Clustering Documentation

Release 0.3.0

Dahua Lin and contributors

Aug 26, 2017

Contents

Clustering.jl is a Julia package for data clustering. The package provides a variety of clustering algorithms, as well as utilities for initialization and result evaluation.

Contents:

CHAPTER 1

Overview

Clustering.jl provides functionalities in three aspects that are related to data clustering:

- Clustering initialization, *e.g.* K-means++ seeding.
- Clustering algorithms, *e.g.* K-means, K-medoids, Affinity propagation, and DBSCAN, etc.
- Clustering evaluation, *e.g.* Silhouettes and variational information.

Inputs

A clustering algorithm, depending on its nature, may accept an input matrix in either of the following forms:

- Sample matrix `X`, where each column `X[:,i]` corresponds to an observed sample.
- Distance matrix `D`, where `D[i,j]` indicates the distance between samples `i` and `j`, or the cost of assigning one to the other.

Common Options

Many clustering algorithms are iterative procedures. There options are usually provided to the users to control the behavior the algorithm.

name	description
`maxiter`	Maximum number of iterations.
`tol`	Tolerable change of objective at convergence. The algorithm is considered to be converged when the change of objective value between consecutive iteration is below the specified value.
`display`	The level of information to be displayed. This should be a symbol, which may take either of the following values: • `:none`: nothing will be shown • **`:final`: only shows a brief summary when the algorithm ends** • `:iter`: shows progess at each iteration

Results

A clustering algorithm would return a struct that captures both the clustering results (*e.g.* assignments of samples to clusters) and information about the clustering procedure (*e.g.* the number of iterations or whether the iterative update converged).

Generally, the resultant struct is defined as an instance of a sub-type of `ClusteringResult`. The following generic methods are implemented for these subtypes (let `R` be an instance):

`nclusters`(*R*)
Get the number of clusters

`assignments`(*R*)
Get a vector of assignments.

Let `a = assignments(R)`, then `a[i]` is the index of the cluster to which the `i`-th sample is assigned.

`counts`(*R*)
Get sample counts of clusters.

Let `c = counts(R)`, then `c[k]` is the number of samples assigned to the `k`-th cluster.

CHAPTER 2

Clustering Initialization

A clustering algorithm usually relies on an initialization scheme to bootstrap the clustering procedure.

Seeding

Seeding is an important family of methods for clustering initialization, which generally refers to an procedure to select a few *seeds* from a data set, each serving as the initial center of a cluster.

Seeding functions

The packages provide two functions `initseeds` and `initseeds_by_costs` for seeding:

`initseeds` (*algname*, *X*, *k*)

Select `k` seeds from a given sample matrix `X`.

It returns an integer vector of length `k` that contains the indexes of chosen seeds.

Here, `algname` indicates the seeding algorithm, which should be a symbol that may take either of the following values:

algname	description
`:rand`	Randomly select a subset as seeds
`:kmpp`	Kmeans++ algorithm, *i.e.* choose seeds sequentially, the probability of an sample to be chosen is proportional to the minimum cost of assigning it to existing seeds. **Reference:** D. Arthur and S. Vassilvitskii (2007). *K-means++: the Advantages of Careful Seeding*. 18th Annual ACM-SIAM symposium on Discrete algorithms, 2007.
`:kmcen`	Choose the `k` samples with highest centrality as seeds.

initseeds_by_costs(*algname*, *C*, *k*)
Select `k` seeds based on a cost matrix `C`.

Here, `C[i, j]` is the cost of binding samples `i` and `j` to the same cluster. One may, for example, use the squared Euclidean distance between samples as the costs.

The argument `algname` determines the choice of algorithm (see above).

In practice, we found that Kmeans++ is the most effective method for initial seeding. Thus, we provide specific functions to simply the use of Kmeans++ seeding:

kmpp(*X*, *k*)
Use *Kmeans++* to choose `k` seeds from a data set given by a sample matrix `X`.

kmpp_by_costs(*C*, *k*)
Use *Kmeans++* to choose `k` seeds based on a cost matrix `C`.

Internals

In this package, each seeding algorithm is represented by a sub-type of `SeedingAlgorithm`. Particularly, the random selection algorithm, Kmean++, and centrality-based algorithm are respectively represented by sub-types `RandSeedAlg`, `KmppAlg`, and `KmCentralityAlg`.

For each sub type, the following methods are implemented:

initseeds!(iseeds, alg, X)
Select seeds from a given sample matrix `X`, and write the results to `iseeds`.

Parameters

- **iseeds** – An pre-allocated array to store the indexes of the chosen seeds.
- **alg** – The algorithm instance.
- **X** – The given data matrix. Each column of `X` is a sample.

Returns `iseeds`

initseeds_by_costs!(iseeds, alg, C)
Select seeds based on a given cost matrix `C`, and write the results to `iseeds`.

Parameters

- **iseeds** – An pre-allocated array to store the indexes of the chosen seeds.
- **alg** – The algorithm instance.
- **C** – The cost matrix. The value of `C[i, j]` is the cost of binding samples `i` and `j` into the same cluster.

Returns `iseeds`

Note: For both functions above, the length of `iseeds` determines the number of seeds to be selected.

To define a new seeding algorithm, one has to first define a sub type of `SeedingAlgorithm` and implement the two functions above.

CHAPTER 3

Clustering Algorithms

This package implements a variety of clustering algorithms:

K-means

K-means is a classic method for clustering or vector quantization. The K-means algorithms produces a fixed number of clusters, each associated with a *center* (also known as a *prototype*), and each sample belongs to a cluster with the nearest center.

From a mathematical standpoint, K-means is a coordinate descent algorithm to solve the following optimization problem:

$$\begin{aligned} &\text{minimize } \|\mathbf{x}_i - \boldsymbol{\mu}_{z_i}\|^2 \\ &\text{w.r.t. } (\boldsymbol{\mu}, z) \end{aligned}$$

Here, $\boldsymbol{\mu}_k$ is the center of the `k`-th cluster, and z_i indicates the cluster for $\mathbf{x}_i$.

This package implements the *K-means* algorithm in the `kmeans` function:

kmeans (*X*, *k*; ...)

Performs K-means clustering over the given dataset.

Parameters

- **X** – The given sample matrix. Each column of `X` is a sample.
- **k** – The number of clusters.

This function returns an instance of `KmeansResult`, which is defined as follows:

```
type KmeansResult{T<:FloatingPoint} <: ClusteringResult
    centers::Matrix{T}          # cluster centers, size (d, k)
    assignments::Vector{Int}    # assignments, length n
    costs::Vector{T}            # costs of the resultant assignments, length n
    counts::Vector{Int}         # number of samples assigned to each cluster,␣
↪length k
```

```
    cweights::Vector{Float64}   # cluster weights, length k
    totalcost::Float64          # total cost (i.e. objective)
    iterations::Int             # number of elapsed iterations
    converged::Bool             # whether the procedure converged
end
```

One may optionally specify some of the options through keyword arguments to control the algorithm:

name	description	default
`init`	Initialization algorithm or initial seeds, which can be either of the following: •a symbol indicating the name of seeding algorithm, `:rand`, `:kmpp`, or `:kmcen` (see *Clustering Initialization*) •an integer vector of length `k` that provides the indexes of initial seeds.	`:kmpp`
`maxiter`	Maximum number of iterations.	`100`
`tol`	Tolerable change of objective at convergence.	`1.0e-6`
`weights`	The weights of samples, which can be either of: •`nothing`: each sample has a unit weight. •a vector of length `n` that gives the sample weights.	`nothing`
`display`	The level of information to be displayed. (see *Common Options*)	`:none`

If you already have a set of initial center vectors, you may use `kmeans!` instead:

`kmeans!(X, centers; ...)`

Performs K-means given initial centers, and updates the centers inplace.

Parameters

- **`X`** – The given sample matrix. Each column of `X` is a sample.
- **`centers`** – The matrix of centers. Each column of `centers` is a center vector for a cluster.

Note: The number of clusters `k` is determined as `size(centers, 2)`.

Like `kmeans`, this function returns an instance of `KmeansResult`.

This function accepts all keyword arguments listed above for `kmeans` (except `init`).

Examples:

```
using Clustering

# make a random dataset with 1000 points
# each point is a 5-dimensional vector
X = rand(5, 1000)
```

```
# performs K-means over X, trying to group them into 20 clusters
# set maximum number of iterations to 200
# set display to :iter, so it shows progressive info at each iteration
R = kmeans(X, 20; maxiter=200, display=:iter)

# the number of resultant clusters should be 20
@assert nclusters(R) == 20

# obtain the resultant assignments
# a[i] indicates which cluster the i-th sample is assigned to
a = assignments(R)

# obtain the number of samples in each cluster
# c[k] is the number of samples assigned to the k-th cluster
c = counts(R)

# get the centers (i.e. mean vectors)
# M is a matrix of size (5, 20)
# M[:,k] is the mean vector of the k-th cluster
M = R.centers
```

Example with plot

```
using RDatasets

iris = dataset("datasets", "iris")
head(iris)

# K-means Clustering unsupervised machine learning example

using Clustering

features = permutedims(convert(Array, iris[:,1:4]), [2, 1])    # use matrix() on Julia␣
↪v0.2
result = kmeans( features, 3 )                                 # onto 3 clusters

using Gadfly

plot(iris, x = "PetalLength", y = "PetalWidth", color = result.assignments, Geom.
↪point)
```

K-medoids

K-medoids is a clustering algorithm that seeks a subset of points out of a given set such that the total costs or distances between each point to the closest point in the chosen subset is minimal. This chosen subset of points are called *medoids*.

This package implements a K-means style algorithm instead of PAM, which is considered to be much more efficient and reliable. Particularly, the algorithm is implemented by the `kmedoids` function.

kmedoids(*C*, *k*; ...)

Performs K-medoids clustering based on a given cost matrix.

Parameters

- **C** – The cost matrix, where `C[i, j]` is the cost of assigning sample `j` to the medoid `i`.

- **k** – The number of clusters.

This function returns an instance of `KmedoidsResult`, which is defined as follows:

```
type KmedoidsResult{T} <: ClusteringResult
    medoids::Vector{Int}        # indices of methods (k)
    assignments::Vector{Int}    # assignments (n)
    acosts::Vector{T}           # costs of the resultant assignments (n)
    counts::Vector{Int}         # number of samples assigned to each cluster (k)
    totalcost::Float64          # total assignment cost (i.e. objective) (k)
    iterations::Int             # number of elapsed iterations
    converged::Bool             # whether the procedure converged
end
```

One may optionally specify some of the options through keyword arguments to control the algorithm:

name	description	default
`init`	Initialization algorithm or initial medoids, which can be either of the following: •a symbol indicating the name of seeding algorithm, `:rand`, `:kmpp`, or `:kmcen` (see *Clustering Initialization*) •an integer vector of length `k` that provides the indexes of initial seeds.	`:kmpp`
`maxiter`	Maximum number of iterations.	`100`
`tol`	Tolerable change of objective at convergence.	`1.0e-6`
`display`	The level of information to be displayed. (see *Common Options*)	`:none`

kmedoids!(C, medoids, ...)

Performs K-medoids clustering based on a given cost matrix.

This function operates on an given set of medoids and updates it inplace.

Parameters

- **C** – The cost matrix, where `C[i, j]` is the cost of assigning sample `j` to the medoid `i`.
- **medoids** – The vector of medoid indexes. The contents of `medoids` serve as the initial guess and will be overrided by the results.

This function returns an instance of `KmedoidsResult`.

One may optionally specify some of the options through keyword arguments to control the algorithm:

name	description	default
`maxiter`	Maximum number of iterations.	`100`
`tol`	Tolerable change of objective at convergence.	`1.0e-6`
`display`	The level of information to be displayed. (see *Common Options*)	`:none`

Affinity Propagation

Affinity propagation is a clustering algorithm based on *message passing* between data points. Similar to *K-medoids*, it finds a subset of points as *exemplars* based on (dis)similarities, and assigns each point in the given data set to the closest exemplar.

This package implements the affinity propagation algorithm based on the following paper:

> Brendan J. Frey and Delbert Dueck. *Clustering by Passing Messages Between Data Points.* Science, vol 315, pages 972-976, 2007.

The implementation is optimized by reducing unnecessary array allocation and fusing loops. Specifically, the algorithm is implemented by the `affinityprop` function:

affinityprop(*S; ...*)
Performs affinity propagation based on a similarity matrix `S`.

Parameters **S** – The similarity matrix. Here, `S[i,j]` is the similarity (or negated distance) between samples `i` and `j` when `i != j`; while `S[i,i]` reflects the *availability* of the `i`-th sample as an exemplar.

This function returns an instance of `AffinityPropResult`, defined as below:

```
type AffinityPropResult <: ClusteringResult
    exemplars::Vector{Int}      # indexes of exemplars (centers)
    assignments::Vector{Int}    # assignments for each point
    iterations::Int             # number of iterations executed
    converged::Bool             # converged or not
end
```

One may optionally specify the following keyword arguments:

name	description	default
`maxiter`	Maximum number of iterations.	`100`
`tol`	Tolerable change of objective at convergence.	`1.0e-6`
`damp`	Dampening coefficient. The value should be in `[0.0, 1.0)`. Larger value of `damp` indicates slower (and probably more stable) update. When `damp = 0`, it means no dampening is performed.	`0.5`
`display`	The level of information to be displayed (see *Common Options*)	`:none`

DBSCAN

Density-based Spatial Clustering of Applications with Noise (DBSCAN) is a data clustering algorithm that finds clusters through density-based expansion of seed points. The algorithm is proposed by:

> Martin Ester, Hans-peter Kriegel, Jörg S, and Xiaowei Xu *A density-based algorithm for discovering clusters in large spatial databases with noise.* 1996.

Density Reachability

DBSCAN's definition of cluster is based on the concept of *density reachability*: a point q is said to be *directly density reachable* by another point p if the distance between them is below a specified threshold ϵ and p is surrounded by sufficiently many points. Then, q is considered to be *density reachable* by p if there exists a sequence $p_1, p_2, \ldots, p_n$ such that $p_1 = p$ and p_{i+1} is directly density reachable from p_i.

A cluster, which is a subset of the given set of points, satisfies two properties:

1. All points within the cluster are mutually *density-connected*, meaning that for any two distinct points p and q in a cluster, there exists a point o sucht that both p and q are density reachable from o.
2. If a point is density connected to any point of a cluster, it is also part of the cluster.

Functions

There are two different implementations of *DBSCAN* algorithm called by `dbscan` function in this package:

1. Using a distance (adjacency) matrix and is O(N^2) in memory usage. Note that the boundary points are not unique.

dbscan (*D*, *eps*, *minpts*)
Perform DBSCAN algorithm based on a given distance matrix.

Parameters

- **D** – The pairwise distance matrix. `D[i,j]` is the distance between points `i` and `j`.
- **eps** – The radius of a neighborhood.
- **minpts** – The minimum number of neighboring points (including self) to qualify a point as a density point.

The algorithm returns an instance of `DbscanResult`, defined as below:

```
type DbscanResult <: ClusteringResult
    seeds::Vector{Int}          # starting points of clusters, size (k,)
    assignments::Vector{Int}    # assignments, size (n,)
    counts::Vector{Int}         # number of points in each cluster, size (k,)
end
```

2. Using an adjacency list which is build on the fly. The performance is much better both in terms of runtime and memory usage. Also, the result is given in a DbscanCluster that provides the indices of all the core points and boundary points, such that boundary points can be associated with multiple clusters.

dbscan (*points*, *radius*, *leafsize=20*, *min_neighbors=1*, *min_cluster_size=1*)
Perform DBSCAN algorithm based on a collection of points.

Parameters

- **points** – matrix of points (column based)
- **radius** – The radius of a neighborhood.
- **leafsize** – number of points binned in each leaf node in the *KDTree*
- **min_neighbors** – minimum number of neighbors to be a core point
- **min_cluster_size** – minimum number of points to be a valid cluster

The algorithm returns an instance of `DbscanCluster`, defined as below:

```
```

```
    immutable DbscanCluster <: ClusteringResult size::Int # number of points in cluster
        core_indices::Vector{Int} # core points indices boundary_indices::Vector{Int} # boundary points
        indices

    end
```

CHAPTER 4

Clustering Validation

This package provides a variety of ways to validate or evaluate clustering results:

Silhouettes

Silhouettes is a method for validating clusters of data. Particularly, it provides a quantitative way to measure how well each item lies within its cluster as opposed to others. The *Silhouette* value of a data point is defined as:

$$s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))}$$

Here, $a(i)$ is the average distance from the `i`-th point to other points within the same cluster. Let $b(i, k)$ be the average distance from the `i`-th point to the points in the `k`-th cluster. Then $b(i)$ is the minimum of all $b(i, k)$ over all clusters that the `i`-th point is not assigned to.

Note that the value of $s(i)$ is not greater than one, and that $s(i)$ is close to one indicates that the `i`-th point lies well within its own cluster.

`silhouettes` (*assignments*, *counts*, *dists*)

Compute silhouette values for individual points w.r.t. a given clustering.

Parameters

- **`assignments`** – the vector of assignments
- **`counts`** – the number of points falling in each cluster
- **`dists`** – the pairwise distance matrix

Returns It returns a vector of silhouette values for individual points. In practice, one may use the average of these silhouette values to assess given clustering results.

`silhouettes` (*R*, *dists*)

This method accepts a clustering result `R` (of a sub-type of `ClusteringResult`).

It is equivalent to `silhouettes(assignments(R), counts(R), dists)`.

Variation of Information

Variation of information (also known as *shared information distance*) is a measure of the distance between two clusterings. It is devised based on mutual information, but it is a true metric, *i.e.* it satisfies symmetry and triangle inequality.

References:

Meila, Marina (2003). *Comparing Clusterings by the Variation of Information.* Learning Theory and Kernel Machines: 173–187.

This package provides the `varinfo` function that implements this metric:

varinfo(*k1*, *a1*, *k2*, *a2*)
Compute the variation of information between two assignments.

Parameters

- **k1** – The number of clusters in the first clustering.
- **a1** – The assignment vector for the first clustering.
- **k2** – The number of clusters in the second clustering.
- **a2** – The assignment vector for the second clustering.

Returns the value of variation of information.

varinfo(*R*, *k0*, *a0*)
This method takes `R`, an instance of `ClusteringResult`, as input, and computes the variation of information between its corresponding clustering with one given by `(k0, a0)`, where `k0` is the number of clusters in the other clustering, while `a0` is the corresponding assignment vector.

varinfo(*R1*, *R2*)
This method takes `R1` and `R2` (both are instances of `ClusteringResult`) and computes the variation of information between them.

Rand indices

Rand index is a measure of the similarity between two data clusterings. From a mathematical standpoint, Rand index is related to the accuracy, but is applicable even when class labels are not used.

References:

Lawrence Hubert and Phipps Arabie (1985). *Comparing partitions.* Journal of Classification 2 (1): 193–218

Meila, Marina (2003). *Comparing Clusterings by the Variation of Information.* Learning Theory and Kernel Machines: 173–187.

This package provides the `randindex` function that implements several metrics:

randindex(*c1*, *c2*)
Compute the tuple of indices (Adjusted Rand index, Rand index, Mirkin's index, Hubert's index) between two assignments.

Parameters

- **c1** – The assignment vector for the first clustering.
- **c2** – The assignment vector for the second clustering.

Returns tuple of indices.

randindex(*R*, *c0*)

This method takes `R`, an instance of `ClusteringResult`, as input, and computes the tuple of indices (see above) where `c0` is the corresponding assignment vector.

randindex(*R1*, *R2*)

This method takes `R1` and `R2` (both are instances of `ClusteringResult`) and computes the tuple of indices (see above) between them.

Index