k-means in Spark

(due date Monday December 4, 2023 4:00 pm EST)

We will implement k-means for k=4 with points in 2-dimensions only. I have provided comments that will provide guidance as to the implementation and left as much skeleton code as possible. Your implementation should use the Spark RDD interface and keep data in Spark RDDs whenever possible. If you are writing a for loop, you are doing it wrong.

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
from pyspark import SparkContext

sc = SparkContext("local", "kmeans2d",)
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, us
e setLogLevel(newLevel).
23/12/06 22:24:04 WARN NativeCodeLoader: Unable to load native-hadoo
p library for your platform... using builtin-java classes where appl
icable
```

In [2]:

```
# k-means helper functions

# assign each point to a cluster based on which centroid is closest
# centroids should be a np.array of shape (4,2), dtype=float32
def assign_class(point, centroids):

    mindist = np.finfo(np.float64).max
    for j in range(len(centroids)):
        distance = np.linalg.norm(point-centroids[j])
        if distance < mindist:
            mindist = distance
            assignclass = j
    return assignclass

# plot the data distribution.
#
# pstriples should be an RDD of type k,v = (int, [float32, float3
2])
# centroids is again np.array of shape (4,2), dtype=float32
def plot_clusters(ptstriples, centroids):

    # extract the points in each cluster
    lcluster0 = ptstriples.filter(lambda x: x[0] == 0).map(lambda
x: x[1])
    lcluster1 = ptstriples.filter(lambda x: x[0] == 1).map(lambda
x: x[1])
    lcluster2 = ptstriples.filter(lambda x: x[0] == 2).map(lambda
x: x[1])
    lcluster3 = ptstriples.filter(lambda x: x[0] == 3).map(lambda
x: x[1])

    # convert data to np.arrays
    cluster0 = np.array(lcluster0.collect())
    cluster1 = np.array(lcluster1.collect())
    cluster2 = np.array(lcluster2.collect())
    cluster3 = np.array(lcluster3.collect())

    # plot the cluster data differentiated by color
    plt.plot(cluster0[:,0], cluster0[:,1], 'b.', markersize=2)
    plt.plot(cluster1[:,0], cluster1[:,1], 'r.', markersize=2)
    plt.plot(cluster2[:,0], cluster2[:,1], 'm.', markersize=2)
    plt.plot(cluster3[:,0], cluster3[:,1], 'c.', markersize=2)

    # overlay the centroids
    plt.plot(centroids[:,0], centroids[:,1], 'ko', markersize=5)

    plt.axis('equal')
    plt.show()

# plot the intial data before there are labels
#
# centroids is again np.array of shape (4,2), dtype=float32
# points is an RDD
def showpoints(points, centroids):
    points = np.array(points.collect())
```

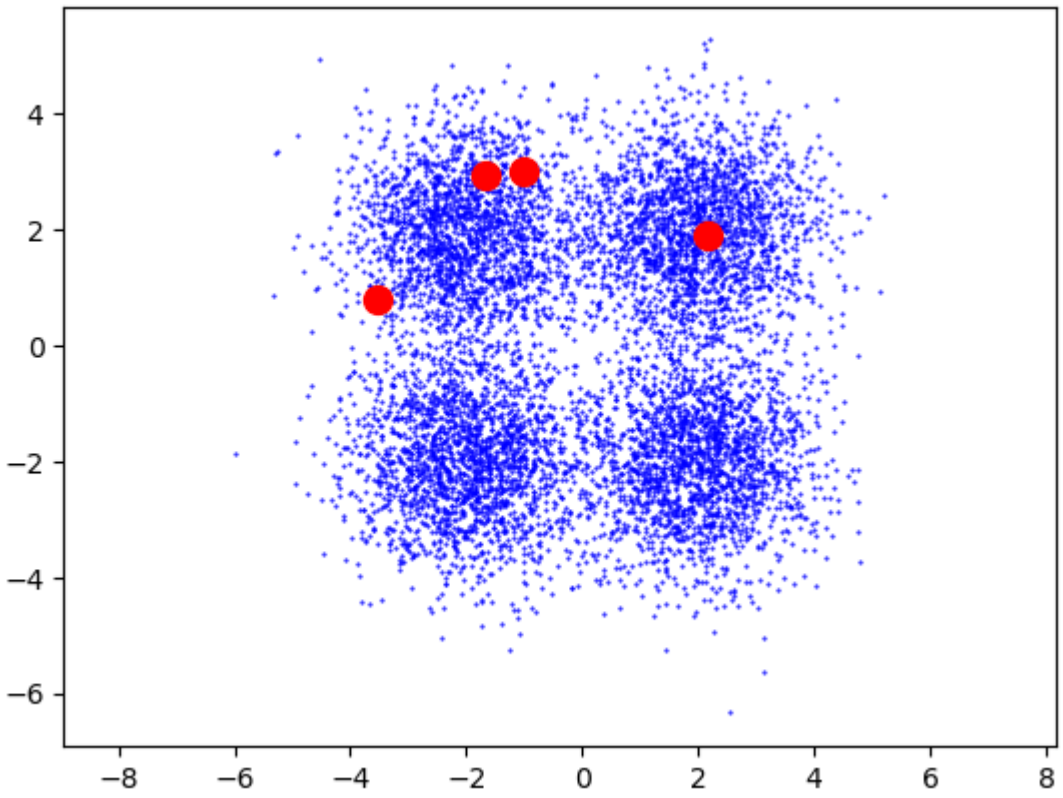
```
    plt.plot(points[:,0], points[:,1], 'b.', markersize=1)
    plt.plot(centroids[:,0], centroids[:,1], 'ro', markersize=10)
    plt.axis('equal')
    plt.show()
```

Generate Data

Create a k-means data set in this spark context. The default is to create 2000 points, 500 each from 4 distributions. You can change then classcount to create small dataset

In [3]:
```
# generate classcount points and permute for each spark partition
def gen2000 (i):

    # 2 data points in each class for a small dataset
    # classcount = 2

    # 500 data points in each class for a large dataset
    classcount = 500

    cov = [[1, 0], [0, 1]]  # diagonal covariance

    points1 = np.random.multivariate_normal([2,2], cov, classcount)
    points2 = np.random.multivariate_normal([2,-2], cov, classcoun
t)
    points3 = np.random.multivariate_normal([-2,2], cov, classcoun
t)
    points4 = np.random.multivariate_normal([-2,-2], cov, classcoun
t)

    # put all points together and permute
    pointsall = np.concatenate((points1, points2, points3, points
4), axis=0)
    pointsall = np.random.permutation(pointsall)

    return pointsall

# number of partitions in dataset
slices = 4

# make points and materialize to an RDD. Then collect.
# This prevents the from being randomly regenerated each iteration.
# This is an array, not an RDD, because we collect.
pointsar = sc.parallelize(range(slices), numSlices=slices).flatMap
(gen2000).collect()

# get the same points as an RDD everytime
points = sc.parallelize(pointsar)

# optionally persist the points to cache for reuse.
# points.persist()
```

In [4]:
```
# take a sample of k points as seeds (comment out the DEBUG line)
centroids = np.array(points.takeSample(False, 4))

# (DEBUG) or use these as an example when debugging
# centroids = np.array([[2.0,2.0],[2.0,-2.0],[-2.0,2.0],[-2.0,-2.
0]])

# keep a copy for rerunning
originalCentroids = centroids

showpoints(points, centroids)
```



In [5]:
```
# assign each point to a class using the assign_class function
# produces an RDD with type (int) with length equal to number of po
ints
clusters = points.map(lambda point: assign_class(point, centroids))
```

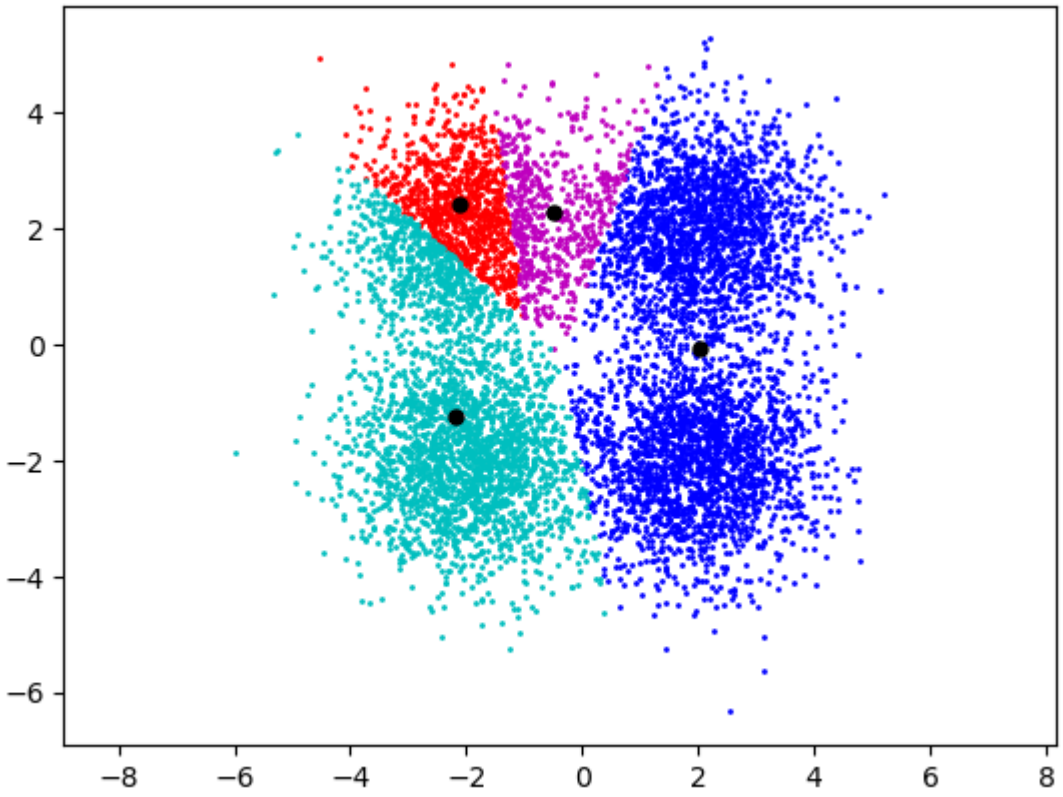
In [6]:
```
# build an RDD of type (int, [float, float]) that specifies the clu
ser and then the point coordinates
# this can be done efficiently with with `zip()` function
def transform_tuple(x):
    return x[0], list(x[1])

ptstriples = clusters.zip(points).map(transform_tuple)
```

Some hints for the next cell

1. use `groupByKey()` to collect data by cluster
2. at the end you are going to have to use the function `np.mean(array, axis=0)` on a iterator. Keep the data in spark RDDs until the last step.
3. it can be hard to materialize RDDs into arrays you need to either `np.array(RDD)` or `np.array(list(RDDiterable))`
4. I wrote a helper function, rather than using a lambda to help take the mean because it was more readable.
5. be careful with the ordering of your centroids. RDDs are not necessarily sorted by key.

In [7]:
```
# update the centroids to be the mean of each cluster of points

def calculate_mean(iterator):
    array = np.array(list(iterator))
    return np.mean(array, axis=0)

# Group points by cluster and calculate the mean for each cluster
cluster_means = ptstriples.groupByKey().mapValues(calculate_mean)

# Convert the RDD of cluster means to a numpy array
centroids = np.array(cluster_means.sortByKey().values().collect())
```

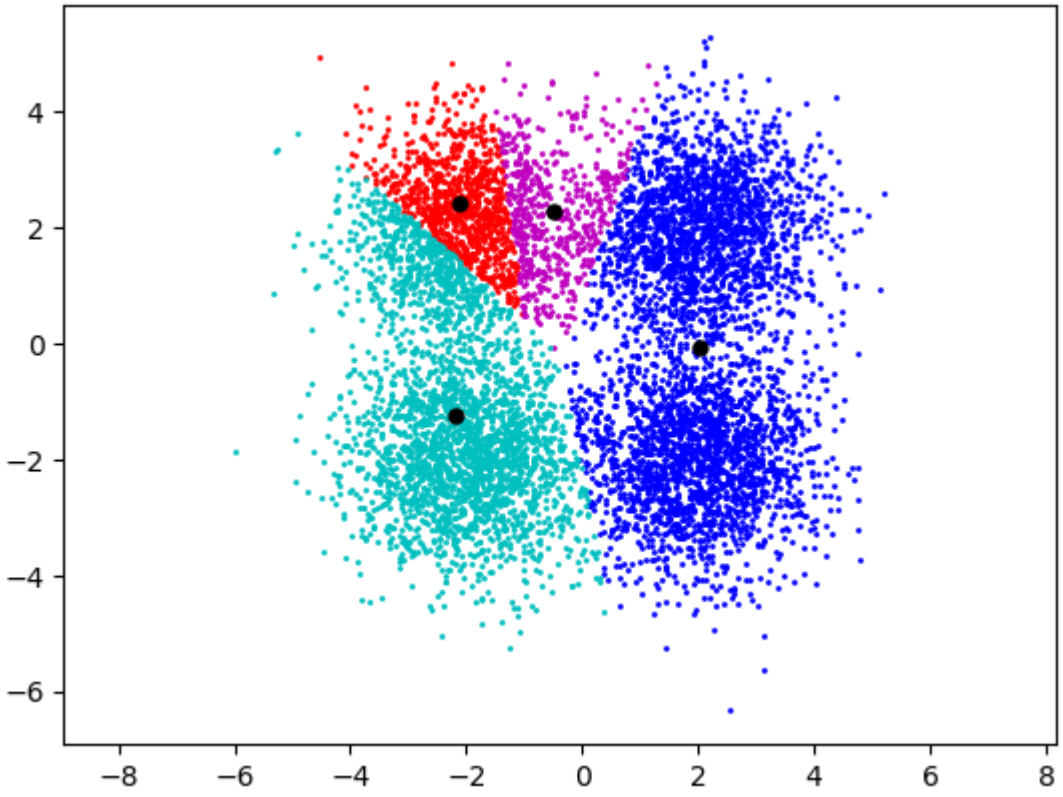
In [8]:
```
# look at the output of your intial clustering
plot_clusters(ptstriples, centroids)
```

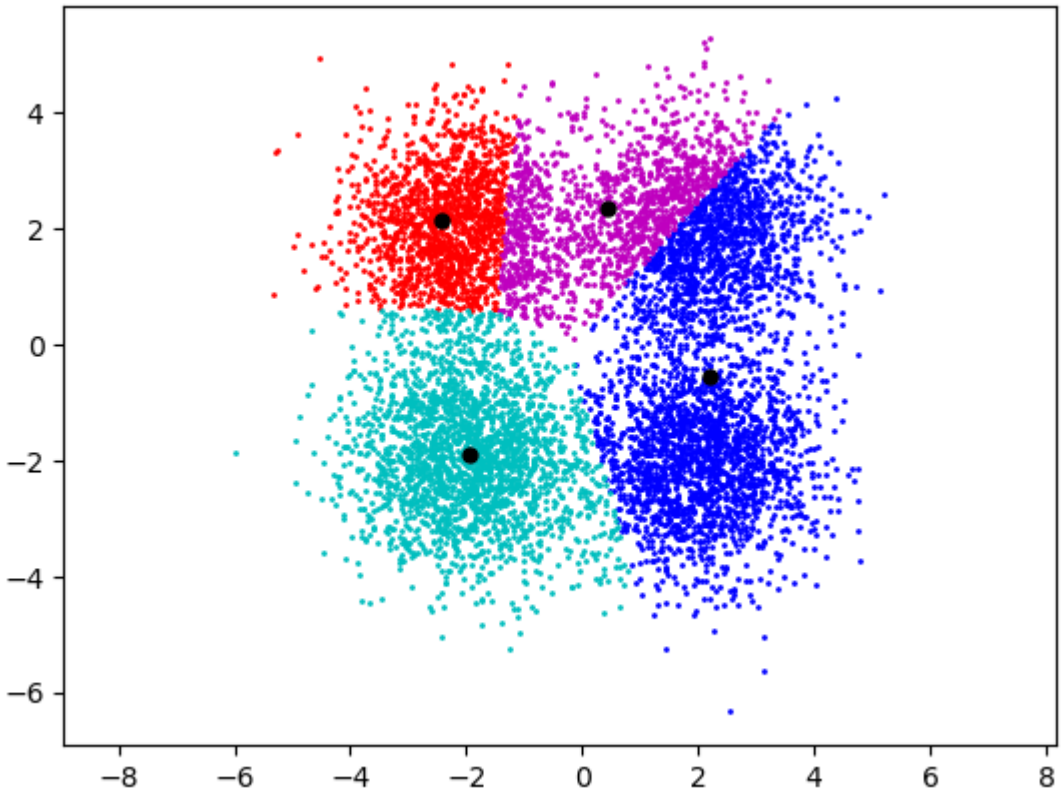


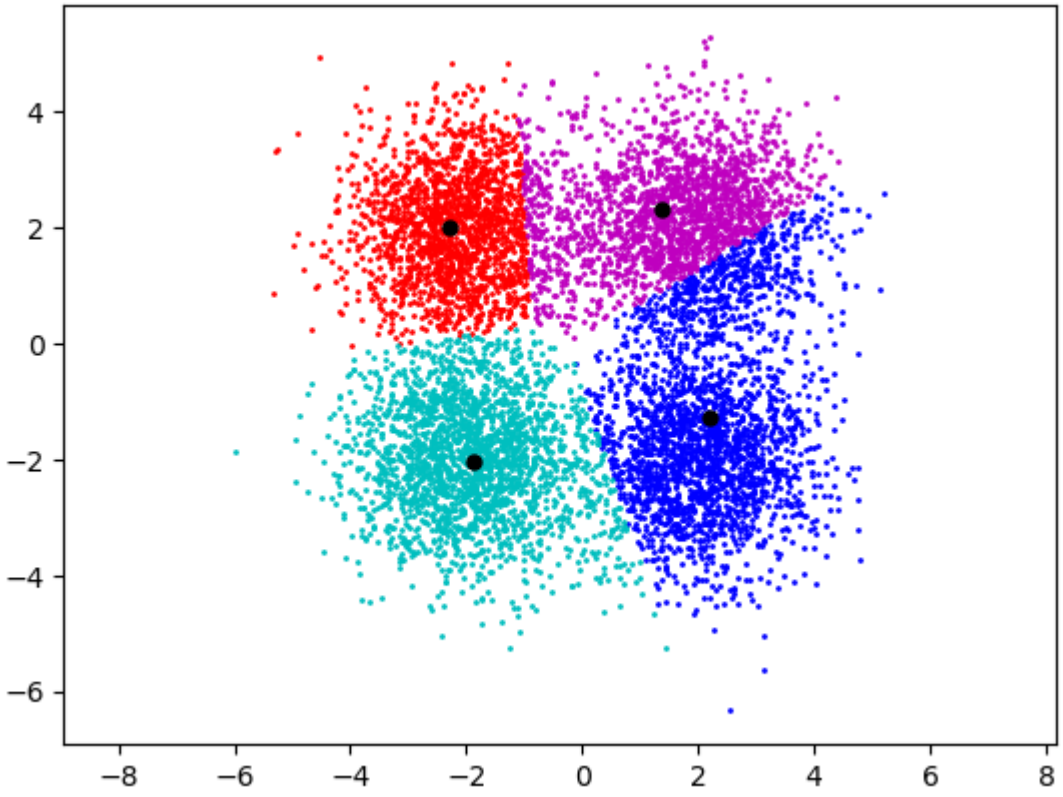
k-means is an iterative algorithm. You will see that the centroids progessively converge to the true means.

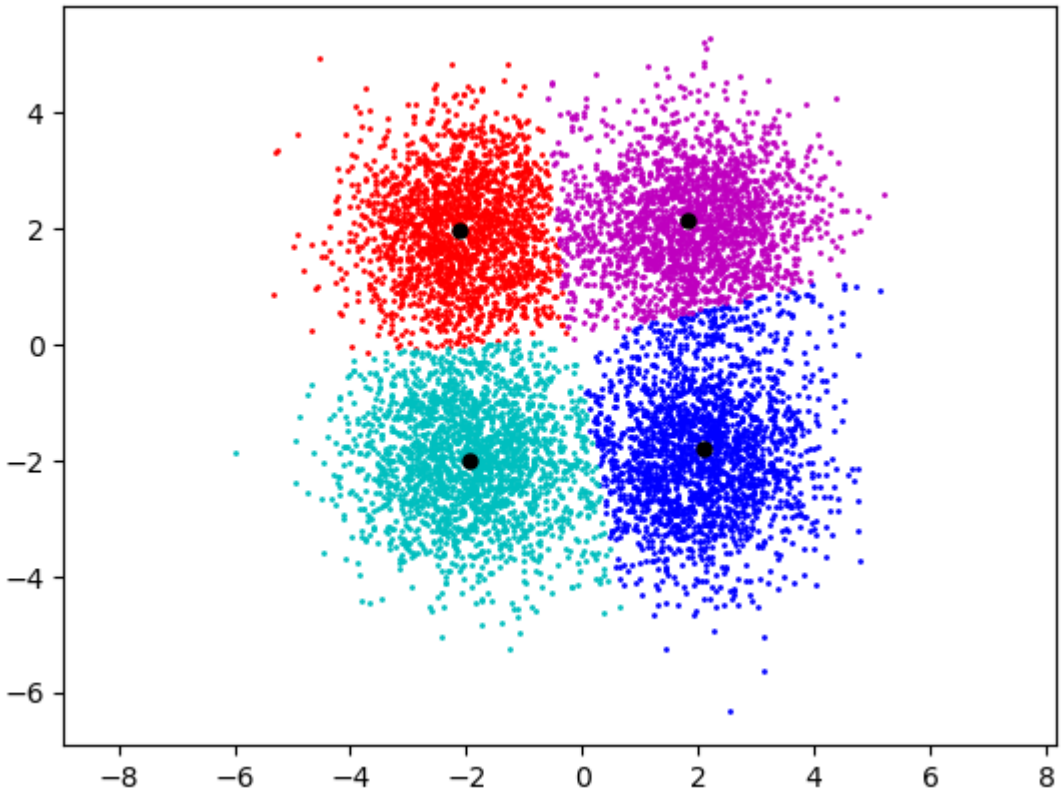
In [9]:

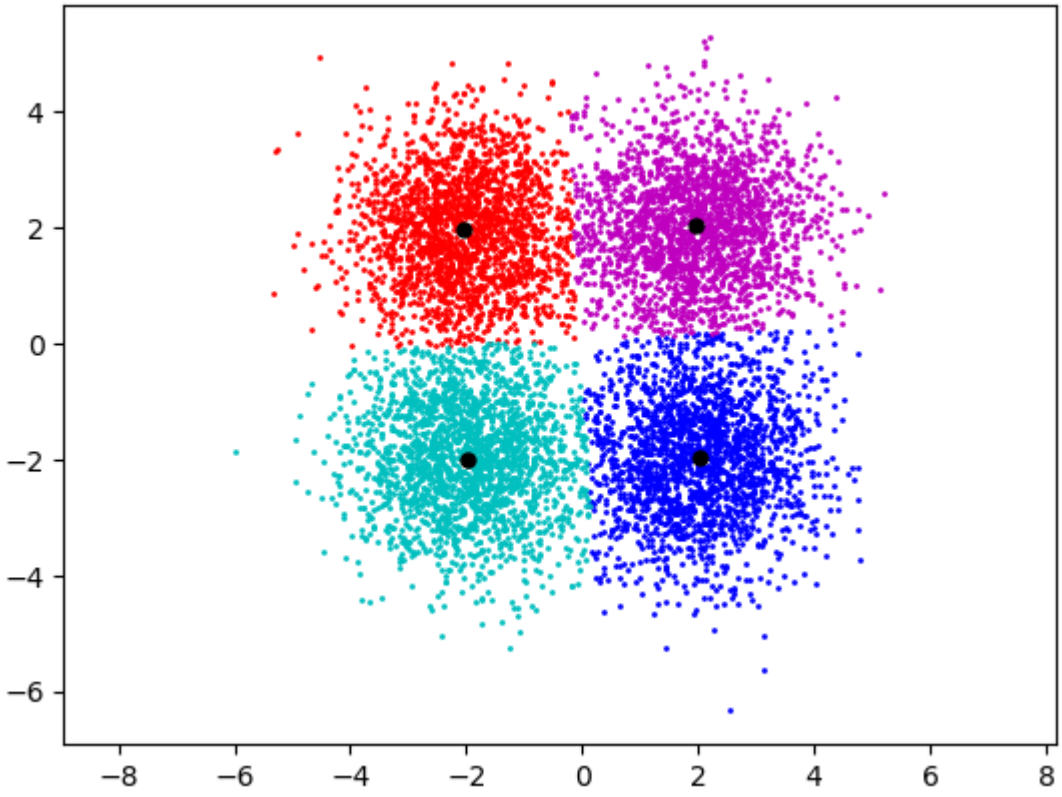
```
# %%timeit -n 1

iterations = 10
centroids = originalCentroids

for i in range(iterations):

    ### TODO
    # run the whole process over and over
    clusters = points.map(lambda point: assign_class(point, centroi
ds))
    ptstriples = clusters.zip(points).map(transform_tuple)
    cluster_means = ptstriples.groupByKey().mapValues(calculate_mea
n)
    centroids = np.array(cluster_means.sortByKey().values().collect
())

    # optionally plot the output (could be slow)
    plot_clusters(ptstriples, centroids)
```

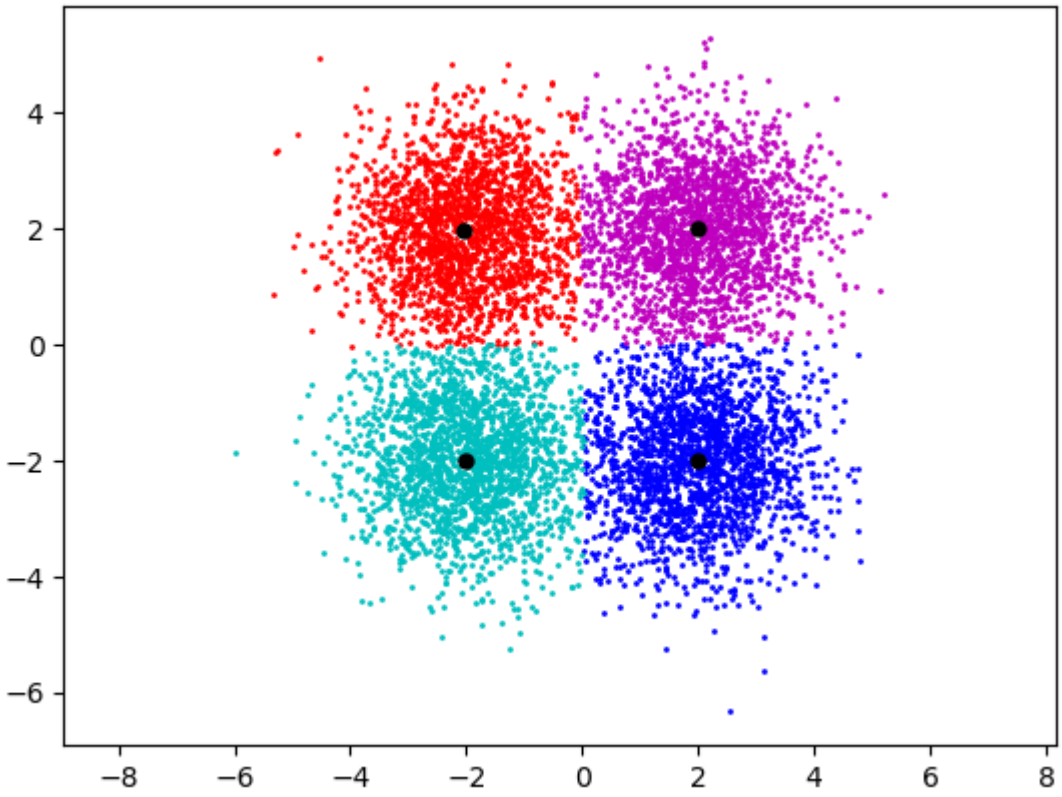

4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8

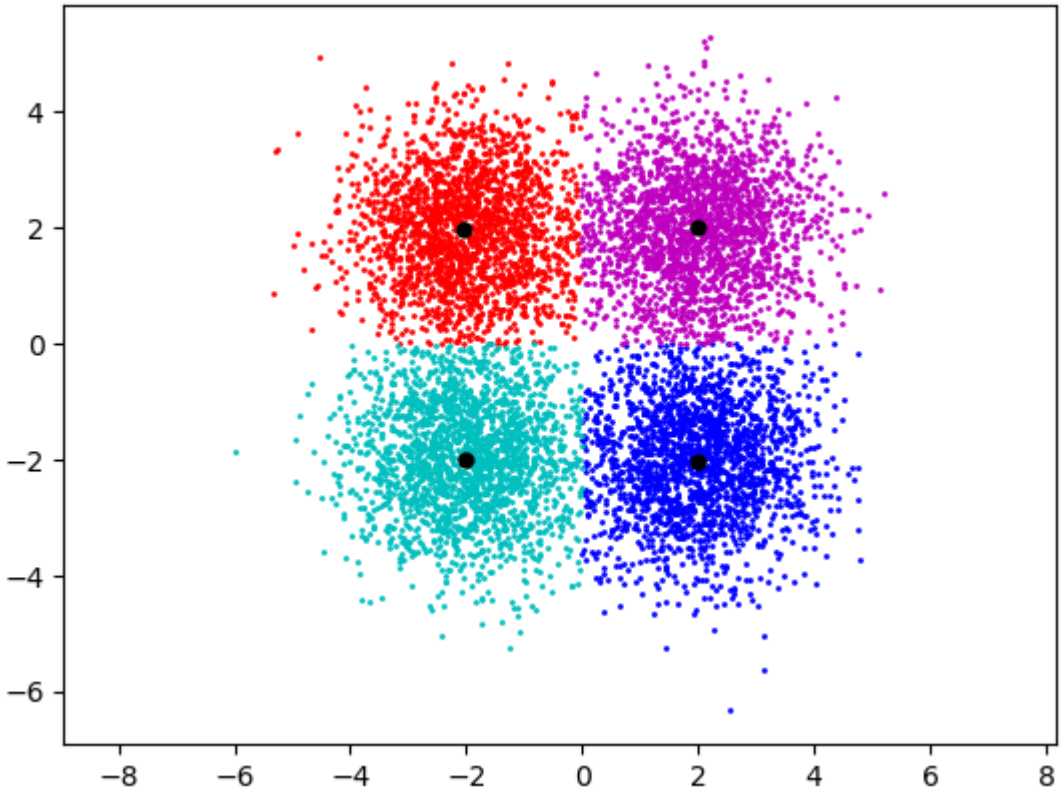

4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8

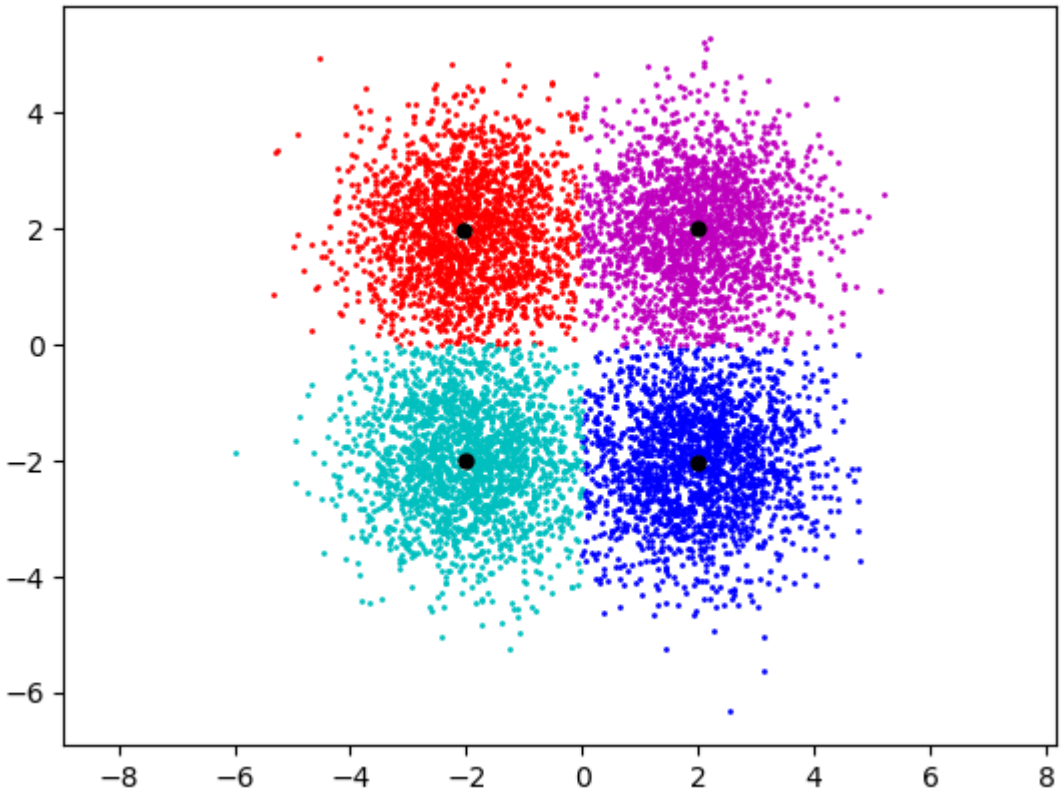

4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8

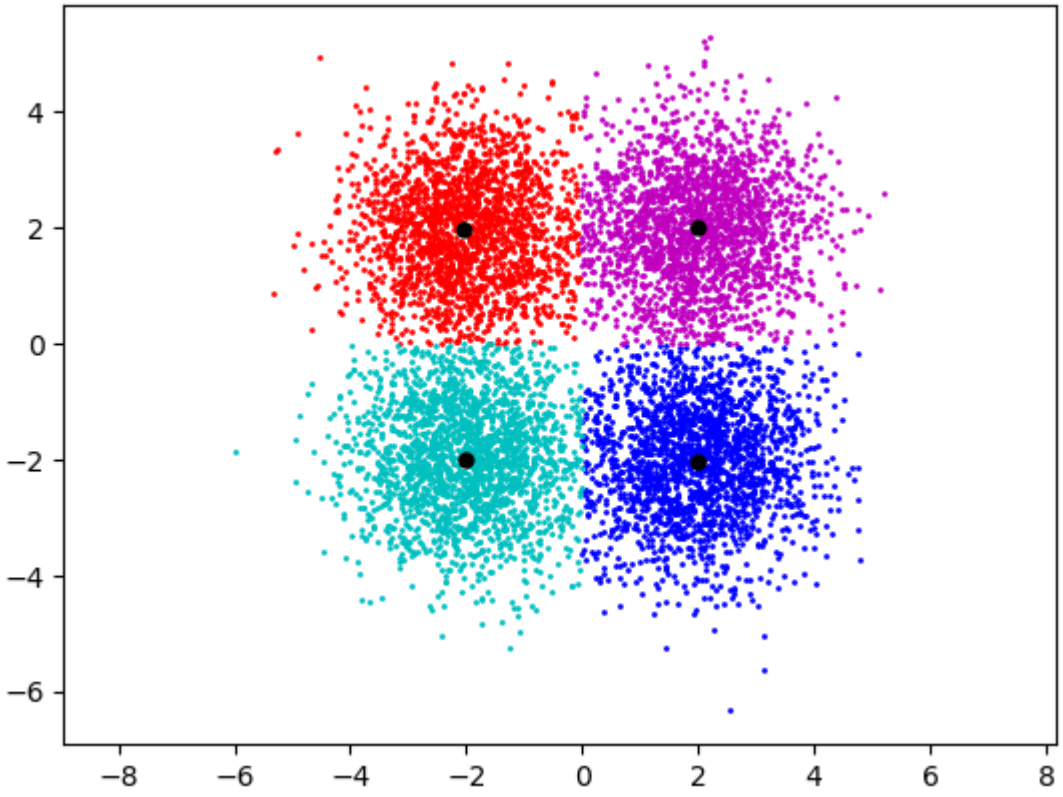

4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8

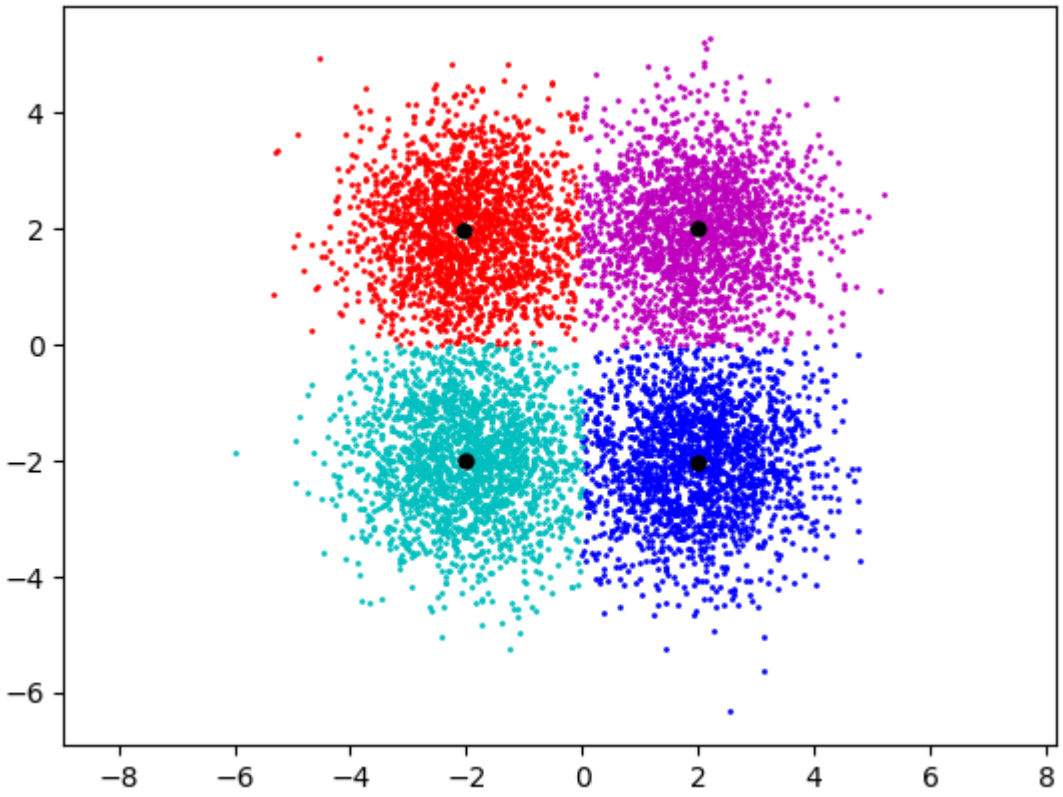

4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8


4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8


4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8


4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8


4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8


4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8

Alternate Implementation

Our first implementation used a `groupBy` to collect data by cluster. This has the disadvantage that it shuffles data and collects data by partition. We will do another implementation that moves no data outside of partitions. This will use the `filter` pattern that is implemented in the `plot_clusters` function.

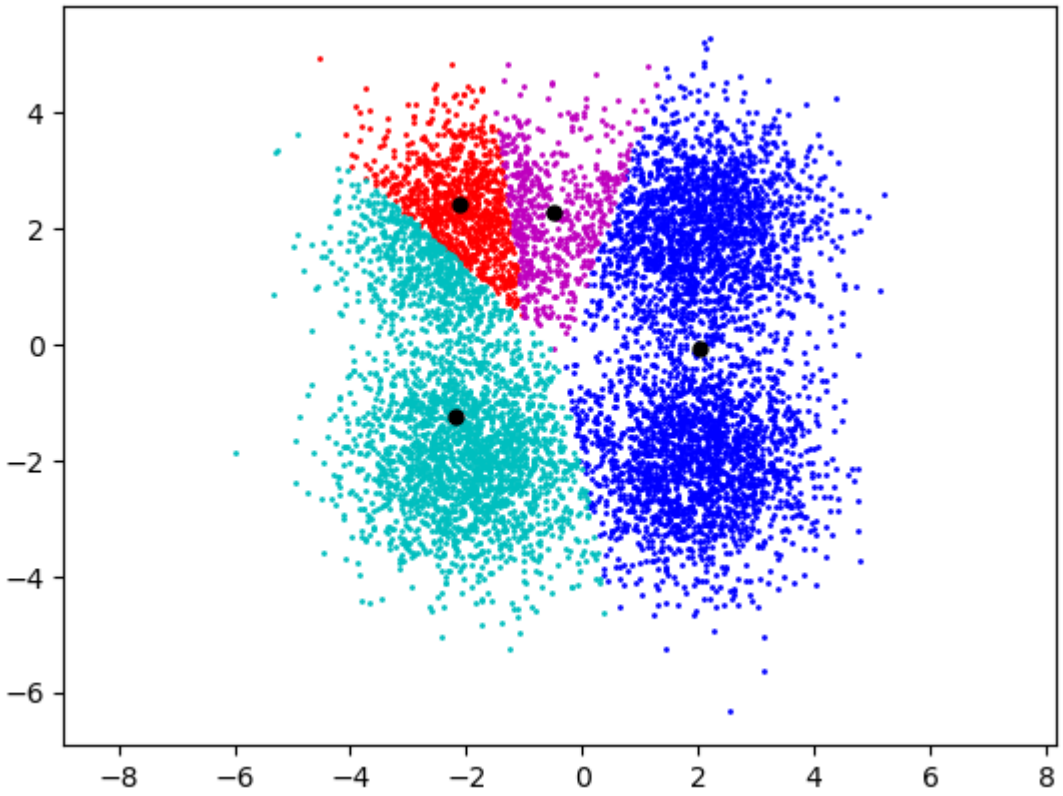
Your program should filter all data within a partition and then aggregate data within each partition. I have given you the `part_sums` helper to aggregate within each partition and the `sums_2_means` helper function to convert the sums into means.

Leave all the `persist()` operations and also all the `%%timeit` directives commented out at this point.

In [10]:

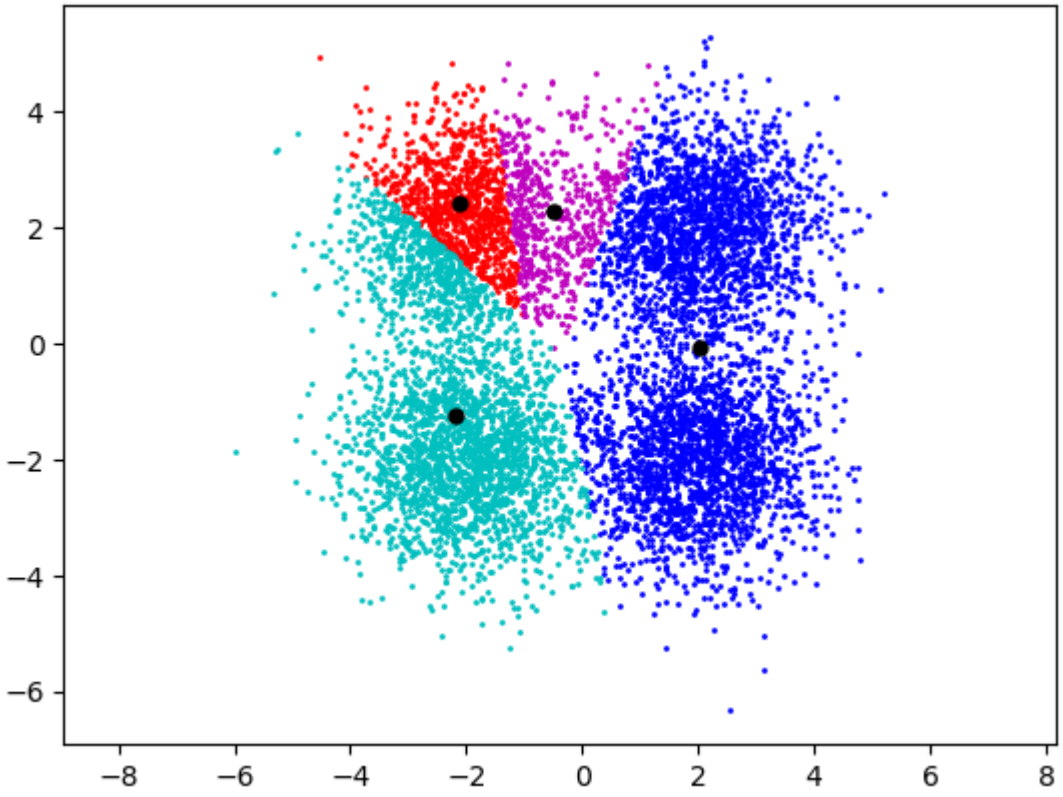
```
# helper function that returns the sum of the points in an RDD.
# HINT: you do want to call this once per partition
def part_sums(clusterrdd):
    ar = np.array(list(clusterrdd))
    return (np.sum(ar, axis=0), ar.shape[0])

# helper function to aggregate the sums and counts into means
def sums_2_means(sumslist):
    sums = sumslist[::2]
    counts = sumslist[1::2]
    return np.sum(sums, axis=0)/np.sum(counts)

# helper function to filter by cluster number
def filter_by_cluster(cluster_number):
    def is_in_cluster(x):
        return x[0] == cluster_number
    return is_in_cluster

# Create an empty array for updated centroids
ucentroids = [ None for i in range(4)]

# use the orginal centroids as input
centroids = originalCentroids

# create clusters and ptstriples as previously
clusters = points.map(lambda point: assign_class(point, centroids))
ptstriples = clusters.zip(points).map(transform_tuple)

# For each of the four clusters (repeat this code for all four clus
ters)

# filter for only the points in this cluster
lcluster0 = ptstriples.filter(filter_by_cluster(0)).map(lambda x: x
[1])

# derive means in each cluster (or do it another way)
cluster0means = lcluster0.mapPartitions(part_sums).collect()

# aggregate means from each partition and update centroids
ucentroids[0] = sums_2_means(cluster0means)

lcluster1 = ptstriples.filter(filter_by_cluster(1)).map(lambda x: x
[1])
cluster1means = lcluster1.mapPartitions(part_sums).collect()
ucentroids[1] = sums_2_means(cluster1means)

lcluster2 = ptstriples.filter(filter_by_cluster(2)).map(lambda x: x
[1])
cluster2means = lcluster2.mapPartitions(part_sums).collect()
ucentroids[2] = sums_2_means(cluster2means)

lcluster3 = ptstriples.filter(filter_by_cluster(3)).map(lambda x: x
[1])
cluster3means = lcluster3.mapPartitions(part_sums).collect()
ucentroids[3] = sums_2_means(cluster3means)
```

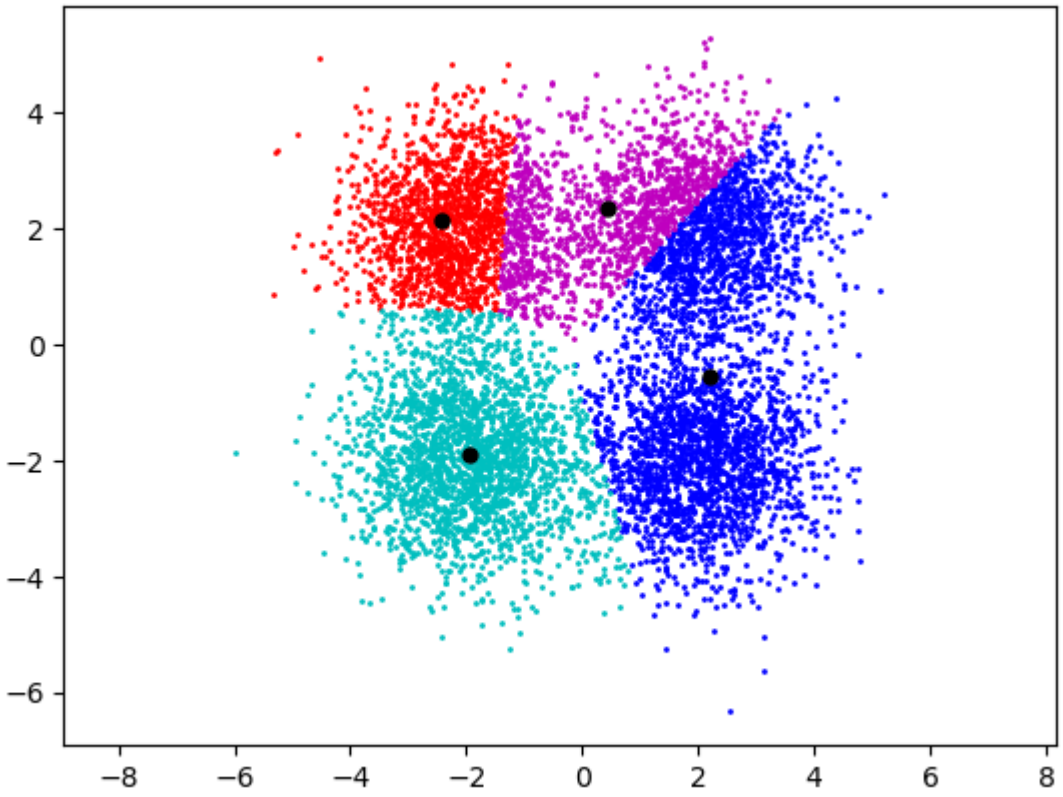
```
# optionally plot the clusters (may be slow)
plot_clusters(ptstriples, np.array(ucentroids))
```

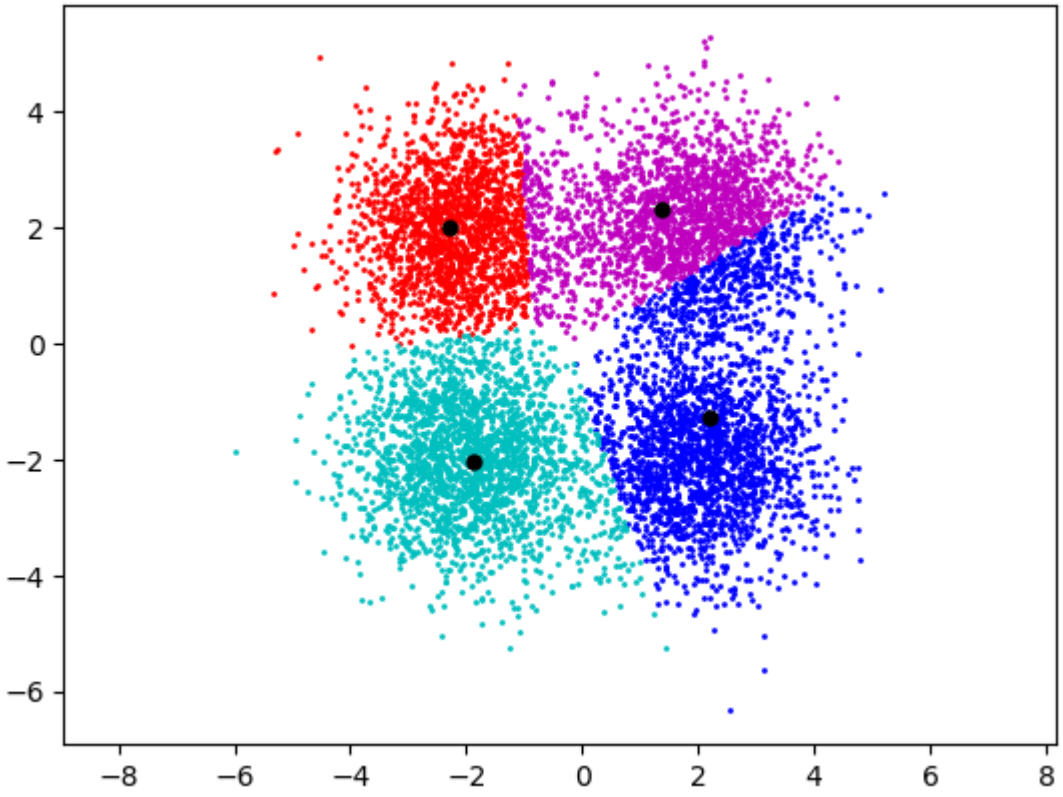


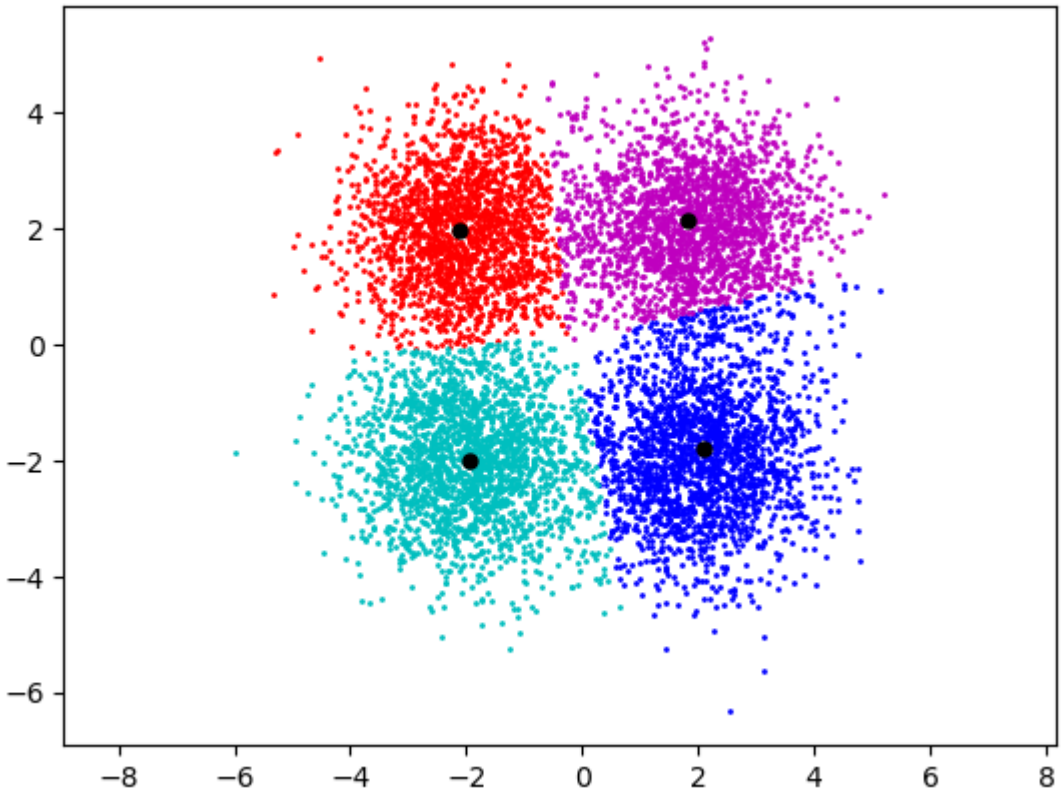
In [11]:

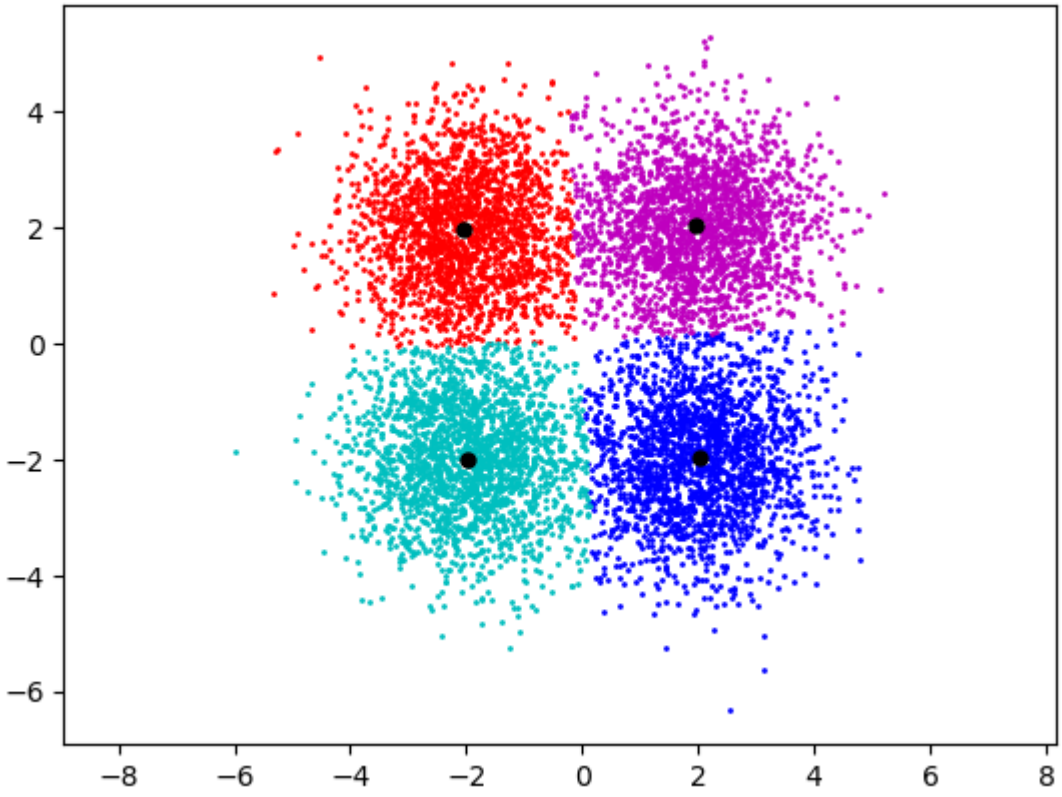
```
# %%timeit -n 1

ucentroids = [ None for i in range(4)]
centroids = originalCentroids

iterations = 10

for i in range(iterations):

    # create clusters and ptstriples as previously
    clusters = points.map(lambda point: assign_class(point, centroi
ds))
    ptstriples = clusters.zip(points).map(transform_tuple)

    # optionally persist the triples for cache reuse
    ptstriples.persist()

    # run the whole process repeatedly

    # filter for only the points in this cluster
    lcluster0 = ptstriples.filter(filter_by_cluster(0)).map(lambda
x: x[1])

    # derive means in each cluster (or do it another way)
    cluster0means = lcluster0.mapPartitions(part_sums).collect()

    # aggregate means from each partition and update centroids
    ucentroids[0] = sums_2_means(cluster0means)

    lcluster1 = ptstriples.filter(filter_by_cluster(1)).map(lambda
x: x[1])
    cluster1means = lcluster1.mapPartitions(part_sums).collect()
    ucentroids[1] = sums_2_means(cluster1means)

    lcluster2 = ptstriples.filter(filter_by_cluster(2)).map(lambda
x: x[1])
    cluster2means = lcluster2.mapPartitions(part_sums).collect()
    ucentroids[2] = sums_2_means(cluster2means)

    lcluster3 = ptstriples.filter(filter_by_cluster(3)).map(lambda
x: x[1])
    cluster3means = lcluster3.mapPartitions(part_sums).collect()
    ucentroids[3] = sums_2_means(cluster3means)

    # apply updated centroids for next iteration
    centroids = np.array(ucentroids)

    # optionally plot the clusters (may be slow)
    plot_clusters(ptstriples, np.array(ucentroids))
```

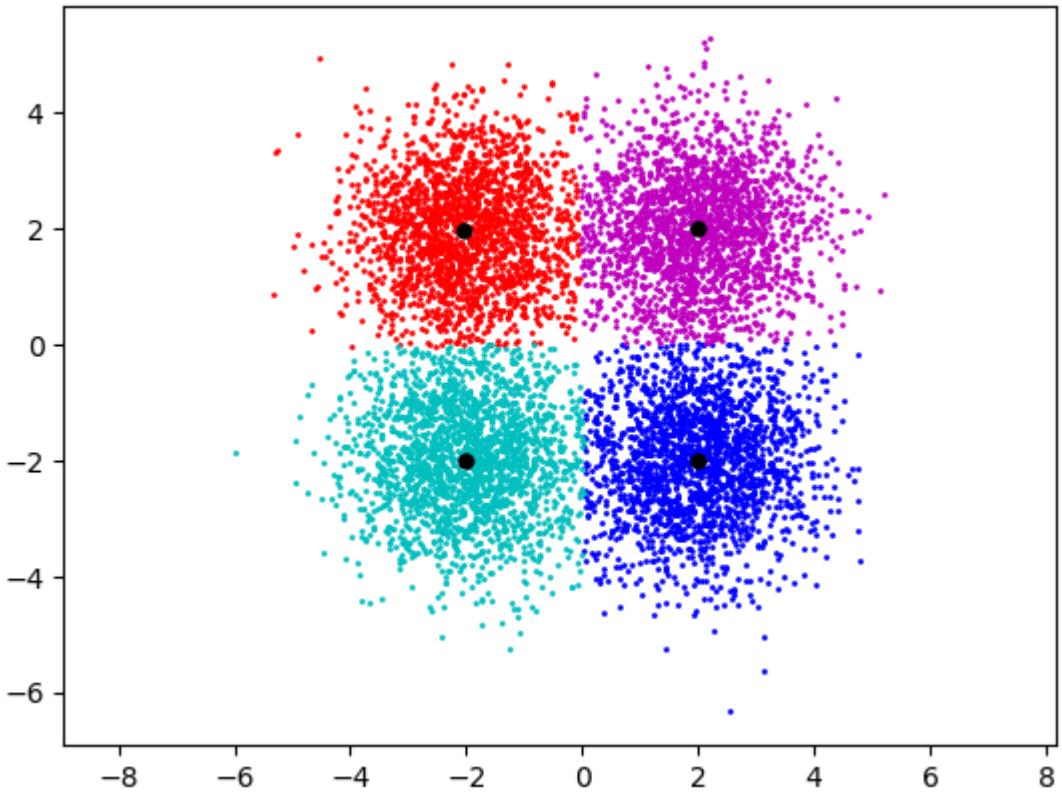

4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8

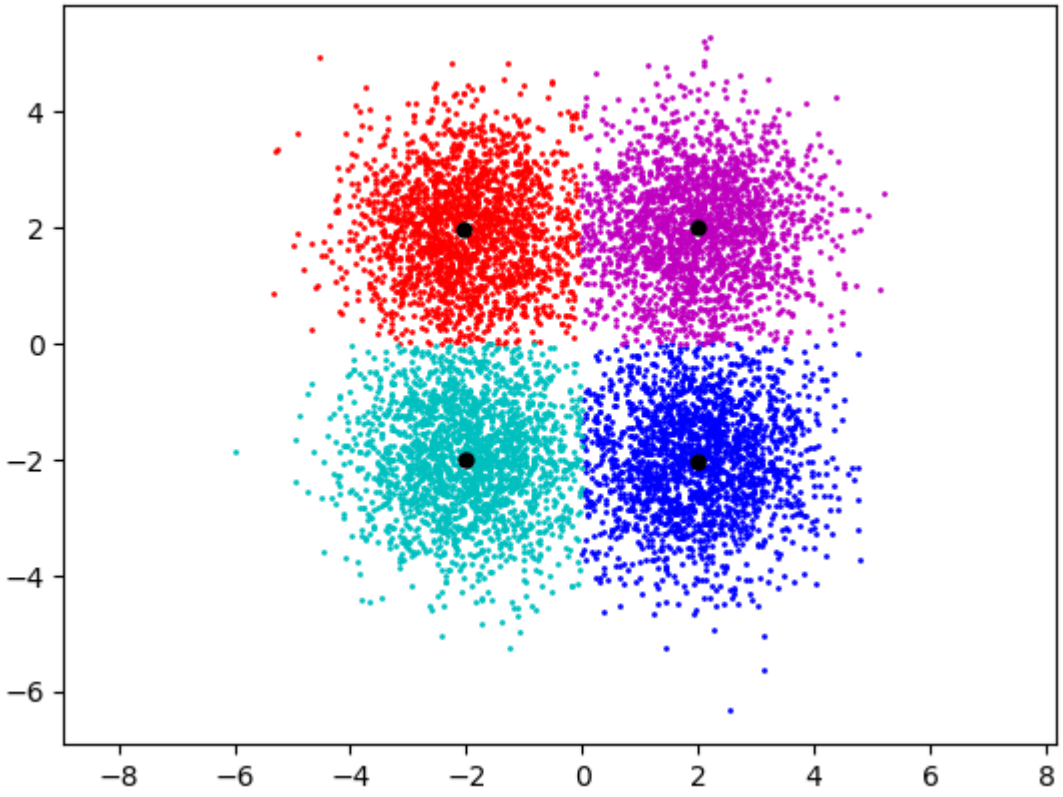

4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8

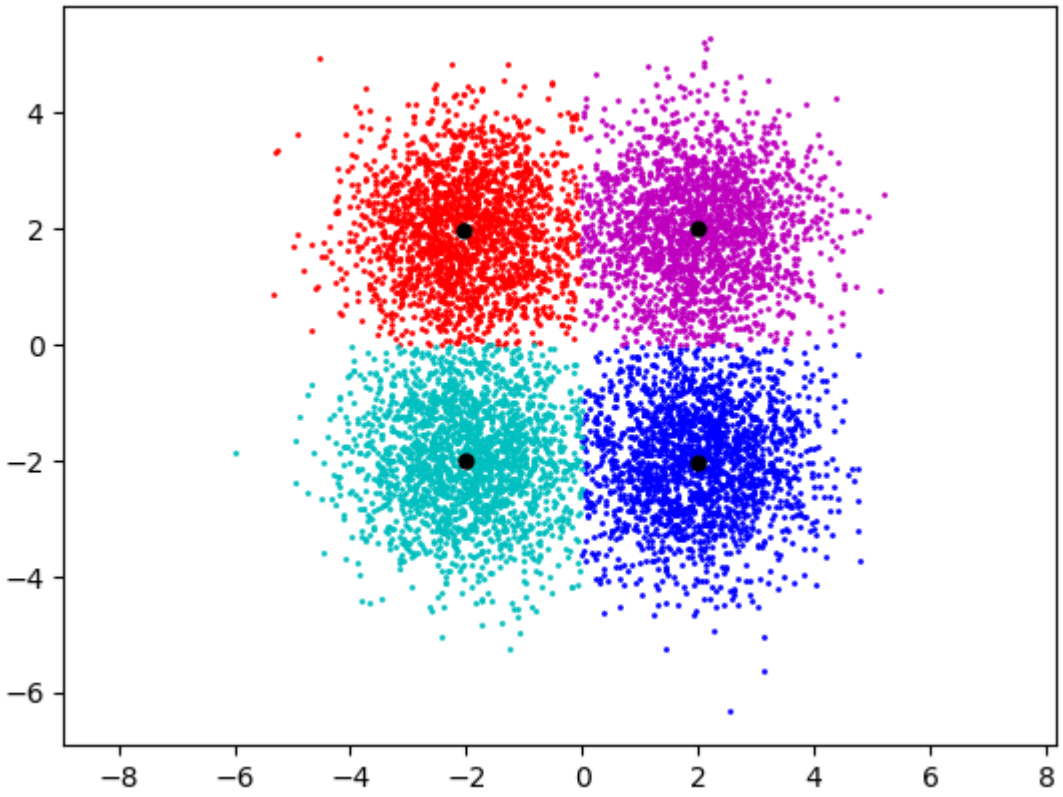

4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8

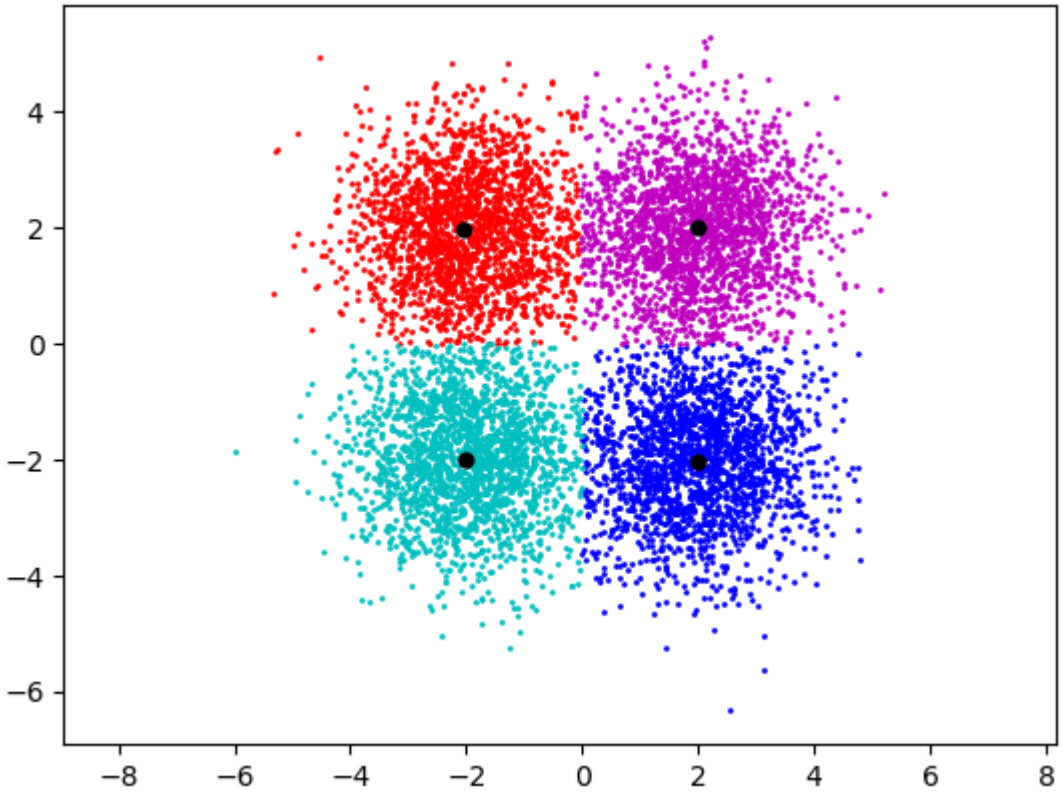

4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8

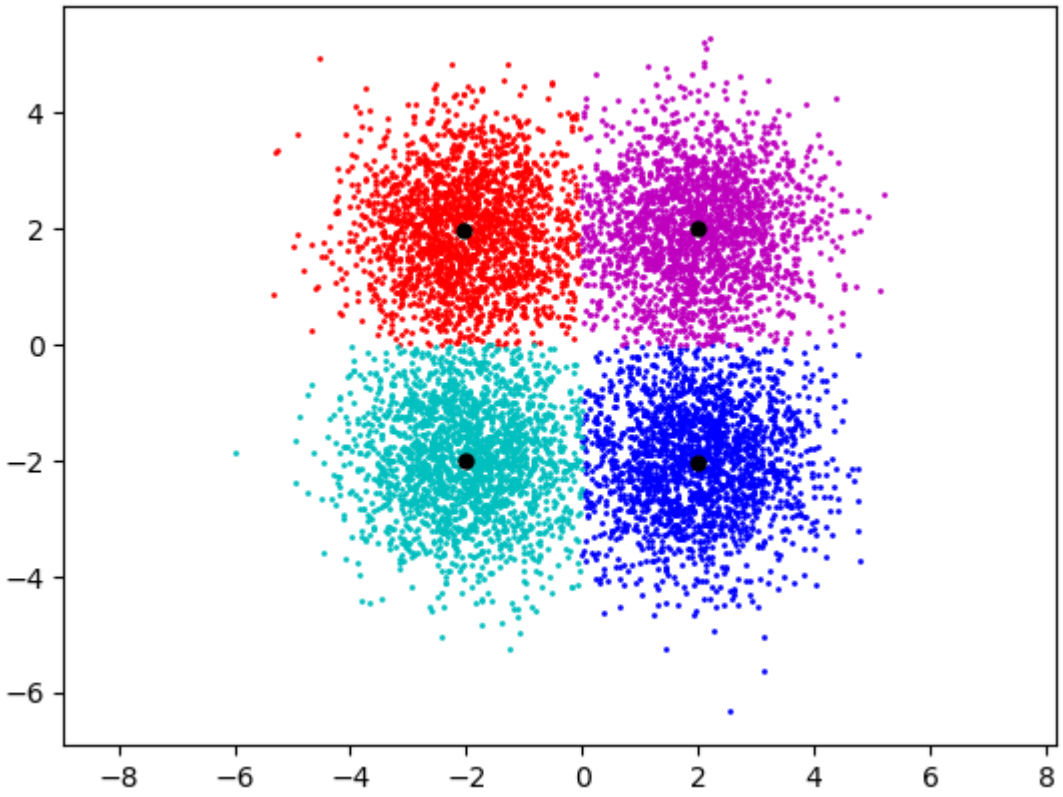

4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8


4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8


4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8


4
2
0
−2
−4
−6
−8
−6
−4
−2
0
2
4
6
8





Stop the context

If you crash, you will often need to close Spark explicitly to reset the system. Just run this cell.

In [12]:
```
sc.stop()
```

Questions

We now turn to an evaluation of the relative performance of our two implementations and a study of the benefit of caching. Performance results are consistent across my 8-core laptop (MacOSX), 12-core laptop (Windows), and an 8-core droplet on Digital Ocean. Your results may vary, but you should be able to explain.

Question 1

Comment out all `plot_clusters` call and uncomment the `%%timeit` decorators. Run the notebook and get the timing information.

- How long did each implementation take to run?
  - `groupBy` : 2.91 s ± 257 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
  - `filter` : 6.12 s ± 81.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
- The `groupBy` implementation is faster than the `filter` implementation. Why?
  - The `groupBy` method efficiently aggregates the data into groups in a single step, typically outperforming the repetitive use of multiple filter functions. This grouping technique optimizes memory and computational efficiency by reducing data shuffling, enabling batch processing of grouped data, improving memory management, optimizing cache usage, and decreasing overall computational complexity.

On a distributed-memory cluster, the `filter` implementation will always be faster.

- Why would the `filter` implementation run faster on distributed memory?
  - The `filter` implementation can parallely execute filter tasks on different nodes within the cluster, which would result in less execution time compared to a non-distributed setting. Additionally, the `groupBy` implementation would take more time because data shuffling across nodes is a comparatively slower process.

Question 2

There are two commented out `persist()` calls in this notebook: one for `points` and one for `ptstriples` in the `filter` implementation. Run four versions of this code and give performance results (timings from `%%timeit`) for each of the following:

- persist neither `points` nor `ptstriples` : 6.07 s ± 146 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
- persist `points` but not `ptstriples` : 6.02 s ± 188 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
- persist `ptstriples` but not `points` : 4.9 s ± 179 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
- persist both `points` and `ptstriples` : 4.88 s ± 215 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

Question 3

- Caching `ptstriples` in the `filter` implementation makes it faster. Explain why.

    - Creating the ptstriples RDD involves applying multiple Spark transformations like map(), zip(), and map() again to transform the original points RDD into the desired triplet format with cluster assignments.
    - Without caching, each iteration would need to reapply all those transformations on the points RDD to recreate the ptstriples RDD from scratch. But by calling persist() on it, Spark keeps the ptstriples RDD partitioned in memory after it is first created.
    - So in later loop iterations, Spark can just reuse the cached ptstriples partitions instead of redoing all the map() and zip() transformations over again. This avoids a lot of redundant computation and saves significant time, especially as the number of iterations grows.
- Why is it more effective to cache `ptstriples` than `points` ?
    - `ptstriples` has more computations attached to it (like map(), zip() and map() again) compared to `points` . The computations on `ptstriples` need to be performed at each iteration but computations on `points` are performed only at the beginning. Hence, it is more effective to cache `ptstriples` than `points` .