k-means in Spark

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 [121…
```
import matplotlib.pyplot as plt
import numpy as np
from pyspark import SparkContext

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

In [122…
```
# 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, float32])
# 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

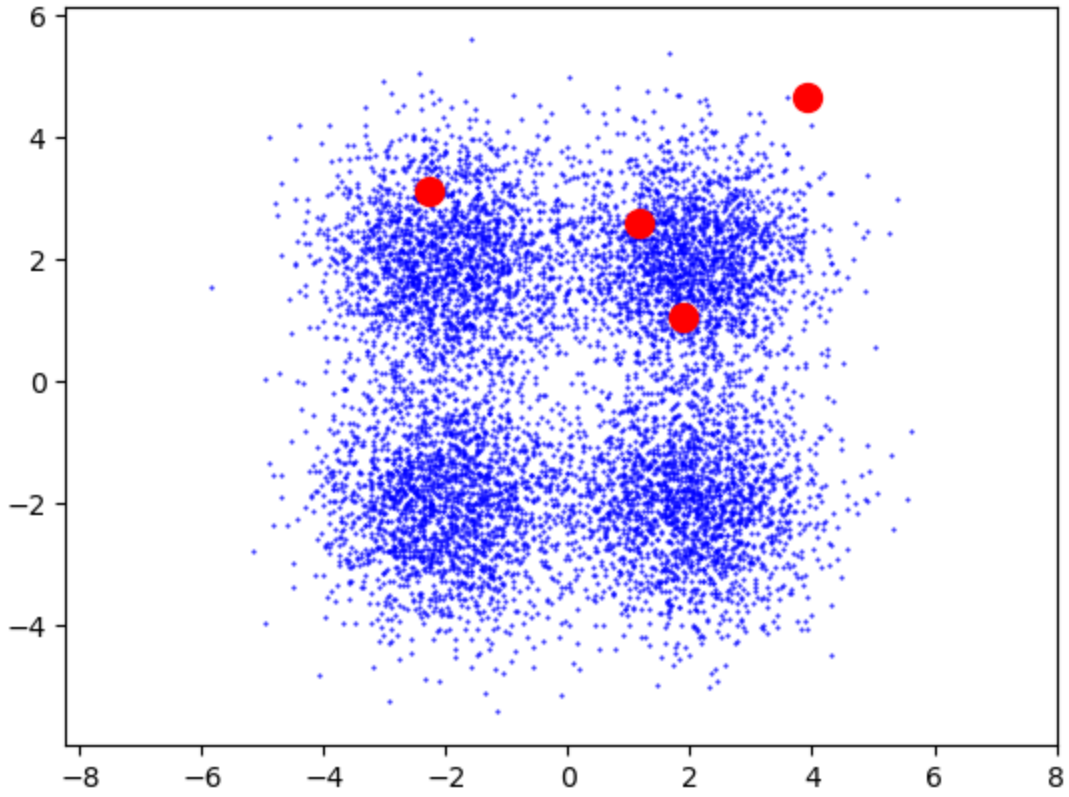
In [123…

```
# 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, classcount)
    points3 = np.random.multivariate_normal([-2,2], cov, classcount)
    points4 = np.random.multivariate_normal([-2,-2], cov, classcount)

    # put all points together and permute
    pointsall = np.concatenate((points1, points2, points3, points4), 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).co

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

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

Out[123]:

```
ParallelCollectionRDD[2] at readRDDFromFile at PythonRDD.scala:289
```

In [124…
```
# take a sample of k points as seeds (comment out the DEBUG line)

# contains 4 samples taken from the points dataset without replacement, selecte
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 [125…
```
# assign each point to a class using the assign_class function
# produces an RDD with type (int) with length equal to number of points
### TODO
clusters = points.map(lambda x: assign_class(x, centroids))
print(clusters.count()) # total points
print(clusters.take(4)) # first 4 elements
```

```
8000
[0, 0, 3, 3]
```

In [126…
```
# build an RDD of type (int, [float, float]) that specifies the cluser and ther
# this can be done efficiently with with `zip()` function
### TODO
ptstriples = clusters.zip(points)
print(ptstriples.take(4)) # specifies cluster and also point-coords
```

```
[(0, array([-3.70668254,  3.28829313])), (0, array([-2.14847358, -1.5796024
])), (3, array([ 0.50858396, -2.33460685])), (3, array([ 2.53318201, -0.197245
08]))]
```

Some hints for the next cell

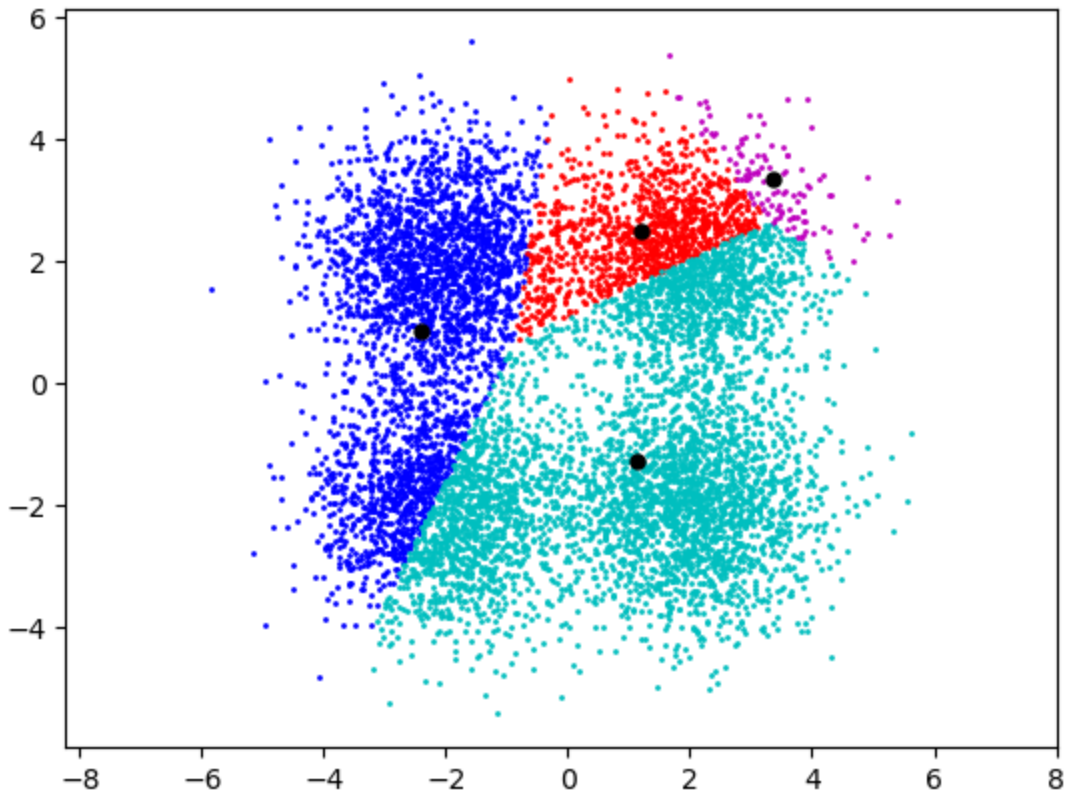
1. use `groupByKey()` to collect data by cluster
2. at the end you are going to have to use the function `np.mean(array, axis=0)` on a iterator. Keep the data in spark RDDs until the last step.
3. it can be hard to materialize RDDs into arrays you need to either `np.array(RDD)` or `np.array(list(RDDiterable))`
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 [127…

```
# update the centroids to be the mean of each cluster of points
###TODO

# helper function to take the mean, more readable than using a lambdas
def mean(points):
    cluster_arrays = np.array(list(points)) # hard to materialize RDDs into ar
    return np.mean(cluster_arrays, axis=0) # use this func on a iterator

# groupByKey() collects data by cluster
grouped_clusters = ptstriples.groupByKey()

grouped_averages = grouped_clusters.mapValues(lambda x: mean(x)) # map the ave
centroids = np.array(grouped_averages.values().collect()) # put those grouped
```

In [128…

```
# 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 [129…

```
%%timeit -n 1

iterations = 10
centroids = originalCentroids

for i in range(iterations):

    ### TODO
    # run the whole process over and over
    clusters = points.map(lambda x: assign_class(x, centroids))
    ptstriples = clusters.zip(points)

    grouped_clusters = ptstriples.groupByKey()
    grouped_averages = grouped_clusters.mapValues(lambda x: mean(x))

    centroids = np.array(grouped_averages.values().collect())

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

```
3.76 s ± 251 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

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.

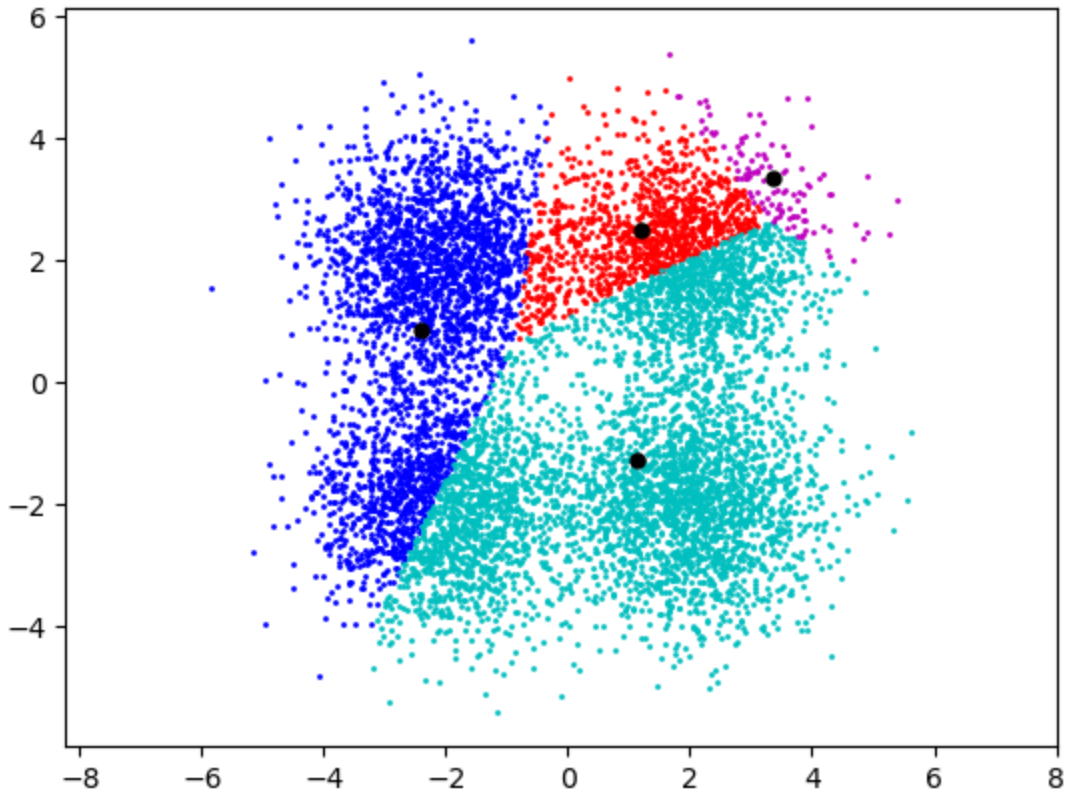
Your program should filter all data within a partition and then aggregate data within each partition. I have given you the `part_sums` helper to aggregate within each partition and the `sums_2_means` helper function to convert the sums into means.

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

In [130…

```
# 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)

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

```
# use the orginal centroids as input
centroids = originalCentroids

###TODO

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

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

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

# derive means in each cluster (or do it another way)
cluster0means = part_sums(lcluster0.collect()) # returns the sum of the points

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

lcluster1 = ptstriples.filter(lambda x: x[0] == 1).map(lambda x: x[1])
cluster1means = part_sums(lcluster1.collect())
ucentroids[1] = sums_2_means(cluster1means)

lcluster2 = ptstriples.filter(lambda x: x[0] == 2).map(lambda x: x[1])
cluster2means = part_sums(lcluster2.collect())
ucentroids[2] = sums_2_means(cluster2means)

lcluster3 = ptstriples.filter(lambda x: x[0] == 3).map(lambda x: x[1])
cluster3means = part_sums(lcluster3.collect())
ucentroids[3] = sums_2_means(cluster3means)

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



In [131…

```
%%timeit -n 1

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

iterations = 10

for i in range(iterations):

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

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

    # run the whole process repeatedly
    lcluster0 = ptstriples.filter(lambda x: x[0] == 0).map(lambda x: x[1])
    cluster0means = part_sums(lcluster0.collect())
    ucentroids[0] = sums_2_means(cluster0means)

    lcluster1 = ptstriples.filter(lambda x: x[0] == 1).map(lambda x: x[1])
    cluster1means = part_sums(lcluster1.collect())
    ucentroids[1] = sums_2_means(cluster1means)

    lcluster2 = ptstriples.filter(lambda x: x[0] == 2).map(lambda x: x[1])
    cluster2means = part_sums(lcluster2.collect())
    ucentroids[2] = sums_2_means(cluster2means)

    lcluster3 = ptstriples.filter(lambda x: x[0] == 3).map(lambda x: x[1])
    cluster3means = part_sums(lcluster3.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))
```

```
6.11 s ± 255 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Stop the context

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

In [132…

```
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` implementation : 3.88 s ± 365 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
  - `filter` implementation : 6.48 s ± 497 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
- The `groupBy` implementation is faster than the `filter` implementation. Why?
  - `groupBy` implementation is faster than the `filter` implementation because this case, there is only one distribution memory cluster and it was divided in k-means where k = 4. With this case, Spark RDDs don't have to worry about shuffling. The `filter` implementation also have to wait on `collect()` for the data to be filtered (seen in the cluster means code above), which causes it to be slower.

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

- Why would the `filter` implementation run faster on distributed memory?
  - Distrbuted memory is broken down into data partitions so shuffling and tranfsering data between them using `groupByKey` implementation is much slower. The

`filter` implementation takes care of this since it moves no data outside of partitions.

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`
  - (groupBy): 3.7 s ± 114 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
  - (filter): 10 s ± 424 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
- persist `points` but not `ptstriples`
  - (groupBy): 3.69 s ± 155 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
  - (filter): 9.73 s ± 381 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
- persist `ptstriples` but not `points`
  - (groupBy): 3.7 s ± 167 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
  - (filter): 6.9 s ± 349 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
- persist both `points` and `ptstriples`
  - (groupBy): 3.73 s ± 181 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
  - (filter): 6.31 s ± 332 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.
  - Caching `ptstriples` in the `filter` implementation makes it faster because there are multiple calls of `ptstriples` for filtering the clusters one by one. Caching `ptstriples` using `persist()` indicates desire to reuse an RDD, encourages Spark to keep it in memory so that the same computations to load the RDD from memory is minimized.
- Why is it more effective to cache `ptstriples` than `points`?
  - It is more effective to cache `ptsriples` than `points` because `ptstriples` is used more throughout the program so when cached, its load overhead would be much less to the overall runtime of the implementation.

In [ ]: