**Introduction to K-Means++**

The K-Means clustering algorithm is more than half a century old, but it is not falling out of fashion; it is still the most popular clustering algorithm for Machine Learning.

However, there can be some problems with its Step 1: "Place k random centroids for the initial clusters", also known as the *seeding* stage. The placement of the initial centroids can sometimes result in:

* Slow convergence
* Poor clustering

In this lesson, we will go over another version of the algorithm, known as **K-Means++ algorithm**. K-Means++ is an updated version of K-Means that aims to avoid the convergence and clustering problems by placing the initial centroids in a more effective way.

Instructions

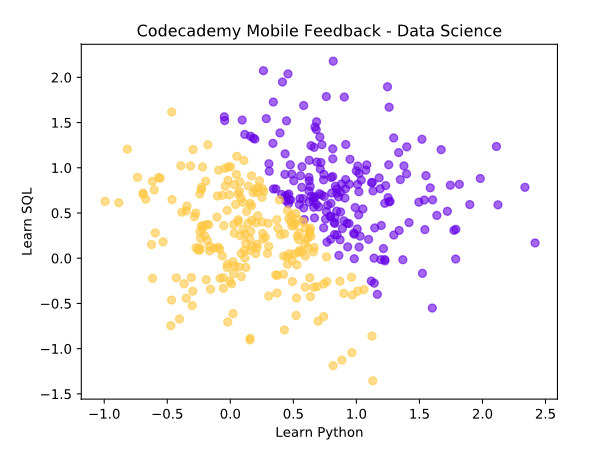
**1.**

To demonstrate slow convergence, run the program in **script.py**.

This program uses the standard K-Means algorithm to group Codecademy learners into two groups. It also runs K-Means 100 times and records the runtimes.

* What is K-means' average runtime?
* What is its worst runtime?

Record these numbers in your notes. Later on in the lesson, we are going to do this again, but for K-Means++.

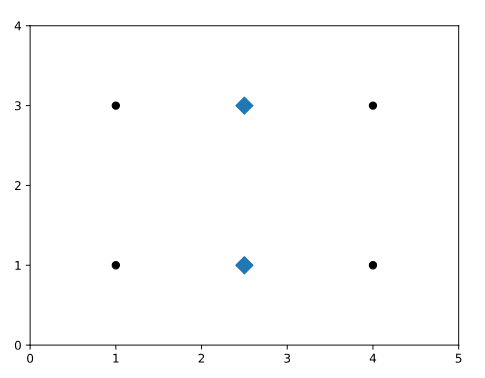


import codecademylib3\_seaborn  
import matplotlib.pyplot as plt  
import numpy as np  
import pandas as pd  
from sklearn.cluster import KMeans  
  
import random  
import timeit  
  
mu = 1  
std = 0.5  
mu2 = 4.188  
  
np.random.seed(100)  
  
xs = np.append(np.append(np.append(np.random.normal(0.25, std, 100), np.random.normal(0.75, std, 100)),  
 np.random.normal(0.25, std, 100)), np.random.normal(0.75, std, 100))  
  
ys = np.append(np.append(np.append(np.random.normal(0.25, std, 100), np.random.normal(0.25, std, 100)),  
 np.random.normal(0.75, std, 100)), np.random.normal(0.75, std, 100))  
  
values = list(zip(xs, ys))  
  
total = 0  
worst = -float("Inf")  
for i in range(100):  
  
 start = timeit.default\_timer()  
  
 model = KMeans(init='random', n\_clusters=2)  
  
 results = model.fit\_predict(values)  
  
 stop = timeit.default\_timer()  
  
 total += stop - start  
 if (stop - start > worst):  
 worst = stop - start  
  
colors = ['#6400e4', '#ffc740']  
  
for i in range(2):  
 points = np.array([values[j] for j in range(len(values)) if results[j] == i])  
 plt.scatter(points[:, 0], points[:, 1], c=colors[i], alpha=0.6)  
  
plt.title('Codecademy Mobile Feedback - Data Science')  
  
plt.xlabel('Learn Python')  
plt.ylabel('Learn SQL')  
  
plt.show()  
  
average = total / 100  
  
print("Average Runtime: ", end='')  
print(average) # Average Runtime: 0.041764466799795626  
print("Worst Runtime: ", end="")  
print(worst) # Worst Runtime: 0.13035153411328793

**Poor Clustering**

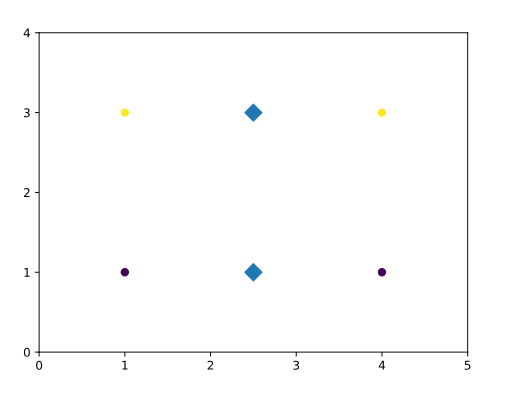
We demonstrated the slow convergence possibility of K-means. Let's take a look at the second possibility: poor clustering.

Suppose we have four data samples that form a rectangle whose width is greater than its height:



If k = 2 and the two initial cluster centers lie at the midpoints of the top and bottom line segments of the rectangle formed by the four data points, the K-means algorithm converges immediately, without moving these cluster centers.

Consequently, the two top data points are clustered together and the two bottom data points are clustered together.



This is a *suboptimal clustering* because the width of the rectangle is greater than its height. The optimal clusters would be the two left points as one cluster and the two right points as one cluster.

Now, consider stretching the rectangle horizontally to an arbitrary width. The original K-means algorithm will continue to cluster the points suboptimally. Oh no!

**Note:** Even though suboptimal clustering *can* happen, it is still rare.

Instructions

**1.**

Suppose we have four data samples with these values:

* (1, 1)
* (1, 3)
* (4, 1)
* (4, 3)

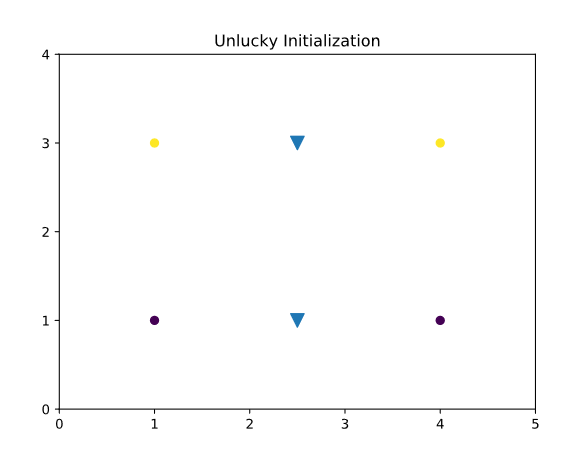
And suppose we perform K-means on this data where the k is 2 and randomized 2 initial centroids at positions:

* (2.5, 1)
* (2.5, 3)

What do you think the result clusters would look like?

Run **script.py** to find out the answer.

import codecademylib3\_seaborn  
import matplotlib.pyplot as plt  
import numpy as np  
from sklearn import datasets  
from copy import deepcopy  
from sklearn.cluster import KMeans   
  
x = [1, 1, 4, 4]  
y = [1, 3, 1, 3]  
  
values = np.array(list(zip(x, y)))  
  
centroids\_x = [2.5, 2.5]  
centroids\_y = [1, 3]  
  
centroids = np.array(list(zip(centroids\_x, centroids\_y)))  
  
model = KMeans(init=centroids, n\_clusters=2)  
  
# Initial centroids  
# plt.scatter(centroids[:, 0], centroids[:, 1], marker='D', s=100)  
  
results = model.fit\_predict(values)  
  
plt.scatter(x, y, c=results, alpha=1)  
  
# Cluster centers  
plt.scatter(model.cluster\_centers\_[:, 0], model.cluster\_centers\_[:, 1], marker='v', s=100)  
  
ax = plt.subplot()  
ax.set\_xticks([0, 1, 2, 3, 4, 5])  
ax.set\_yticks([0, 1, 2, 3, 4])  
  
plt.title('Unlucky Initialization')  
plt.show()  
  
# print(centroids)  
  
print('(1,1):')  
print(results[0])  
print(results[1])  
print(results[2])  
print(results[3])  
  
print(model.cluster\_centers\_)



**What is K-Means++?**

To recap, the Step 1 of the K-Means algorithm is "Place k random centroids for the initial clusters".

The K-Means++ algorithm replaces Step 1 of the K-Means algorithm and adds the following:

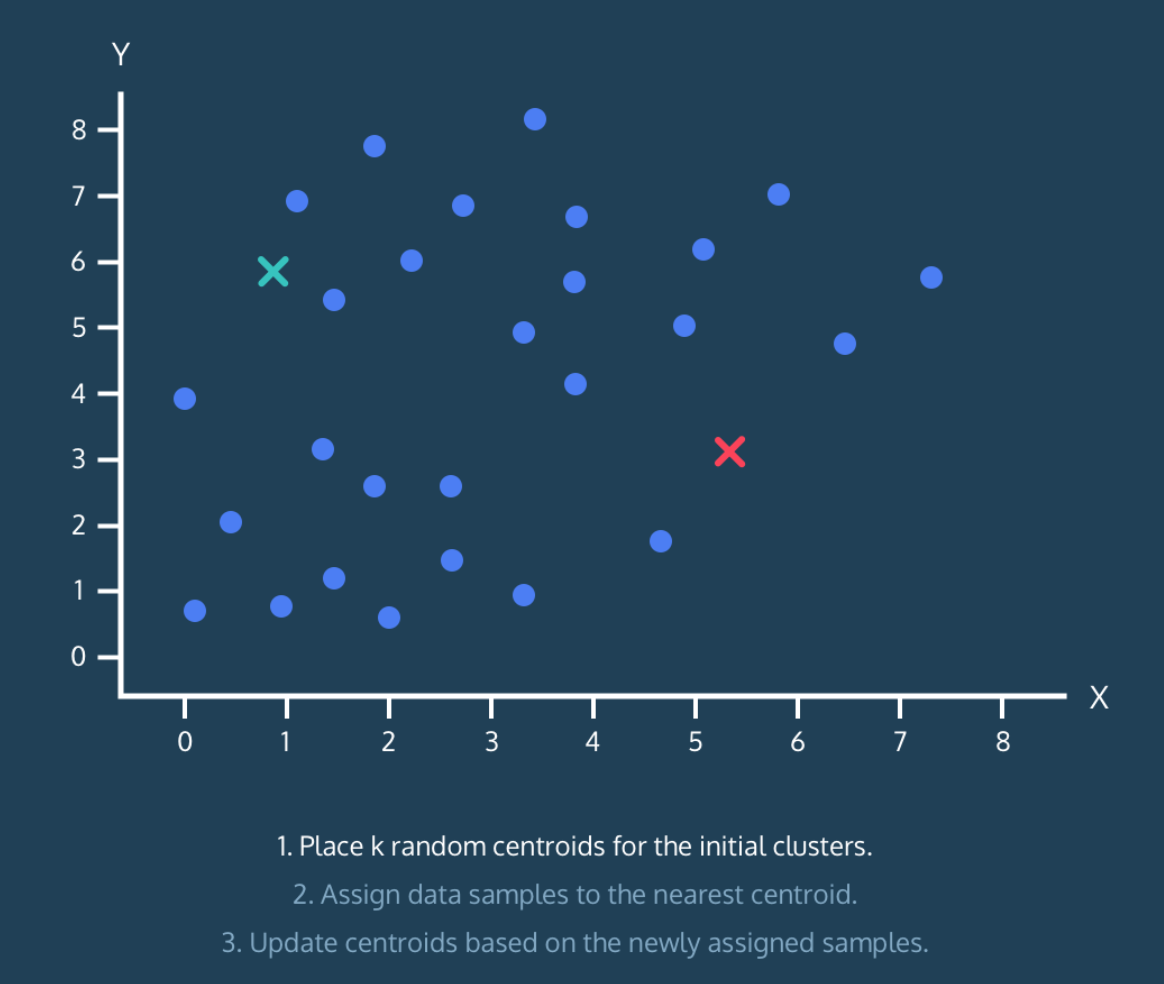
* **1.1** The first centroid is randomly picked from the data samples, and put into set centroids.
* **1.2** For each non-centroid sample, compute its minimum distance to any of centroids in set centroids.
* **1.3** Assign probabilities of selecting each non-centroid sample to add to set centroids.
* **1.4** Add to centroids one randomly picked non-centroid data sample based on the above probabilities.

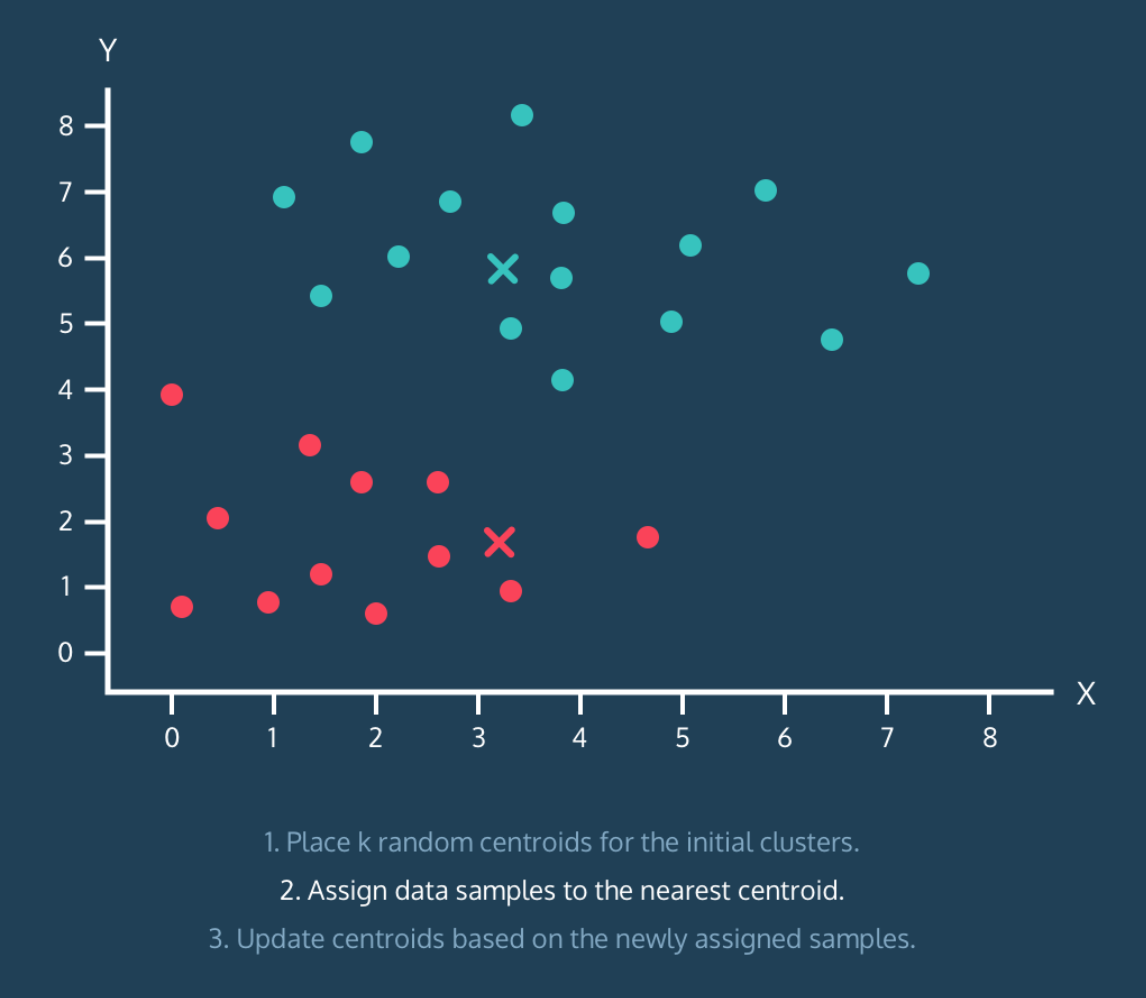
**Repeat 1.2 - 1.4** until k centroids are chosen.

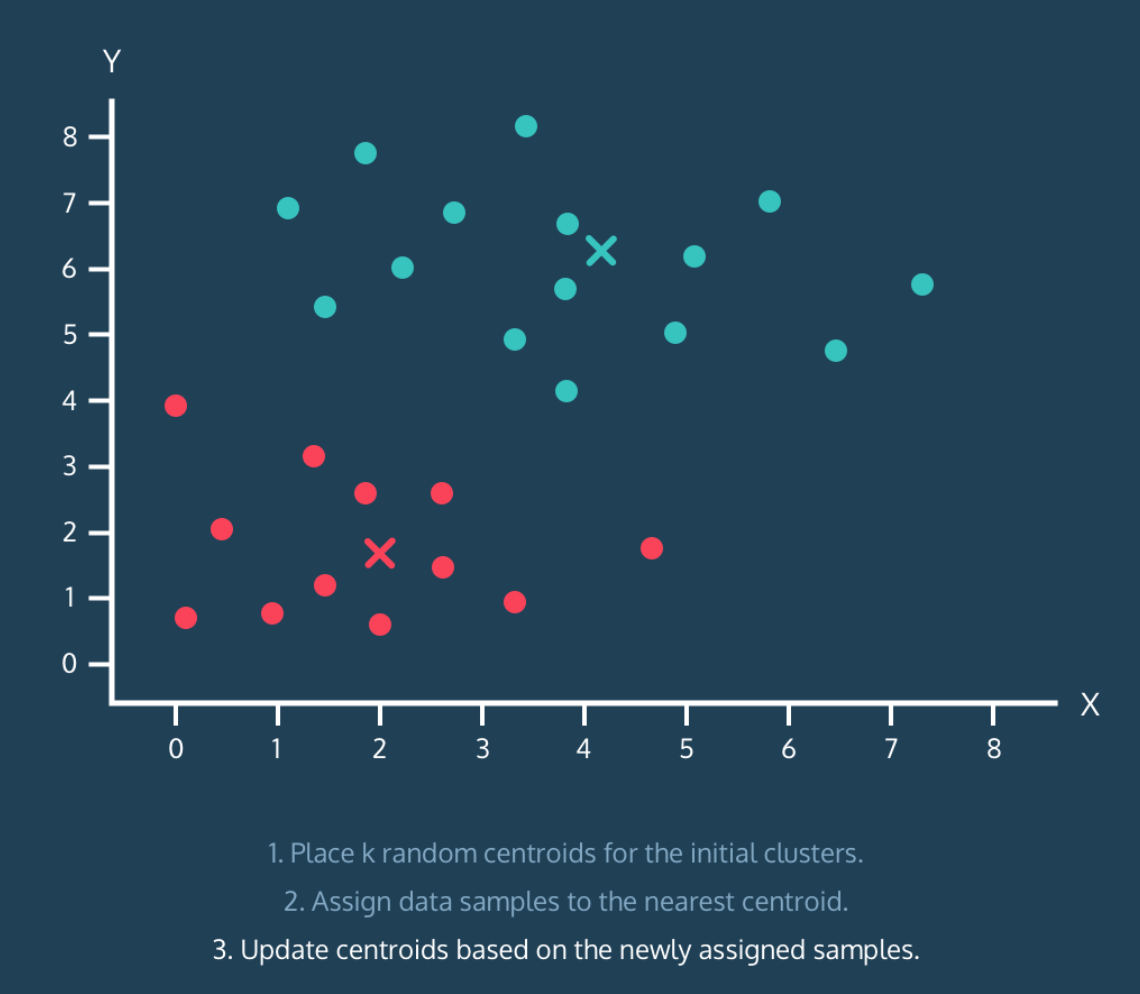
Take a look at the animation. It showcases the steps of K-Means++ that replaces the Step 1 of K-Means.

Instructions

Now let's see how you can run K-Means++ using scikit-learn.







**K-Means++ using Scikit-Learn**

Using the [scikit-learn](http://scikit-learn.org/stable/) library and its cluster module , you can use the KMeans() method to build an original K-Means model that finds 6 clusters like so:

model = KMeans(n\_clusters=6, init='random')

The init parameter is used to specify the initialization and init='random' specifies that initial centroids are chosen as random (the original K-Means).

But how do you implement K-Means++?

There are two ways and they both require little change to the syntax:

**Option 1:** You can adjust the parameter to init='k-means++'.

test = KMeans(n\_clusters=6, init='k-means++')

**Option 2:** Simply drop the parameter.

test = KMeans(n\_clusters=6)

This is because that init=k-means++ is actually *default* in scikit-learn.

Instructions

**1.**

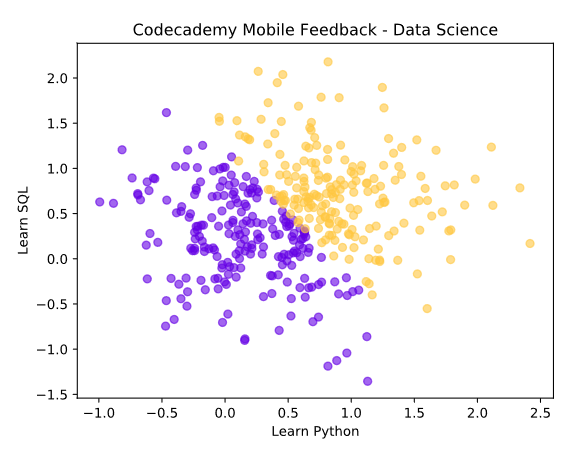
In **script.py**, change the init parameter so that KMeans() is running K-Means++ instead of K-Means.

* What is K-means++'s average runtime?
* What is its worst runtime?

Compare with those of K-Means:

*Which performed better?*

import codecademylib3\_seaborn  
import matplotlib.pyplot as plt  
import numpy as np  
import pandas as pd  
from sklearn.cluster import KMeans  
  
import random  
import timeit  
  
mu = 1  
std = 0.5  
mu2 = 4.188  
  
np.random.seed(100)  
  
xs = np.append(np.append(np.append(np.random.normal(0.25, std, 100), np.random.normal(0.75, std, 100)),  
 np.random.normal(0.25, std, 100)), np.random.normal(0.75, std, 100))  
  
ys = np.append(np.append(np.append(np.random.normal(0.25, std, 100), np.random.normal(0.25, std, 100)),  
 np.random.normal(0.75, std, 100)), np.random.normal(0.75, std, 100))  
  
values = list(zip(xs, ys))  
  
total = 0  
  
worst = 0  
  
for i in range(100):  
  
 start = timeit.default\_timer()  
  
 model = KMeans(init='k-means++', n\_clusters=2)  
  
 results = model.fit\_predict(values)  
  
 stop = timeit.default\_timer()  
  
 run = stop - start  
  
 total += run  
  
 if run > worst:  
 worst = run  
  
# plt.scatter(xs, ys, c=results, alpha=0.6)  
  
colors = ['#6400e4', '#ffc740']  
  
for i in range(2):  
 points = np.array([values[j] for j in range(len(values)) if results[j] == i])  
 plt.scatter(points[:, 0], points[:, 1], c=colors[i], alpha=0.6)  
  
plt.title('Codecademy Mobile Feedback - Data Science')  
  
plt.xlabel('Learn Python')  
plt.ylabel('Learn SQL')  
  
plt.show()  
  
average = total / 100  
  
print("Average Runtime: ", end='')  
print(average) # Average Runtime: 0.04415750313550234  
  
print("Worst Runtime: ", end='')  
print(worst) # Worst Runtime: 0.1533062569797039



~~K++~~

~~Average Runtime: 0.04415750313550234~~

~~Worst Runtime: 0.1533062569797039~~

~~K - Original~~

~~Average Runtime: 0.043659383468329906~~

~~Worst Runtime: 0.06372083723545074~~

**Review**

Congratulations, now your K-Means model is improved and ready to go!

For a review of this lesson: K-Means' Step 1: "Place k random centroids for the initial clusters" can sometimes result in:

* Slow convergence
* Poor clustering

K-Means++ improves K-Means by placing initial centroids more strategically. In result, it outperforms K-Means in both speed and accuracy.

You can implement K-Means++ in [scikit-learn](http://scikit-learn.org/stable/) library similar to how you implement K-Means.

The KMeans() function has an init parameter, which specifies the method for initialization:

* 'random'
* 'k-means++'

**Note:** scikit-learn's KMeans() uses 'k-means++' by default, but it is a good idea to be explicit!