```

`PerturbationClustering` supports parallel computing using the **ncore** parameter (default `ncore = 2`):

```
result <- PerturbationClustering(data = AML2004$Gene, ncore = 8)
```

Print out the number of clusters:

```
result$k
```

```
## [1] 4
```

Print out the cluster membership:

```
result$cluster
```

```
##  ALL_Bcell_1  ALL_Bcell_2  ALL_Bcell_3  ALL_Bcell_4  ALL_Bcell_5
##            1            1            2            1            1
##  ALL_Bcell_6  ALL_Bcell_7  ALL_Bcell_8  ALL_Bcell_9 ALL_Bcell_10
##            1            2            2            2            2
## ALL_Bcell_11 ALL_Bcell_12 ALL_Bcell_13 ALL_Bcell_14 ALL_Bcell_15
##            1            1            2            1            1
## ALL_Bcell_16 ALL_Bcell_17 ALL_Bcell_18 ALL_Bcell_19  ALL_Tcell_1
##            2            1            1            1            3
##  ALL_Tcell_2  ALL_Tcell_3  ALL_Tcell_4  ALL_Tcell_5  ALL_Tcell_6
##            3            3            3            3            3
##  ALL_Tcell_7  ALL_Tcell_8        AML_1        AML_2        AML_3
##            3            3            4            1            4
##        AML_4        AML_5        AML_6        AML_7        AML_8
##            4            4            4            4            4
##        AML_9       AML_10       AML_11
##            4            4            4
```

Compare the result with the known sutypes [6]:

```
condition <- seq(unique(AML2004$Group[, 2]))
names(condition) = unique(AML2004$Group[, 2])
plot(prcomp(AML2004$Gene)$x, col = result$cluster,
     pch = condition[AML2004$Group[, 2]], main = "AML2004")
legend("bottomright", legend = paste("Cluster ", sort(unique(result$cluster)), sep = ""),
       fill = sort(unique(result$cluster)))
legend("bottomleft", legend = names(condition), pch = condition)
```

AML2004
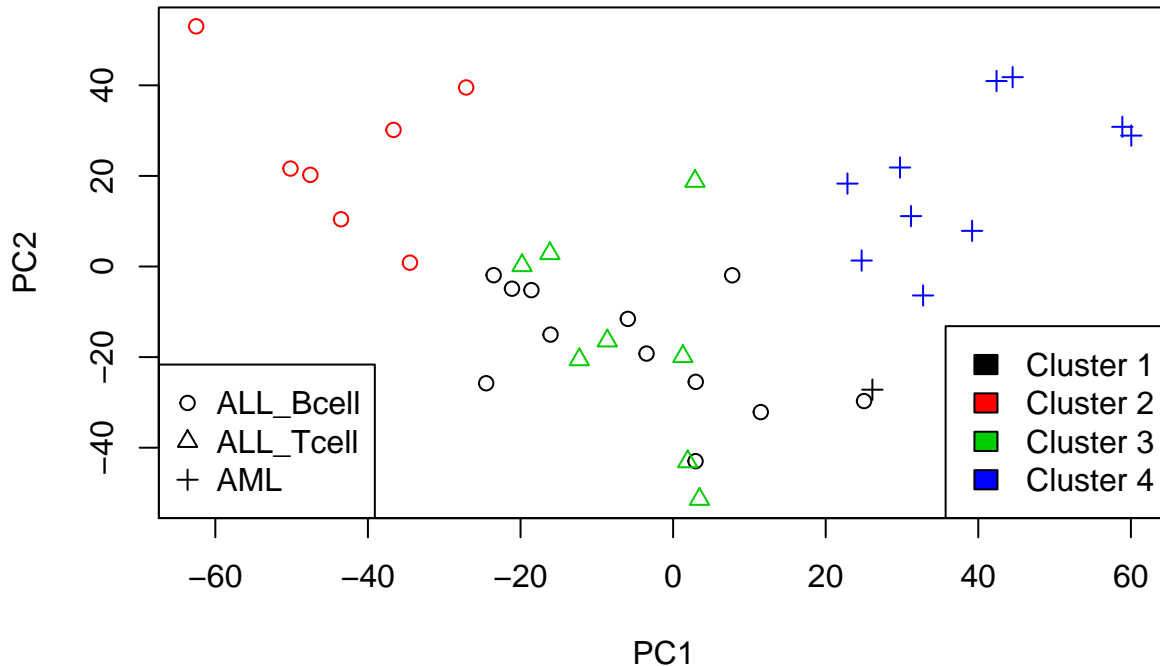
By default, `PerturbationClustering` runs with `kMax = 10` and `kmeans` as the basic algorithm. `PerturbationClustering` performs `kmeans` clustering to partition the input data with $k \in [2, 10]$ and then computes the optimal value of k.

```
result <- PerturbationClustering(data = AML2004$Gene, kMax = 10,
                                 clusteringMethod = "kmeans")
```

To switch to other basic algorithms, use the `clusteringMethod` argument:

```
result <- PerturbationClustering(data = AML2004$Gene, kMax = 10,
                                 clusteringMethod = "pam")
```

or

```
result <- PerturbationClustering(data = AML2004$Gene, kMax = 10,
                                 clusteringMethod = "hclust")
```

By default, `kmeans` clustering runs with parameters `nstart = 20` and `iter.max = 1000`. Users can pass new values to `clusteringOptions` to change these values:

```
result <- PerturbationClustering(
    data = AML2004$Gene,
    clusteringMethod = "kmeans",
    clusteringOptions = list(nstart = 100, iter.max = 500),
    verbose = FALSE
)
```

Instead of using the built-in clustering algorithms such as `kmeans`, `pam`, and `hclust`, users can also pass their own clustering algorithm via the `clusteringFunction` argument.

```
result <- PerturbationClustering(data = AML2004$Gene,
    clusteringFunction = function(data, k){
    # this function must return a vector of cluster
```

```
    kmeans(x = data, centers = k, nstart = k*10, iter.max = 2000)$cluster
})
```

In the above example, we use our version of `kmeans` instead of the built-in `kmeans` where the value of `nstart` parameter is dependent on the number of clusters `k`. Note that the implementation of `clusteringFunction` must accept two arguments: (1) `data` - the input matrix, and (2) `k` - the number of clusters. It must return a vector indicating the cluster to which each item is allocated.

By default, `PerturbationClustering` adds noise to perturbate the data before clustering. The `noise` perturbation method by default accepts two arguments: `noise = NULL` and `noisePercent = "median"`. To change these parameters, users can pass new values to `perturbOptions`:

```
result <- PerturbationClustering(data = AML2004$Gene,
                                 perturbMethod = "noise",
                                 perturbOptions = list(noise = 1.23))
```

or

```
result <- PerturbationClustering(data = AML2004$Gene,
                                 perturbMethod = "noise",
                                 perturbOptions = list(noisePercent = 10))
```

If the `noise` parameter is specified, the `noisePercent` parameter will be skipped.

`PerturbationClustering` provides another built-in perturbation method called `subsampling` with a `percent` parameter:

```
result <- PerturbationClustering(data = AML2004$Gene,
                                 perturbMethod = "subsampling",
                                 perturbOptions = list(percent = 80))
```

If users wish to use their own perturbation method, they can pass it to the `perturbFunction` parameter:

```
result <- PerturbationClustering(data = AML2004$Gene, perturbFunction = function(data){
    rowNum <- nrow(data)
    colNum <- ncol(data)
    epsilon <-
        matrix(
            data = rnorm(rowNum * colNum, mean = 0, sd = 1.23456),
            nrow = rowNum, ncol = colNum
        )

    list(
        data = data + epsilon,
        ConnectivityMatrixHandler = function(connectivityMatrix, iter, k) {
            connectivityMatrix
        },
        MergeConnectivityMatrices = function(oldMatrix, newMatrix, iter, k){
            return((oldMatrix*(iter-1) + newMatrix)/iter)
        }
    )
})
```

The one argument `perturbFunction` takes is `data` - the original input matrix. The `perturbFunction` must return a list object which contains the following entities:

- `data`: a matrix after perturbating from input `data` and is ready for clustering.
- `ConnectivityMatrixHandler`: a function that takes three arguments: i) `connectivityMatrix` - the

connectivity matrix generated after clustering, ii) `iter` - the current iteration, and iii) `k` - the number of clusters. This function must return a compatible connectivity matrix with the original connectivity matrix. It aims to correct the `connectivityMatrix` if needed and returns its corrected version.

- `MergeConnectivityMatrices`: a function that takes four arguments: i) `oldMatrix`, ii) `newMatrix`, iii) `k`, and iv) `iter`. `oldMatrix` and `newMatrix` are two connectivity matrices that need to be merged. `k` is the cluster number. `iter` is the current number of iterations. This function must return a connectivity matrix that is merged from `oldMatrix` and `newMatrix`.

`PerturbationClustering` provides several arguments to control stopping criterias:

- `iterMax`: the maximum number of iterations.
- `iterMin`: the minimum number of iterations that allows `PerturbationClustering` to calculate the stability of the perturbed connectivity matrix based on its AUC (Area Under the Curve) with the original one. If the perturbed connectivity matrix for current processing `k` is stable (based on `madMin` and `msdMin`), the iteration for this `k` will be stopped.
- `madMin`: the minimum of Mean Absolute Deviation of AUC of Connectivity matrices.
- `msdMin`: the minimum of Mean Square Deviation of AUC of Connectivity matrices.

```
result <- PerturbationClustering(data = AML2004$Gene, iterMax = 200,
                                 iterMin = 10, madMin = 1e-2, msdMin = 1e-4)
```

SubtypingOmicsData

SubtypingOmicsData automatically finds the optimum number of subtypes and its membership from multi-omics data through two processing stages:

- Stage I: The algorithm first partitions each data type using the function PerturbationClustering and then merges the connectivities across data types into similarity matrices. Similarity-based clustering algorithms such as partitioning around medoids (`pam`) and hierarchical clustering (`hclust`) are used to partition the built similarity. The algorithm returns the partitioning that agrees the most with individual data types.
- Stage II: The algorithm attempts to split each discovered group if there is a strong agreement between data types, or if the subtyping in Stage I is very unbalanced.

Preparing data

```
# Load the kidney cancer carcinoma data
data(KIRC)
# SubtypingOmicsData`'s input data must be a list of
# numeric matrices or data frames that have the same number of rows:
dataList <- list (KIRC$GE, KIRC$ME, KIRC$MI)
names(dataList) <- c("GE", "ME", "MI")
# Run `SubtypingOmicsData`:
result <- SubtypingOmicsData(dataList = dataList)
```

By default, `SubtypingOmicsData` runs with parameters `agreementCutoff = 0.5` and `kMax = 10`. `SubtypingOmicsData` uses the `PerturbationClustering` function to cluster each data type. The parameters for `PerturbationClustering` are described above in the previous part of this document. If users wish to change the parameters for `PerturbationClustering`, they can pass it directly to the function:

```
result <- SubtypingOmicsData(
    dataList = dataList,
    clusteringMethod = "kmeans",
    clusteringOptions = list(nstart = 50)
)
```

Plot the Kaplan-Meier curves and calculate Cox p-value:

```r
library(survival)
cluster1=result$cluster1;cluster2=result$cluster2
a <- intersect(unique(cluster2), unique(cluster1))
names(a) <- intersect(unique(cluster2), unique(cluster1))
a[setdiff(unique(cluster2), unique(cluster1))] <-
    seq(setdiff(unique(cluster2), unique(cluster1))) + max(cluster1)
colors <- a[levels(factor(cluster2))]
coxFit <- coxph(
    Surv(time = Survival, event = Death) ~ as.factor(cluster2),
    data = KIRC$survival,
    ties = "exact"
)
mfit <- survfit(Surv(Survival, Death == 1) ~ as.factor(cluster2), data = KIRC$survival)
plot(
    mfit, col = colors, main = "Survival curves for KIRC, level 2",
    xlab = "Days", ylab = "Survival",lwd = 2
)
legend("bottomright",
    legend = paste(
        "Cox p-value:", round(summary(coxFit)$sctest[3], digits = 5), sep = ""
    )
)
legend(
    "bottomleft",
    fill = colors,
    legend = paste("Group ", levels(factor(cluster2)), ": ",
        table(cluster2)[levels(factor(cluster2))], sep =""
    )
)
```
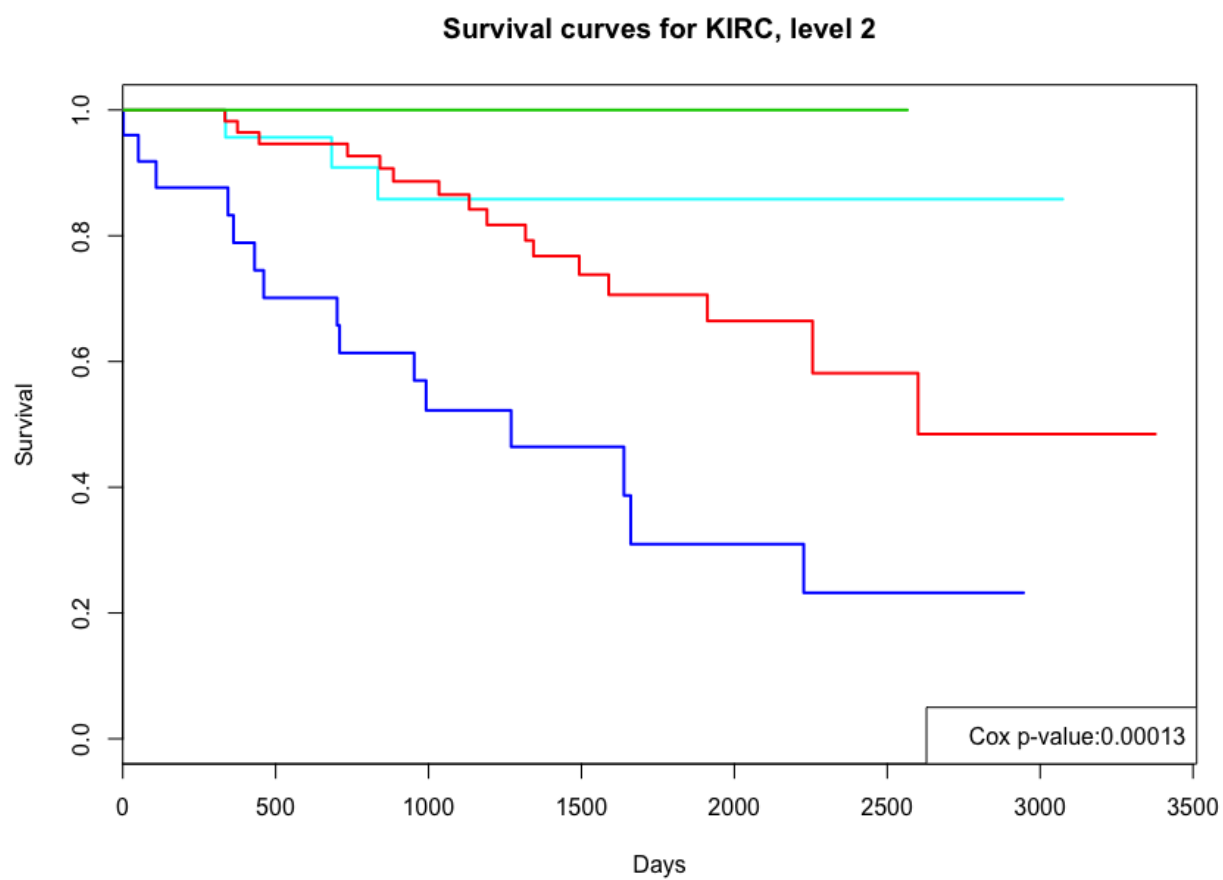
Figure 1: KIRC result

References

[1] T. Nguyen, R. Tagett, D. Diaz, and S. Draghici, "A novel approach for data integration and disease subtyping," *Genome Research*, vol. 27, no. 12, pp. 2025–2039, 2017.

[2] B. Wang, A. M. Mezlini, F. Demir, M. Fiume, Z. Tu, M. Brudno, B. Haibe-Kains, and A. Goldenberg, "Similarity network fusion for aggregating data types on a genomic scale," *Nature Methods*, vol. 11, no. 3, pp. 333–337, 2014.

[3] S. Monti, P. Tamayo, J. Mesirov, and T. Golub, "Consensus clustering: A resampling-based method for class discovery and visualization of gene expression microarray data," *Machine Learning*, vol. 52, nos. 1-2, pp. 91–118, 2003.

[4] Q. Mo, S. Wang, V. E. Seshan, A. B. Olshen, N. Schultz, C. Sander, R. S. Powers, M. Ladanyi, and R. Shen, "Pattern discovery and cancer gene identification in integrated cancer genomic data," *Proceedings of the National Academy of Sciences*, vol. 110, no. 11, pp. 4245–4250, 2013.

[5] T. Nguyen, "Horizontal and vertical integration of bio-molecular data," PhD thesis, Wayne State University, 2017.

[6] J.-P. Brunet, P. Tamayo, T. R. Golub, and J. P. Mesirov, "Metagenes and molecular pattern discovery using matrix factorization," *Proceedings of the National Academy of Sciences*, vol. 101, no. 12, pp. 4164–4169, Mar. 2004.