DAMLAS-HW03-vonOven

August 2, 2016

```
In [2]: %%javascript
        /*********************************************************************
        Known Mathjax Issue with Chrome - a rounding issue adds a border to the rig
        https://github.com/mathjax/MathJax/issues/1300
        A quick hack to fix this based on stackoverflow discussions:
        http://stackoverflow.com/questions/34277967/chrome-rendering-mathjax-equati
        *********************************************************************

        $('.math>span').css("border-left-color","transparent")
```

```
<IPython.core.display.Javascript object>
```

```
In [3]: %reload_ext autoreload
        %autoreload 2
```

1 DAMLAS - Machine Learning At Scale

1.1 Assignment - HW3

Data Analytics and Machine Learning at Scale Target, Minneapolis

---

Name: Mark von Oven
Class: DAMLAS (Section Summer 2016)
Email: Mark.Vonoven@Target.com
Week: 03

2 Table of Contents

3.2. MLlib-centric Kmeans
3.3. Homegrown KMeans in Spark
3.4. Making Homegrown KMeans more efficient
3.5. OPTIONAL Weighted KMeans
3.6. OPTIONAL Linear Regression
3.7. OPTIONAL Error surfaces

1 Instructions Back to Table of Contents * Homework submissions are due by Tueday, 08/02/2016 at 11AM (CT).

- Prepare a single Jupyter note, please include questions, and question numbers in the questions and in the responses. Submit your homework notebook via the following form:
- Submission Link - Google Form

2.0.1 Documents:

- IPython Notebook, published and viewable online.
- PDF export of IPython Notebook.

2 Useful References Back to Table of Contents

- Karau, Holden, Konwinski, Andy, Wendell, Patrick, & Zaharia, Matei. (2015). Learning Spark: Lightning-fast big data analysis. Sebastopol, CA: O'Reilly Publishers.
- Hastie, Trevor, Tibshirani, Robert, & Friedman, Jerome. (2009). The elements of statistical learning: Data mining, inference, and prediction (2nd ed.). Stanford, CA: Springer Science+Business Media. **(Download for free here)**

- **2.1 Ryza, Sandy, Laserson, Uri, Owen, Sean, & Wills, Josh. (2015). Advanced analytics with Spark: Patterns for learning from data at scale. Sebastopol, CA: O'Reilly Publishers.**

- Slides for Supervised-ML-Classification-via-GradientDescent
- Slides from High Entropy Friday

---

3 HW Problems Back to Table of Contents
HW3.0: Short answer questions
Back to Table of Contents

What is Apache Spark and how is it different to Apache Hadoop? Apache Spark is an open source cluster computing framework. Originally developed at the University of California, Berkeley's AMPLab, the Spark codebase was later donated to the Apache Software Foundation, which has maintained it since. Spark provides an interface for programming entire clusters with implicit data parallelism and fault-tolerance.

Fill in the blanks: Spark API consists of interfaces to develop applications based on it in Java, Scala, R & Python languages (list languages).

Using Spark, resource management can be done either in a single server instance or using a framework such as Mesos or Hadoop YARN in a distributed manner.

What is an RDD and show a fun example of creating one and bringing the first element back to the driver program.

3.

Below is an example of creating a spark context environment, as well as a "fun example" of creating an RDD. Thank you Abbott and Costello.

```
In [1]: import os
        import sys #current as of 9/26/2015
        import pyspark
        from pyspark.sql import SQLContext
        # We can give a name to our app (to find it in Spark WebUI) and configure e
        # In this case, it is local multicore execution with "local[*]"
        app_name = "example-logs"
        master = "local[*]"
        conf = pyspark.SparkConf().setAppName(app_name).setMaster(master)
        sc = pyspark.SparkContext(conf=conf)
        sqlContext = SQLContext(sc)
        print(sc)
        print(sqlContext)
        # Import some libraries to work with dates
        import dateutil.parser
        import dateutil.relativedelta as dateutil_rd
```

```
<pyspark.context.SparkContext object at 0x7fc480074320>
<pyspark.sql.context.SQLContext object at 0x7fc480ed45f8>
```

```
In [2]: %%writefile funRDD.txt
        who is on first
        what is on second
        I dunno is on third
```

```
Writing funRDD.txt
```

```
In [4]: rdd = sc.textFile('funRDD.txt')
        rdd.first()
```

```
Out[4]: 'who is on first'
```

HW3.1 WordCount plus sorting

Back to Table of Contents

The following notebooks will be useful to jumpstart this collection of Homework exercises:

- Example Notebook with Debugging tactics in Spark

- Word Count Quiz
- Work Count Solution

In Spark write the code to count how often each word appears in a text document (or set of documents). Please use this homework document (with no solutions in it) as a the example document to run an experiment. Report the following: * provide a sorted list of tokens in decreasing order of frequency of occurence limited to [top 20 most frequent only] and [bottom 10 least frequent].

OPTIONAL Feel free to do a secondary sort where words with the same frequncy are sorted alphanumerically increasing. Plseas refer to the following notebook for examples of secondary sorts in Spark. Please provide the following [top 20 most frequent terms only] and [bottom 10 least frequent terms]

NOTE [Please incorporate all referenced notebooks directly into this master notebook as cells for HW submission. I.e., HW submissions should comprise of just one notebook]__

```
In [4]: import re, string
        # HW3.1 WordCount plus sorting
        def cleanWords(line):
            regex = re.compile('[%s]' % re.escape(string.punctuation))
            line = regex.sub(' ', str(line).lower())
            line = re.sub( '\s+', ' ', line )
            words = line.split()
            return words

        myFile = 'DAMLAS-HW03-Template-2016-07-22.ipynb'
        text_file = sc.textFile(myFile)
        counts = text_file.flatMap(lambda line: cleanWords(line)) \
            .map(lambda word: (word, 1)) \
            .reduceByKey(lambda a, b: a + b) \
            .sortByKey() \
            .sortBy(lambda a: -a[1])

        wordCounts = counts.collect()
        sizeWC = len(wordCounts)
        print("***Top 20***")
        for v in wordCounts[:20]:
            print (v)
        print("\n***Bottom 10***")
        for w in wordCounts[sizeWC-10:]:
            print (w)

***Top 20***
('n', 219)
('the', 76)
('of', 51)
('code', 50)
('a', 48)
```

```
('metadata', 48)
('cell', 47)
('type', 47)
('source', 46)
('and', 40)
('hw3', 39)
('count', 35)
('to', 35)
('in', 34)
('collapsed', 29)
('execution', 29)
('outputs', 29)
('true', 28)
('h2', 26)
('kmeans', 26)

***Bottom 10***
('w01', 1)
('week', 1)
('wendell', 1)
('will', 1)
('wills', 1)
('wordcount', 1)
('wordcountdebugging', 1)
('write', 1)
('z', 1)
('zaharia', 1)
```

HW3.1.1

Back to Table of Contents

Modify the above word count code to count words that begin with lower case letters (a-z) and report your findings. Again sort the output words in decreasing order of frequency.

```
In [5]: import re, string
        # HW3.1.1 Modified
        def cleanWords(line):
            regex = re.compile('[%s]' % re.escape(string.punctuation))
            line = regex.sub(' ', str(line))
            line = re.sub( '\s+', ' ', line )
            words = line.split()
            return words

        def checkLower(word):
            if word[0].islower():
                return word
            else:
                return "*UPPERS*"
```

```
myFile = 'DAMLAS-HW03-Template-2016-07-22.ipynb'
text_file = sc.textFile(myFile)
counts = text_file.flatMap(lambda line: cleanWords(line)) \
    .map(lambda word: (checkLower(word), 1) ) \
    .reduceByKey(lambda a, b: a + b) \
    .sortByKey() \
    .sortBy(lambda a: -a[1])

wordCounts = counts.collect()
sizeWC = len(wordCounts)
print("***Top 20***")
for v in wordCounts[:20]:
    if v[0] != "*UPPERS*":
        print (v)
print("\n***Bottom 10***")
for w in wordCounts[sizeWC-10:]:
    if v[0] != "*UPPERS*":
        print (w)
```

```
***Top 20***
('n', 219)
('the', 72)
('of', 50)
('metadata', 48)
('cell', 47)
('type', 47)
('a', 46)
('source', 46)
('code', 41)
('and', 40)
('to', 35)
('count', 32)
('in', 32)
('collapsed', 29)
('execution', 29)
('outputs', 29)
('true', 28)
('h2', 26)
('data', 22)

***Bottom 10***
('versus', 1)
('vgmpivsi4rvqz0s', 1)
('viewable', 1)
('w0', 1)
('w01', 1)
('weight', 1)
```

```
('weighted', 1)
('will', 1)
('write', 1)
('z', 1)
```

```
In [86]: ## Drivers & Runners
```

```
In [87]: ## Run Scripts, S3 Sync
```

HW3.2: MLlib-centric KMeans
Back to Table of Contents
Using the following MLlib-centric KMeans code snippet:
NOTE
The **kmeans_data.txt** is available here https://www.dropbox.com/s/q85t0ytb9apggnh/kmeans_data.txt?dl=
TASKS * Run this code snippet and list the clusters that your find. * compute the Within Set Sum of Squared Errors for the found clusters. Comment on your findings.

```
In [6]: from pyspark.mllib.clustering import KMeans, KMeansModel
        from numpy import array
        from math import sqrt
        # HW3.2: MLlib-centric KMeans

        # Load and parse the data
        # NOTE  kmeans_data.txt is available here
        #          https://www.dropbox.com/s/q85t0ytb9apggnh/kmeans_data.txt?dl=0
        data = sc.textFile("kmeans_data.txt")
        parsedData = data.map(lambda line: array([float(x) for x in line.split(' ')

        # Build the model (cluster the data)
        clusters = KMeans.train(parsedData, 2, maxIterations=10,
                runs=10, initializationMode="random")

        # Evaluate clustering by computing Within Set Sum of Squared Errors
        def error(point):
            center = clusters.centers[clusters.predict(point)]
            return sqrt(sum([x**2 for x in (point - center)]))

        WSSSE = parsedData.map(lambda point: error(point)).reduce(lambda x, y: x +
        print("Within Set Sum of Squared Error = " + str(WSSSE))

        # Save and load model
        clusters.save(sc, "myModelPath")
        sameModel = KMeansModel.load(sc, "myModelPath")
        answers = clusters.clusterCenters
        for answer in answers:
            print(answer)
```

```
/usr/local/spark/python/pyspark/mllib/clustering.py:176: UserWarning: Support for r
  "Support for runs is deprecated in 1.6.0. This param will have no effect in 1.7.0


Within Set Sum of Squared Error = 0.6928203230275529
[ 0.1  0.1  0.1]
[ 9.1  9.1  9.1]
```

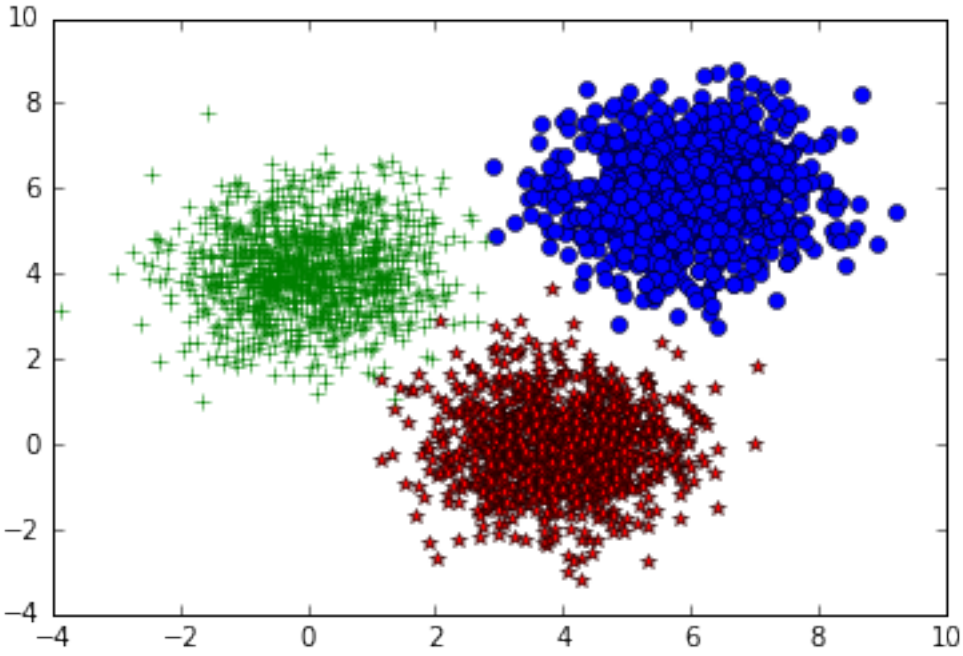
These clusters are pretty well contained within a small space, therefore the WSSSE isn't too bad.

```
In [90]: ## Run Scripts, S3 Sync
```

HW3.3: Homegrown KMeans in Spark
Back to Table of Contents
Download the following KMeans notebook.
Generate 3 clusters with 100 (one hundred) data points per cluster (using the code provided). Plot the data. Then run MLlib's Kmean implementation on this data and report your results as follows:

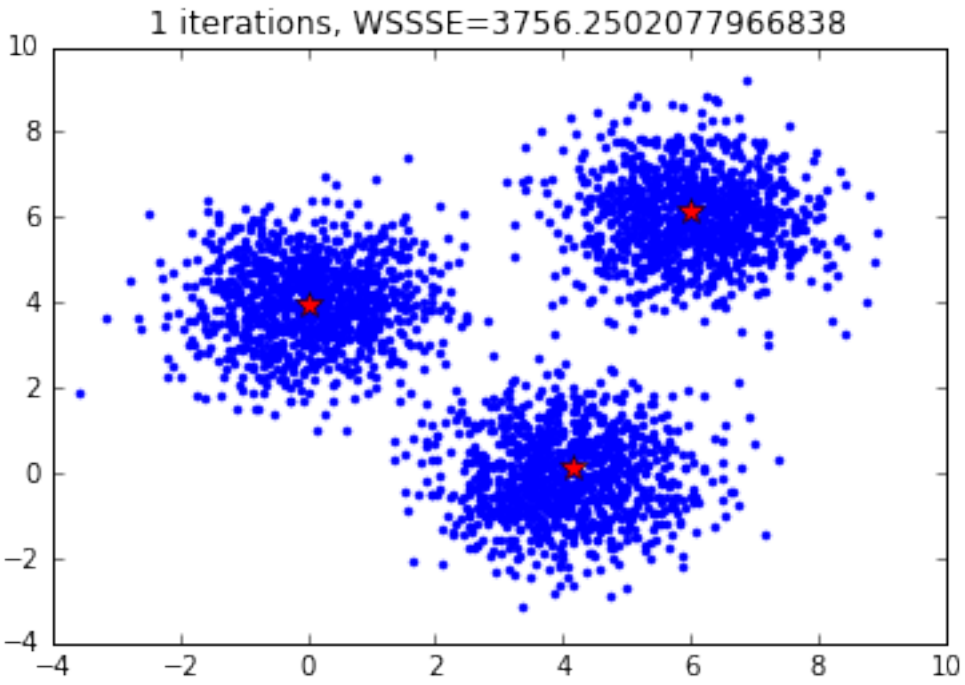
- plot the resulting clusters after 1 iteration, 10 iterations, after 20 iterations, after 100 iterations.
- in each plot please report the Within Set Sum of Squared Errors for the found clusters (as part of the title WSSSE). Comment on the progress of this measure as the KMEans algorithms runs for more iterations. Then plot the WSSSE as a function of the iteration (1, 10, 20, 30, 40, 50, 100).

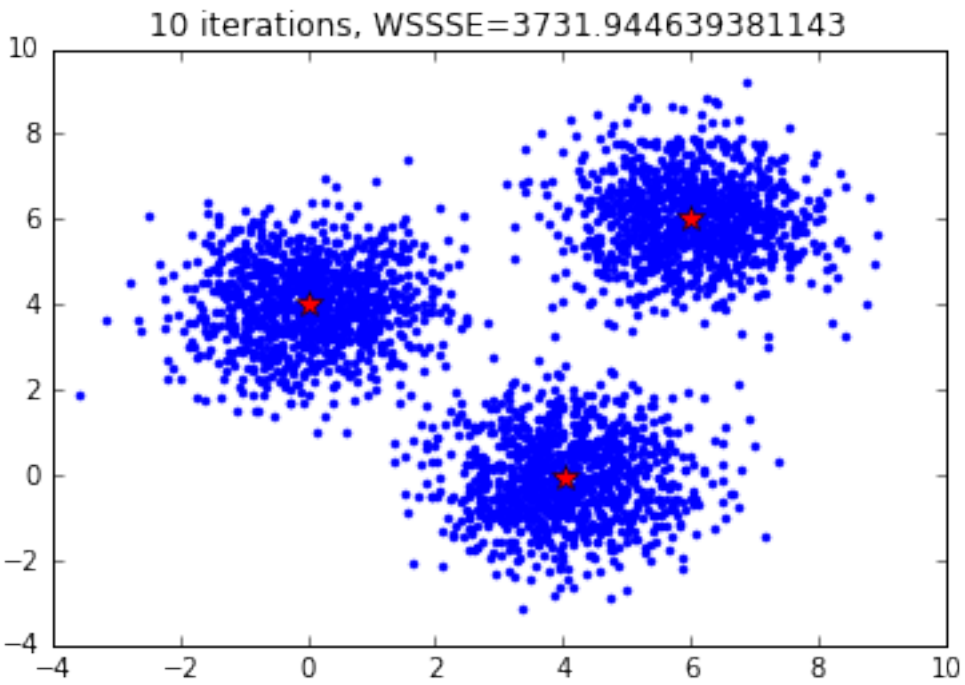
```
In [2]:  %matplotlib inline
         import numpy as np
         import pylab
         import json
         size1 = size2 = size3 = 1000
         samples1 = np.random.multivariate_normal([4, 0], [[1, 0],[0, 1]], size1)
         data = samples1
         samples2 = np.random.multivariate_normal([6, 6], [[1, 0],[0, 1]], size2)
         data = np.append(data,samples2, axis=0)
         samples3 = np.random.multivariate_normal([0, 4], [[1, 0],[0, 1]], size3)
         data = np.append(data,samples3, axis=0)
         # Randomlize data
         data = data[np.random.permutation(size1+size2+size3),]
         np.savetxt('data.csv',data,delimiter = ',')

         pylab.plot(samples1[:, 0], samples1[:, 1],'*', color = 'red')
         pylab.plot(samples2[:, 0], samples2[:, 1],'o',color = 'blue')
         pylab.plot(samples3[:, 0], samples3[:, 1],'+',color = 'green')
         pylab.show()
```

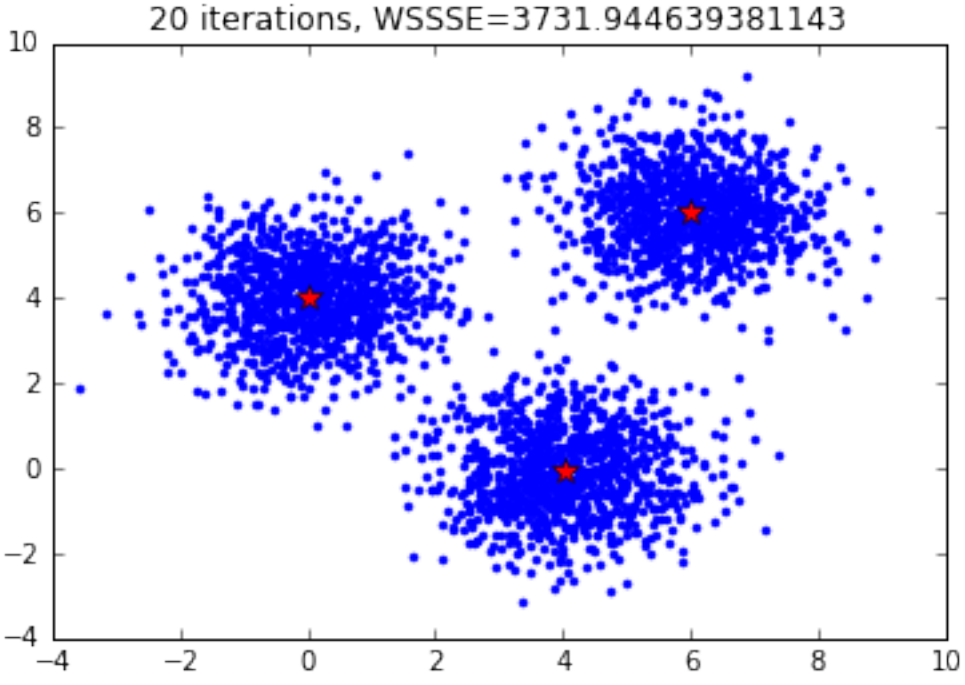


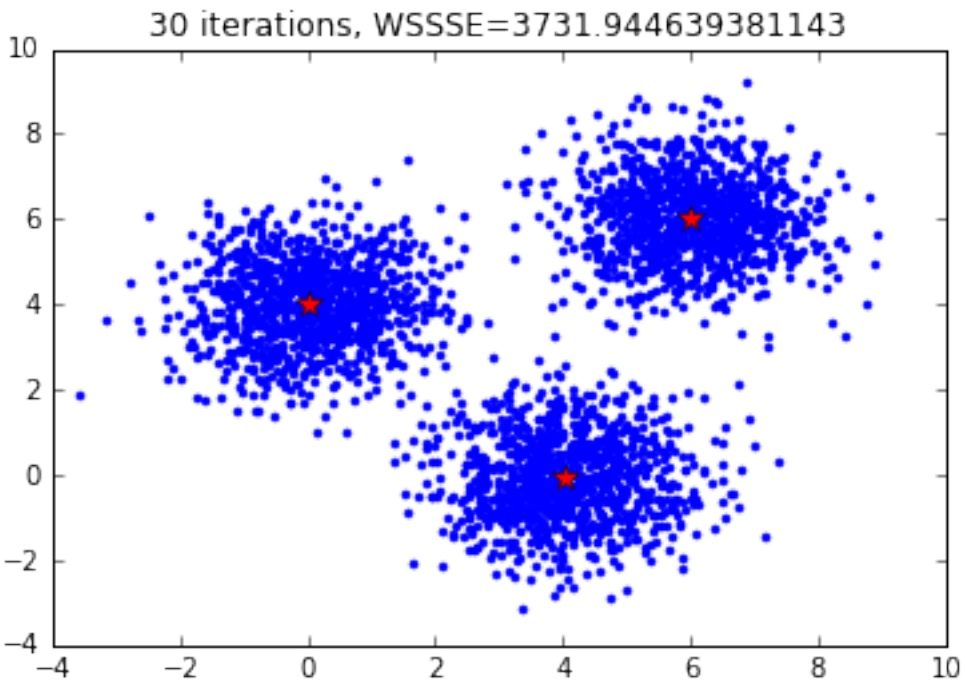

```
In [84]: from pyspark.mllib.clustering import KMeans, KMeansModel
         from numpy import array
         from math import sqrt
         # HW3.3: Homegrown KMeans in Spark

         #plot centroids and data points for each iteration
         def plot_iteration(means, title):
             pylab.plot(samples1[:, 0], samples1[:, 1], '.', color = 'blue')
             pylab.plot(samples2[:, 0], samples2[:, 1], '.', color = 'blue')
             pylab.plot(samples3[:, 0], samples3[:, 1],'.', color = 'blue')
             pylab.plot(means[0][0], means[0][1],'*',markersize =10,color = 'red')
             pylab.plot(means[1][0], means[1][1],'*',markersize =10,color = 'red')
             pylab.plot(means[2][0], means[2][1],'*',markersize =10,color = 'red')
             pylab.title(title)
             pylab.show()

         # Load and parse the data
         # NOTE  kmeans_data.txt is available here
         #          https://www.dropbox.com/s/q85t0ytb9apggnh/kmeans_data.txt?dl=0
         superIterator = [1, 10, 20, 30, 40, 50, 100]
         WSSSEplotter = []
         data = sc.textFile("data.csv")
         parsedData = data.map(lambda line: array([float(x) for x in line.split(','

         for i in superIterator:
```

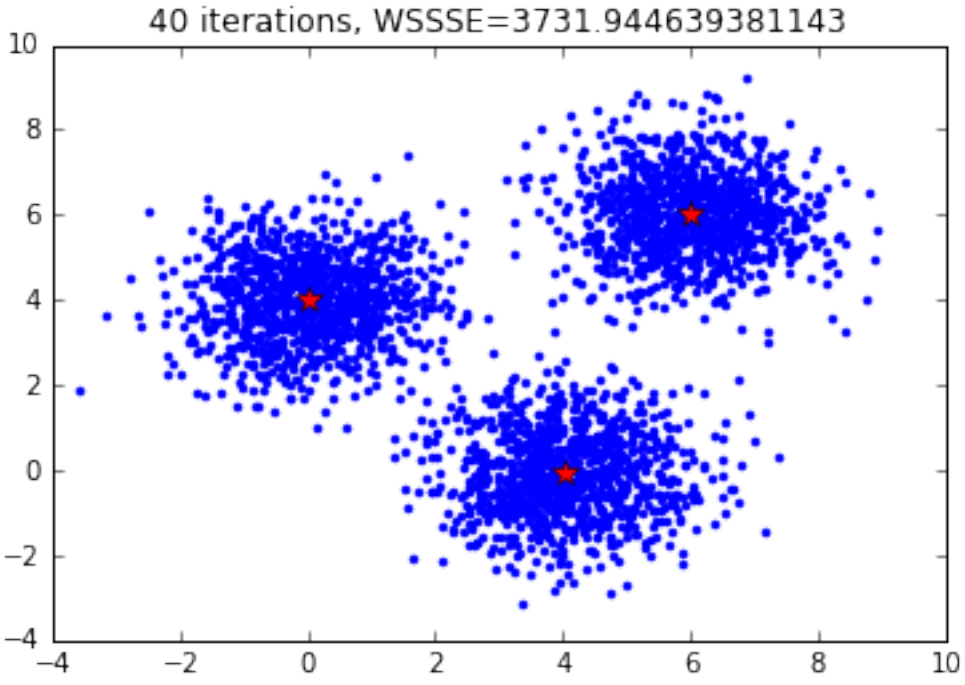
```
        # Build the model (cluster the data)
        clusters = KMeans.train(parsedData, 3, maxIterations=i, initializatio

        # Evaluate clustering by computing Within Set Sum of Squared Errors
        def error(point):
            center = clusters.centers[clusters.predict(point)]
            return sqrt(sum([x**2 for x in (point - center)]))

        WSSSE = parsedData.map(lambda point: error(point)).reduce(lambda x, y:
        #print("***For %s iterations***" % i)
        #print("Within Set Sum of Squared Error = " + str(WSSSE))
        WSSSEplotter.append(WSSSE)
        title = "%s iterations, WSSSE=%s" % (i, WSSSE)

        # Save and load model
        modelPath = "myModelPath"+str(i)
        clusters.save(sc, modelPath)
        sameModel = KMeansModel.load(sc, modelPath)
        answers = clusters.clusterCenters
        plot_iteration(answers, title)

#plot the WSSSE as a function of the iteration
pylab.plot(superIterator, WSSSEplotter, 'r--')
pylab.title("WSSSE as a function of the iteration")
pylab.show()
```

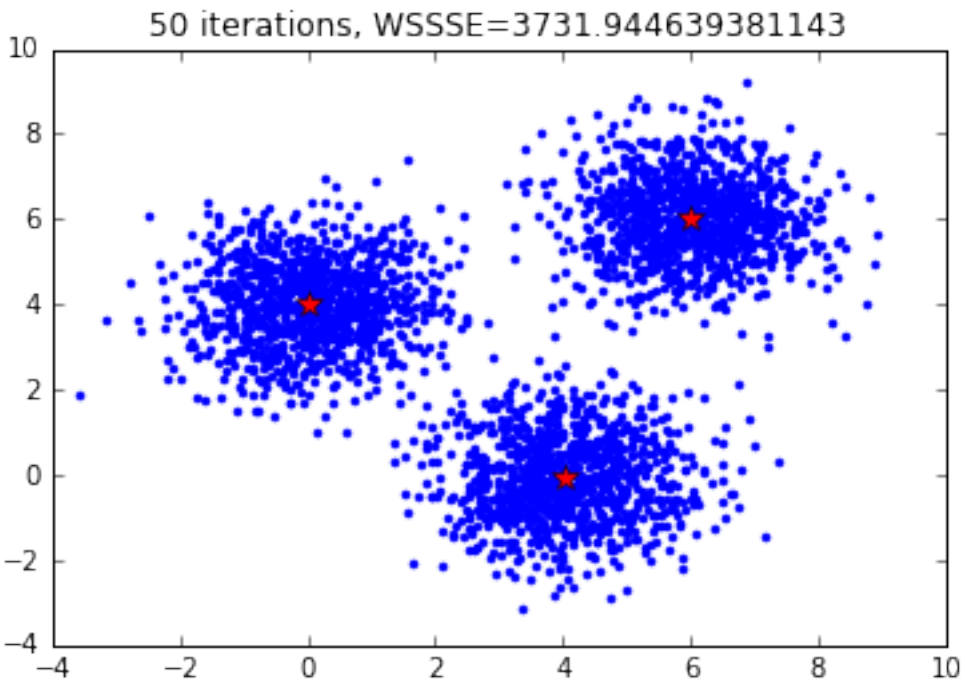


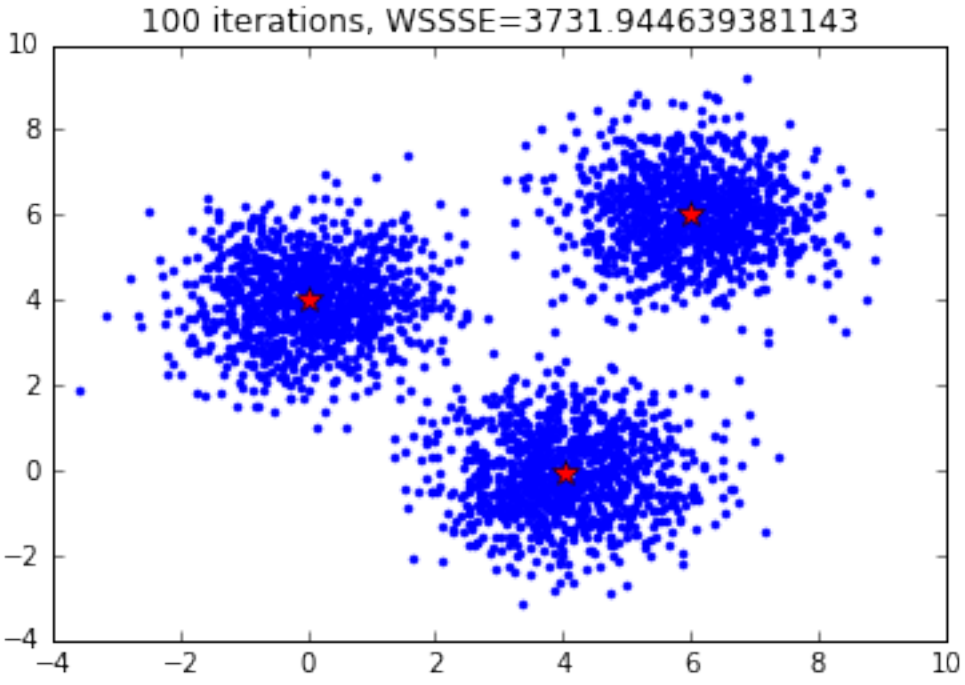

10 iterations, WSSSE=3731.944639381143
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10

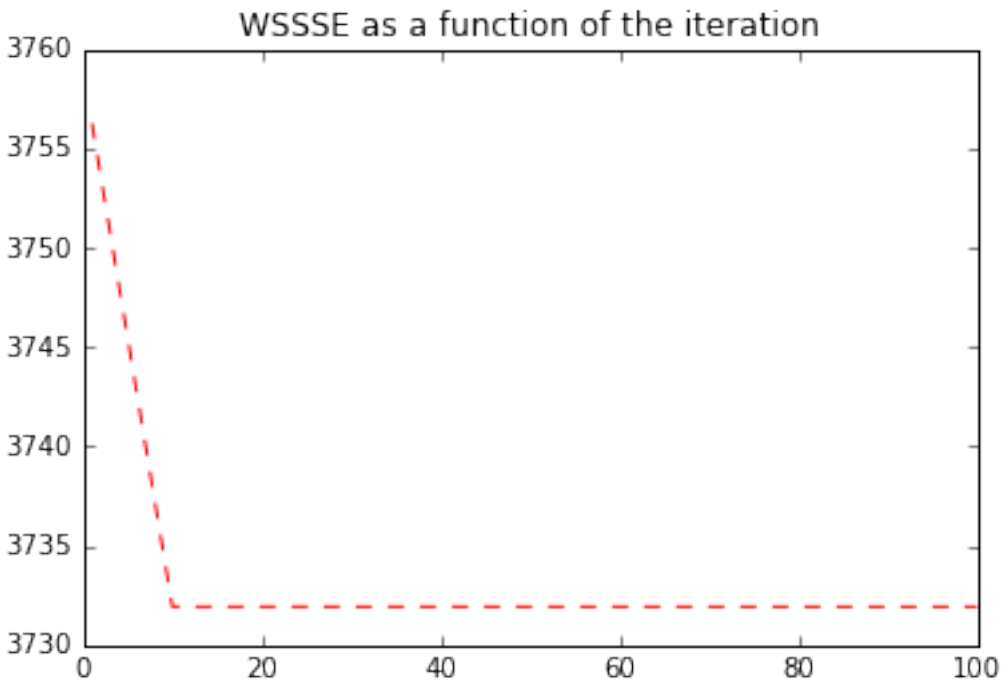

20 iterations, WSSSE=3731.944639381143
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


30 iterations, WSSSE=3731.944639381143
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


40 iterations, WSSSE=3731.944639381143
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


50 iterations, WSSSE=3731.944639381143
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


100 iterations, WSSSE=3731.944639381143
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10




```
In [92]: ## Drivers & Runners
```

```
In [93]: ## Run Scripts, S3 Sync
```

HW3.4: KMeans Experiments

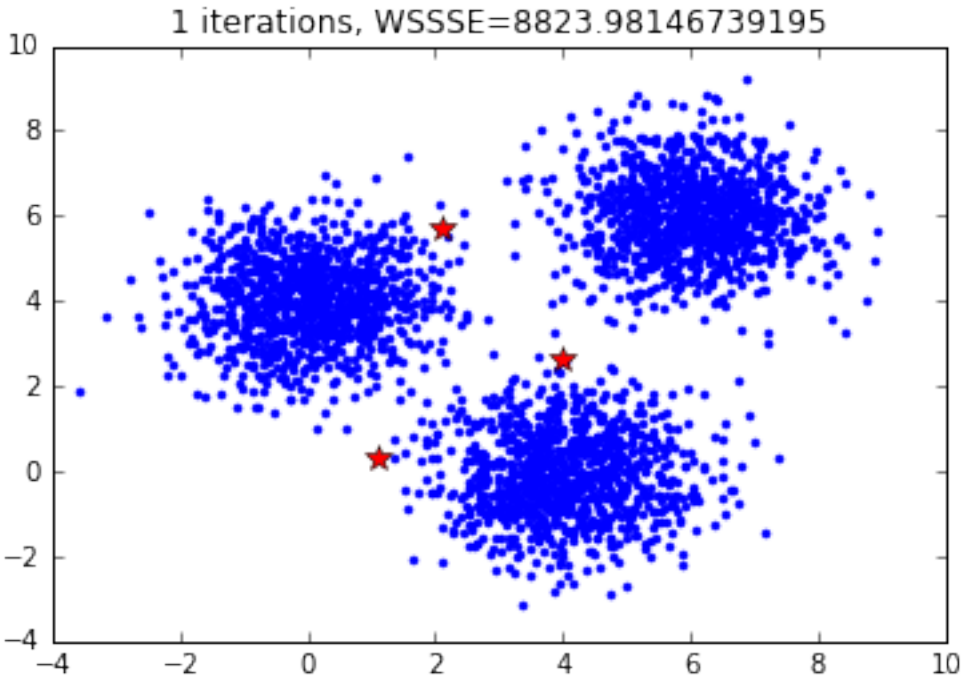
Back to Table of Contents

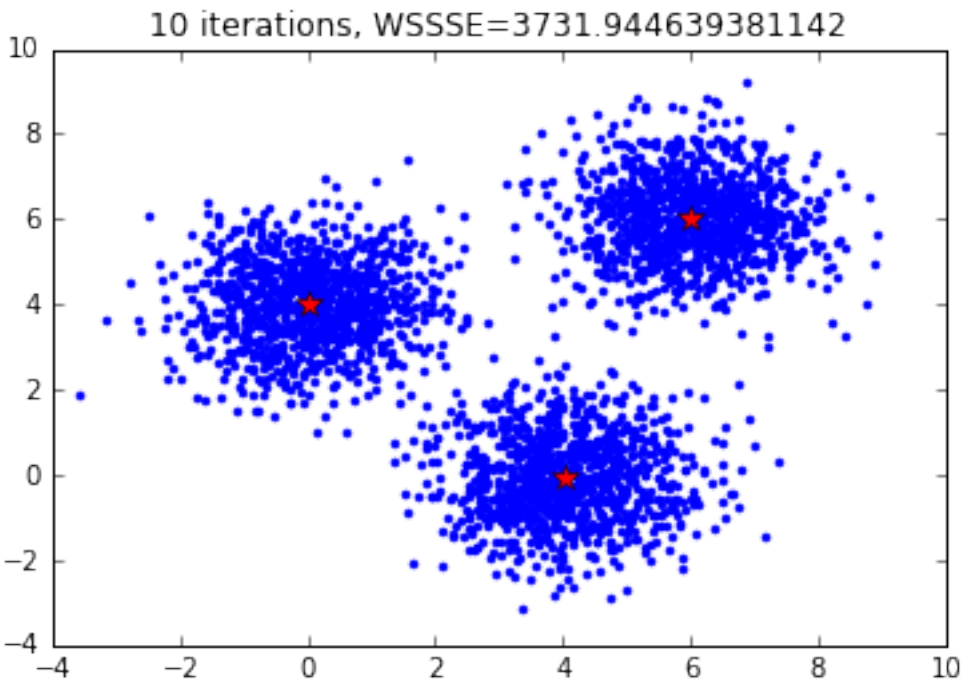
Using this provided homegrown Kmeans code repeat the experiments in HW3.3. Explain any differences between the results in HW3.3 and HW3.4.

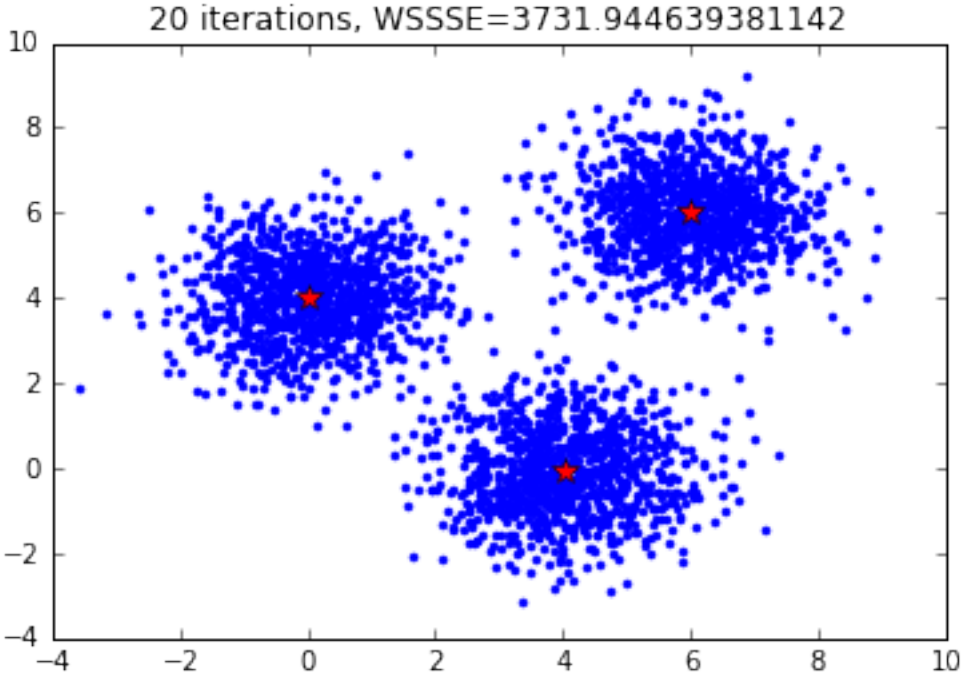
```
In [103]:  %matplotlib inline
           import numpy as np
           import pylab
           from math import sqrt
           from numpy import array
           # HW3.4: KMeans Experiments


           #Calculate which class each data point belongs to
           def nearest_centroid(line):
               x = np.array([float(f) for f in line.split(',')])
               closest_centroid_idx = np.sum((x - centroids)**2, axis=1).argmin()
               return (closest_centroid_idx,(x,1))

           def closest_centroid_xy(centroids, point):
               closest_centroid_idx = np.sum((point - centroids)**2, axis=1).argmin
```

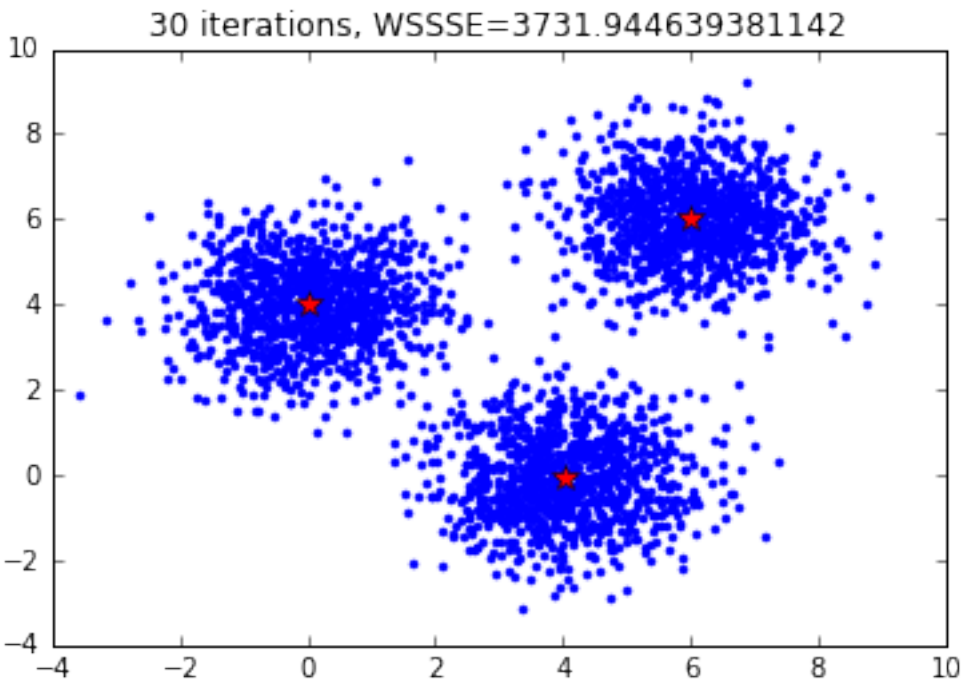
```
    #the return is a single centroid that is closest to the point
    return ((centroids)[closest_centroid_idx])

#plot centroids and data points for each iteration
def plot_iteration(means, title):
    pylab.plot(samples1[:, 0], samples1[:, 1], '.', color = 'blue')
    pylab.plot(samples2[:, 0], samples2[:, 1], '.', color = 'blue')
    pylab.plot(samples3[:, 0], samples3[:, 1],'.', color = 'blue')
    pylab.plot(means[0][0], means[0][1],'*',markersize =10,color = 'red')
    pylab.plot(means[1][0], means[1][1],'*',markersize =10,color = 'red')
    pylab.plot(means[2][0], means[2][1],'*',markersize =10,color = 'red')
    pylab.title(title)
    pylab.show()

# Evaluate clustering by computing Within Set Sum of Squared Errors
def error(center, point):
    #center = centroids_new
    return sqrt(sum([x**2 for x in (point - center)]))

# Load and parse the data
# NOTE  kmeans_data.txt is available here
#          https://www.dropbox.com/s/q85t0ytb9apggnh/kmeans_data.txt?dl=0
superIterator = [1, 10, 20, 30, 40, 50, 100]
WSSSEplotter = []
D = sc.textFile("./data.csv").cache()


K = 3
# Initialization: initialization of parameter is fixed to show an example
centroids = np.array([[0.0,0.0],[2.0,2.0],[0.0,7.0]])

for iterMax in superIterator:
    iter_num = 0
    for i in range(iterMax):
        res = D.map(nearest_centroid).reduceByKey(lambda x,y : (x[0]+y[0]
        res = sorted(res,key = lambda x : x[0])  #sort based on clusted
        centroids_new = np.array([x[1][0]/x[1][1] for x in res])  #divide
        centroids = centroids_new
        iter_num = iter_num + 1


        WSSSE = 0
        for k in D.map(lambda line: line.split(",")).collect():
            #creating a single point array directly from the file
            xy_point=np.array([np.float(k[0]), np.float(k[1])])
            #determine the closest cluster center to a point
            center = closest_centroid_xy(centroids, xy_point)
            #calculate the distance from the cluster center to the point
```

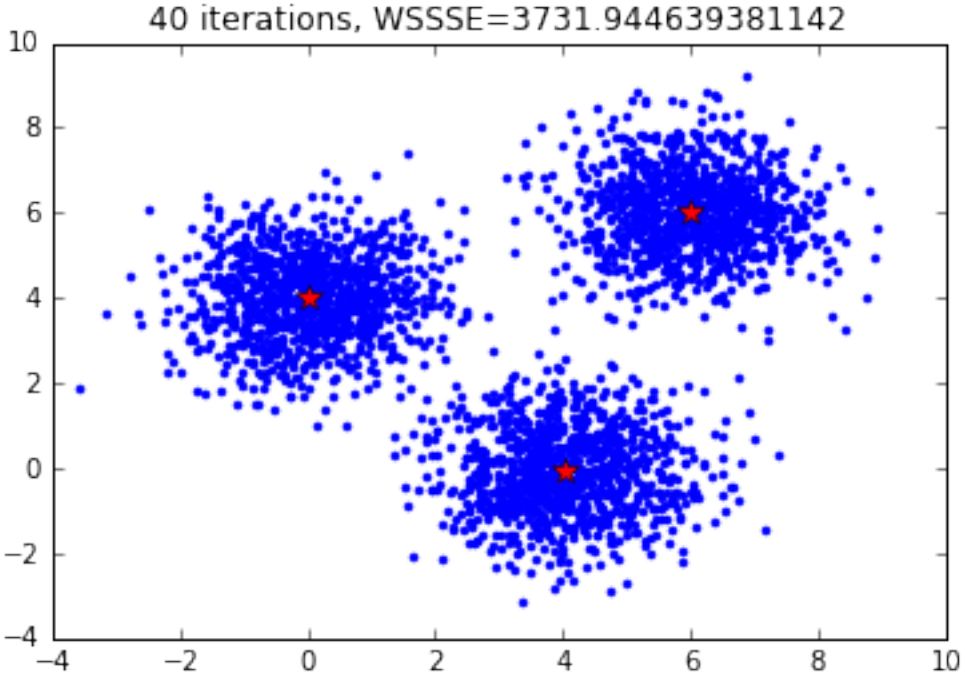
```
            WSSSE += error(center, xy_point)
        #WSSSE = D.map(lambda point: error(centroids, point)).reduce(lambda x
        title = "%s iterations, WSSSE=%s" % (iter_num, WSSSE)
        WSSSEplotter.append(WSSSE)
        plot_iteration(centroids, title)

#plot the WSSSE as a function of the iteration
pylab.plot(superIterator, WSSSEplotter, 'r--')
pylab.title("WSSSE as a function of the iteration")
pylab.show()
```

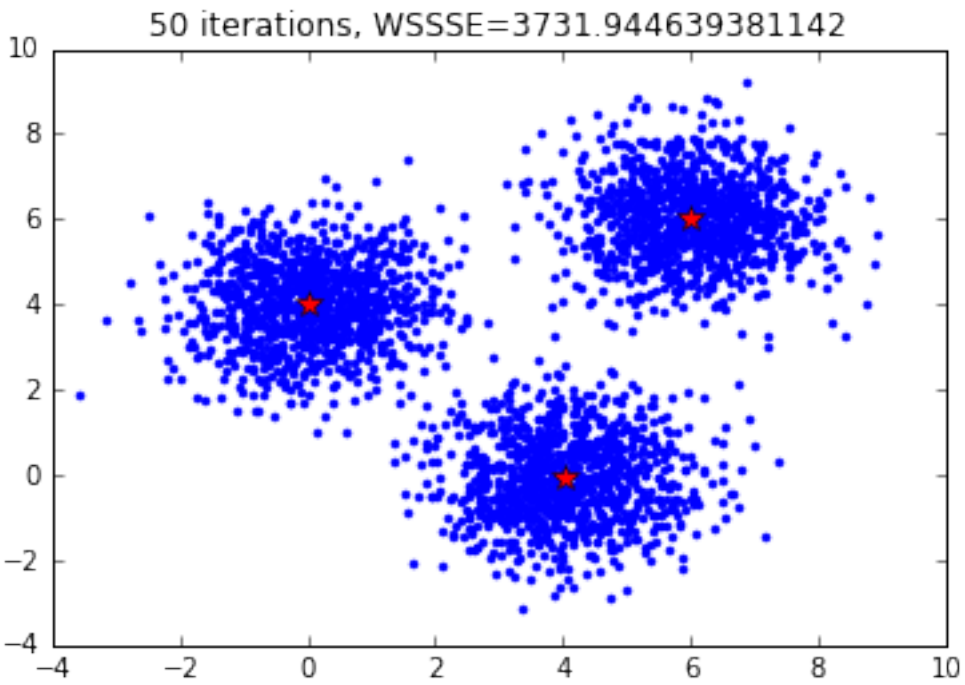


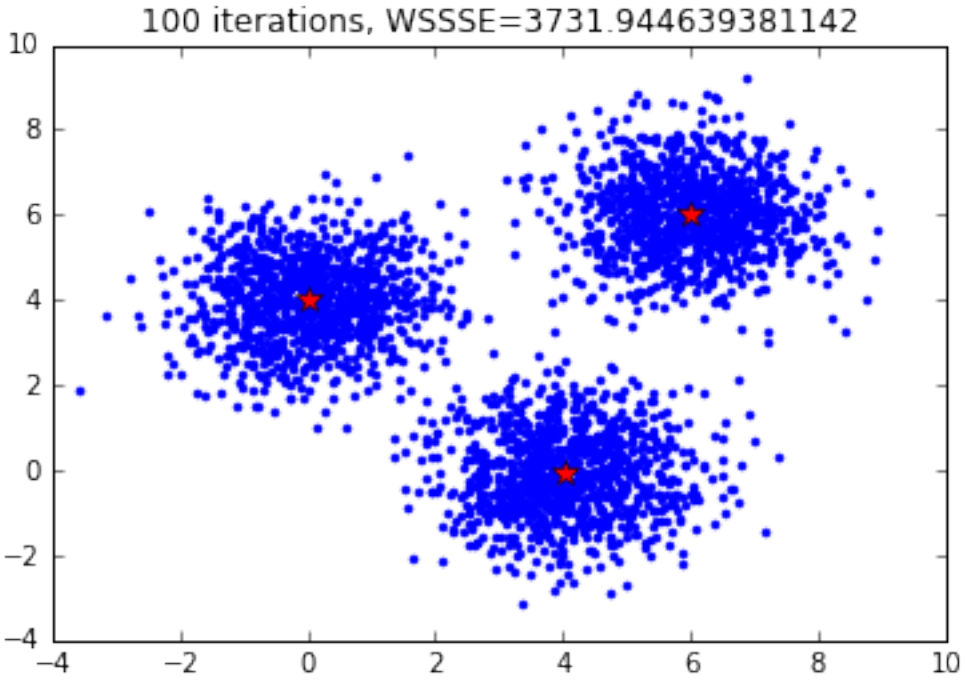

10 iterations, WSSSE=3731.944639381142
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10

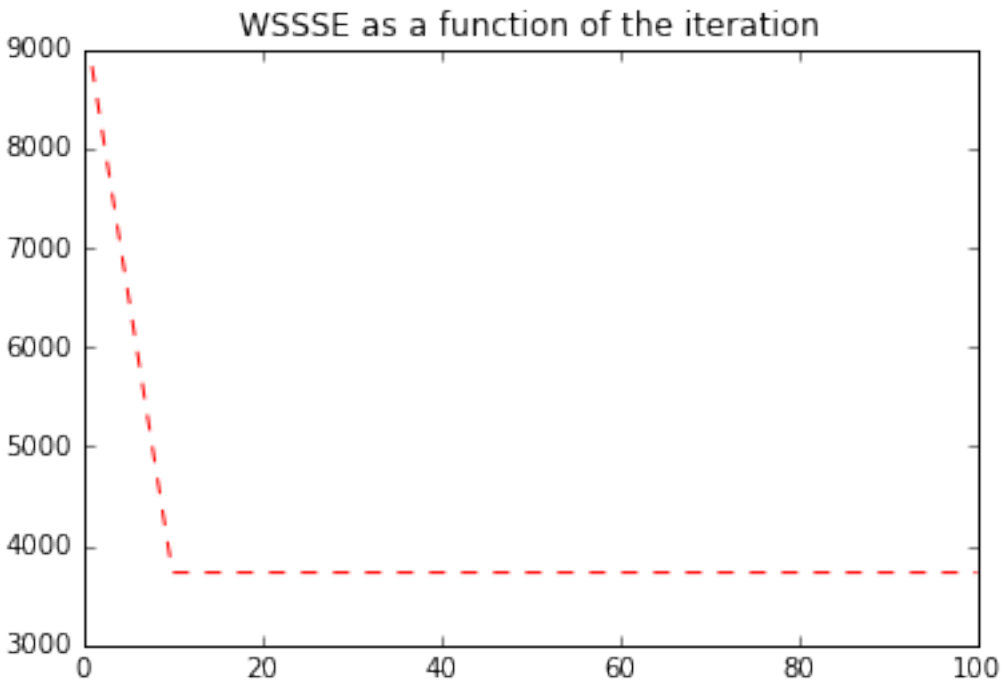

20 iterations, WSSSE=3731.944639381142
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


30 iterations, WSSSE=3731.944639381142
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


40 iterations, WSSSE=3731.944639381142
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


50 iterations, WSSSE=3731.944639381142
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


100 iterations, WSSSE=3731.944639381142
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10




```
In [95]: ## Drivers & Runners
```

```
In [96]: ## Run Scripts, S3 Sync
```

HW3.4.1: Making Homegrown KMeans more efficient

Back to Table of Contents

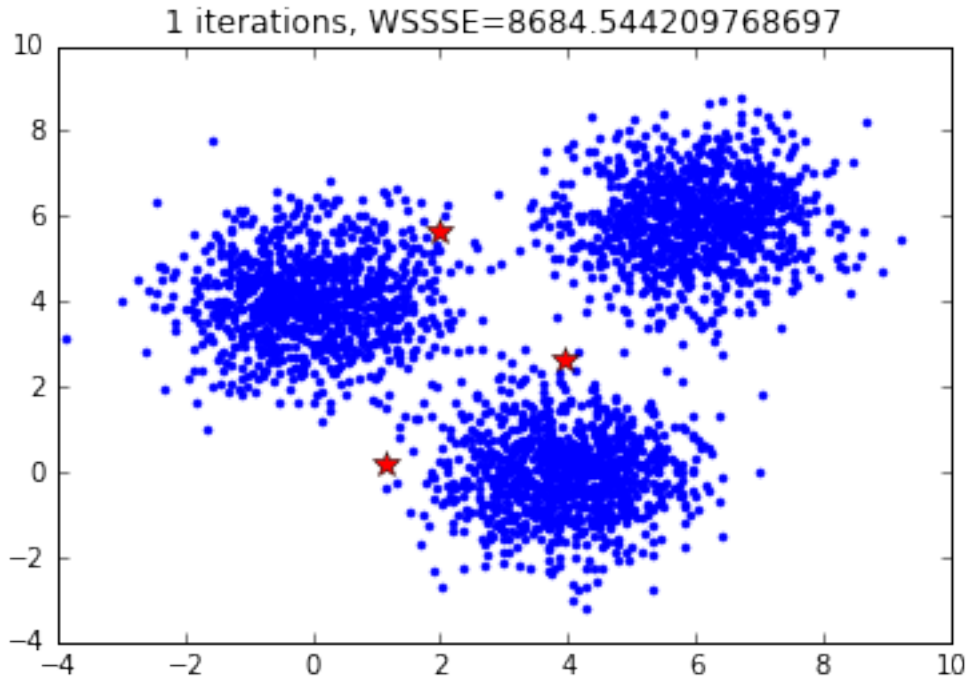
The above provided homegrown KMeans implentation in not the most efficient. How can you make it more efficient? Make this change in the code and show it work and comment on the gains you achieve.

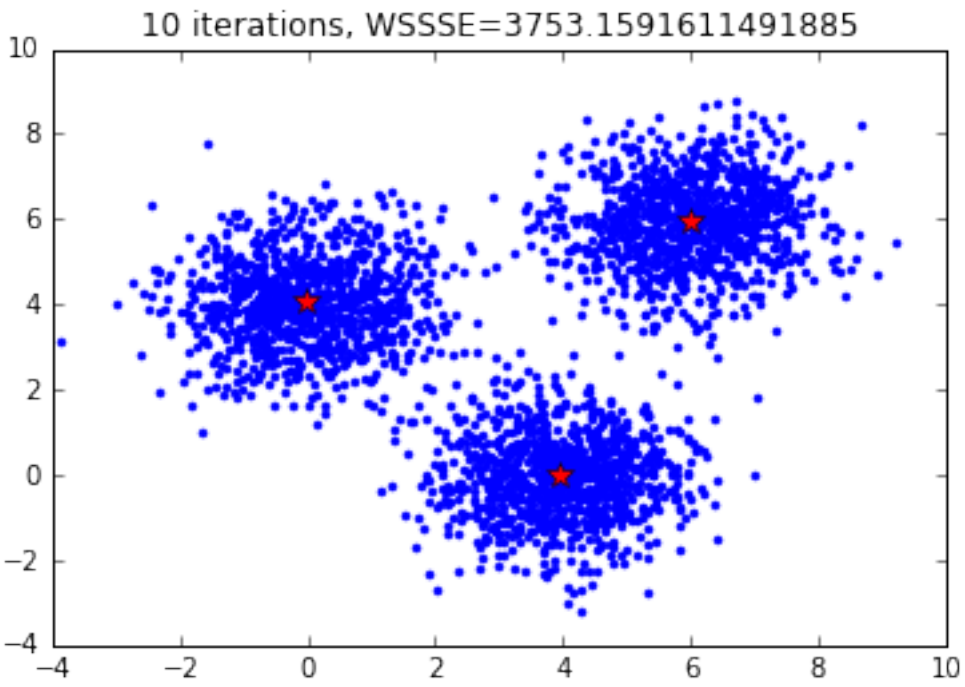
2.1.1 HINT: have a look at this linear regression notebook

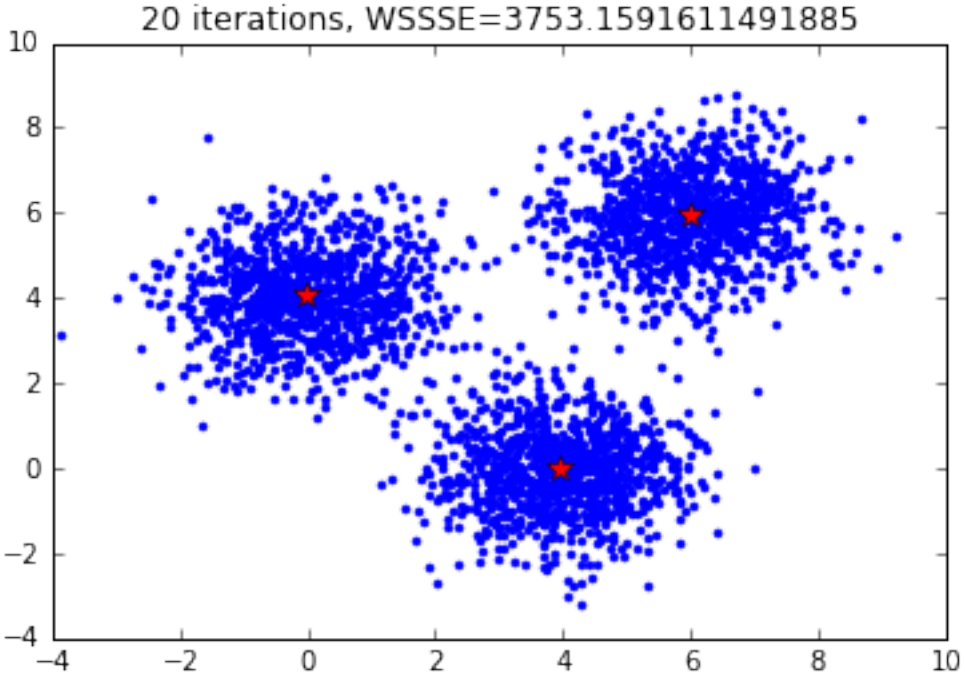
```
In [3]:  %matplotlib inline
         import numpy as np
         import pylab
         from math import sqrt
         from numpy import array
         # HW3.4.1: Making Homegrown KMeans more efficient

         #Calculate which class each data point belongs to
         def nearest_centroid(line):
             x = np.array([float(f) for f in line.split(',')])
             closest_centroid_idx = np.sum((x - centroidBC.value)**2, axis=1).argmir
             return (closest_centroid_idx,(x,1))
```

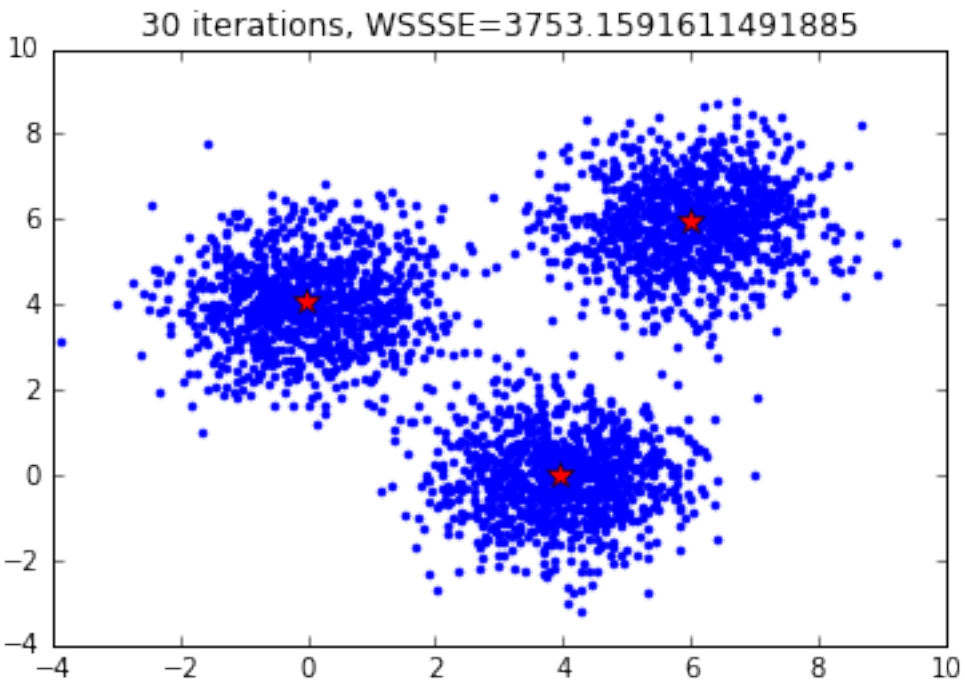
```
def closest_centroid_xy(point):
    closest_centroid_idx = np.sum((point - centroidBC.value)**2, axis=1).ar
    #the return is a single centroid that is closest to the point
    return ((centroidBC.value)[closest_centroid_idx])

#plot centroids and data points for each iteration
def plot_iteration(means, title):
    pylab.plot(samples1[:, 0], samples1[:, 1], '.', color = 'blue')
    pylab.plot(samples2[:, 0], samples2[:, 1], '.', color = 'blue')
    pylab.plot(samples3[:, 0], samples3[:, 1],'.', color = 'blue')
    pylab.plot(means[0][0], means[0][1],'*',markersize =10,color = 'red')
    pylab.plot(means[1][0], means[1][1],'*',markersize =10,color = 'red')
    pylab.plot(means[2][0], means[2][1],'*',markersize =10,color = 'red')
    pylab.title(title)
    pylab.show()

# Evaluate clustering by computing Within Set Sum of Squared Errors
def error(center, point):
    #center = centroids_new
    return sqrt(sum([x**2 for x in (point - center)]))

# Load and parse the data
# NOTE  kmeans_data.txt is available here
#          https://www.dropbox.com/s/q85t0ytb9apggnh/kmeans_data.txt?dl=0
superIterator = [1, 10, 20, 30, 40, 50, 100]
WSSSEplotter = []
D = sc.textFile("./data.csv").cache()


K = 3
# Initialization: initialization of parameter is fixed to show an example
centroids = np.array([[0.0,0.0],[2.0,2.0],[0.0,7.0]])
centroidBC = sc.broadcast(centroids)

for iterMax in superIterator:
    iter_num = 0
    for i in range(iterMax):
        res = D.map(nearest_centroid).reduceByKey(lambda x,y : (x[0]+y[0],x
        res = sorted(res,key = lambda x : x[0])  #sort based on clusted ID
        centroids_new = np.array([x[1][0]/x[1][1] for x in res])  #divide k
        centroids = centroids_new
        centroidBC = sc.broadcast(centroids)
        iter_num = iter_num + 1

        WSSSE = 0
        for k in D.map(lambda line: line.split(",")).collect():
            #creating a single point array directly from the file
            xy_point=np.array([np.float(k[0]), np.float(k[1])])
```

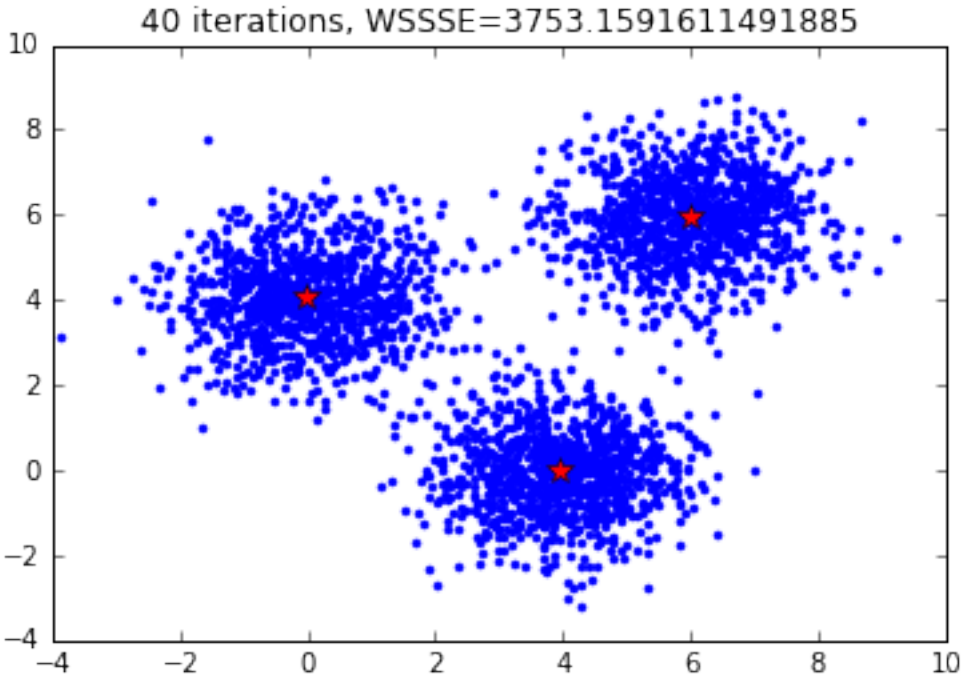
```
            #determine the closest cluster center to a point
            center = closest_centroid_xy(xy_point)
            #calculate the distance from the cluster center to the point
            WSSSE += error(center, xy_point)
    #WSSSE = D.map(lambda point: error(centroids, point)).reduce(lambda x,
    title = "%s iterations, WSSSE=%s" % (iter_num, WSSSE)
    WSSSEplotter.append(WSSSE)
    plot_iteration(centroids, title)

#plot the WSSSE as a function of the iteration
pylab.plot(superIterator, WSSSEplotter, 'r--')
pylab.title("WSSSE as a function of the iteration")
pylab.show()
```

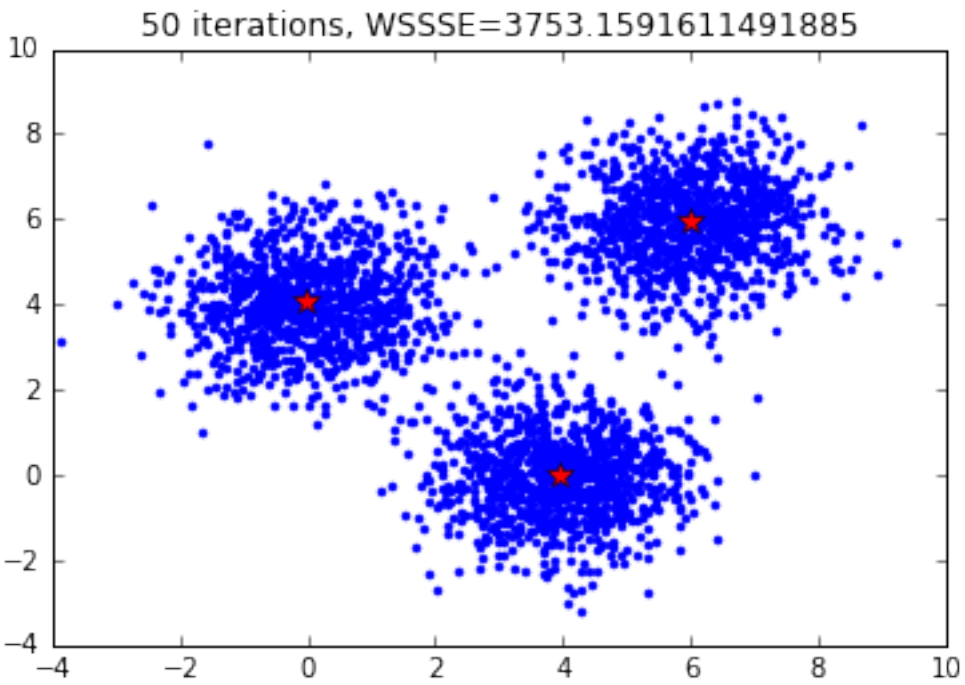


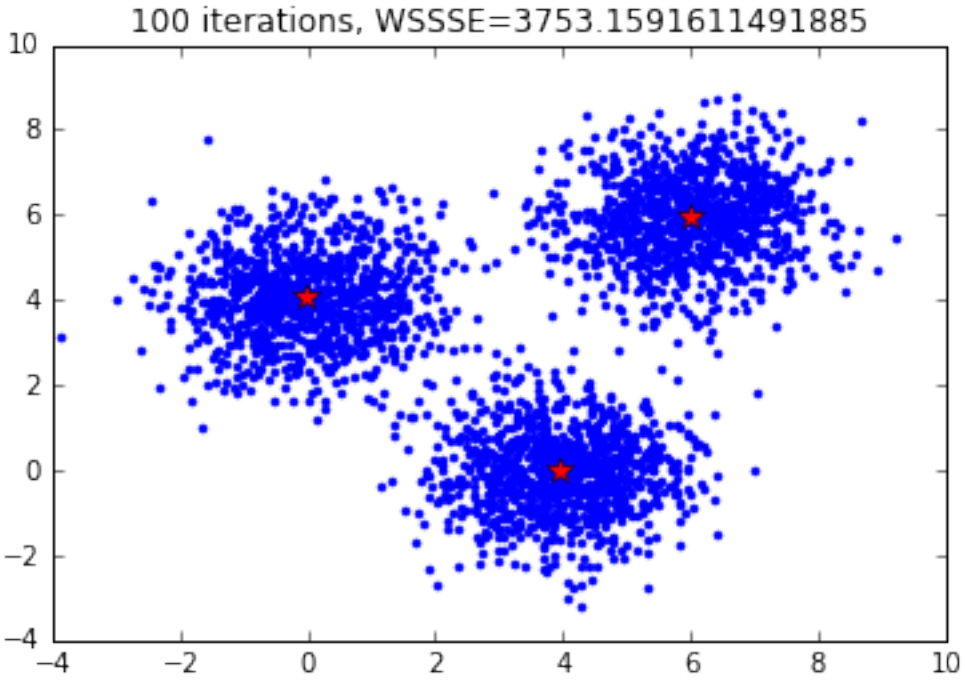

10 iterations, WSSSE=3753.1591611491885
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10

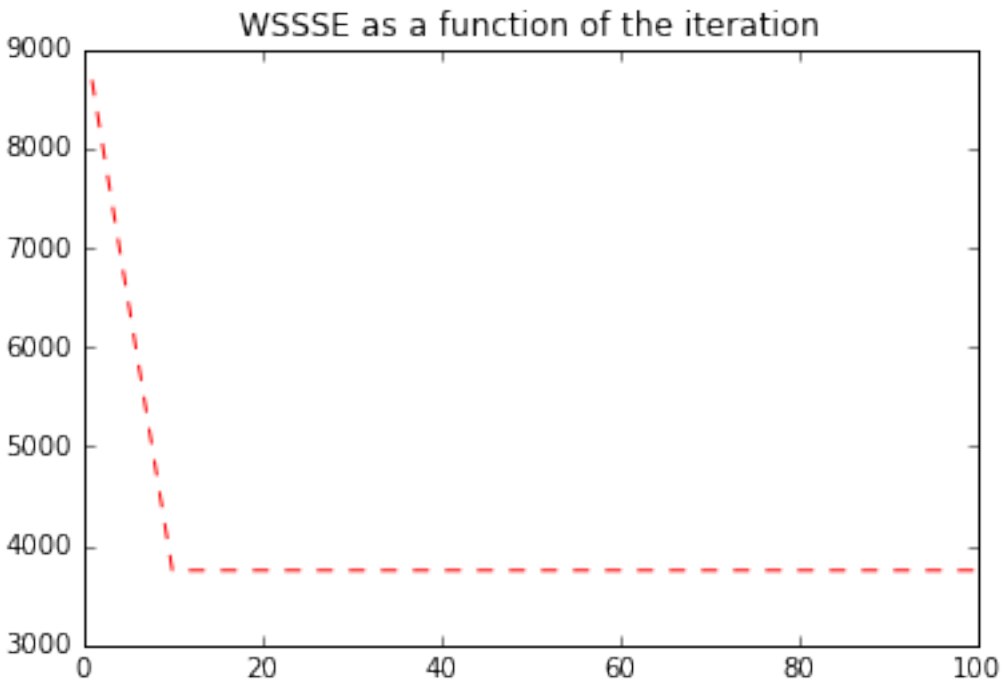

20 iterations, WSSSE=3753.1591611491885
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


30 iterations, WSSSE=3753.1591611491885
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


40 iterations, WSSSE=3753.1591611491885
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


50 iterations, WSSSE=3753.1591611491885
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


100 iterations, WSSSE=3753.1591611491885
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10



By broadcasting the centroids, we do not have to rely on encapsulation. . . speeding things up by avoiding unnecessary use of bandwidth.

```
In [99]: ## Run Scripts, S3 Sync
```

HW3.5: OPTIONAL Weighted KMeans

Back to Table of Contents

Using this provided homegrown Kmeans code, modify it to do a weighted KMeans and repeat the experiements in HW3.3. Explain any differences between the results in HW3.3 and HW3.5.

NOTE: Weight each example as follows using the inverse vector length (Euclidean norm):

weight(X)= 1/||X||,

where ||X|| = SQRT(X.X)= SQRT(X1^2 + X2^2)

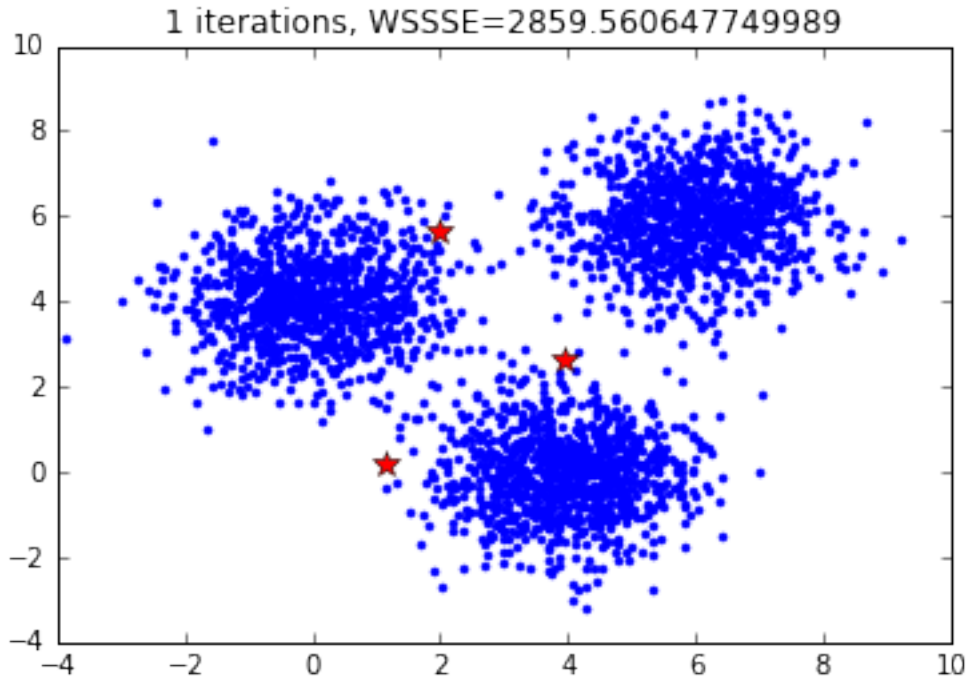
Here X is vector made up of two values X1 and X2.

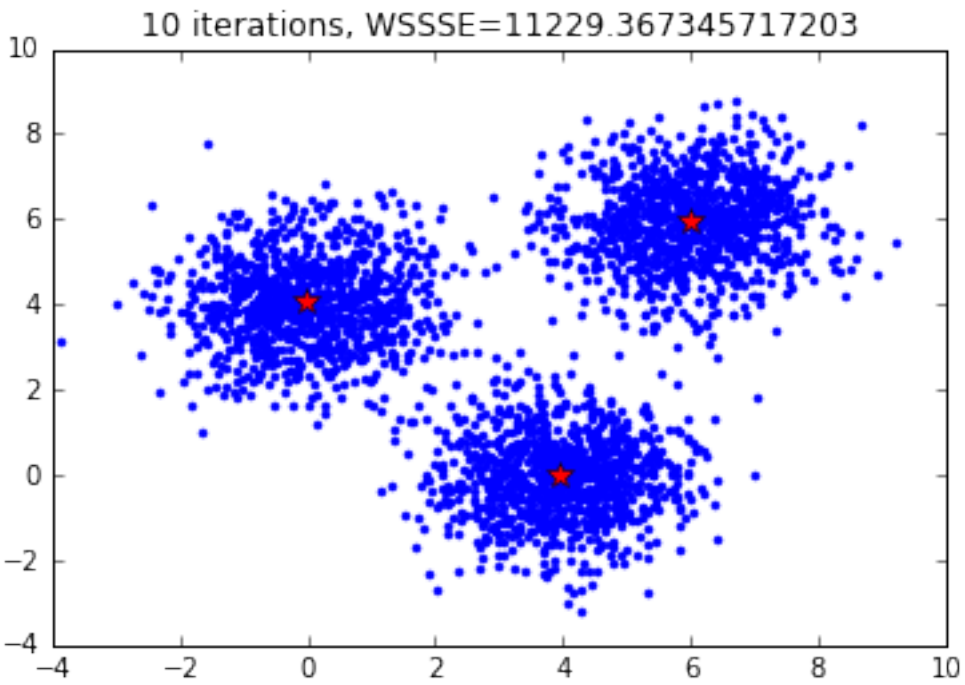
[Please incorporate all referenced notebooks directly into this master notebook as cells for HW submission. I.e., HW submissions should comprise of just one notebook]

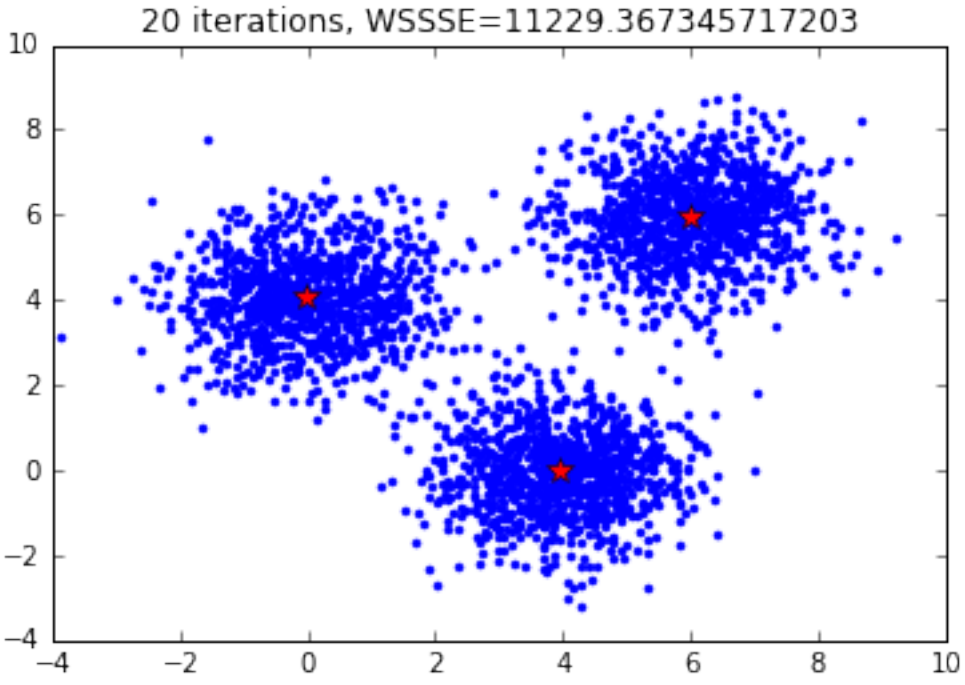
```
In [9]:  %matplotlib inline
         import numpy as np
         import pylab
         from math import sqrt
         from numpy import array
         # HW3.5: OPTIONAL Weighted KMeans

         #Calculate which class each data point belongs to
```

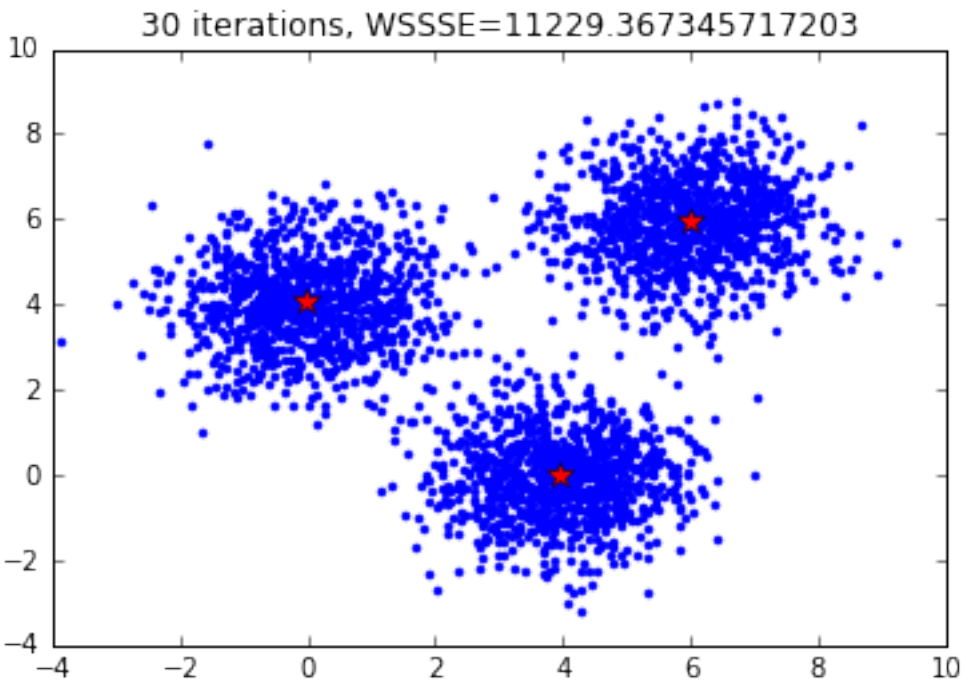
```
def nearest_centroid(line):
    x = np.array([float(f) for f in line.split(',')])
    closest_centroid_idx = np.sum((x - centroidBC.value)**2, axis=1).argmin
    return (closest_centroid_idx,(x,1))

def closest_centroid_xy(point):
    closest_centroid_idx = np.sum((point - centroidBC.value)**2, axis=1).ar
    #the return is a single centroid that is closest to the point
    return ((centroidBC.value)[closest_centroid_idx])

#plot centroids and data points for each iteration
def plot_iteration(means, title):
    pylab.plot(samples1[:, 0], samples1[:, 1], '.', color = 'blue')
    pylab.plot(samples2[:, 0], samples2[:, 1], '.', color = 'blue')
    pylab.plot(samples3[:, 0], samples3[:, 1],'.', color = 'blue')
    pylab.plot(means[0][0], means[0][1],'*',markersize =10,color = 'red')
    pylab.plot(means[1][0], means[1][1],'*',markersize =10,color = 'red')
    pylab.plot(means[2][0], means[2][1],'*',markersize =10,color = 'red')
    pylab.title(title)
    pylab.show()

# Evaluate clustering by computing Within Set Sum of Squared Errors
def error(center, point):
    #center = centroids_new
    return sqrt(sum([x**2 for x in (point - center)]))

def weightweightdonttellme(xy_point):
    return 1/(sqrt(xy_point[0]**2 + xy_point[1]**2))

# Load and parse the data
# NOTE  kmeans_data.txt is available here
#          https://www.dropbox.com/s/q85t0ytb9apggnh/kmeans_data.txt?dl=0
superIterator = [1, 10, 20, 30, 40, 50, 100]
WSSSEplotter = []
D = sc.textFile("./data.csv").cache()


K = 3
# Initialization: initialization of parameter is fixed to show an example
centroids = np.array([[0.0,0.0],[2.0,2.0],[0.0,7.0]])
centroidBC = sc.broadcast(centroids)

for iterMax in superIterator:
    iter_num = 0
    for i in range(iterMax):
        res = D.map(nearest_centroid).reduceByKey(lambda x,y : (x[0]+y[0],x
        res = sorted(res,key = lambda x : x[0])  #sort based on clusted ID
        centroids_new = np.array([x[1][0]/x[1][1] for x in res])  #divide b
```

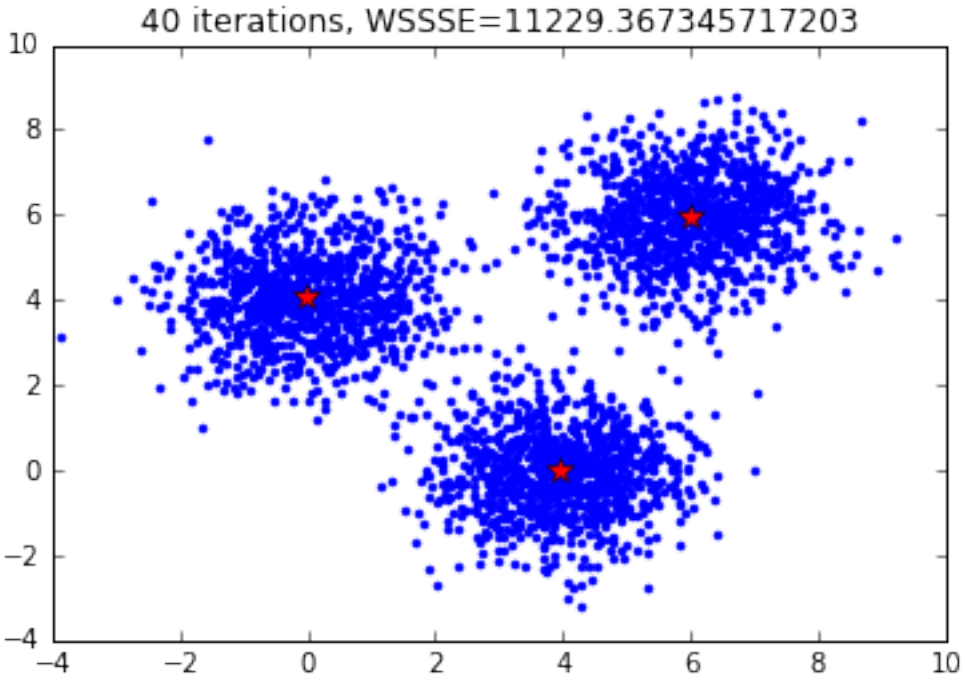
```
            centroids = centroids_new
            centroidBC = sc.broadcast(centroids)
            iter_num = iter_num + 1

            WSSSE = 0
            for k in D.map(lambda line: line.split(",")).collect():
                #creating a single point array directly from the file
                xy_point=np.array([np.float(k[0]), np.float(k[1])])
                xy_point=weightweightdonttellme(xy_point)
                #determine the closest cluster center to a point
                center = closest_centroid_xy(xy_point)
                #calculate the distance from the cluster center to the point
                WSSSE += error(center, xy_point)
        #WSSSE = D.map(lambda point: error(centroids, point)).reduce(lambda x,
        title = "%s iterations, WSSSE=%s" % (iter_num, WSSSE)
        WSSSEplotter.append(WSSSE)
        plot_iteration(centroids, title)

#plot the WSSSE as a function of the iteration
pylab.plot(superIterator, WSSSEplotter, 'r--')
pylab.title("WSSSE as a function of the iteration")
pylab.show()
```

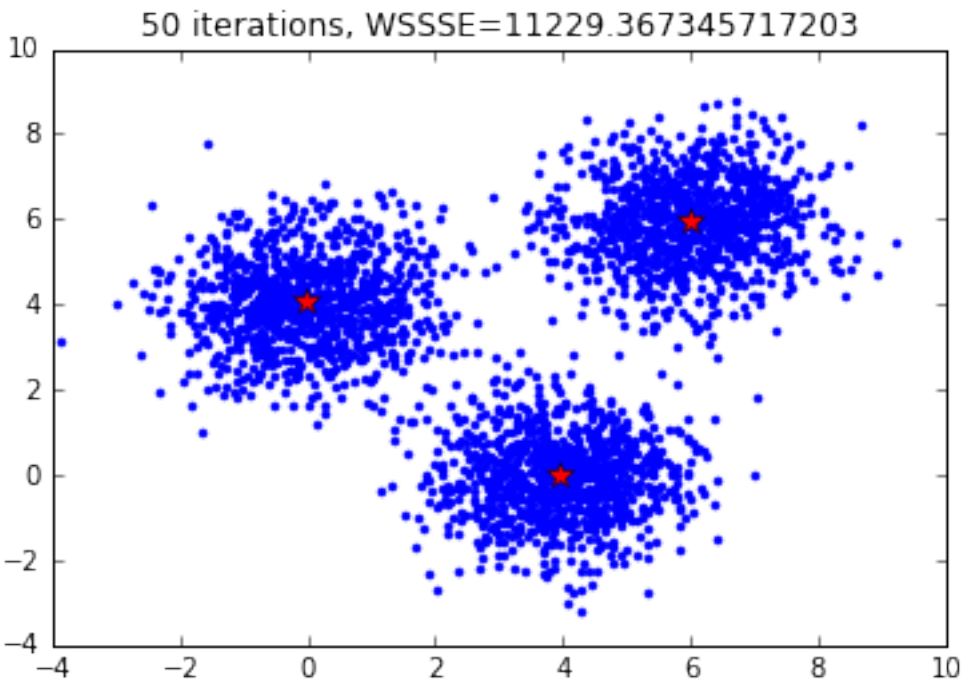


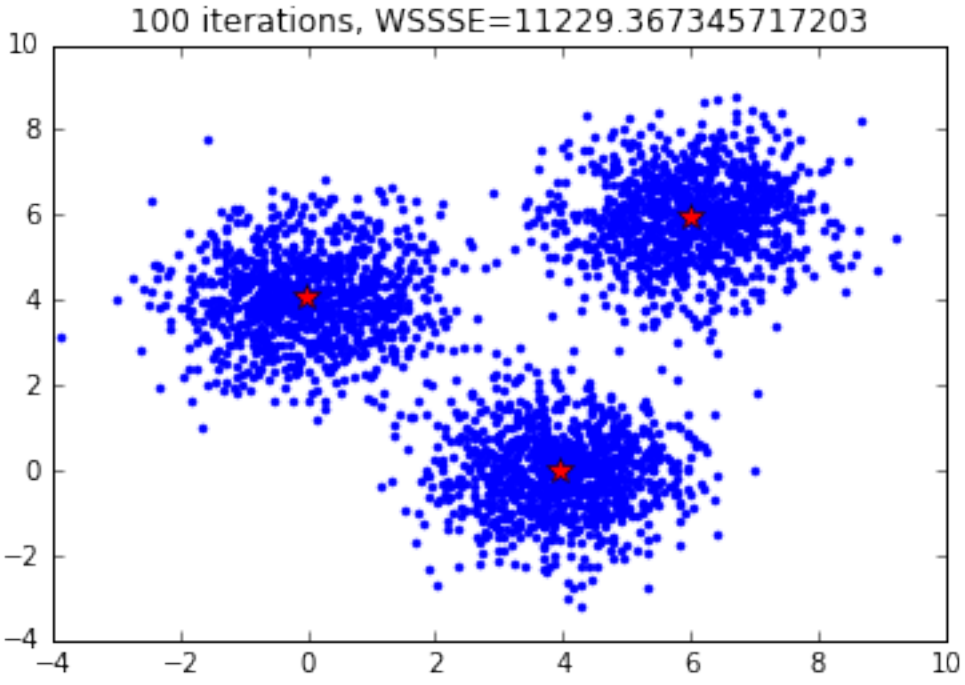

10 iterations, WSSSE=11229.367345717203
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10

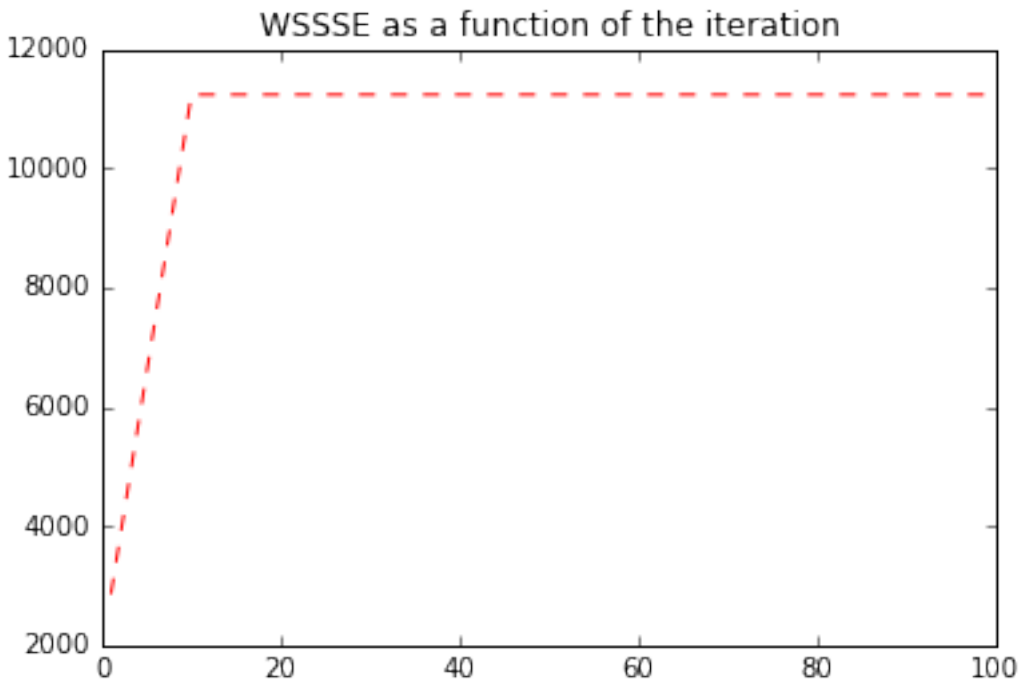

20 iterations, WSSSE=11229.367345717203
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


30 iterations, WSSSE=11229.367345717203
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


40 iterations, WSSSE=11229.367345717203
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


50 iterations, WSSSE=11229.367345717203
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10


100 iterations, WSSSE=11229.367345717203
10
8
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8
10



Our WSSSE function gets inverted as a result of the weighting, although the number of iterations required to achieve stability is the same

```
In [102]: ## Run Scripts, S3 Sync
```

HW3.6 OPTIONAL Linear Regression
Back to Table of Contents
HW3.6.1 OPTIONAL Linear Regression
Back to Table of Contents
Using this linear regression notebook:

- Generate 2 sets of data with 100 data points using the data generation code provided and plot each in separate plots. Call one the training set and the other the testing set.
- Using MLLib's LinearRegressionWithSGD train up a linear regression model with the training dataset and evaluate with the testing set. What a good number of iterations for training the linear regression model? Justify with plots (e.g., plot MSE as a function of the number of iterations) and words.

HW3.6.2 OPTIONAL Linear Regression
Back to Table of Contents
In the notebook provided above, in the cell labeled "Gradient descent (regularization)".

- Fill in the blanks and get this code to work for LASS0 and RIDGE linear regression.
- Using the data from 3.6.1 tune the hyper parameters of your LASS0 and RIDGE regression. Report your findings with words and plots.

```
In [103]: ## Code goes here

In [104]: ## Drivers & Runners

In [105]: ## Run Scripts, S3 Sync
```

HW3.7 OPTIONAL Error surfaces

Back to Table of Contents

Here is a link to R code with 1 test drivers that plots the linear regression model in model space and in the domain space:

https://www.dropbox.com/s/3xc3kwda6d254l5/PlotModelAndDomainSpaces.R?dl=0

Here is a sample output from this script:

https://www.dropbox.com/s/my3tnhxx7fr5qs0/image%20%281%29.png?dl=0

Please use this as inspiration and code a equivalent error surface and heatmap (with isolines) in Spark and show the trajectory of learning taken during gradient descent(after each n-iterations of Gradient Descent):

Using Spark and Python (using the above R Script as inspiration), plot the error surface for the linear regression model using a heatmap and contour plot. Also plot the current model in the original domain space for every 10th iteration. Plot them side by side if possible for each iteration: lefthand side plot is the model space(w0 and w01) and the righthand side plot is domain space (plot the corresponding model and training data in the problem domain space) with a final pair of graphs showing the entire trajectory in the model and domain space. Make sure to label your plots with iteration numbers, function, model space versus original domain space, MSE on the training data etc.

Also plot the MSE as a function of each iteration (possibly every 10th iteration). Dont forget to label both axis and the graph also. **[Please incorporate all referenced notebooks directly into this master notebook as cells for HW submission. I.e., HW submissions should comprise of just one notebook]**

```
In [106]: ## Code goes here

In [107]: ## Drivers & Runners

In [108]: ## Run Scripts, S3 Sync
```

Back to Table of Contents

——- END OF HWK 9 ———