L09 K-means Algorithm

Ivan Slapničar

May 8, 2018

1 K-means Algorithm

Data clustering is one of the main mathematical applications variety of algorithms have been developed to tackle the problem. K-means is one of the basic algorithms for data clustering.

1.1 Prerequisites

The reader should be familiar with basic linear algebra.

1.2 Competences

The reader should be able to recognise applications where K-means algorithm can be efficiently used and use it.

Credits: The notebook is based on [Mir05].

References

[Mir05] I. Mirošević, 'Spectral Graph Partitioning and Application to Knowledge Extraction', M.Sc. Thesis, University of Zagreb, 2005 (in Croatian).

1.3 Definitions

Data clustering problem is the following: partition the given set of m objects of the same type into k subsets according to some criterion. Additional request may be to find the optimal k.

K-means clustering problem is the following: partition the set $X = \{x_1, x_2, \cdots, x_m\}$, where $x_i \in \mathbb{R}^n$, into k **clusters** $\pi = \{C_1, C_2, ..., C_k\}$ such that

$$J(\pi) = \sum_{i=1}^k \sum_{x\in C_i} \|x - c_i\|_2^2 \to \min$$

over all possible partitions. Here $c_i = \displaystyle\frac{1}{|C_i|} \sum_{x\in C_i} x$ is the mean of points in C_i and $|C_i|$ is the cardinality of C_i.

K-means clustering algorithm is the following: 1. *Initialization*: Choose initial set of k means $\{c_1,\ldots,c_k\}$ (for example, by choosing randomly k points from X). 2. *Assignment step*: Assign each point x to one nearest mean c_i. 3. *Update step*: Compute the new means. 4. *Convergence*: Repeat Steps 2 and 3 until the assignment no longer changes.

A **first variation** of a partition $\pi=\{C_1,\ldots,C_k\}$ is a partition $\pi'=\{C_1',\cdots,C_k'\}$ obtained by moving a single point x from a cluster C_i to a cluster C_j. Notice that π is a first variation of itself.

A **next partition** of the partition π is a partition $next(\pi)=\underset{\pi'}{\operatorname{argmin}}\, J(\pi')$.

First Variation clustering algorithm is the following: 1. Choose initial partition π. 2. Compute $next(\pi)$ 3. If $J(next(\pi))<J(\pi)$, set $\pi=next(\pi)$ and go to Step 2 4. Stop.

1.4 Facts

1. The k-means clustering problem is NP-hard.
2. In the k-means algorithm, $J(\pi)$ decreases in every iteration.
3. K-means algorithm can converge to a local minimum.
4. Each iteration of the k-means algorithm requires $O(mnk)$ operations.
5. K-means algorithm is implemented in the function `kmeans()` in the package Clustering.jl.
6. $J(\pi)=\operatorname{trace}(S_W)$, where

$$S_W=\sum_{i=1}^k\sum_{x\in C_i}(x-c_i)(x-c_i)^T=\sum_{i=1}^k\frac{1}{2|C_i|}\sum_{x\in C_i}\sum_{y\in C_i}(x-y)(x-y)^T.$$

   Let c denote the mean of X. Then $S_W=S_T-S_B$, where

$$S_T=\sum_{x\in X}(x-c)(x-c)^T=\frac{1}{2m}\sum_{i=1}^m\sum_{j=1}^m(x_i-x_j)(x_i-x_j)^T,$$
$$S_B=\sum_{i=1}^k|C_i|(c_i-c)(c_i-c)^T=\frac{1}{2m}\sum_{i=1}^k\sum_{j=1}^k|C_i||C_j|(c_i-c_j)(c_i-c_j)^T.$$

7. In order to try to avoid convergence to local minima, the k-means algorithm can be enhanced with first variation by adding the following steps:
   1. Compute $next(\pi)$.
   2. If $J(next(\pi))<J(\pi)$, set $\pi=next(\pi)$ and go to Step 2.

1.4.1 Example - Random clusters

We generate k random clusters around points with integer coordinates.

In [1]:
```
function myKmeans(X::Array, k::Int64)
    # X is Array of Arrays
    m,n=length(X),length(X[1])
    T=typeof((X[1])[1])
    C=Array{Int64}(m)
    # Choose random k means among X
    c=X[randperm(m)[1:k]]
    # This is just to start the while loop
    cnew=copy(c)
    cnew[1]=cnew[1]+[one(T);one(T)]
    # Loop
    iterations=0
    while cnew!=c
        iterations+=1
        cnew=copy(c)
        # Assignment
        for i=1:m
            C[i]=findmin([norm(X[i]-c[j]) for j=1:k])[2]
        end
        # Update
        for j=1:k
            c[j]=mean(X[C.==j])
        end
    end
    C,c,iterations
end
```

Out[1]:
```
myKmeans (generic function with 1 method)
```

In [2]:
```
# Pkg.checkout("Winston")
using Winston
using Colors
```

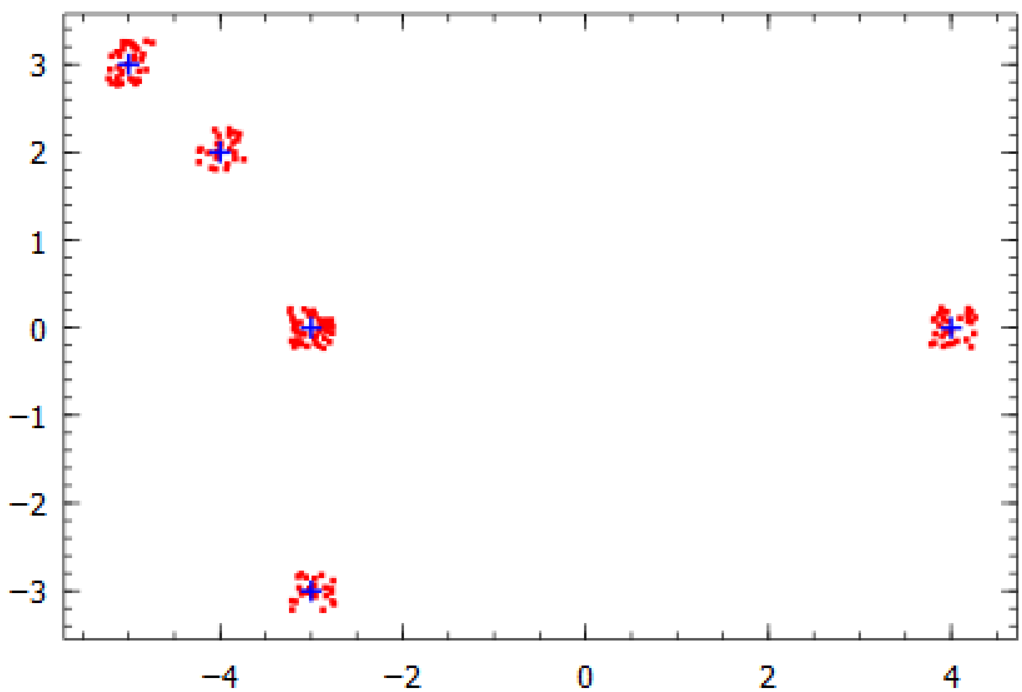
In [3]:
```
# Generate points
k=5
s=srand(421)
centers=rand(-5:5,k,2)
# Number of points in cluster
sizes=rand(10:50,k)
# X is array of arrays
X=Array{Array{Float64,1}}(sum(sizes))
first=0
last=0
for j=1:k
    first=last+1
    last=last+sizes[j]
    for i=first:last
```

```
            X[i]=vec(centers[j,:])+(rand(2)-0.5)/2
        end
    end
    centers, sizes
```

Out[3]: ([4 0; -3 -3; ... ; -5 3; -4 2], [30, 24, 48, 40, 26])

In [4]:
```
# Prepare for plot
function extractxy(X::Array)
    x=[X[i][1] for i=1:length(X)]
    y=[X[i][2] for i=1:length(X)]
    x,y
end
x,y=extractxy(X)
plot(x,y,"r.",centers[:,1],centers[:,2],"b+")
```

Out[4]:



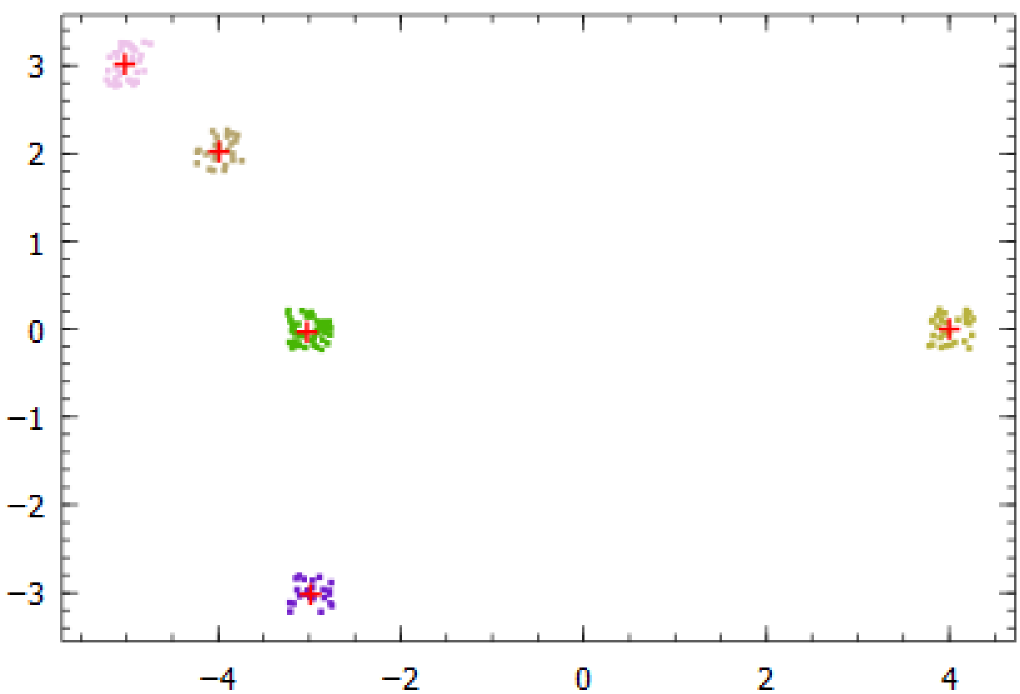
In [5]:
```
# Plot the solution
function plotKmeansresult(C::Array,c::Array,X::Array)
    p=FramedPlot()
    x,y=extractxy(X)
```

```
    # Custers
    for j=1:k
        # Random color
        col=RGB(rand(),rand(),rand())
        p1=Points(x[find(C.==j)],y[find(C.==j)],
        "color",col,symbolkind="dot")
        add(p,p1)
    end
    # Means
    cx,cy=extractxy(c)
    p2=Points(cx,cy,color="red",symbolkind="plus")
    add(p,p2)
end
```

Out[5]: plotKmeansresult (generic function with 1 method)

In [6]:
```
# Cluster the data, repeat several times
C,c,iterations=myKmeans(X,k)
plotKmeansresult(C,c,X)
```

Out[6]:



What happens?

We see that the algorithm, although simple, may converge to a local minimum.

Let us try the function `kmeans()` from the package `Clustering.jl`. In this function, the inpout is a matrix where columns are points, number of cluster we are looking for, and, optionally, the method to compute initial means.

If we choose `init=:rand`, the results are similar. If we choose `init=:kmpp`, wjich is the default, the results are better, but convergence to a local minimum is still possible.

Run the clustering several times!

```
seeding_algorithm(s::Symbol) =
    s == :rand ? RandSeedAlg() :
    s == :kmpp ? KmppAlg() :
    s == :kmcen ? KmCentralityAlg() :
    error("Unknown seeding algorithm $s")
```

```
In [7]: # Pkg.add("Clustering")
        using Clustering
```

```
In [8]: methods(kmeans)
```

```
Out[8]: # 1 method for generic function "kmeans":
        kmeans(X::Array{T,2}, k::Int64; weights, init, maxiter, tol, display, distance) where T<
```

```
In [9]: X1=[x y]'
        out=kmeans(X1,k,init=:kmpp)
```

```
Out[9]: Clustering.KmeansResult{Float64}([-3.99062 4.00132 ... -3.02492 -5.02088; 2.01964 0.0030
```

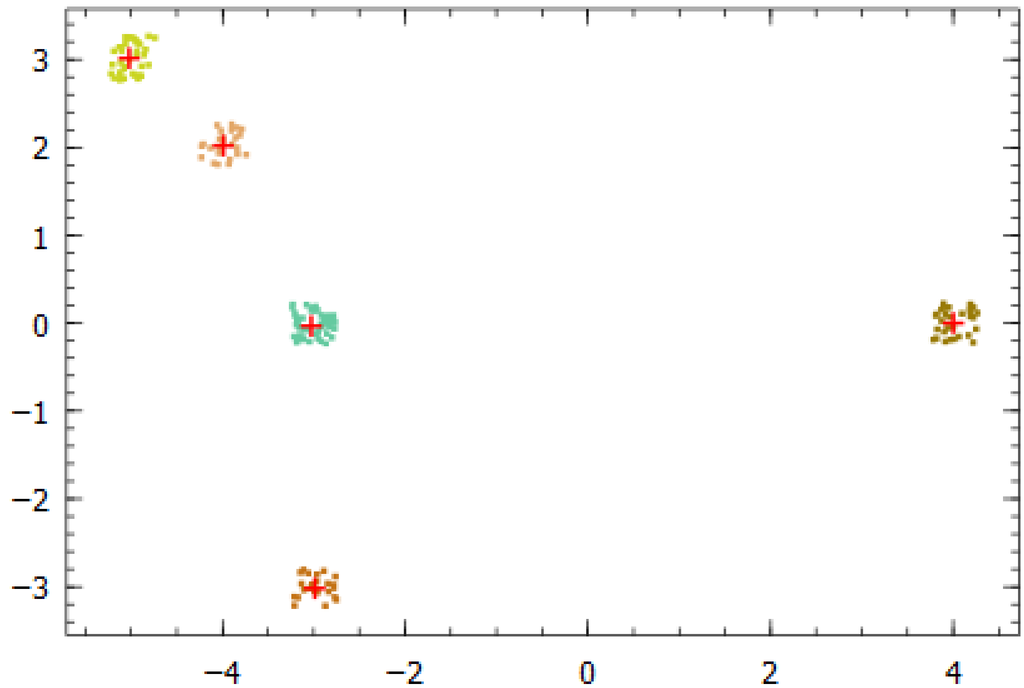
```
In [10]: fieldnames(KmeansResult)
```

```
Out[10]: 8-element Array{Symbol,1}:
          :centers
          :assignments
          :costs
          :counts
          :cweights
          :totalcost
          :iterations
          :converged
```

```
In [11]: out.centers
```

```
Out[11]: 2×5 Array{Float64,2}:
          -3.99062  4.00132     -2.9848   -3.02492    -5.02088
           2.01964  0.00302082  -3.01761  -0.0305806   3.01373
```

```
In [12]: println(out.assignments)

[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3,


In [13]: # We need to modify the plotting function
         function plotKmeansresult(out::KmeansResult,X::Array)
             p=FramedPlot()
             # Custers
             k=size(out.centers,2)
             for j=1:k
                 # Random color
                 col=RGB(rand(),rand(),rand())
                 p1=Points(X[1,find(out.assignments.==j)],
                 X[2,find(out.assignments.==j)],"color",col,symbolkind="dot")
                 add(p,p1)
             end
             # Means
             p2=Points(out.centers[1,:],out.centers[2,:],
             color="red",symbolkind="plus")
             add(p,p2)
         end

Out[13]: plotKmeansresult (generic function with 2 methods)

In [14]: plotKmeansresult(out,X1)

Out[14]:
```



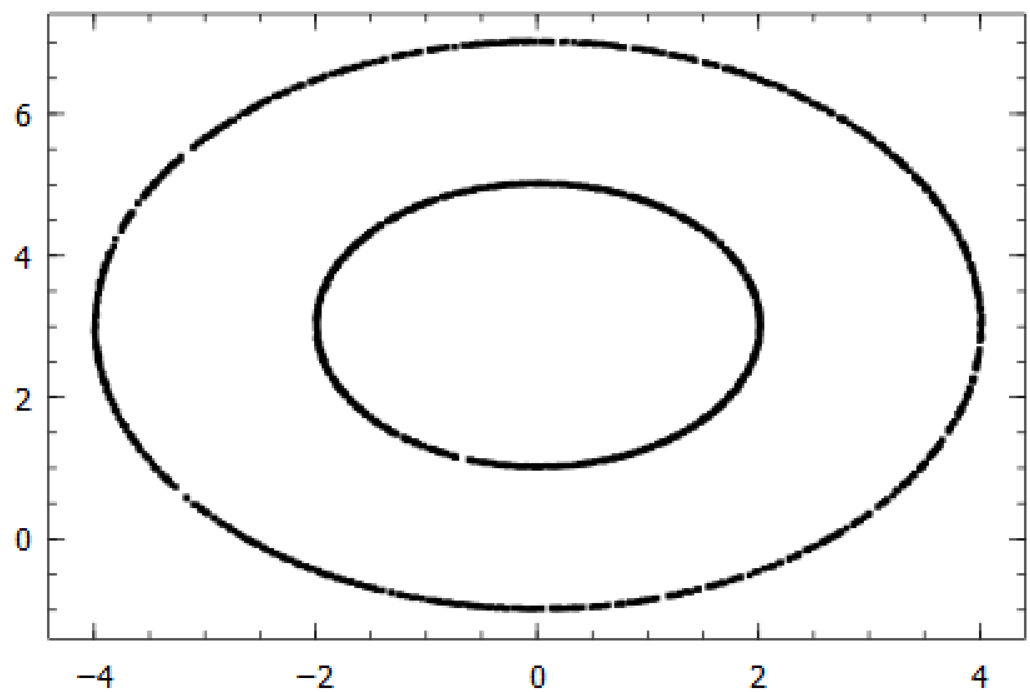
1.4.2 Example - Concentric rings

The k-means algorithm works well if clusters can be separated by hyperplanes. In this example it is not the case.

```
In [15]: # Number of rings, try also k=3
         k=2
         # Center
         center=[rand(-5:5);rand(-5:5)]
         # Radii
         radii=randperm(10)[1:k]
         # Number of points in circles
         sizes=rand(1000:2000,k)
         center,radii,sizes

Out[15]: ([0, 3], [2, 4], [1029, 1069])

In [16]: # Points
         X=Array{Float64}(2,sum(sizes))
         first=0
         last=0
         for j=1:k
```

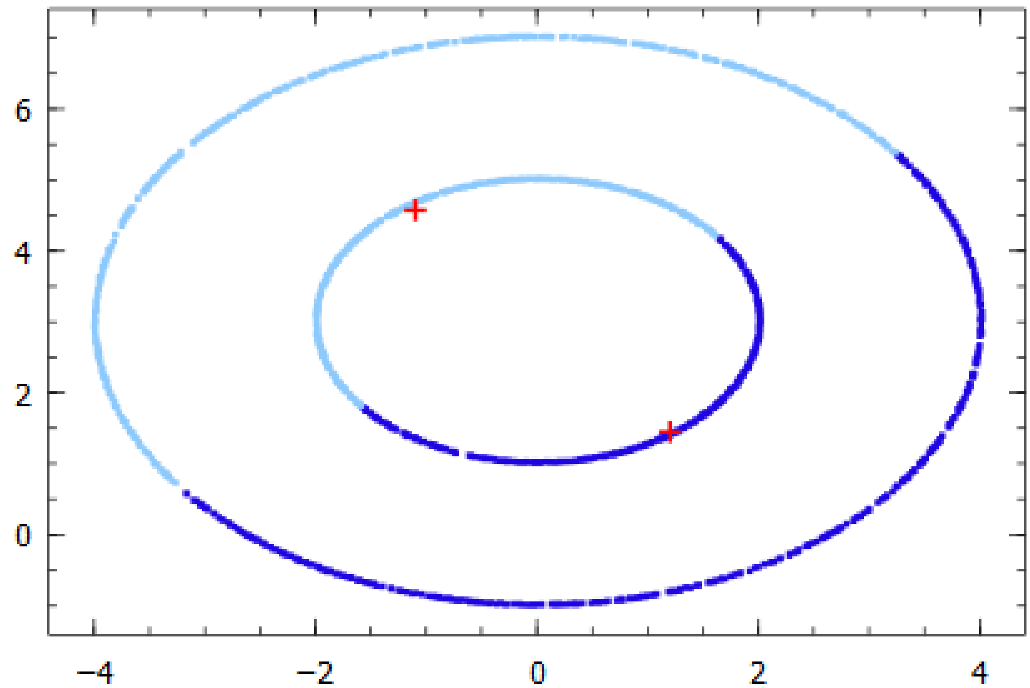
```
        first=last+1
        last=last+sizes[j]
        # Random angles
        ϕ=2*π*rand(sizes[j])
        for i=first:last
            l=i-first+1
            X[:,i]=center+radii[j]*[cos(ϕ[l]);sin(ϕ[l])]+(rand(2)-0.5)/50
        end
    end
    plot(X[1,:],X[2,:],".")
```

Out[16]:



In [17]:
```
out=kmeans(X,k)
plotKmeansresult(out,X)
```

Out[17]:



In [ ]: