A Gentle Introduction to Package ClusterR

143095 Francesco Zuccato, 142135 Alessio Corrado

Contents

Introduction **1**

Unsupervised Learning **2**

Cluster Analysis **2**

Connectivity based . . . 2

Centroid based . . . 2

K-means and K-medoids . . . 3

Distribution based . . . 3

GMM: Gaussian Mixture Model . . . 3

Density based . . . 4

Data Generation **4**

The Package **5**

Testing GMM . . . 5

Choose the optimal number of centroids . . . 9

Elbow Method . . . 9

Testing K-Means . . . 9

Testing K-Medoids . . . 12

Silhouette Dissimilarity Plot . . . 14

Performance Comparisons (ALL vs ALL) **15**

Introduction

ClusterR is an R package for clustering data. It contains some built-in functions for K-Means and Gaussian Mixture Models.

Documentation can be found at https://cran.r-project.org/web/packages/ClusterR/.

Unsupervised Learning

Statistical learning problems can be divided in two main categories: supervised and unsupervised. They both aim to extract information from a set of observations x_i, with $1 \leq i \leq n$. Each x_i is a vector of p predictor variables.

The supervised framework assumes to have in input also the associated response y_i foreach corresponding x_i. A classical example of supervised learning is the logistic regression.

The unsupervised learning instead relies only on the observations x_i, without any associated response y_i. So, since we do not have y_i we cannot apply any type of checks on the error between real and predicted values for the response variable.

So, we can state that given $X_1, ..., X_p$ variables the goal is to discover any type of correlation or pattern between the variables. In the unsupervised framework, the most known class of methods is the Cluster Analysis.

Cluster Analysis

Clustering is the task of dividing the population or data points into a number of groups such that data points in the same groups are more similar to other data points in the same group and dissimilar to the data points in other groups. It is basically a collection of objects on the basis of similarity and dissimilarity between them.

Clustering is suitable for explaining the possible data-generating process, for finding possible relations or hierarchies between data or as a part of the exploratory data analysis.

For doing clustering is necessary to define a measure of similarity: similar items should belong to the same group. A measure of similarity must be non-negative, symmetric and must have a value of zero only if the items are equal. There are various possible similarity measures, for example: euclidean distance, manhattan distance and maximum distance. A clustering library should give the possibility to specify the similarity function to be used.

A predictive algorithm based on clustering may for example use the value of nearest cluster to give the response for a new data point. It is also possible to combine the informations of all clusters, weighted by the cluster distances. There are four main approaches in clustering:

- Connectivity based
- Centroid based
- Distribution based
- Density based

Connectivity based

Connectivity-based clustering tries to create connections between the clusters. In this way the algorithm creates a tree (or a forest), which can gives an hierarchial view of the data. Two possible uses can be creating a cause-effect relationship between items or reconstructing a possible evolution of a population.

Algorithms in this class: Agglomerative (Divisive) Hierarchical Clustering.

Centroid based

Centroid-based clustering is based on centroids: each item belongs to the class of its nearest centroid. The centroids can be restricted to locations of the items (medoid) or generic points (centroid). The number of

centroids can be hard-coded or searched by the algorithm; the position of the centroids is optimized by the algorithm.

Algorithms in this class: K-Means, K-Medoids.

K-means and K-medoids

K-means initially creates k random points which represents the centroids. Then, for each iteration, the algorithm moves the centroid positions to optimize the clustering. The k-medoid algorithms is a variation of k-means, which changes the way to calculate the center of each cluster:

- **k-means**: uses the mean of all its points (which is not necessarily a point of the cluster)
  - pros: center of the cluster coincides with the mean point over all its coordinates
- **k-medoids** uses the median point of the cluster (so a point of the cluster)
  - pros: center of the cluster coincides is not influenced by noise or outliers.

Distribution based

Distribution-based clustering uses a probabilistic model to explain the original data-generating model, by minimizing the discrepancy. By doing so, it is possible to create specific models that accurately describe the data and give an idea of the prediction confidence.

Algorithms in this class: Gaussian Mixture Models.

GMM: Gaussian Mixture Model

The most known and effective distribution-based algorithm is GMM. GMM models the data-generating process as a set of gaussian models, one for each cluster. Doing so, each gaussian (cluster) gives the probability density function of the points it generates. A point belongs to the cluster that maximized the probability of generating it there.

The GMM can be seen as an extension of the K-means, which assumes that the clusters are non-overlapping and have a circular shape. GMM instead is effective also if the first assumption does not hold and the second is relaxed, requesting an ellipsoidal shape.

GMM uses generalized gaussian distributions, so each cluster $k \in \{1, .., K\}$ has its own independent mean and variance. The goal is to estimate the parameter $\theta_k = (\mu_k, \Sigma_k, \pi_k)$, where:

- **mean** μ_k: defines the center
- **covariance matrix** Σ_k: defines the width
- **"size"** π_k: defines how big the function will be (π_k is a probability itself, so: $\Sigma_{i=1}^{K} \pi_i = 1$)

To discover the above parameters, is used the Expectation-Maximization (EM) Algorithm. Basically, is based on two steps which alternates each other:

- **Expectation**: computes the posterior probabilities
- **Maximization**: updates the parameters given the newly computed posterior

Density based

Density-based clustering groups data using distance between points. Dense connected regions of points should be part of the same cluster. In this way there is not a fixed number of clusters, but the number of clusters depends on how many different regions of data points are there and on the chosen density threshold.

Algorithms in this class: DBSCAN, OPTICS.

Data Generation

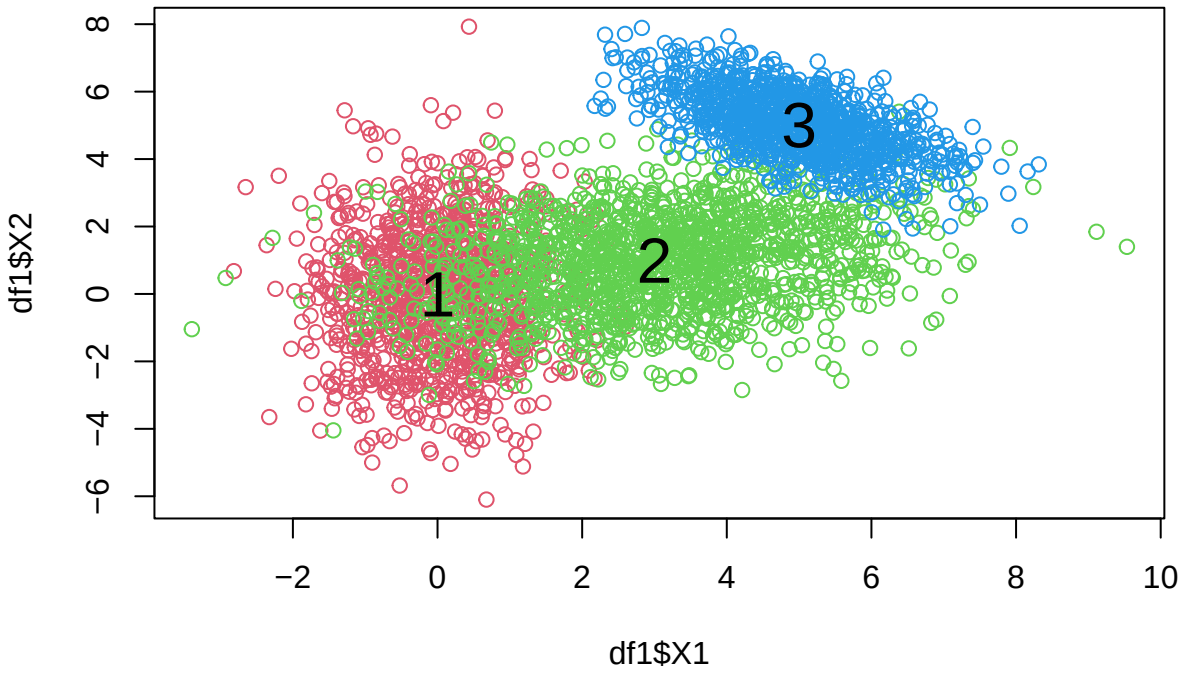
To test the various cluster algorithms first of all we need some data. We have created a synthetic dataset drawn from $k = 3$ different 2-dimensional gaussians (so $d = 2$).

To do it we used the library MASS, which defines the function `mvrnorm`: specifying the number n of samples, the mean $\mu \in \mathbb{R}^d$ and the correlation matrix $\Sigma \in \mathbb{R}^{d*d}$, it returns a matrix representing n observations $x_1, ..., x_n$, with each $x_i \in \mathbb{R}^d$ generated from a simulation of a Multivariate Normal Distribution.

We generated and combined three matrixes (one for each gaussian generating process) into a single dataframe, where the column "color" specifies the true class for each point.

Finally, we visualized the data using a scatterplot, where the digits indicate the positions of the centroids.

```
{
plot(x=df1$X1, y=df1$X2, col=df1$color + 1)
plot_points(points_mean, num_points=3)
}
```



The Package

ClusterR is a package designed for cluster generation and analysis. It supports Distribution-based and Centroid-based algorithms. It also contains some helper functions and example datasets.

The package is subdivided in different parts:

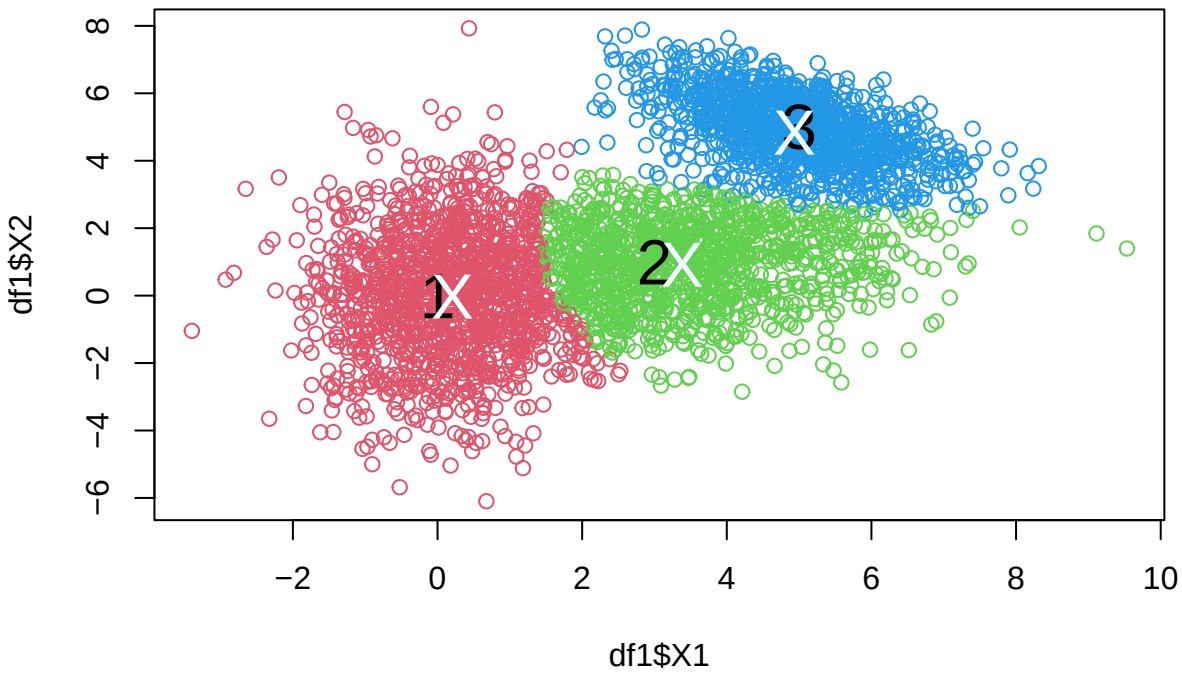
- clustering algorithms
- search of the optimal number of clusters
- prediction using the generated clusters
- visualization of the clustering goodness with 2D scatterplots and silouhette dissimilarity plots
- validation of the clustering goodness using ground truth labels
- helper methods

Testing GMM

In the library, to compute the Gaussian Mixture Models there is the function `GMM`, which has the following arguments:

- (mandatory) **data**: the matrix with the observations (one row per item, one column per component)
- (mandatory) **n_gaus**: the number of gaussian processes
- **dist_mode**: specifies if the training algorithm should use euclidean or manhattan distance
- **seed_mode**: specifies the initial placement of the centroids for the iterative algorithm (static/random subset/spread)
- **km_iter**: number of iterations for the k-means algorithm
- **em_iter**: number of iterations for the expectation-maximization algorithm
- **var_floor**: smallest possible values for diagonal covariances
- **seed**: integer for the random number generator (to allow replicability)

```
gmm_3 = GMM(df1_no_label, 3, dist_mode = "maha_dist", seed_mode = "random_subset")
gmm_3 = reorder_data(gmm_3, "centroids", list("weights", "covariance_matrices"))
df1["y_3gmm"] = predict(gmm_3, newdata = df1_no_label)
```



It is also possible to use the function `plot_2d`, defined by by the ClusterR package, for a 2D visualization of the predicted points along with the centroid/medoid positions. The arguments are:

- (mandatory) **data**: 2-dimensional matrix specifying the item positions
- (mandatory) **clusters**: a list specifying the class for each item (numeric vector)
- (mandatory) **centroids_medoids**: the position of the centroids (or medoids)

```
plot_2d(data = df1_no_label, clusters = df1$y_3gmm, centroids_medoids = gmm_3$centroids)
```

The function `external_validation`, also defined in the ClusterR package, gives us the possibility to extract some useful metrics measuring the goodness of the fit. It requires only the predicted values and annotated values.

The arguments are:

- (mandatory) **true_labels**, **clusters**: respectively the annotated values and the predicted values
- **method**: specify which summary statistic should the function return
- **summary_stats**: whether or not to print all the summary statistics

```
external_validation(df1$color, df1$y_3gmm, summary_stats = T)
```

```
##
## ---------------------------------------
## purity                          : 0.8706
## entropy                         : 0.3303
```

```
## normalized mutual information  : 0.6737
## variation of information       : 1.0234
## normalized var. of information : 0.492
## ----------------------------------------
## specificity                    : 0.8884
## sensitivity                    : 0.757
## precision                      : 0.7826
## recall                         : 0.757
## F-measure                      : 0.7695
## ----------------------------------------
## accuracy OR rand-index         : 0.8428
## adjusted-rand-index            : 0.6503
## jaccard-index                  : 0.6254
## fowlkes-mallows-index          : 0.7696
## mirkin-metric                  : 1925298
## ----------------------------------------
```

```
## [1] 0.650293
```

The GMM object returned by the function is a list with 5 components:

- **centroids**: a matrix $\in \mathbb{R}^{k*d}$ that specifies the position of each centroid
- **covariance_matrices**: a matrix $\in \mathbb{R}^{k*d}$ that specifies the diagonal values of each covariance matrix
- **weights** a vector $\in \mathbb{R}^{k}$ for each gaussian component (list)
- **Log_likelihood**: a matrix $\in \mathbb{R}^{n*k}$ foreach training item
- **call**: a list containing the values of the parameters used in the invokation

```
gmm_3$centroids
```

```
##           [,1]        [,2]
## [1,] 0.2047424 -0.00156993
## [2,] 3.3832735  0.94360627
## [3,] 4.9344644  4.86183228
```

```
gmm_3$covariance_matrices
```

```
##          [,1]     [,2]
## [1,] 1.004037 3.555754
## [2,] 2.178807 1.536963
## [3,] 1.106012 1.184378
```

```
gmm_3$weights
```

```
## [1] 0.3620318 0.3236028 0.3143654
```

```
head(gmm_3$Log_likelihood)
```

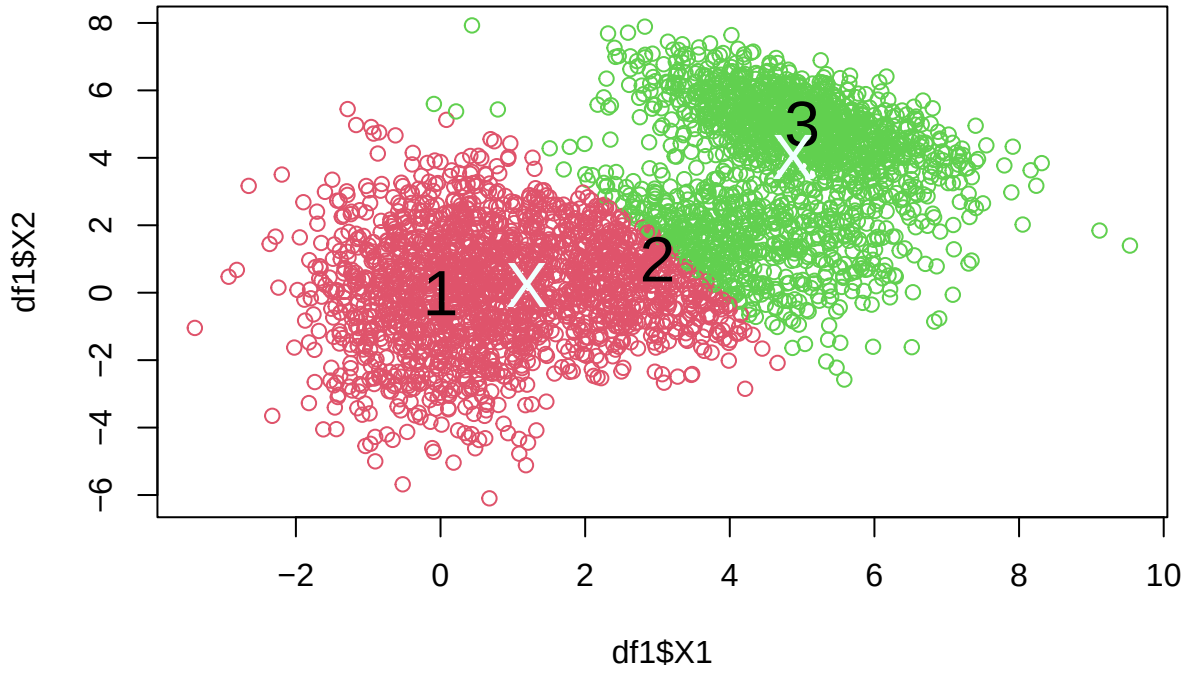
```
##           [,1]      [,2]      [,3]
## [1,] -24.37676 -5.274902 -2.713859
## [2,] -25.88778 -6.127775 -2.627882
```

```
## [3,] -35.46065 -8.799086 -3.296493
## [4,] -38.52503 -9.935926 -3.735107
## [5,] -29.10837 -6.725882 -2.742982
## [6,] -20.49818 -4.309937 -2.790042
```

Since it is possible to specify the number of clusters, we tried running the GMM algorithm specifying $gaussian\_comps = 2$.

We can see that the clustering still makes sense, even if the external validation statistics are worse. One of the centroids remains on the cluster nr.1, while the other centroid is in the middle of clusters 2 and 3.

```
gmm_2 = GMM(df1_no_label, 2, dist_mode = "maha_dist", seed_mode = "random_subset")
gmm_2 = reorder_data(gmm_2, "centroids", list("weights", "covariance_matrices"), k=2)
df1["y_2gmm"] = predict(gmm_2, newdata = df1_no_label)
```




```
external_validation(df1$color, df1$y_2gmm, summary_stats = T)
```

```
##
## ---------------------------------------
## purity                          : 0.5697
## entropy                         : 0.2775
## normalized mutual information   : 0.4373
## variation of information        : 1.4377
## normalized var. of information : 0.7201
## ---------------------------------------
```

```
## specificity                    : 0.6238
## sensitivity                    : 0.7356
## precision                      : 0.5093
## recall                         : 0.7356
## F-measure                      : 0.6019
## ---------------------------------------
## accuracy OR rand-index         : 0.6625
## adjusted-rand-index            : 0.3254
## jaccard-index                  : 0.4305
## fowlkes-mallows-index          : 0.6121
## mirkin-metric                  : 4132800
## ---------------------------------------

## [1] 0.3254161
```

Choose the optimal number of centroids

A non-simple question is how to find the optimal number of centroids? Plotting the data as above with colors only works for small datasets representable in two dimensions. The function `Optimal_Clusters_GMM` calculates the goodness of the fit for each number of clusters between 1 and *max_clusters*.

A criterion to evaluate a model can be either **AIC (Alkaine Information Criterion)** or **BIC (Bayes Information Criterion)**. A rule of thumb is that we should select the smallest (simplest) model with good performances, so the best model should be the one which minimizes one of these criterions.

Elbow Method

The elbow method is a well-known heuristic method used to determine the number of clusters. Given the plot with on the y-axis a metric (AIC, BIC, etc) and on the x-axis the number of clusters, the method consists of picking the elbow of the curve as the number of clusters to use. The idea is that the *elbow* can be seen as a **cutoff point** which means that choosing a number of clusters greater than it would not give much better modeling of the data. The Elbow Method is pretty simple but can used effectively only when the curve has a clear point where the improvement does not highly increase any more, which can not always be the case.

In the example, we can see that a model with 3 clusters should be fine, because after this value the BIC stops increasing sharply. Models with a lot of clusters have a even lower value for BIC but may be too complex, and probably would just overfit the data. This shows why it is important also to watch the clustering results using a scatterplot, if possible.

```
opt_gmm = Optimal_Clusters_GMM(df1_no_label, max_clusters = 10, criterion = "BIC",
    dist_mode = "eucl_dist", seed_mode = "random_subset", plot_data = T)
```

Testing K-Means

ClusterR supports k-means algorithm with two different implementations: `KMeans_arma` and `KMeans_rcpp`. Their interface is similar to GMM, but specific for the k-means task. The arguments shared by both are:

- (mandatory) **data**: matrix with the observations (one row per item, one column per component)
- (mandatory) **clusters**: number of clusters
- **CENTROIDS**: matrix with the initial cluster centroids
- **verbose**: whether or not to print some logs during the process

`KMeans_rcpp` gives the user more choice on setting the parameters, in fact it allows to:

- initialize various parameters such as **initializer**, **fuzzy**, **tol_optimal_init**, **seed**
- set the running time and convergence:
  - **num_init**: number of different initializations, if > 1 then is returned the best fit, according to `within-cluster-sum-of-squared-error (WCSS)`
  - **max_iters**, **tol**: stopping criterias based on the number of iterations or the accuracy

The `KMeans_rcpp` object returned by the function is a list with 8 components:

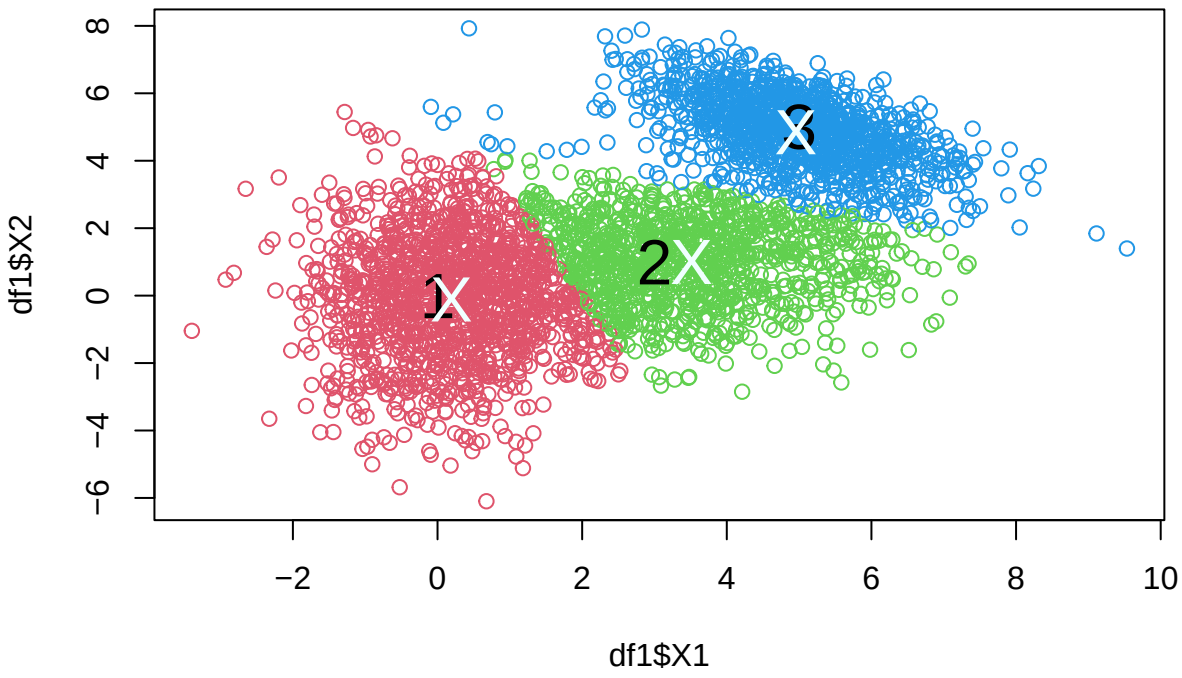
- **call**, **centroids**: as for the `GMM`
- **clusters**: a vector of n elements with the predicted cluster foreach item
- **best_initialization**: an integer indicating which was the best initialization (useful when $num_init > 1$)
- some metrics: **total_SSE** (squared distance of each point to its centroid), **WCSS_per_cluster**, **obs_per_cluster** (counts items predicted per cluster)

Although `KMeans_arma` is less flexible and has a more *ready-to-go* approach, it is faster than KMeans_rcpp since:

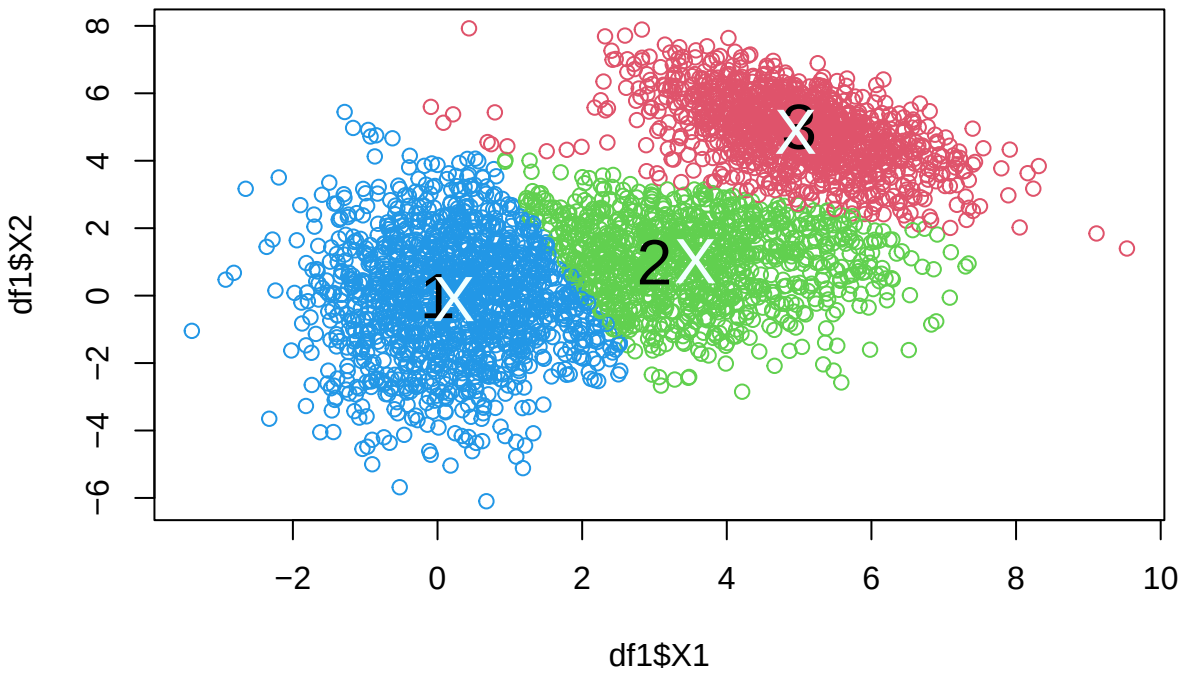
- it is slighter, it returns just the matrix of the centroids
- can work in parallel thanks to its implementation with OpenMP (if enabled).

In the following the result of KMeans_rcpp with $k = 3$.

```
kmeans_rcpp = KMeans_rcpp(df1_no_label, 3)
kmeans_rcpp = reorder_data(kmeans_rcpp, "centroids", list("WCSS_per_cluster", "obs_per_cluster"))
kmeans_rcpp$clusters <- NULL #drop it now, it's wrong, use the predict below
df1["y_3Mn_rcpp"] = predict(kmeans_rcpp, newdata = df1_no_label)
```




```
kmeans_arma = list()
kmeans_arma$centroids = matrix(KMeans_arma(df1_no_label, 3), nrow=3)
df1["y_3Mn_arma"] = predict_KMeans(df1_no_label, kmeans_arma$centroids)
```



Testing K-Medoids

The most common implementation of the k-medoid is the **Partitioning Around Medoids (PAM)** algorithm, which is based on two stages: **BUILD** and **SWAP**.

1. **BUILD**: an initial solution is built by:

a. Choosing as the first medoid the object with the lowest mean dissimilarity w.r.t. the whole dataset (the most central object)
b. Selecting iteratively medoids that further minimize the overall dissimilarity of each object from its nearest medoid.

2. **SWAP**: given the set of k medoids all the *neighbour solutions* are evaluated. A neighbour solution is constructed by swapping one of the current medoids with one of the non-selected objects. Thus the size of the neighborhood of a given solution is $(n-k)*k = O(n)$ assuming k to be a constant.

In the ClusterR package, as for the K-Means, there are two different implementations of the k-medoids:

- `Cluster_Medoids`: corresponds to the **PAM**
- `Clara_Medoids` (**C**lustering **LAR**ge **A**pplications): applies the PAM to a small sample of the data

The parameters of the `Cluster_Medoids` are:

- (mandatory) **data**, **clusters**: as for the k-means

- **distance_metric**: the distance method to be used: euclidean, manhattan, chebyshev, hamming, etc.
- **threads**: number of cores (for parallelism)
- **swap_phase**: whether or not to apply also the SWAP phase

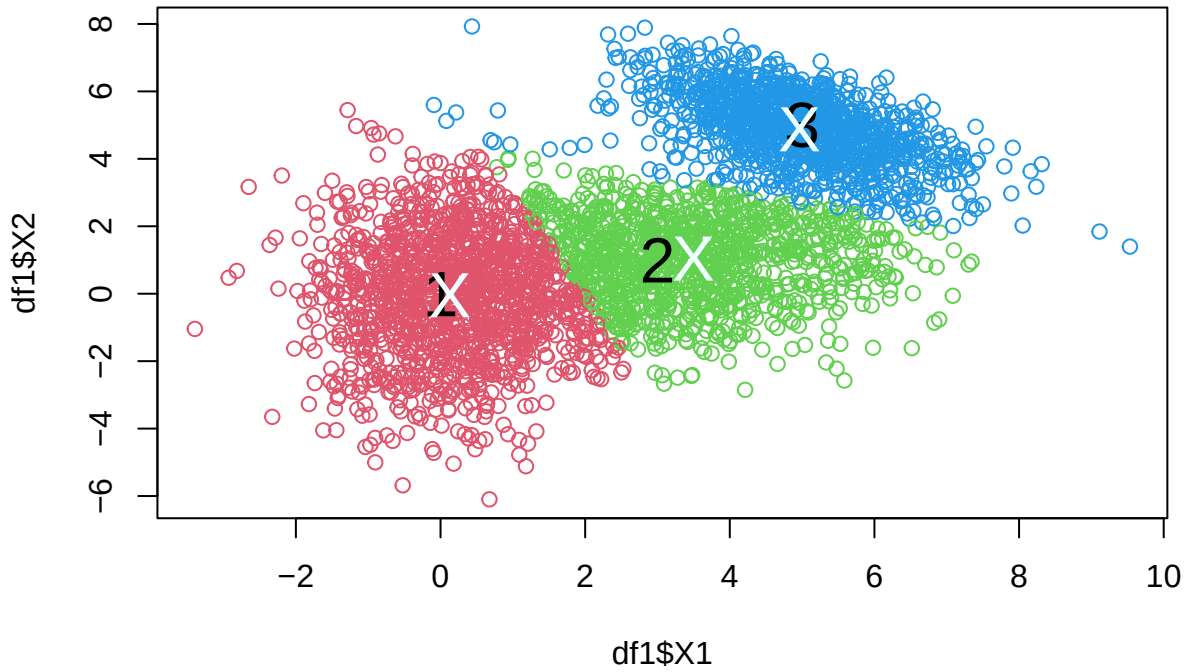
`Clara_Medoids` has two additional parameters:

- (mandatory) **samples**: number of samples to draw from the data set
- (mandatory) **sample_size** $\in (0, 1]$: percentage of the data to draw in each sample iteration

The `Cluster_Medoids` object returned by the function is a list with 10 components:

- **call**, **medoids**, **clusters**: as for the `Kmeans` (with medoids instead of centroids)
- **silhouette_matrix**: dataframe $\in \mathbb{R}^{n*7}$
- **dissimilarity_matrix**: matrix $\in \mathbb{R}^{n*n}$ with the distance foreach couple of items
- **best_dissimilarity**: an overall statistics
- **clustering_stats** dataframe $\in \mathbb{R}^{k*6}$ with per-cluster statistics: **clusters**, **average_dissimilarity**, **max_dissimilarity**, **diameter**, **separation**, **number_obs**

```
kmedoids_pam = Cluster_Medoids(df1_no_label, 3, distance_metric="euclidean")
```

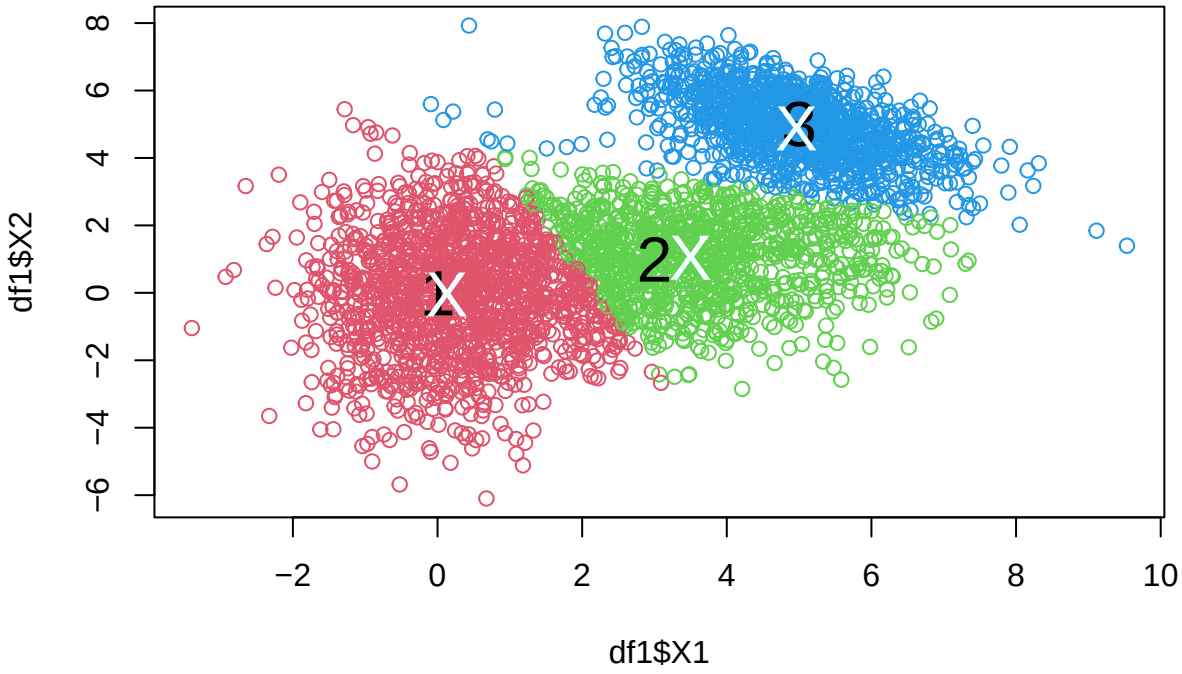


The function `Clara_Medoids` returns a list with 10 components, where the additional one is the **sample_indices**: the elements used in the sampling process. Also, the are some differences in:

- **dissimilarity_matrix** is now reduced to $\mathbb{R}^{s*s}$, since it contains only the sampled elements ($s \leq samples * sample\_size$, equals only if there have been no repetitions in any sampling).

- **clustering_stats**: instead of having **diameter** and **separation**, the **isolation** is given.

```
kmedoids_cla = Clara_Medoids(df1_no_label, 3, samples=4, sample_size=0.1, "euclidean")
```



Silhouette Dissimilarity Plot

The Silhouette Dissimilarity provides an easy graphical representation of how well each object has been associated to its own cluster (**cohesion**) compared to other clusters (**separation**). The silhouette range is $[-1, +1]$, where high values indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters. If most objects have a high value, then the clustering configuration is appropriate, otherwise the model is not reflecting correctly the data, maybe there are too many or too few clusters.

In the package there is the function `Silhouette_Dissimilarity_Plot`, which builds the plot given the output of one of the medoids' algorithms. Its input parameters are:

- (mandatory) **object**: the output of either `Cluster_Medoids` or `Clara_Medoids`
- **silhouette**: if TRUE plots the average silhouette width, otherwise the average dissimilarity

```
invisible(Silhouette_Dissimilarity_Plot(kmedoids_pam, silhouette = TRUE))
```

```
invisible(Silhouette_Dissimilarity_Plot(kmedoids_pam, silhouette = FALSE))
```

Performance Comparisons (ALL vs ALL)

Often is useful to compare models to seek for the best, to understand which are the possible limitations of another etc. Given the overall statistics of the different algorithms launched with the different parameters, it's pretty simple to build a dataframe the contains all of them, by just executing the `external_validation` function foreach metric desired and foreach algorithm. This is exactly what the function `compare_metrics` does, note that we implemented that, it is not present in the package. Besides to the printing of the dataframe, we can also see the various barplots. From the charts we can easily see how the performance are more or less equal between them, except -as expected- the 2-dimensional GMM.

```
metrics_df = compare_metrics(df1[,-c(1:2)], metrics, metrics_abb)
metrics_df
```

```
##                   RI       ARI   Jaccard Fowlkes-Mallows    Purity   Entropy
## y_3gmm     0.8427879 0.6502930 0.6254062       0.7696448 0.8705714 0.3302632
## y_2gmm     0.6625322 0.3254161 0.4304680       0.6120563 0.5697143 0.2774854
## y_3Mn_rcpp 0.8286340 0.6189729 0.5987497       0.7491146 0.8562857 0.3562607
## y_3Mn_arma 0.8261631 0.6136912 0.5945199       0.7457822 0.8534286 0.3561097
## y_3Md_pam  0.8214734 0.6037756 0.5867451       0.7396088 0.8474286 0.3572842
```