**Clustering With K-Means in Python**

A very common task in data analysis is that of grouping a set of objects into subsets such that all elements within a group are more similar among them than they are to the others. The practical applications of such a procedure are many: given a medical image of a group of cells, a clustering algorithm could aid in [identifying the centers of the cells](http://www.academia.edu/2496077/Automatic_Segmentation_of_Skin_Cancer_Images_using_Adaptive_Color_Clustering); looking at the GPS data of a user’s mobile device, their more frequently visited locations within a certain radius can be revealed; for any set of [unlabeled observations](http://en.wikipedia.org/wiki/Unsupervised_learning), clustering helps establish the existence of some sort of structure that might indicate that the data is separable.

**Mathematical background**

The k-means algorithm takes a dataset *X* of *N* points as input, together with a parameter *K* specifying how many clusters to create. The output is a set of *K* cluster centroids and a labeling of *X* that assigns each of the points in *X* to a unique cluster. All points within a cluster are closer in distance to their centroid than they are to any other centroid.

The mathematical condition for the *K* clusters _k and the *K* centroids mu_k can be expressed as:

Minimize displaystyle \sum_{k=1}^K \sum_{\mathrm{x}_n \in C_k} ||\mathrm{x}_n - \mu_k ||^2 with respect to displaystyle C_k, \mu_k.

**Lloyd’s algorithm**

Finding the solution is unfortunately [NP hard](http://en.wikipedia.org/wiki/NP-hard). Nevertheless, an iterative method known as Lloyd’s algorithm exists that converges (albeit to a local minimum) in few steps. The procedure alternates between two operations. (1) Once a set of centroids mu_k is available, the clusters are updated to contain the points closest in distance to each centroid. (2) Given a set of clusters, the centroids are recalculated as the means of all points belonging to a cluster.

displaystyle \mu_k = \frac{1}{C_k}\sum_{\mathrm{x}_n \in C_k}\mathrm{x}_n\qquad(2)

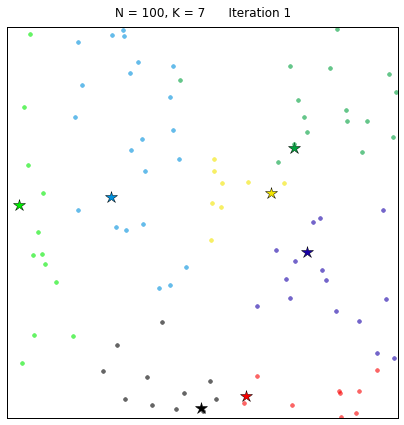
The two-step procedure continues until the assignments of clusters and centroids no longer change. As already mentioned, the convergence is guaranteed but the solution might be a local minimum. In practice, the algorithm is run multiple times and averaged. For the starting set of centroids, several methods can be employed, for instance random assignation.

Below is a simple implementation of Lloyd’s algorithm for performing k-means clustering in python:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | import numpy as np    def cluster\_points(X, mu):      clusters  = {}      for x in X:          bestmukey = min([(i[0], np.linalg.norm(x-mu[i[0]])) \                      for i in enumerate(mu)], key=lambda t:t[1])[0]          try:              clusters[bestmukey].append(x)          except KeyError:              clusters[bestmukey] = [x]      return clusters    def reevaluate\_centers(mu, clusters):      newmu = []      keys = sorted(clusters.keys())      for k in keys:          newmu.append(np.mean(clusters[k], axis = 0))      return newmu    def has\_converged(mu, oldmu):      return (set([tuple(a) for a in mu]) == set([tuple(a) for a in oldmu])    def find\_centers(X, K):      # Initialize to K random centers      oldmu = random.sample(X, K)      mu = random.sample(X, K)      while not has\_converged(mu, oldmu):          oldmu = mu          # Assign all points in X to clusters          clusters = cluster\_points(X, mu)          # Reevaluate centers          mu = reevaluate\_centers(oldmu, clusters)      return(mu, clusters) |

**Clustering in action**

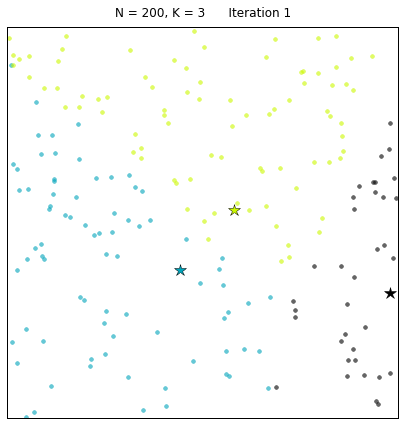
Let’s see the algorithm in action! For an ensemble of 100 random points on the plane, we set the k-means function to find 7 clusters. The code converges in 7 iterations after initializing with random centers. In the following plots, dots correspond to the target data points *X* and stars represent the centroids mu_k of the clusters. Each cluster is distinguished by a different color.

[](https://datasciencelab.files.wordpress.com/2013/12/p_n100_k7.gif)

The initial configuration of points for the algorithm is created as follows:

|  |  |
| --- | --- |
| 1  2  3  4  5 | import random    def init\_board(N):      X = np.array([(random.uniform(-1, 1), random.uniform(-1, 1)) for i in range(N)])      return X |

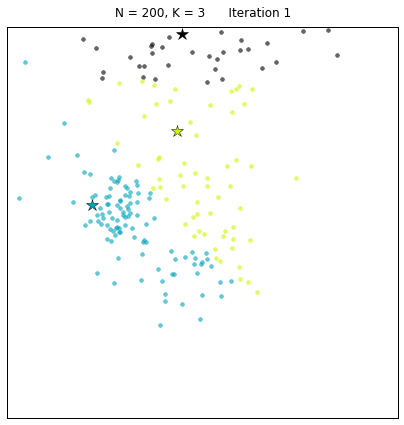
For a configuration with twice as many points and a target of 3 clusters, often the algorithm needs more iterations to converge.

[](https://datasciencelab.files.wordpress.com/2013/12/p_n200_k3.gif)

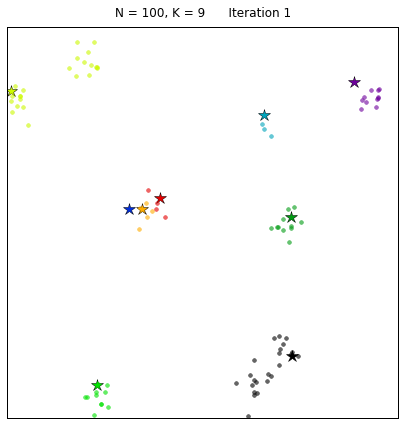
Obviously, an ensemble of randomly generated points does not possess a natural cluster-like structure. To make things slightly more tricky, we want to modify the function that generates our initial data points to output a more interesting structure. The following routine constructs a specified number of Gaussian distributed clusters with random variances:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | def init\_board\_gauss(N, k):      n = float(N)/k      X = []      for i in range(k):          c = (random.uniform(-1, 1), random.uniform(-1, 1))          s = random.uniform(0.05,0.5)          x = []          while len(x) < n:              a, b = np.array([np.random.normal(c[0], s), np.random.normal(c[1], s)])              # Continue drawing points from the distribution in the range [-1,1]              if abs(a) < 1 and abs(b) < 1:                  x.append([a,b])          X.extend(x)      X = np.array(X)[:N]      return X |

Let us look at a data set constructed as X = init\_board\_gauss(200,3): 7 iterations are needed to find the 3 centroids.

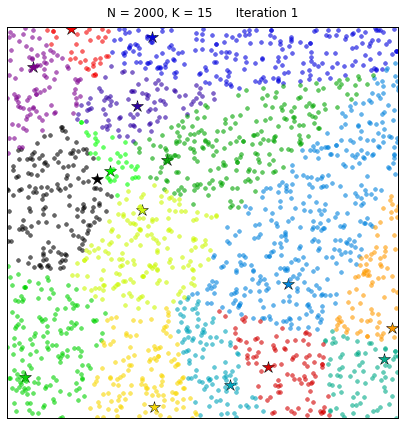
[](https://datasciencelab.files.wordpress.com/2013/12/p_n200_k3_g.gif)

If the target distribution is disjointedly clustered and only one instantiation of Lloyd’s algorithm is used, the danger exists that the local minimum reached is not the optimal solution. This is shown in the example below, where initial data using very peaked Gaussians is constructed:

[](https://datasciencelab.files.wordpress.com/2013/12/p_n100_k9_g.gif)

The yellow and black stars serve two different clusters each, while the orange, red and blue centroids are cramped within one unique blob due to an unfortunate random initialization. For this type of cases, a cleverer election of initial clusters should help.

To finalize our [table-top experiment](https://datasciencelab.wordpress.com/category/experiments/) on k-means clustering, we might want to take a look at what happens when the original data space is densely populated:

[](https://datasciencelab.files.wordpress.com/2013/12/p_n2000_k15_.gif)

The k-means algorithm induces a partition of the observations corresponding to a [Voronoi tessellation](http://en.wikipedia.org/wiki/Voronoi_diagram)generated by the *K* centroids. And it is certainly very pretty!

**Table-top data experiment take-away message**

Lloyd’s two-step implementation of the k-means algorithm allows to cluster data points into groups represented by a centroid. This technique is employed in many facets of machine learning, from unsupervised learning algorithms to dimensionality reduction problems. The general clustering problem is NP hard, but the iterative procedure converges always, albeit to a local minimum. Proper initialization of the centroids is important. Additionally, this algorithm does not supply information as to which *K* for the k-means is optimal; that has to be found out by alternative methods.

**Update:** We explore the gap statistic as a method to determine the optimal *K* for clustering in this post: [Finding the K in K-Means Clustering](https://datasciencelab.wordpress.com/2013/12/27/finding-the-k-in-k-means-clustering/) and the (K) method: [Selection of K in K-means Clustering, Reloaded](https://datasciencelab.wordpress.com/2014/01/21/selection-of-k-in-k-means-clustering-reloaded/).

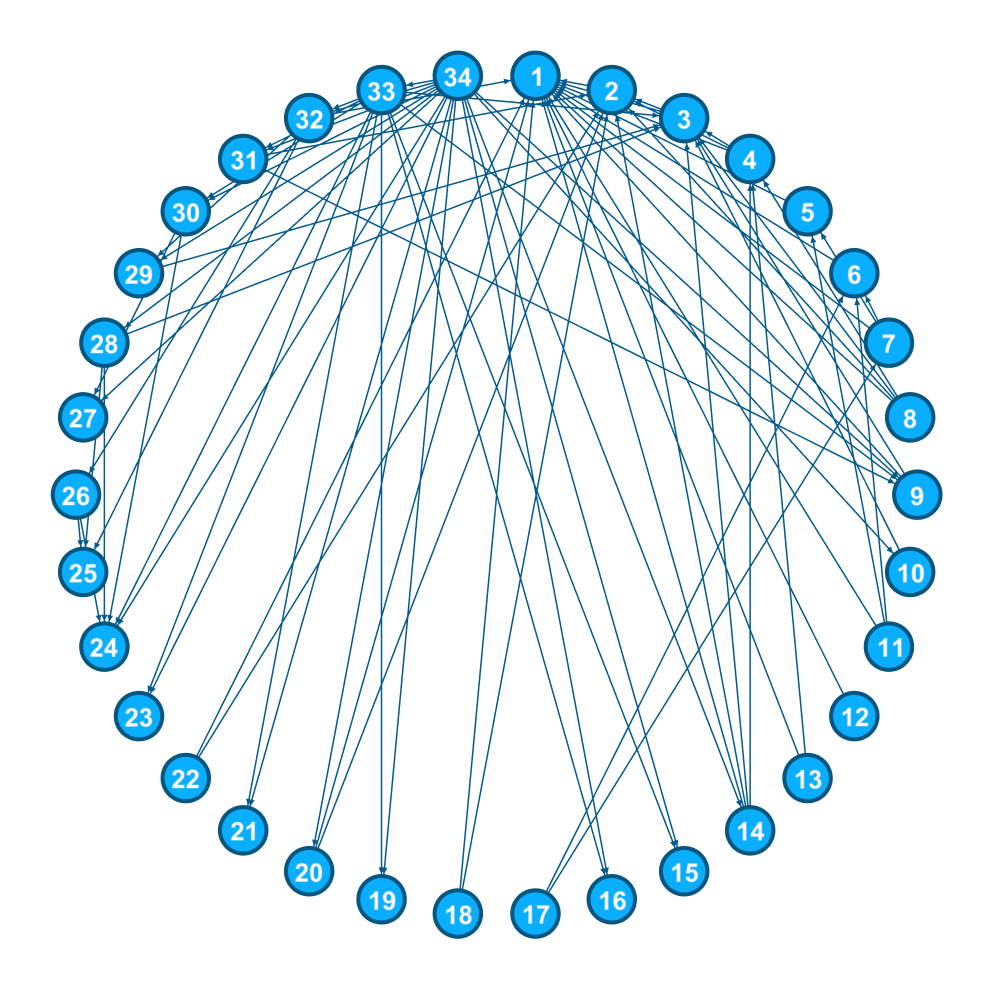
**Update:** For a proper initialization of the centroids at the start of the k-means algorithm, we implement the [improved k-means++ seeding procedure](https://datasciencelab.wordpress.com/2014/01/15/improved-seeding-for-clustering-with-k-means/).

# K-Means & Other Clustering Algorithms: A Quick Intro with Python

Clustering is the grouping of objects together so that objects belonging in the same group (cluster) are more similar to each other than those in other groups (clusters). In this intro cluster analysis tutorial, we'll check out a few algorithms in Python so you can get a basic understanding of the fundamentals of clustering on a real dataset.

### The Dataset

For the clustering problem, we will use the famous Zachary’s Karate Club dataset. The story behind the data set is quite simple: There was a Karate Club that had an administrator “John A” and an instructor “Mr. Hi” (both pseudonyms). Then a conflict arose between them, causing the students (Nodes) to split into two groups. One that followed John and one that followed Mr. Hi.

Visualization of social relationships between members of the Karate Club

### Getting Started with Clustering in Python

But enough with the introductory talk, let’s get to main reason you are here, the code itself. First of all, you need to install both [scikit-learn](http://scikit-learn.org/) and [networkx](https://networkx.github.io/) libraries to complete this tutorial. If you don’t know how, the links above should help you. Also, feel free to follow along by grabbing the source code for this tutorial over on [Github](https://github.com/LearnDataSci/blog-post-resources/blob/master/Karate%20Club%20Clustering/Classifiers.py).

Usually, the datasets that we want to examine are available in text form (JSON, Excel, simple txt file, etc.) but in our case, [networkx](https://networkx.github.io/) provide it for us. Also, to compare our algorithms, we want the truth about the members (who followed whom) which unfortunately is not provided. But with these two lines of code, you will be able to load the data and store the truth (from now on we will refer it as ground truth):

# Load and Store both data and groundtruth of Zachary's Karate Club

G = nx.karate\_club\_graph()

groundTruth = [0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,0,1,0,1,0,1,1,1,1,1,1,1,1,1,1,1,1]

The final step of the data preprocessing, is to transform the graph into a matrix (desirable input for our algorithms). This is also quite simple:

**def** graphToEdgeMatrix(G):

# Initialize Edge Matrix

edgeMat = [[0 **for** x **in** range(len(G))] **for** y **in** range(len(G))]

# For loop to set 0 or 1 ( diagonal elements are set to 1)

**for** node **in** G:

tempNeighList = G.neighbors(node)

**for** neighbor **in** tempNeighList:

edgeMat[node][neighbor] = 1

edgeMat[node][node] = 1

**return** edgeMat

Before we get going with the Clustering Techniques, I would like you to get a visualization on our data. So, let’s compile a simple function to do that:

**def** drawCommunities(G, partition, pos):

# G is graph in networkx form

# Partition is a dict containing info on clusters

# Pos is base on networkx spring layout (nx.spring\_layout(G))

# For separating communities colors

dictList = defaultdict(list)

nodelist = []

**for** node, com **in** partition.items():

dictList[com].append(node)

# Get size of Communities

size = len(set(partition.values()))

# For loop to assign communities colors

**for** i **in** range(size):

amplifier = i % 3

multi = (i / 3) \* 0.3

red = green = blue = 0

**if** amplifier == 0:

red = 0.1 + multi

**elif** amplifier == 1:

green = 0.1 + multi

**else**:

blue = 0.1 + multi

# Draw Nodes

nx.draw\_networkx\_nodes(G, pos,

nodelist=dictList[i],

node\_color=[0.0 + red, 0.0 + green, 0.0 + blue],

node\_size=500,

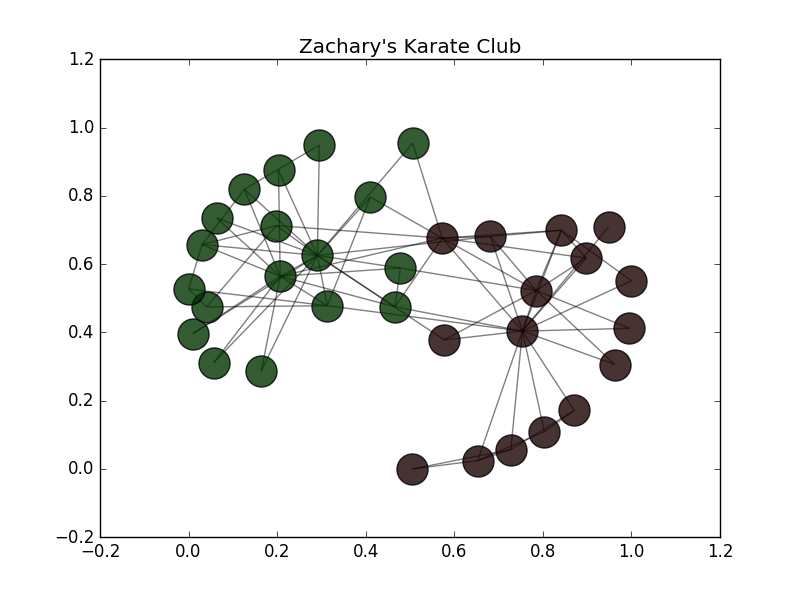
alpha=0.8)

# Draw edges and final plot

plt.title("Zachary's Karate Club")

nx.draw\_networkx\_edges(G, pos, alpha=0.5)

What that function does is to simply extract the number of clusters that are in our result and then assign a different color to each of them (up to 10 for the given time is fine) before plotting them.



### Clustering Algorithms

Some clustering algorithms will cluster your data quite nicely and others will end up failing to do so. That is one of the main reasons why clustering is such a difficult problem. But don’t worry, we won’t let you drown in an ocean of choices. We'll go through a few algorithms that are known to perform very well.

#### **K-Means Clustering**

Step Restart

N (the number of node):

K (the number of cluster):

New

Source: [github.com/nitoyon/tech.nitoyon.com](https://github.com/nitoyon/tech.nitoyon.com)

K-means is considered by many the gold standard when it comes to clustering due to its simplicity and performance, and it's the first one we'll try out. When you have no idea at all what algorithm to use, K-means is usually the first choice. Bear in mind that K-means might under-perform sometimes due to its concept: spherical clusters that are separable in a way so that the mean value converges towards the cluster center. To simply construct and train a K-means model, use the follow lines:

# K-means Clustering Model

kmeans = cluster.KMeans(n\_clusters=kClusters, n\_init=200)

kmeans.fit(edgeMat)

# Transform our data to list form and store them in results list

results.append(list(kmeans.labels\_))

#### **Agglomerative Clustering**

The main idea behind agglomerative clustering is that each node starts in its own cluster, and recursively merges with the pair of clusters that minimally increases a given linkage distance. The main advantage of agglomerative clustering (and hierarchical clustering in general) is that you don’t need to specify the number of clusters. That of course, comes with a price: performance. But, in scikit’s implementation, you can specify the number of clusters to assist the algorithm’s performance. To create and train an agglomerative model use the following code:

# Agglomerative Clustering Model

agglomerative = cluster.AgglomerativeClustering(n\_clusters=kClusters, linkage="ward")

agglomerative.fit(edgeMat)

# Transform our data to list form and store them in results list

results.append(list(agglomerative.labels\_))

#### **Spectral**

The Spectral clustering technique applies clustering to a projection of the normalized Laplacian. When it comes to image clustering, spectral clustering works quite well. See the next few lines of Python for all the magic:

# Spectral Clustering Model

spectral = cluster.SpectralClustering(n\_clusters=kClusters, affinity="precomputed", n\_init= 200)

spectral.fit(edgeMat)

# Transform our data to list form and store them in results list

results.append(list(spectral.labels\_))

#### **Affinity Propagation**

Well this one is a bit different. Unlike the previous algorithms, you can see AF does not require the number of clusters to be determined before running the algorithm. AF, performs really well on several computer vision and biology problems, such as clustering pictures of human faces and identifying regulated transcripts:

# Affinity Propagation Clustering Model

affinity = cluster.affinity\_propagation(S=edgeMat, max\_iter=200, damping=0.6)

# Transform our data to list form and store them in results list

results.append(list(affinity[1]))

### Metrics & Plotting

Well, it is time to choose which algorithm is more suitable for our data. A simple visualization of the result might work on small datasets, but imagine a graph with one thousand, or even ten thousand, nodes. That would be slightly chaotic for the human eye. So, let me show how to calculate the Adjusted Rand Score (ARS) and the Normalized Mutual Information (NMI):

# Append the results into lists

**for** x **in** results:

nmiResults.append(normalized\_mutual\_info\_score(groundTruth, x))

arsResults.append(adjusted\_rand\_score(groundTruth, x))

If you're unfamiliar with these metrics, here's a quick explanation:

#### **Normalized Mutual Information (NMI)**

Mutual Information of two random variables is a measure of the mutual dependence between the two variables. Normalized Mutual Information is a normalization of the Mutual Information (MI) score to scale the results between 0 (no mutual information) and 1 (perfect correlation). In other words, 0 means dissimilar and 1 means perfect match.

#### **Adjusted Rand Score (ARS)**

Adjusted Rand Score on the other hand, computes a similarity measure between two clusters by considering all pairs of samples and counting pairs that are assigned in the same or different clusters in the predicted and true clusters. If that's a little weird to think about, have in mind that, for now, 0 is the lowest similarity and 1 is the highest.

So, to get a combination of these metrics (the NMI and ARS), we simply calculate the average value of their sum. And remember, the higher the number, the better the result.

Below, I have plotted the score evaluation so we can get a better understanding of our results. We could plot them in many ways, as points, as a straight line, but I think a bar chart is the better choice for our case. To do so, just use the following code:

# Code for plotting results

# Average of NMI and ARS

y = [sum(x) / 2 **for** x **in** zip(nmiResults, arsResults)]

xlabels = ['Spectral', 'Agglomerative', 'Kmeans', 'Affinity Propagation']

fig = plt.figure()

ax = fig.add\_subplot(111)

# Set parameters for plotting

ind = np.arange(len(y))

width = 0.35

# Create barchart and set the axis limits and titles

ax.bar(ind, y, width,color='blue', error\_kw=dict(elinewidth=2, ecolor='red'))

ax.set\_xlim(-width, len(ind)+width)

ax.set\_ylim(0,2)

ax.set\_ylabel('Average Score (NMI, ARS)')

ax.set\_title('Score Evaluation')

# Add the xlabels to the chart

ax.set\_xticks(ind + width / 2)

xtickNames = ax.set\_xticklabels(xlabels)

plt.setp(xtickNames, fontsize=12)

# Add the actual value on top of each chart

**for** i, v **in** enumerate(y):

ax.text( i, v, str(round(v, 2)), color='blue', fontweight='bold')

# Show the final plot

plt.show()

As you can see in the chart below, K-means and Agglomerative clustering have the best results for our dataset (best possible outcome). That of course, does not mean that Spectral and AF are low-performing algorithms, just that the did not fit in our data.

