Zuehlke DAMLAS Target DS Homework - 2aug2016

August 2, 2016

1 DAMLAS - Machine Learning At Scale

1.1 Assignment - HW3

Data Analytics and Machine Learning at Scale Target, Minneapolis

---

Name: Scott Zuehlke
Class: DAMLAS Summer 2016
Email: Scott.Zuehlke@Target.com
Week: 03

2 Table of Contents

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 a general data processing and analytics framework. It can be run on either a single node or a distributed cluster, and is extremely fast as it uses in-memory computations.

By comparison, Hadoop is a distributed computing cluster. Hadoop uses MapReduce, which is great for analyzing big data, but writes data to disk, and so is much slower than Spark.

Fill in the blanks: Spark API consists of interfaces to develop applications based on it in Java, Scala, R, and Python.

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

An RDD (resilient distributed dataset) is a dataset that has distributed storage, is lazily evaluated, can be transformed, and can be either a base RDD or a pair RDD.

```
In [1]: %%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 [2]: # Find and load Spark context

        import os
        import sys

        import pyspark
        from pyspark.sql import SQLContext


        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 dateutil.parser
        import dateutil.relativedelta as dateutil_rd
```

```
<pyspark.context.SparkContext object at 0x7fceab4c6a90>
<pyspark.sql.context.SQLContext object at 0x7fceab4c6ef0>
```

Example:

```
In [3]: %%writefile funexample.txt
        Fun examples are fun
        But how fun are they?  You will not even see this line in the below cell.
```

```
Overwriting funexample.txt
```

```
In [4]: rdd = sc.textFile("funexample.txt")
        rdd.first()
```

```
Out[4]: 'Fun examples are fun'
```

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 [5]:
```
import re

hw31top20 = sc.textFile('HW3.txt') \
    .flatMap(lambda line: re.findall(r'[\w]+', line)) \
    .filter(lambda x: len(x) > 1) \
    .map(lambda x: (x, 1)) \
    .reduceByKey(lambda x, y: x + y) \
    .takeOrdered(20, key = lambda x: -x[1])
print('Most frequent 20 words')
for h in hw31top20:
    print(h)
```

```
Most frequent 20 words
('the', 71)
('of', 49)
('and', 40)
('to', 33)
('in', 32)
('HW3', 19)
('Contents', 17)
('this', 17)
('for', 17)
('with', 17)
('Table', 17)
('Back', 16)
('data', 15)
('as', 14)
('KMeans', 14)
('model', 12)
('Spark', 12)
('plot', 12)
('notebook', 12)
('code', 12)
```

In [6]:
```
import re
```

```
        hw31bottom10 = sc.textFile('HW3.txt') \
            .flatMap(lambda line: re.findall(r'[\w]+', line)) \
            .filter(lambda x: len(x) > 1) \
            .map(lambda x: (x, 1)) \
            .reduceByKey(lambda x, y: x + y) \
            .takeOrdered(10, key = lambda x: x[1])
        print('Least frequent 10 words')
        for h in hw31bottom10:
            print(h)
```

```
Least frequent 10 words
('Java', 1)
('appears', 1)
('driver', 1)
('implentation', 1)
('At', 1)
('PlotModelAndDomainSpaces', 1)
('50', 1)
('versus', 1)
('both', 1)
('numpy', 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 [7]: import re

        hw311top20 = sc.textFile('HW3.txt') \
            .flatMap(lambda line: re.findall(r'\b[a-z]\w+\b', line)) \
            .filter(lambda x: len(x) > 1) \
            .map(lambda x: (x, 1)) \
            .reduceByKey(lambda x, y: x + y) \
            .takeOrdered(20, key = lambda x: -x[1])
        print('Most frequent 20 words')
        for h in hw311top20:
            print(h)
```

```
Most frequent 20 words
('the', 71)
('of', 49)
('and', 40)
('to', 33)
('in', 32)
('this', 17)
('for', 17)
('with', 17)
```

```
('data', 15)
('as', 14)
('model', 12)
('plot', 12)
('here', 12)
('notebook', 12)
('code', 12)
('is', 10)
('goes', 9)
('clusters', 9)
('space', 9)
('regression', 8)
```

```
In [8]: import re

        hw311bottom10 = sc.textFile('HW3.txt') \
            .flatMap(lambda line: re.findall(r'\b[a-z]\w+\b', line)) \
            .filter(lambda x: len(x) > 1) \
            .map(lambda x: (x, 1)) \
            .reduceByKey(lambda x, y: x + y) \
            .takeOrdered(10, key = lambda x: x[1])
        print('Least frequent 10 words')
        for h in hw311bottom10:
            print(h)
```

```
Least frequent 10 words
('equations', 1)
('appears', 1)
('prediction', 1)
('driver', 1)
('implentation', 1)
('regularization', 1)
('axis', 1)
('make', 1)
('form', 1)
('color', 1)
```

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 [9]: !curl -L https://www.dropbox.com/s/q85t0ytb9apggnh/kmeans_data.txt?dl=0 -o
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:--  0:00:06 --:--:--     0
100    72  100    72    0     0      7      0  0:00:10  0:00:09  0:00:01    96
```

```
In [10]: from pyspark.mllib.clustering import KMeans, KMeansModel
         from numpy import array
         from math import sqrt


         # 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_32 = parsedData.map(lambda point: error(point)).reduce(lambda x, y:
         print("WSSSE = " + str(WSSSE_32))
         print("Centroids = " + str(clusters.centers))
```

```
/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
```

```
WSSSE = 0.6928203230275529
Centroids = [array([ 0.1,  0.1,  0.1]), array([ 9.1,  9.1,  9.1])]
```

.69 is a small WSSSE, so this would indicate the different cluster points are close to their respective centroids without much spread.

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:

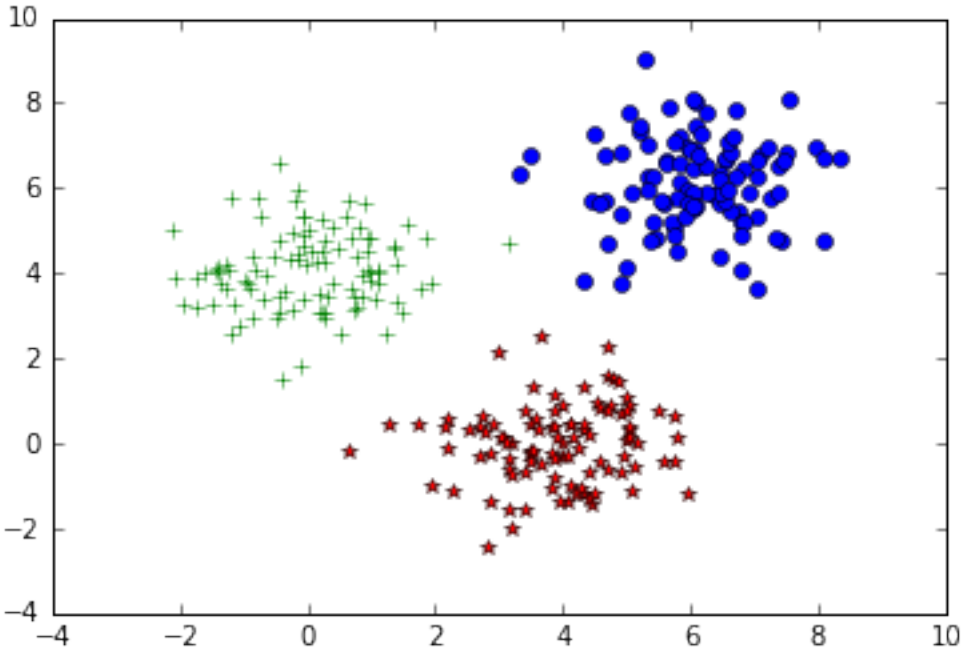
- plot the resulting clusters after 1 iteration, 10 iterations, after 20 iterations, after 100 iterations.

- 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 [11]:
```
%matplotlib inline
import numpy as np
import pylab
import json
size1 = size2 = size3 = 100
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 = ',')
```

```
/opt/conda/lib/python3.5/site-packages/matplotlib/font_manager.py:273: UserWarning:
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take
/opt/conda/lib/python3.5/site-packages/matplotlib/font_manager.py:273: UserWarning:
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take
```

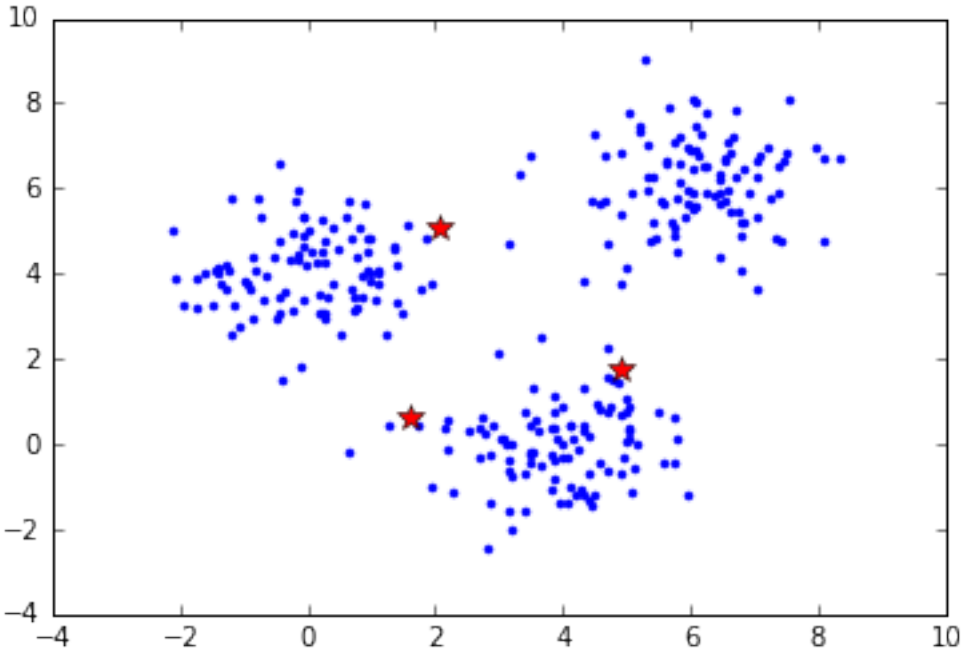
In [12]:
```
pylab.plot(samples1[:, 0], samples1[:, 1],'*', color = 'red')
pylab.plot(samples2[:, 0], samples2[:, 1],'o',color = 'blue')
pylab.plot(samples3[:, 0], samples3[:, 1],'+',color = 'green')
pylab.show()
```



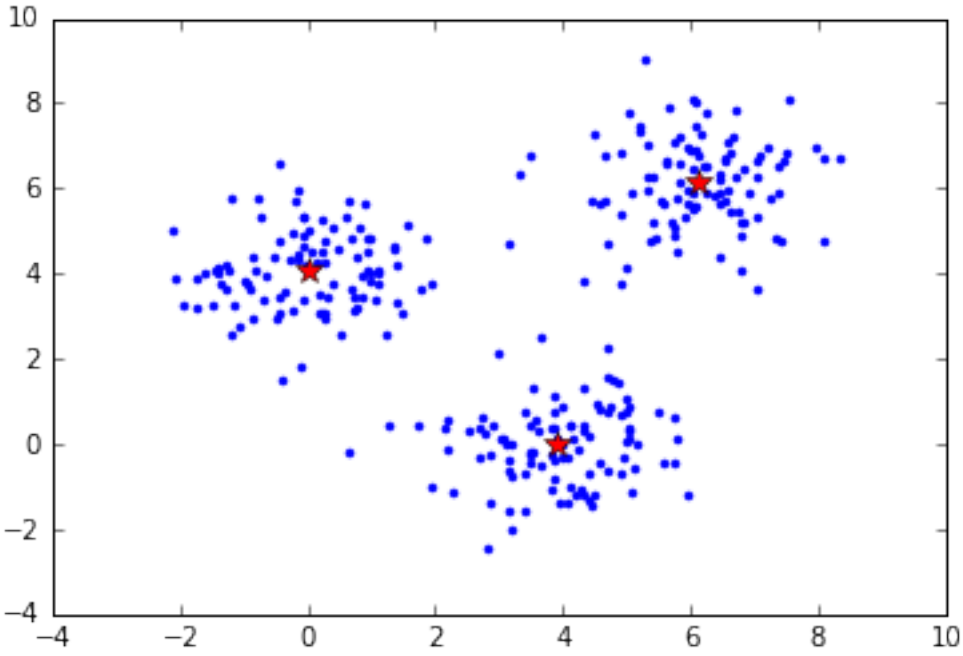
In [13]:
```
import numpy as np
import pylab as plt
from numpy import array
from math import sqrt
from pyspark.mllib.clustering import KMeans, KMeansModel

def errorHW33(point):
    center = clustersHW33.centers[clustersHW33.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))

#plot centroids and data points for each iteration
def plot_iterationHW33(means):
    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.show()

iterationsHW33 = [1, 10, 20, 30, 40, 50, 100]
for_plotHW33 = []

dataHW33 = sc.textFile("data.csv") \
    .map(lambda line: array([float(x) for x in line.split(',')])) \
    .cache()

for iteration in iterationsHW33:
    # Build the model (cluster the data)
    clustersHW33 = KMeans.train(dataHW33, 3, maxIterations = iteration, ru

# Evaluate clustering by computing Within Set Sum of Squared Errors

    WSSSE_HW33 = dataHW33.map(lambda point: errorHW33(point)).reduce(lambd


    if iteration == 1 or iteration == 10 or iteration == 20 or iteration =
    iteration == 40 or iteration == 50 or iteration == 100:
        print("Iteration number: " + str(iteration))
        print("WSSSE is " + str(WSSSE_HW33) + " after " + str(iteration) +
        print("Cluster Centers after " + str(iteration) + " iterations:")
        print(clustersHW33.centers[0], clustersHW33.centers[1], clustersHW
        plot_iterationHW33(clustersHW33.centers)
        for_plotHW33.insert(iteration,WSSSE_HW33)
```

```
        plt.plot(iterationsHW33, for_plotHW33)
        plt.title("WSSSE as function of iterations for MLLib Kmeans")
        plt.xlabel("Iterations")
        plt.ylabel("WSSSE")
        plt.show()
```

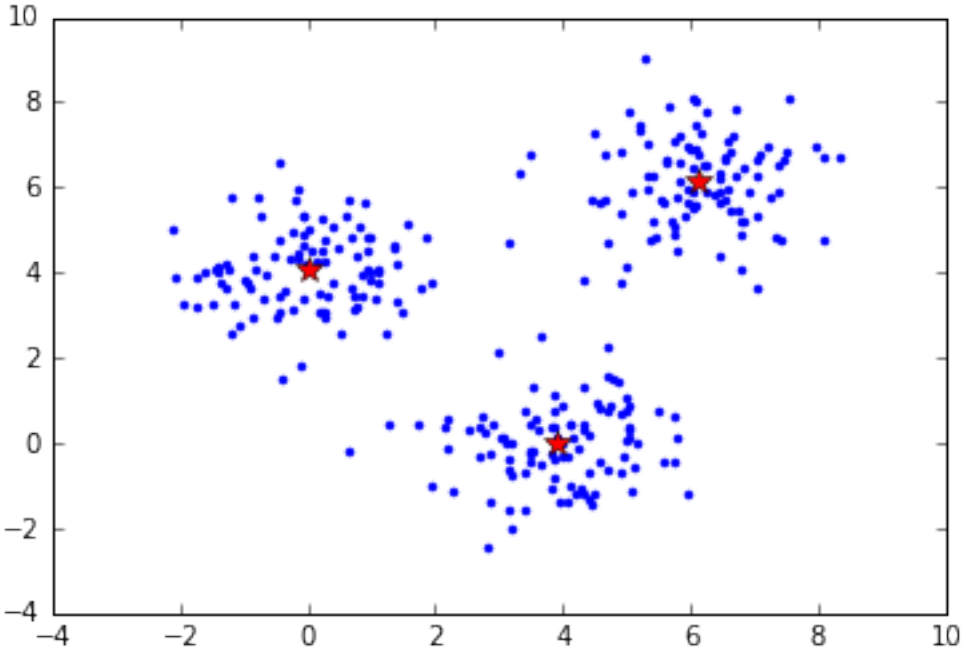
```
Iteration number: 1
WSSSE is 828.7723002782129 after 1 iterations.
Cluster Centers after 1 iterations:
[ 2.07821643  5.08484706] [ 1.58616974  0.60221601] [ 4.92940317  1.78153253]
```



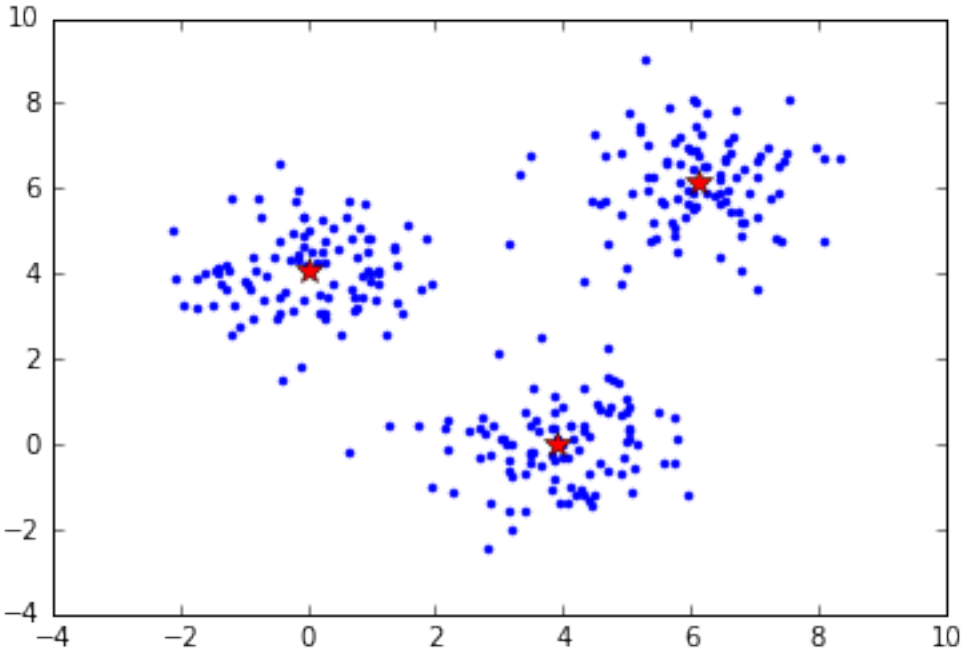

```
Iteration number: 10
WSSSE is 369.60125446154785 after 10 iterations.
Cluster Centers after 10 iterations:
[ 0.01632869  4.08348349] [ 6.1013494   6.14437711] [ 3.90210723 -0.01553607]
```



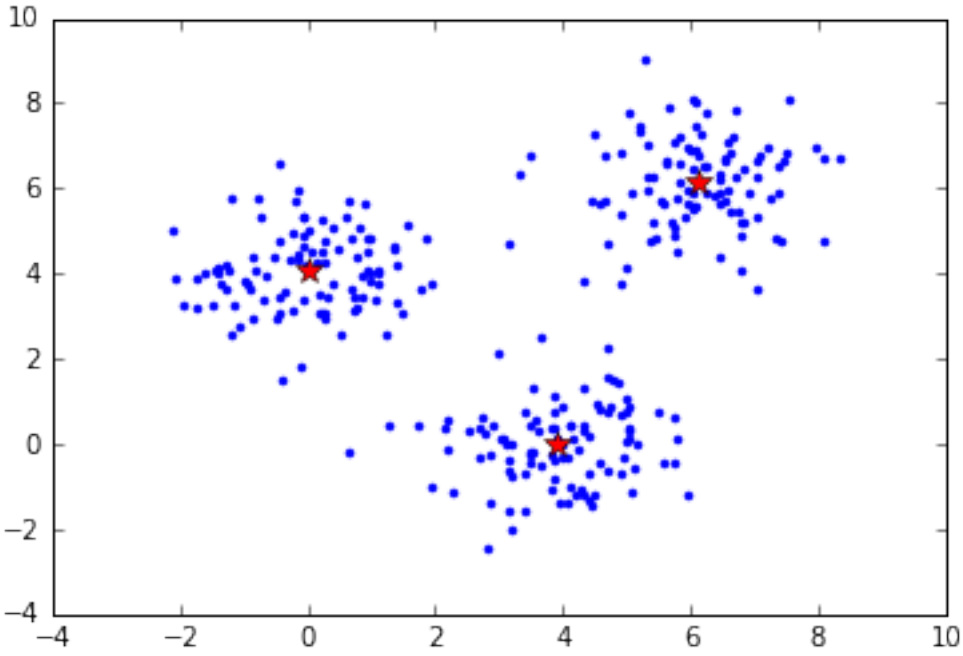

```
Iteration number: 20
WSSSE is 369.60125446154785 after 20 iterations.
Cluster Centers after 20 iterations:
[ 3.90210723 -0.01553607] [ 0.01632869  4.08348349] [ 6.1013494   6.14437711]
```



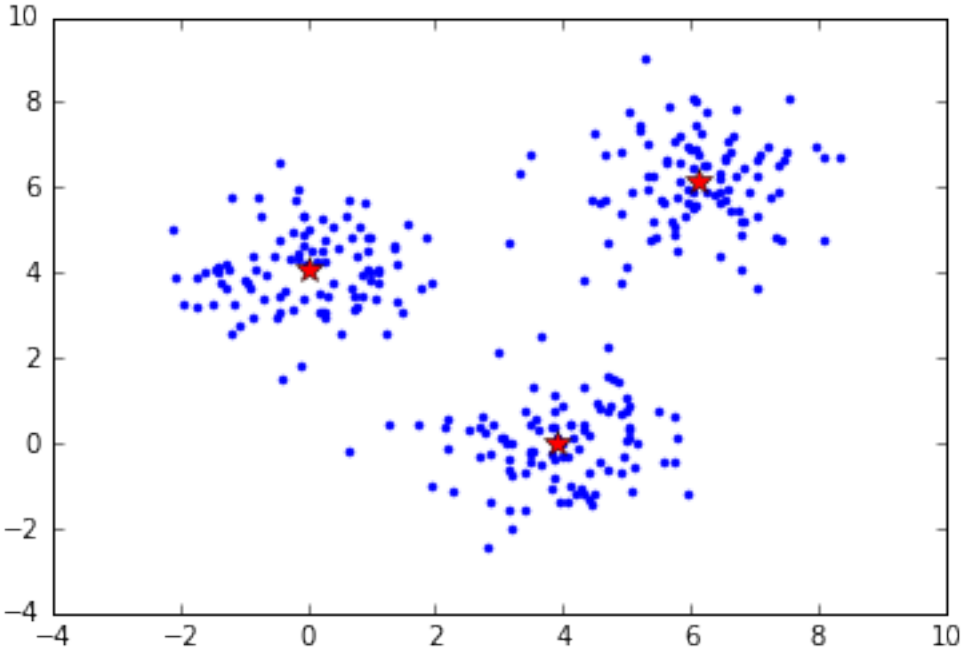
```
Iteration number: 30
WSSSE is 369.60125446154785 after 30 iterations.
Cluster Centers after 30 iterations:
[ 6.1013494   6.14437711] [ 0.01632869  4.08348349] [ 3.90210723 -0.01553607]
```



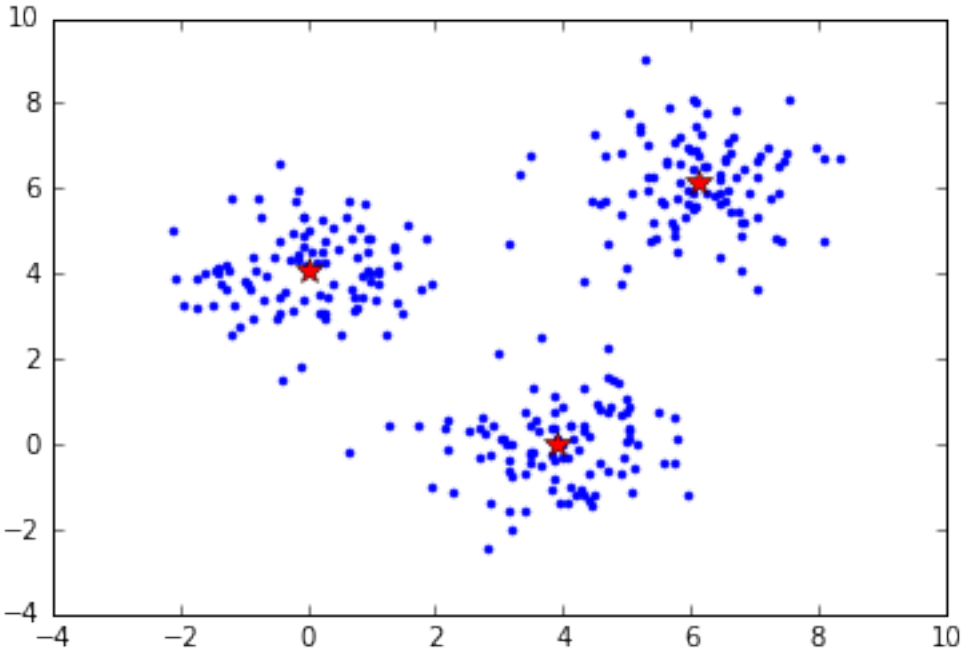

```
Iteration number: 40
WSSSE is 369.60125446154785 after 40 iterations.
Cluster Centers after 40 iterations:
[ 0.01632869  4.08348349] [ 6.1013494   6.14437711] [ 3.90210723 -0.01553607]
```

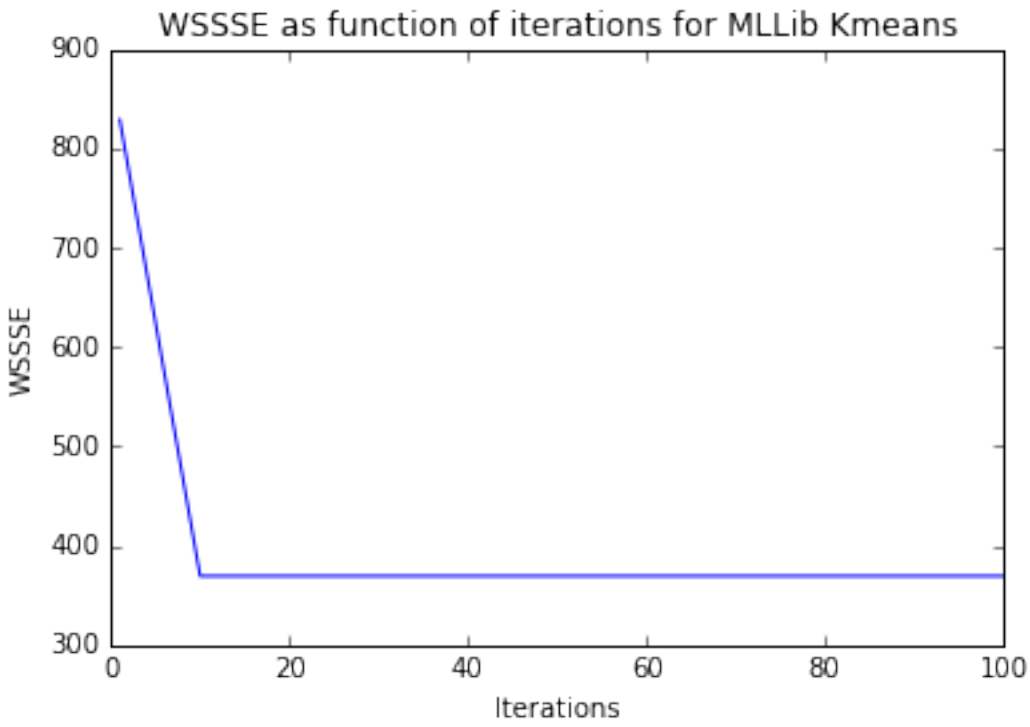



```
Iteration number: 50
WSSSE is 369.60125446154785 after 50 iterations.
Cluster Centers after 50 iterations:
[ 3.90210723 -0.01553607] [ 6.1013494   6.14437711] [ 0.01632869  4.08348349]
```



```
Iteration number: 100
WSSSE is 369.60125446154785 after 100 iterations.
Cluster Centers after 100 iterations:
[ 3.90210723 -0.01553607] [ 6.1013494   6.14437711] [ 0.01632869  4.08348349]
```





HW3.4: KMeans Experiments

Back to Table of Contents

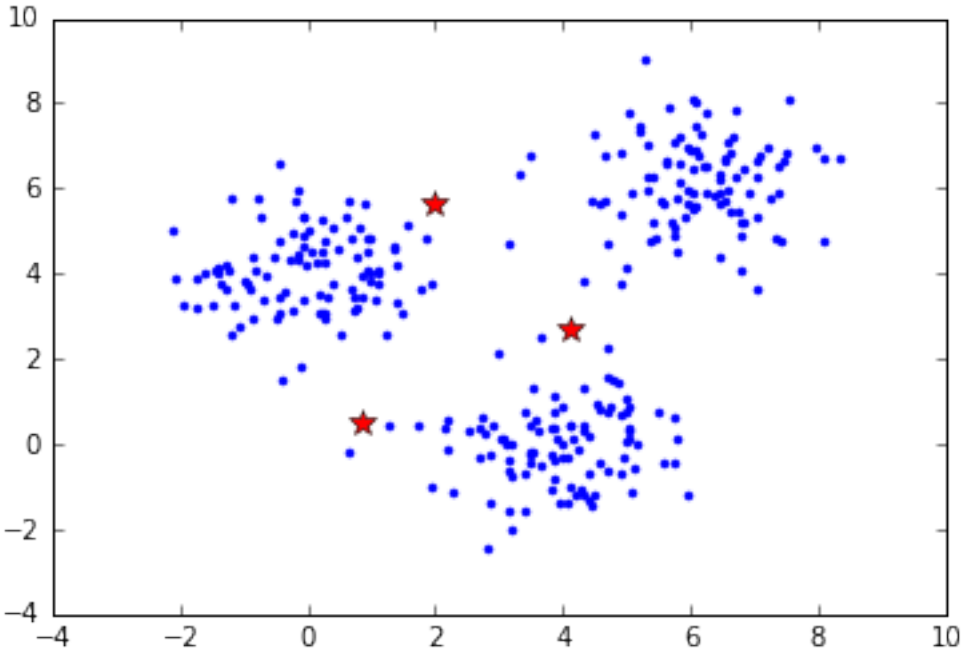
Using this provided homegrown Kmeans code repeat the experiments in HW3.3. Explain any differences between the results in HW3.3 and HW3.4.

```
In [14]: from timeit import default_timer as timer

         start = timer()

         import numpy as np

         #Calculate which class each data point belongs to
         def nearest_centroidHW34(line):
             x = np.array([float(f) for f in line.split(',')])
             closest_centroid_idxHW34 = np.sum((x - centroidsHW34)**2, axis=1).argm
             return (closest_centroid_idxHW34,(x,1))

         def errorHW34(line):
             x = np.array([float(f) for f in line.split(',')])
             closest_centroid_idxHW34 = np.sum((x - centroidsHW34)**2, axis=1).argm
             errorHW34 =  sqrt(np.sum((x-centroidsHW34[closest_centroid_idxHW34])**
             return (closest_centroid_idxHW34, errorHW34)

         #plot centroids and data points for each iteration
```

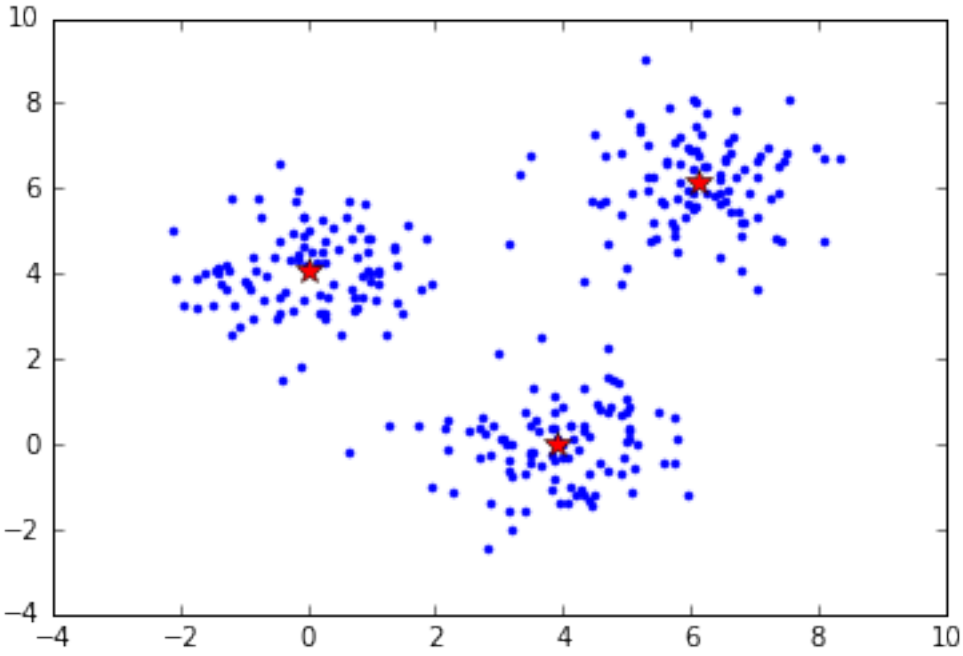
```
def plot_iterationHW34(means):
    pylab.plot(samples1[:, 0], samples1[:, 1], '.', color = 'blue')
    pylab.plot(samples2[:, 0], samples2[:, 1], '.', color = 'blue')
    pylab.plot(samples3[:, 0], samples3[:, 1],'.', color = 'blue')
    pylab.plot(means[0][0], means[0][1],'*',markersize =10,color = '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.show()

iterationsHW34 = [0, 9, 19, 29, 39, 49, 99]

# Initialization: initialization of parameter is fixed to show an example
centroidsHW34 = np.array([[0.0,0.0], [2.0,2.0], [0.0,7.0]])
homegwssse = []
for_plotHW34 = []
dataHW34 = sc.textFile("data.csv").cache()

for iteration in range(101):
    res = dataHW34.map(nearest_centroidHW34).reduceByKey(lambda x,y : (x[0
    #res [(0, (array([  2.66546663e+00,   3.94844436e+03]), 1001)  ),
    #     (2, (array([ 6023.84995923,  5975.48511018]), 1000)),
    #     (1, (array([ 3986.85984761,    15.93153464]), 999))]
    # res[1][1][1] returns 1000 here

    res = sorted(res, key = lambda x : x[0])  #sort based on clusted ID
    centroids_newcentroidsHW34 = np.array([x[1][0]/x[1][1] for x in res])

    WSSSE_HW34 = dataHW34.map(errorHW34).reduceByKey(lambda x, y: x + y).c
    hgwssse = 0

    for w in WSSSE_HW34:
        hgwssse += w[1]
    homegwssse.append(hgwssse)
    #if np.sum(np.absolute(centroids_new-centroids))<0.01:
    #    break

    centroidsHW34 = centroids_newcentroidsHW34

    if iteration == 0 or iteration == 9 or iteration == 19 or iteration ==
    iteration == 39 or iteration == 49 or iteration == 99:
        print("Iteration number: " + str(iteration+1))
        print("WSSSE is " + str(homegwssse[iteration]) + " after " + str(i
        print("Cluster Centers after " + str(iteration+1) + " iterations:'
        print(centroidsHW34)
        plot_iterationHW34(centroidsHW34)
        for_plotHW34.insert(iteration,homegwssse[iteration])

plt.plot(iterationsHW34, for_plotHW34)
```

```
        plt.title("WSSSE as function of iterations for Homegrown Kmeans")
        plt.xlabel("Iterations")
        plt.ylabel("WSSSE")
        plt.show()

        end = timer()
        print(str(end - start) + " seconds" )
```

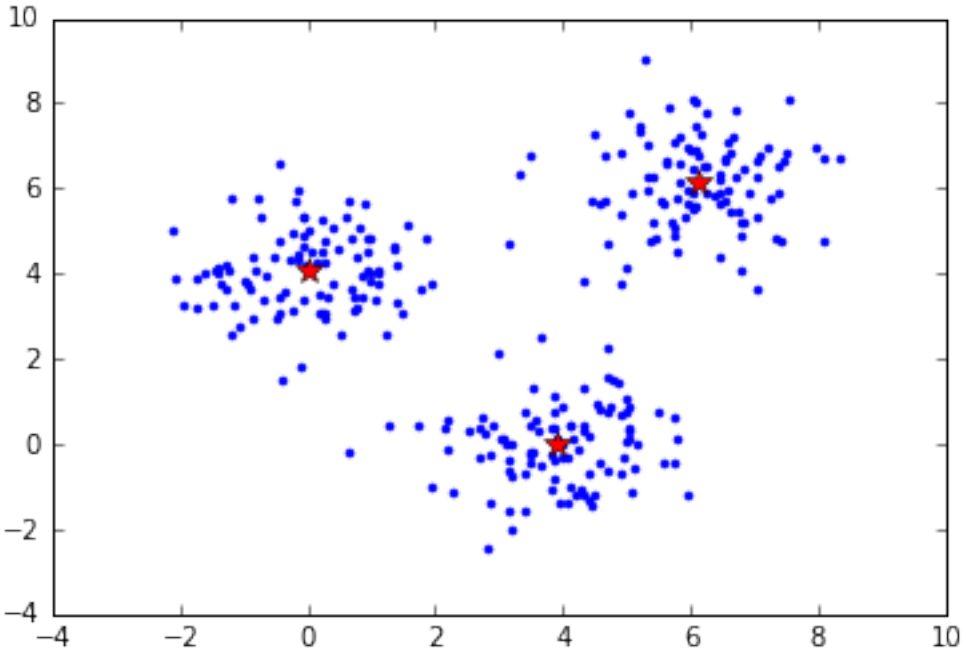
```
Iteration number: 1
WSSSE is 1101.985269836844 after 1 iterations.
Cluster Centers after 1 iterations:
[[ 0.85561352  0.47385729]
 [ 4.12492414  2.682101  ]
 [ 1.96975072  5.64603102]]
```



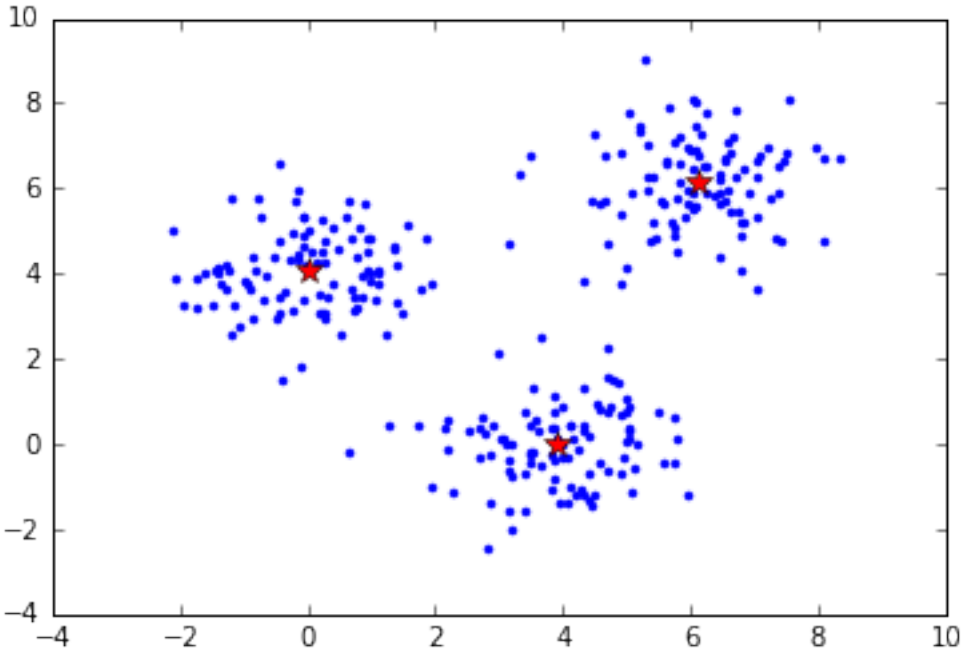

```
Iteration number: 10
WSSSE is 369.6012544615479 after 10 iterations.
Cluster Centers after 10 iterations:
[[ 3.90210723 -0.01553607]
 [ 6.1013494   6.14437711]
 [ 0.01632869  4.08348349]]
```



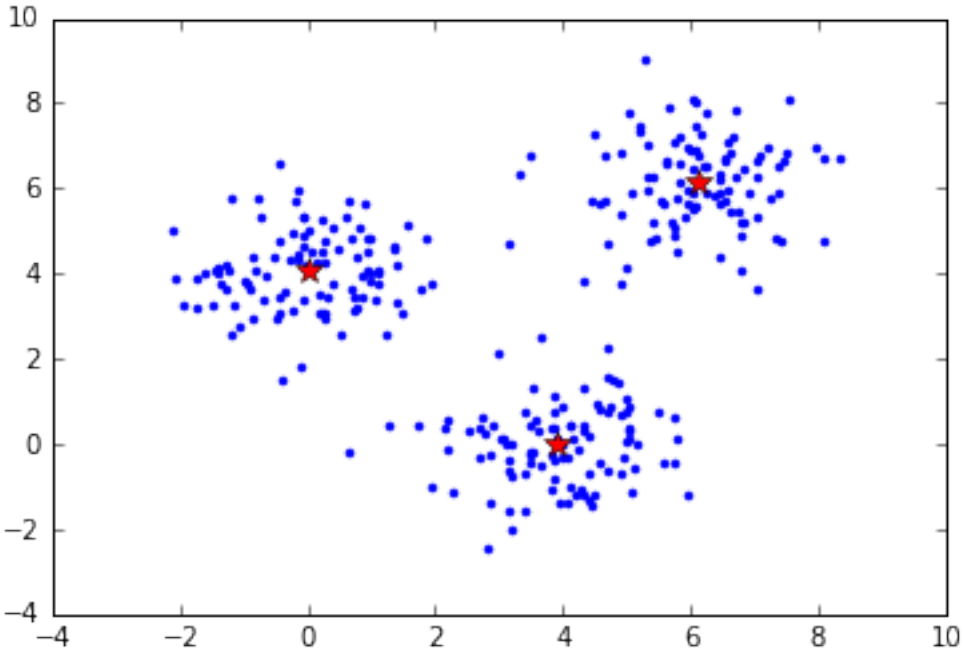

```
Iteration number: 20
WSSSE is 369.6012544615479 after 20 iterations.
Cluster Centers after 20 iterations:
[[ 3.90210723 -0.01553607]
 [ 6.1013494   6.14437711]
 [ 0.01632869  4.08348349]]
```



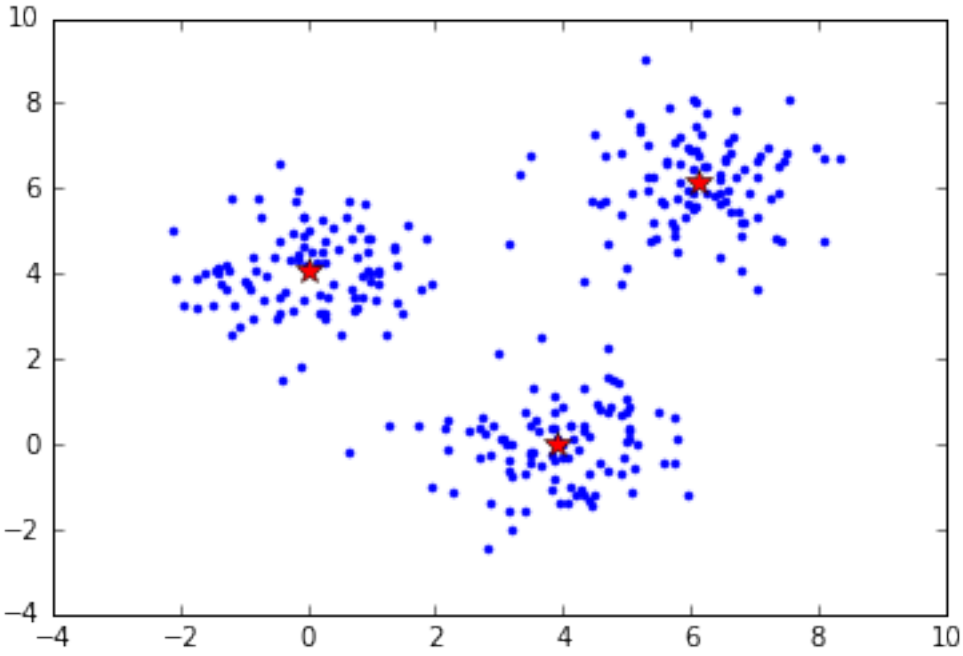

```
Iteration number: 30
WSSSE is 369.6012544615479 after 30 iterations.
Cluster Centers after 30 iterations:
[[ 3.90210723 -0.01553607]
 [ 6.1013494   6.14437711]
 [ 0.01632869  4.08348349]]
```



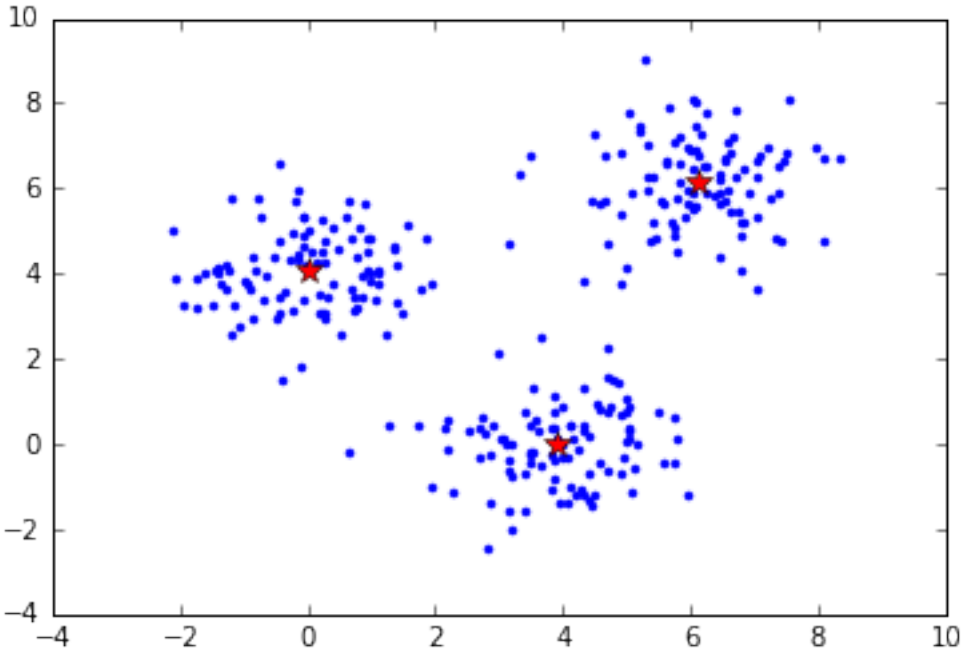

```
Iteration number: 40
WSSSE is 369.6012544615479 after 40 iterations.
Cluster Centers after 40 iterations:
[[ 3.90210723 -0.01553607]
 [ 6.1013494   6.14437711]
 [ 0.01632869  4.08348349]]
```

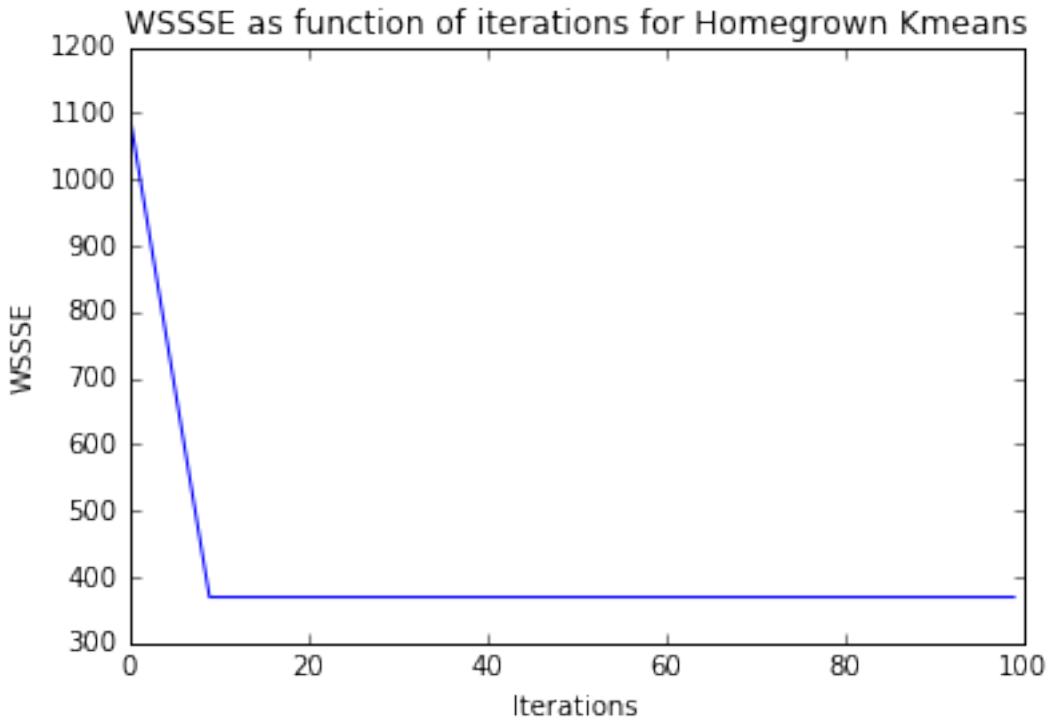



```
Iteration number: 50
WSSSE is 369.6012544615479 after 50 iterations.
Cluster Centers after 50 iterations:
[[ 3.90210723 -0.01553607]
 [ 6.1013494   6.14437711]
 [ 0.01632869  4.08348349]]
```




```
Iteration number: 100
WSSSE is 369.6012544615479 after 100 iterations.
Cluster Centers after 100 iterations:
[[ 3.90210723 -0.01553607]
 [ 6.1013494   6.14437711]
 [ 0.01632869  4.08348349]]
```






```
26.932072223999057 seconds
```

Both methods converge quickly, though the homegrown's initial guess is much worse so the initial WSSSE is very high. However, both converge to the same cluster centroids, and so the same WSSSE.

HW3.4.1: Making Homegrown KMeans more efficient

Back to Table of Contents

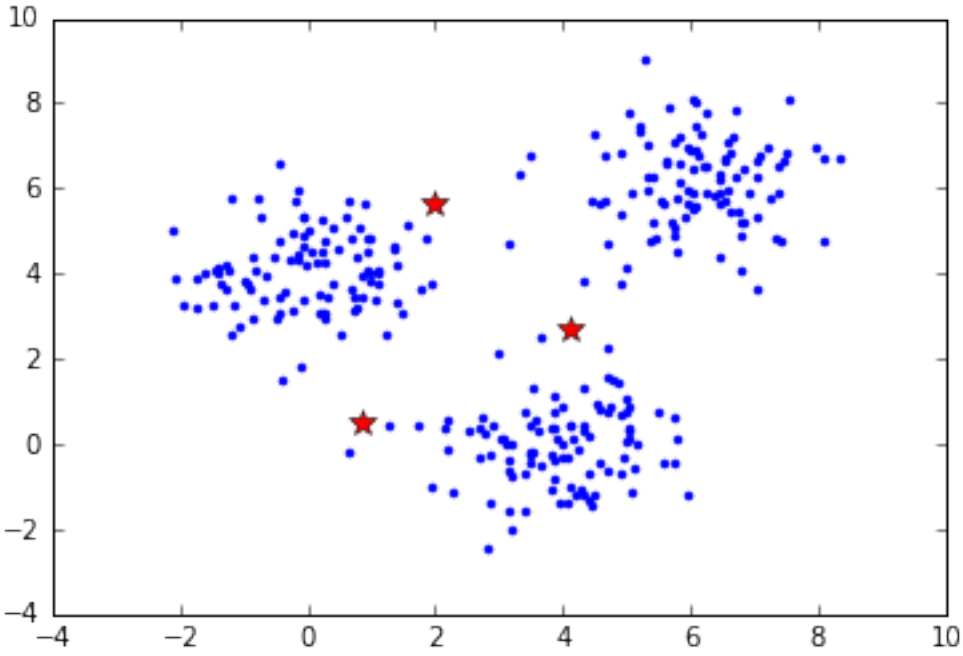
The above provided homegrown KMeans implentation in not the most efficient. How can you make it more efficient? Make this change in the code and show it work and comment on the gains you achieve.

2.1.1 HINT: have a look at this linear regression notebook

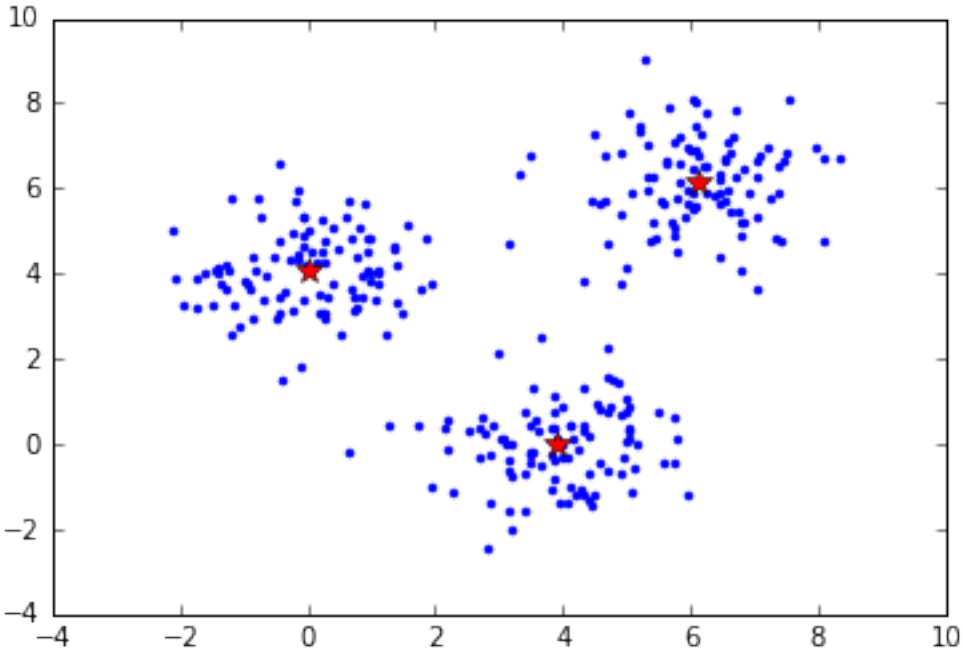
In [15]:
```
from timeit import default_timer as timer

start = timer()

import numpy as np

#Calculate which class each data point belongs to
def nearest_centroidHW341(line):
    x = np.array([float(f) for f in line.split(',')])
    closest_centroid_idxHW341 = np.sum((x - centroidBroadcast.value)**2, a
    return (closest_centroid_idxHW341,(x,1))

def errorHW341(line):
    x = np.array([float(f) for f in line.split(',')])
    closest_centroid_idxHW341 = np.sum((x - centroidBroadcast.value)**2, a
    errorHW341 =  sqrt(np.sum((x-centroidsHW341[closest_centroid_idxHW341]
    return (closest_centroid_idxHW341, errorHW341)

#plot centroids and data points for each iteration
def plot_iterationHW341(means):
    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.show()

iterationsHW341 = [0, 9, 19, 29, 39, 49, 99]

# Initialization: initialization of parameter is fixed to show an example
centroidsHW341 = np.array([[0.0,0.0], [2.0,2.0], [0.0,7.0]])
homegwssse341 = []
for_plotHW341 = []
dataHW341 = sc.textFile("data.csv").cache()

for iteration in range(100):
```

```
            centroidBroadcast = sc.broadcast(centroidsHW341)
            res = dataHW341.map(nearest_centroidHW341).reduceByKey(lambda x,y : (x
            #res [(0, (array([  2.66546663e+00,   3.94844436e+03]), 1001)  ),
            #     (2, (array([ 6023.84995923,  5975.48511018]), 1000)),
            #     (1, (array([ 3986.85984761,    15.93153464]), 999))]
            # res[1][1][1] returns 1000 here

            res = sorted(res, key = lambda x : x[0])  #sort based on clusted ID
            centroids_newHW341 = np.array([x[1][0]/x[1][1] for x in res])  #divide

            WSSSE_HW341 = dataHW341.map(errorHW341).reduceByKey(lambda x, y: x + y
            hgwssse341 = 0

            for w in WSSSE_HW341:
                hgwssse341 += w[1]
            homegwssse341.append(hgwssse341)
            #if np.sum(np.absolute(centroids_new-centroids))<0.01:
            #    break

            centroidsHW341 = centroids_newHW341

            if iteration == 0 or iteration == 9 or iteration == 19 or iteration ==
            iteration == 39 or iteration == 49 or iteration == 99:
                print("Iteration number: " + str(iteration+1))
                print("WSSSE is " + str(homegwssse341[iteration]) + " after " + st
                print("Cluster Centers after " + str(iteration+1) + " iterations:"
                print(centroidsHW341)
                plot_iterationHW341(centroidsHW341)
                for_plotHW341.insert(iteration,homegwssse341[iteration])

        plt.plot(iterationsHW341, for_plotHW341)
        plt.title("WSSSE as function of iterations for Improved Homegrown Kmeans")
        plt.xlabel("Iterations")
        plt.ylabel("WSSSE")
        plt.show()

        end = timer()
        print(str(end - start) + " seconds" )
```

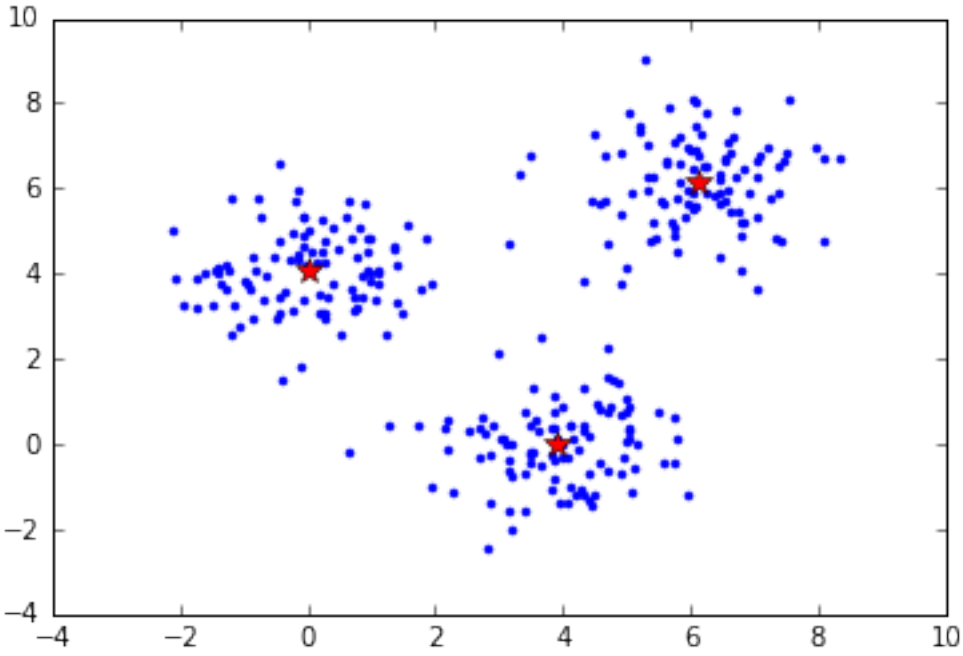
```
Iteration number: 1
WSSSE is 1101.985269836844 after 1 iterations.
Cluster Centers after 1 iterations:
[[ 0.85561352  0.47385729]
 [ 4.12492414  2.682101  ]
 [ 1.96975072  5.64603102]]
```



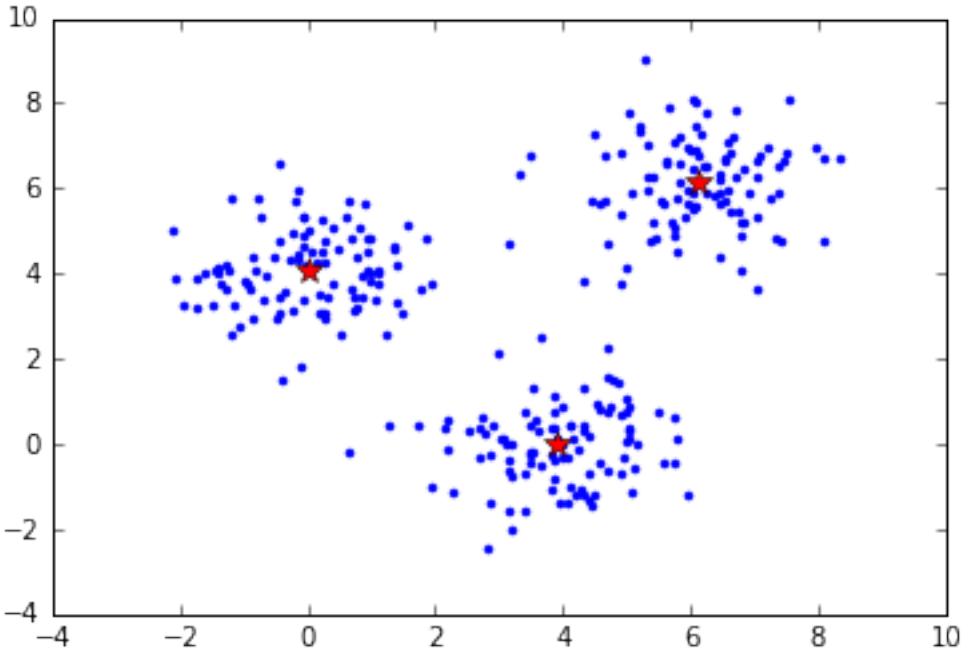

```
Iteration number: 10
WSSSE is 369.6012544615479 after 10 iterations.
Cluster Centers after 10 iterations:
[[ 3.90210723 -0.01553607]
 [ 6.1013494   6.14437711]
 [ 0.01632869  4.08348349]]
```



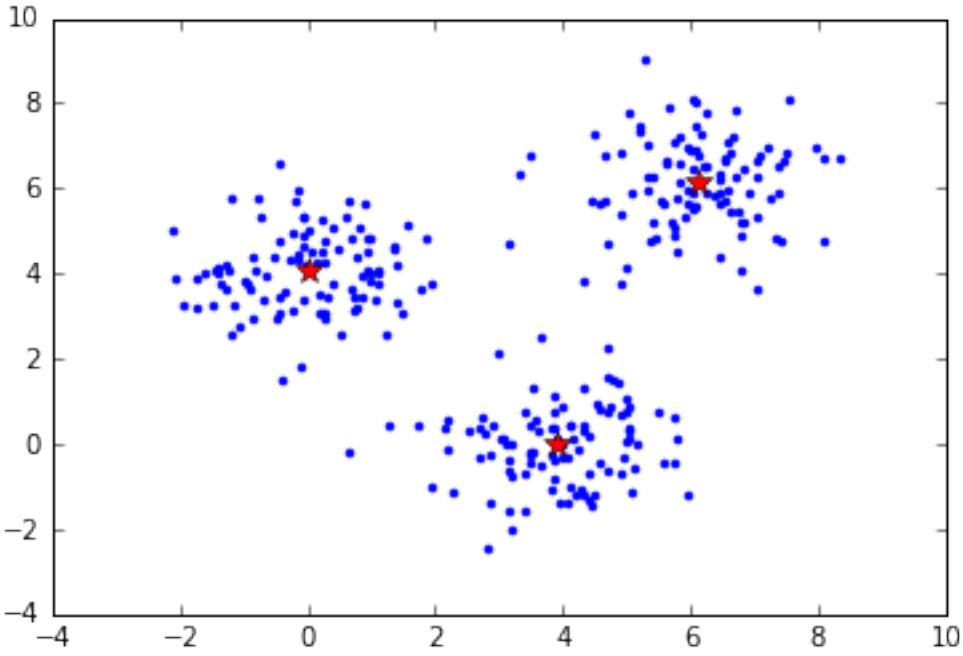

```
Iteration number: 20
WSSSE is 369.6012544615479 after 20 iterations.
Cluster Centers after 20 iterations:
[[ 3.90210723 -0.01553607]
 [ 6.1013494   6.14437711]
 [ 0.01632869  4.08348349]]
```



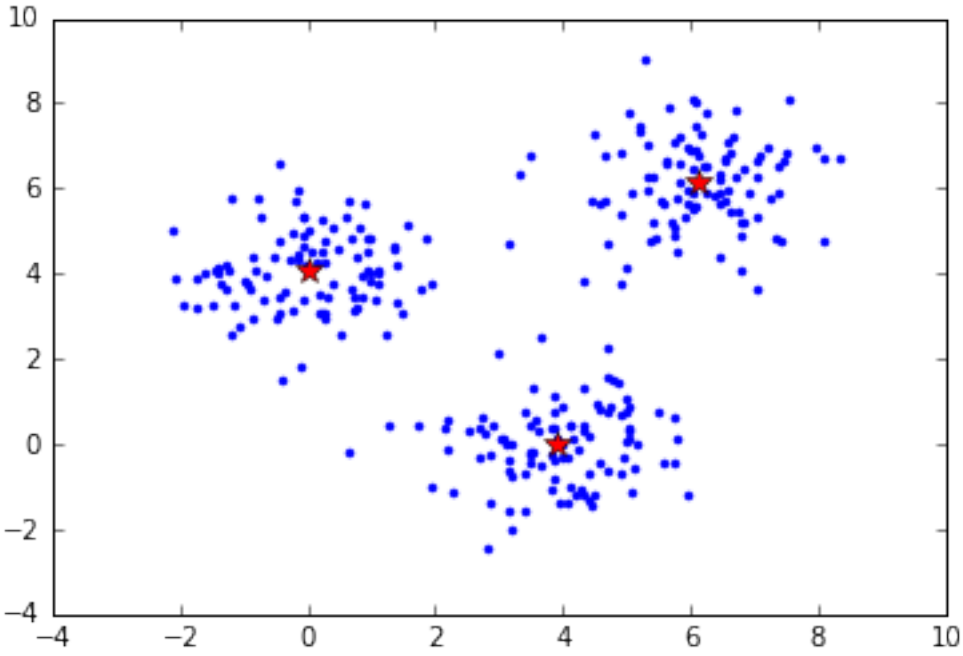

```
Iteration number: 30
WSSSE is 369.6012544615479 after 30 iterations.
Cluster Centers after 30 iterations:
[[ 3.90210723 -0.01553607]
 [ 6.1013494   6.14437711]
 [ 0.01632869  4.08348349]]
```



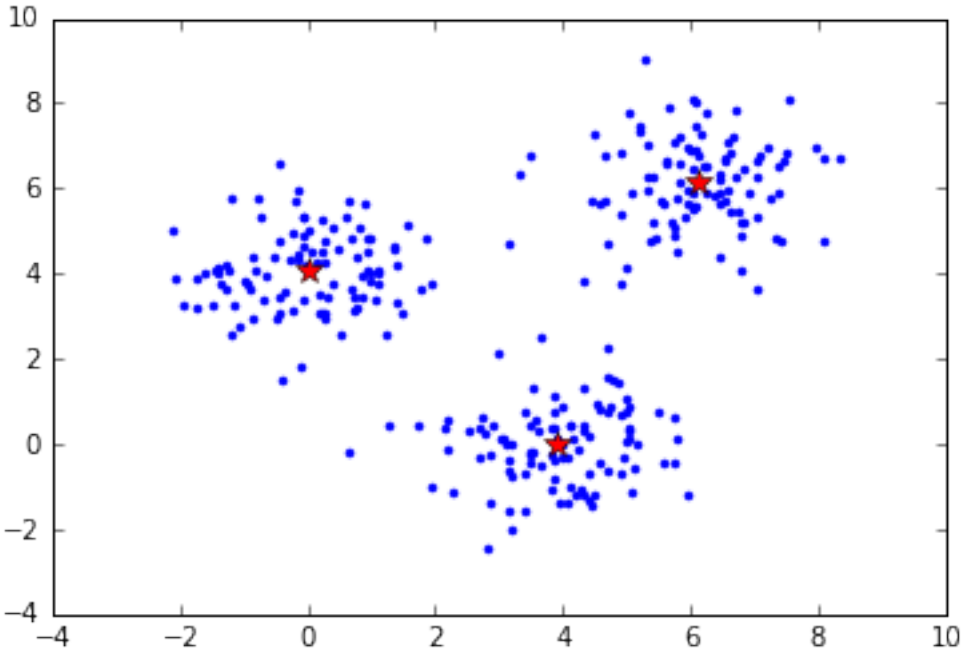

```
Iteration number: 40
WSSSE is 369.6012544615479 after 40 iterations.
Cluster Centers after 40 iterations:
[[ 3.90210723 -0.01553607]
 [ 6.1013494   6.14437711]
 [ 0.01632869  4.08348349]]
```

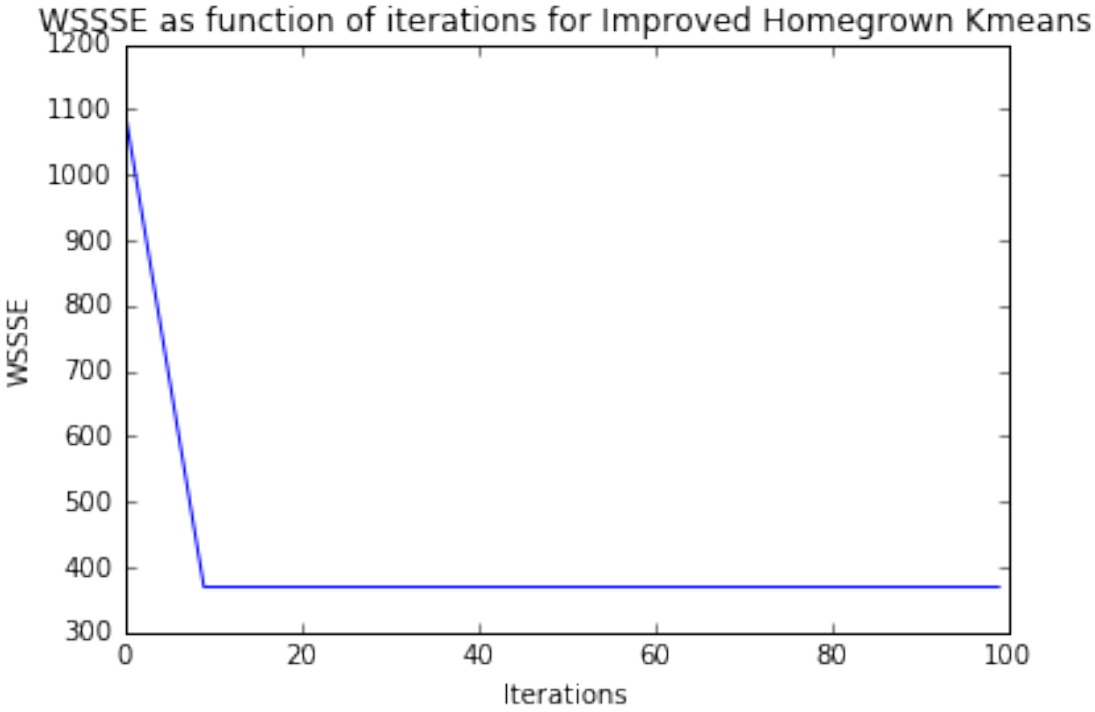



```
Iteration number: 50
WSSSE is 369.6012544615479 after 50 iterations.
Cluster Centers after 50 iterations:
[[ 3.90210723 -0.01553607]
 [ 6.1013494   6.14437711]
 [ 0.01632869  4.08348349]]
```




```
Iteration number: 100
WSSSE is 369.6012544615479 after 100 iterations.
Cluster Centers after 100 iterations:
[[ 3.90210723 -0.01553607]
 [ 6.1013494   6.14437711]
 [ 0.01632869  4.08348349]]
```






```
26.273201120999147 seconds
```

3.4 time was 26.93 seconds.

3.4.1 time was 26.27 seconds, so it did run faster, but not noticably with 100 data points and on a local machine.

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.

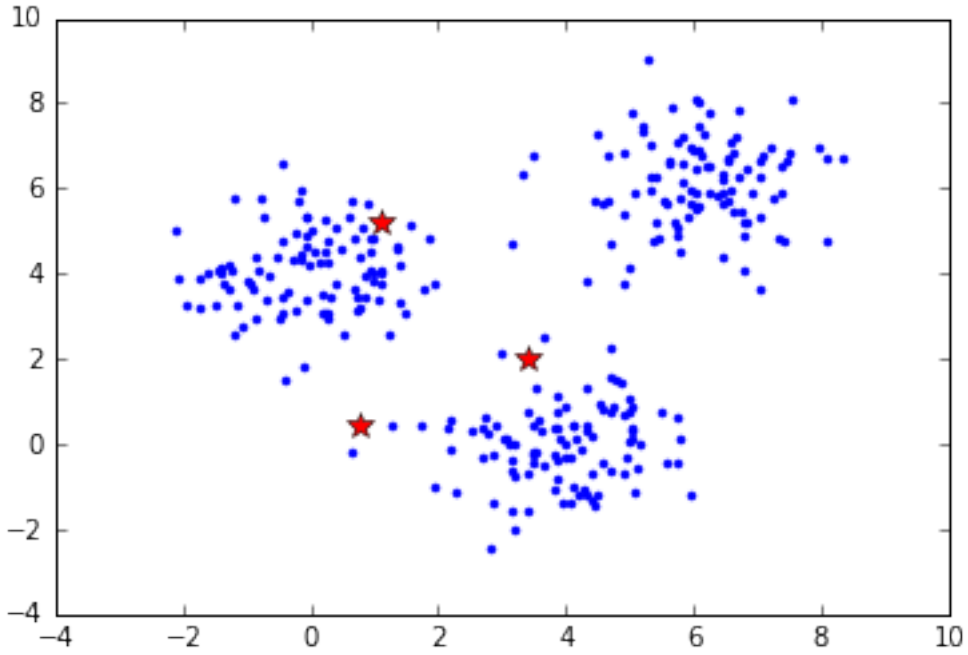
[Please incorporate all referenced notebooks directly into this master notebook as cells for HW submission. I.e., HW submissions should comprise of just one notebook]

In [17]:
```
from timeit import default_timer as timer

start = timer()

from numpy.linalg import norm

def point_weight(value):
    closest_centroid = value[0]
    point = value[1][0]
    nweight = 1.0/norm(point)

    return closest_centroid, (point * nweight, nweight)

#Calculate which class each data point belongs to
def nearest_centroidHW34(line):
    x = np.array([float(f) for f in line.split(',')])
    closest_centroid_idxHW34 = np.sum((x - centroidsHW34)**2, axis=1).argm
    return (closest_centroid_idxHW34,(x,1))

def errorHW34(line):
    x = np.array([float(f) for f in line.split(',')])
    closest_centroid_idxHW34 = np.sum((x - centroidsHW34)**2, axis=1).argm
    errorHW34 =  sqrt(np.sum((x-centroidsHW34[closest_centroid_idxHW34])**
    return (closest_centroid_idxHW34, errorHW34)

#plot centroids and data points for each iteration
def plot_iterationHW34(means):
    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')
```

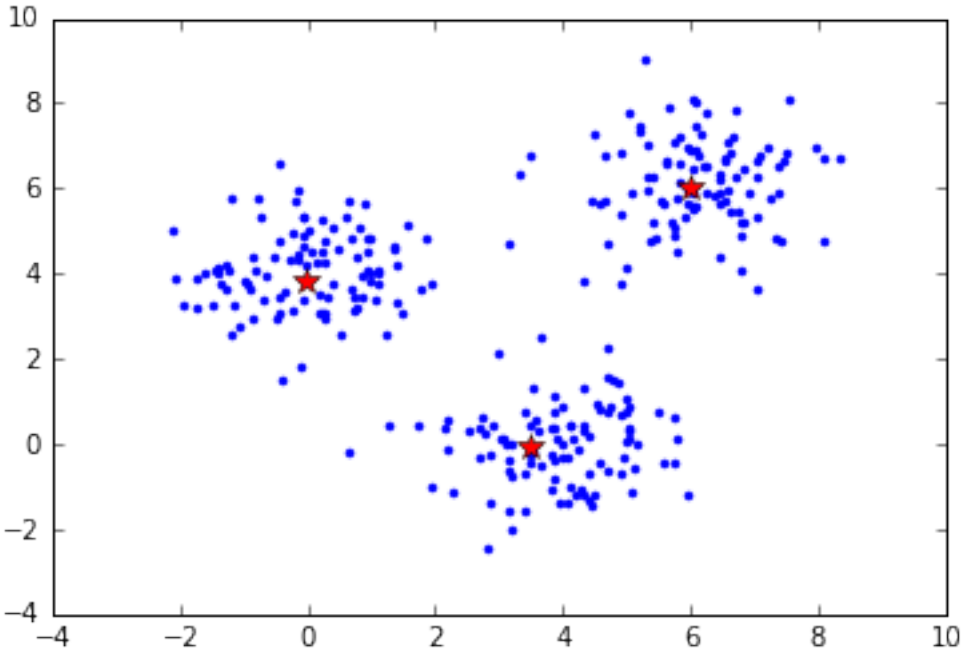
```
    pylab.plot(means[2][0], means[2][1],'*',markersize =10,color = 'red')
    pylab.show()

iterationsHW34 = [0, 9, 19, 29, 39, 49, 99]

# Initialization: initialization of parameter is fixed to show an example
centroidsHW34 = np.array([[0.0,0.0], [2.0,2.0], [0.0,7.0]])
homegwssse = []
for_plotHW34 = []
dataHW34 = sc.textFile("data.csv").cache()

for iteration in range(100):
    res = dataHW34.map(nearest_centroidHW34).map(point_weight).reduceByKey
    #res [(0, (array([  2.66546663e+00,   3.94844436e+03]), 1001)  ),
    #     (2, (array([ 6023.84995923,  5975.48511018]), 1000)),
    #     (1, (array([ 3986.85984761,    15.93153464]), 999))]
    # res[1][1][1] returns 1000 here

    res = sorted(res, key = lambda x : x[0])  #sort based on clusted ID
    centroids_newcentroidsHW34 = np.array([x[1][0]/x[1][1] for x in res])

    WSSSE_HW34 = dataHW34.map(errorHW34).reduceByKey(lambda x, y: x + y).c
    hgwssse = 0

    for w in WSSSE_HW34:
        hgwssse += w[1]
    homegwssse.append(hgwssse)
    #if np.sum(np.absolute(centroids_new-centroids))<0.01:
    #    break

    centroidsHW34 = centroids_newcentroidsHW34

    if iteration == 0 or iteration == 9 or iteration == 19 or iteration ==
    iteration == 39 or iteration == 49 or iteration == 99:
        print("Iteration number: " + str(iteration+1))
        print("WSSSE is " + str(homegwssse[iteration]) + " after " + str(i
        print("Cluster Centers after " + str(iteration+1) + " iterations:"
        print(centroidsHW34)
        plot_iterationHW34(centroidsHW34)
        for_plotHW34.insert(iteration,homegwssse[iteration])

plt.plot(iterationsHW34, for_plotHW34)
plt.title("WSSSE as function of iterations for Weighted Homegrown Kmeans")
plt.xlabel("Iterations")
plt.ylabel("WSSSE")
plt.show()

end = timer()
```

```
        print(str(end - start) + " seconds" )
```

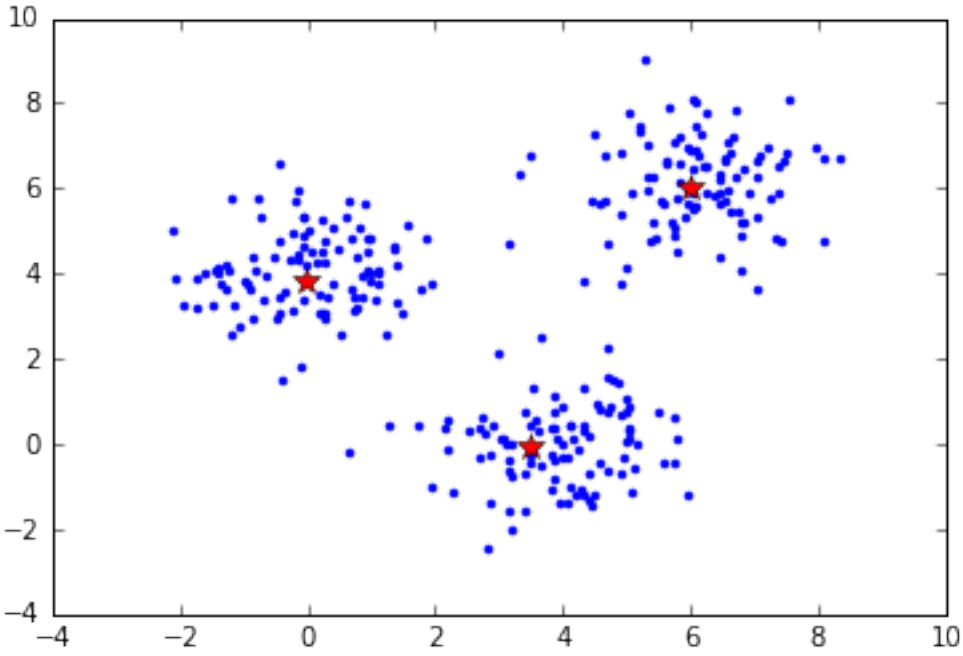
```
Iteration number: 1
WSSSE is 1101.985269836844 after 1 iterations.
Cluster Centers after 1 iterations:
[[ 0.76947064  0.44330139]
 [ 3.41097629  2.02257758]
 [ 1.09490075  5.22526621]]
```



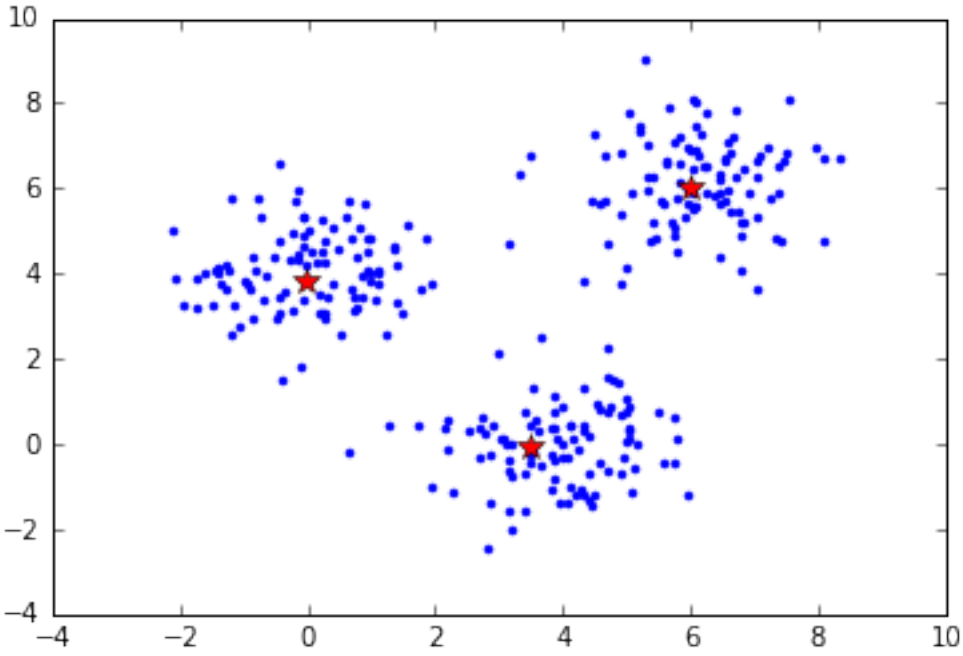

```
Iteration number: 10
WSSSE is 378.6053650187612 after 10 iterations.
Cluster Centers after 10 iterations:
[[ 3.49936518 -0.03079605]
 [ 5.97908178  6.02643422]
 [-0.03034403  3.85444832]]
```



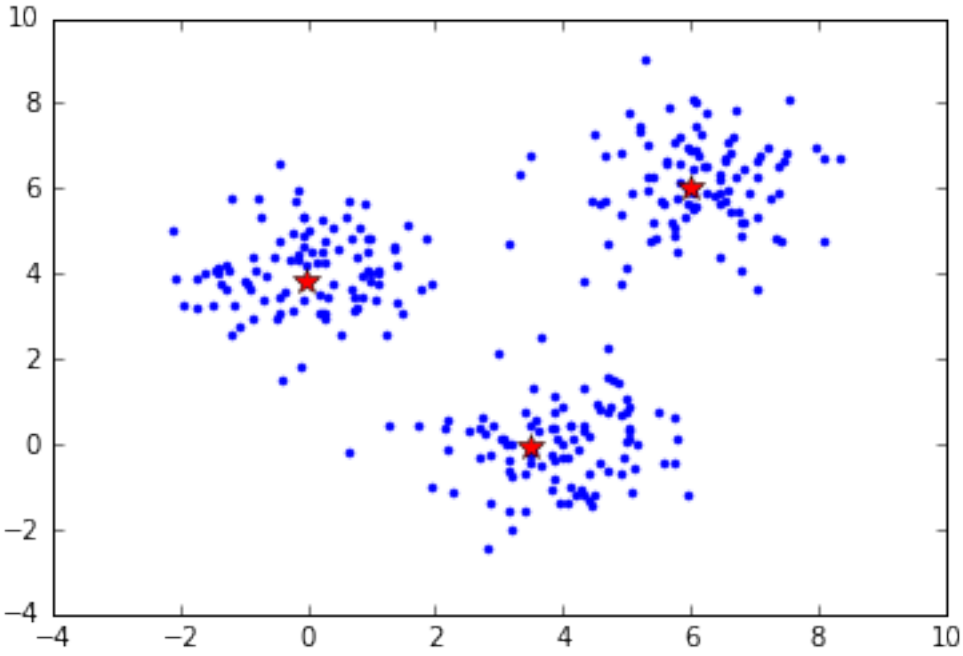

```
Iteration number: 20
WSSSE is 378.6053650187612 after 20 iterations.
Cluster Centers after 20 iterations:
[[ 3.49936518 -0.03079605]
 [ 5.97908178  6.02643422]
 [-0.03034403  3.85444832]]
```



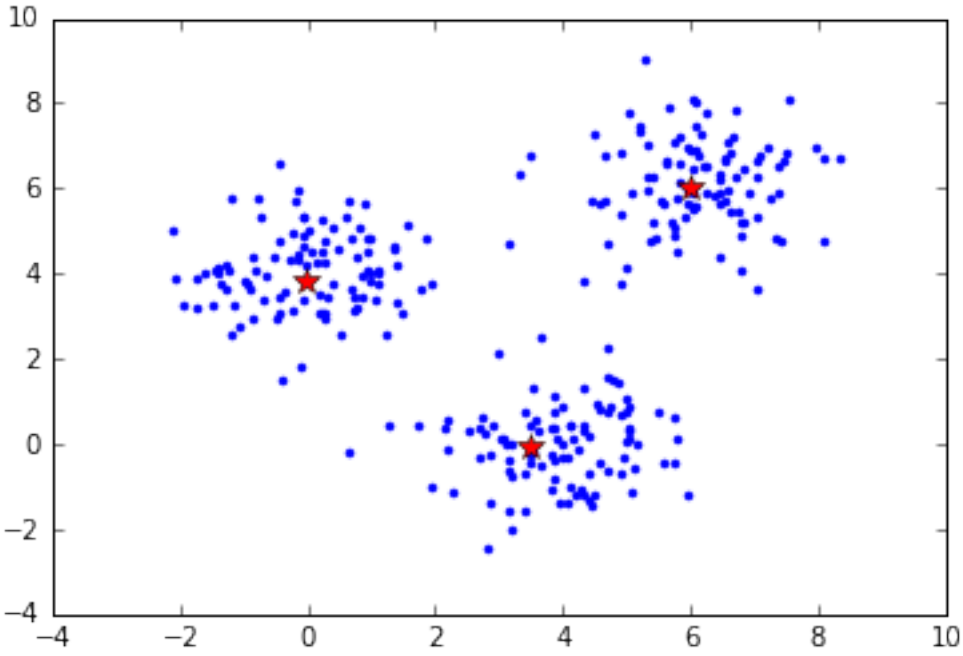

```
Iteration number: 30
WSSSE is 378.6053650187612 after 30 iterations.
Cluster Centers after 30 iterations:
[[ 3.49936518 -0.03079605]
 [ 5.97908178  6.02643422]
 [-0.03034403  3.85444832]]
```



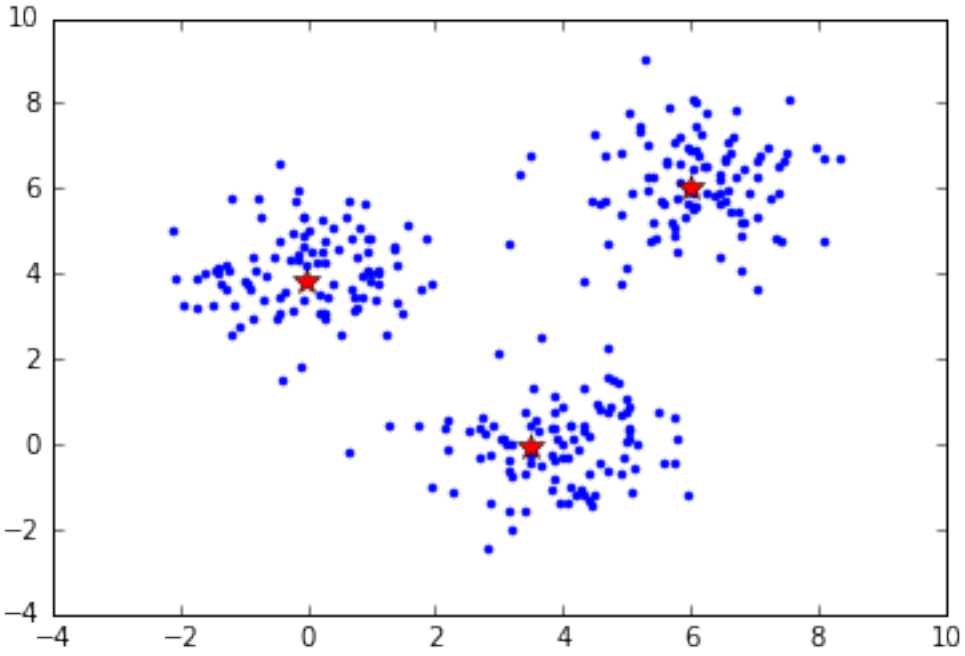

```
Iteration number: 40
WSSSE is 378.6053650187612 after 40 iterations.
Cluster Centers after 40 iterations:
[[ 3.49936518 -0.03079605]
 [ 5.97908178  6.02643422]
 [-0.03034403  3.85444832]]
```

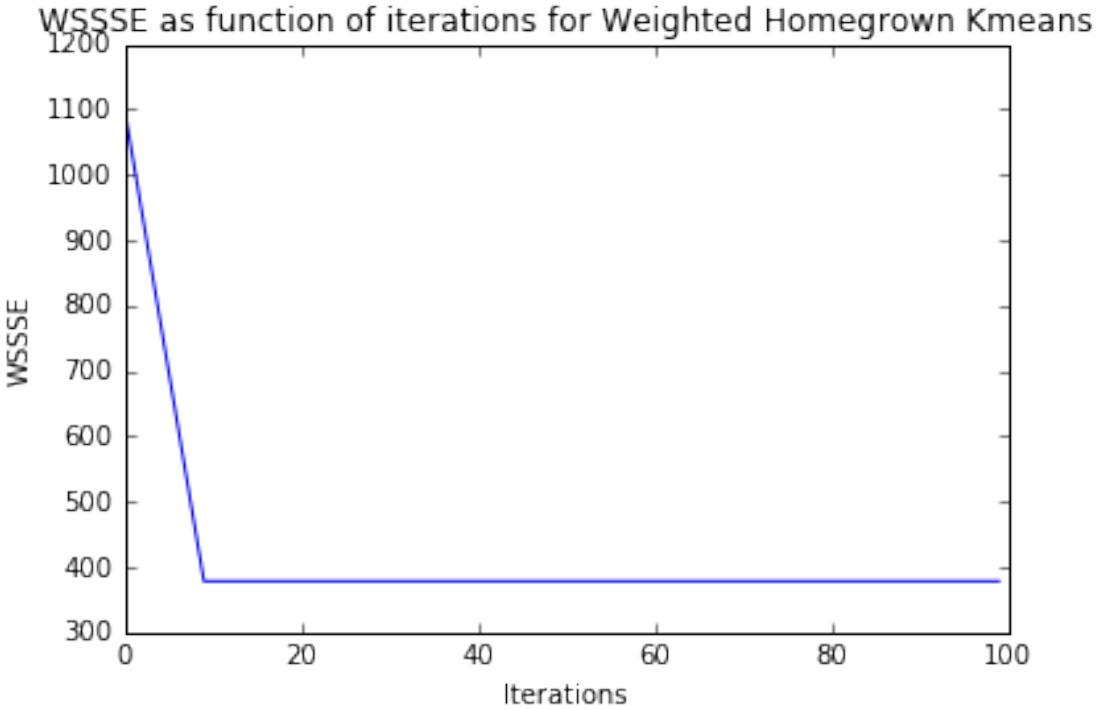



```
Iteration number: 50
WSSSE is 378.6053650187612 after 50 iterations.
Cluster Centers after 50 iterations:
[[ 3.49936518 -0.03079605]
 [ 5.97908178  6.02643422]
 [-0.03034403  3.85444832]]
```




```
Iteration number: 100
WSSSE is 378.6053650187612 after 100 iterations.
Cluster Centers after 100 iterations:
[[ 3.49936518 -0.03079605]
 [ 5.97908178  6.02643422]
 [-0.03034403  3.85444832]]
```






```
25.517934105000677 seconds
```

2.2 After 100 iterations, HW3.4 looks like this...

Iteration number: 100 WSSSE is 369.6012544615479 after 100 iterations. Cluster Centers after 100 iterations: [[3.90210723 -0.01553607] [6.1013494 6.14437711] [0.01632869 4.08348349]]

2.3 After 100 iterations, HW3.5, looks like this...

Iteration number: 100 WSSSE is 378.6053650187612 after 100 iterations. Cluster Centers after 100 iterations: [[3.49936518 -0.03079605] [5.97908178 6.02643422] [-0.03034403 3.85444832]]

By comparison, the weighted kmeans is closer to the x axis than the non weighted, but has a larger WSSSE. The farther the point is from the x axis, the more it is weighted (pulled). It does, however, increase the error, but not by much.

It also runs somewhat faster than the unweighted, which is surprising.

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.

```
In [18]: import numpy as np
         import csv
         def data_generate(fileName, w=[0,0], size=100):
             np.random.seed(0)
             x = np.random.uniform(-4, 4, size)
             noise = np.random.normal(0, 2, size)
             y = (x * w[0] + w[1] + noise)
             data = zip(y, x)
             with open(fileName,'w') as f:
                 writer = csv.writer(f)
                 for row in data:
                     writer.writerow(row)
             return True
```

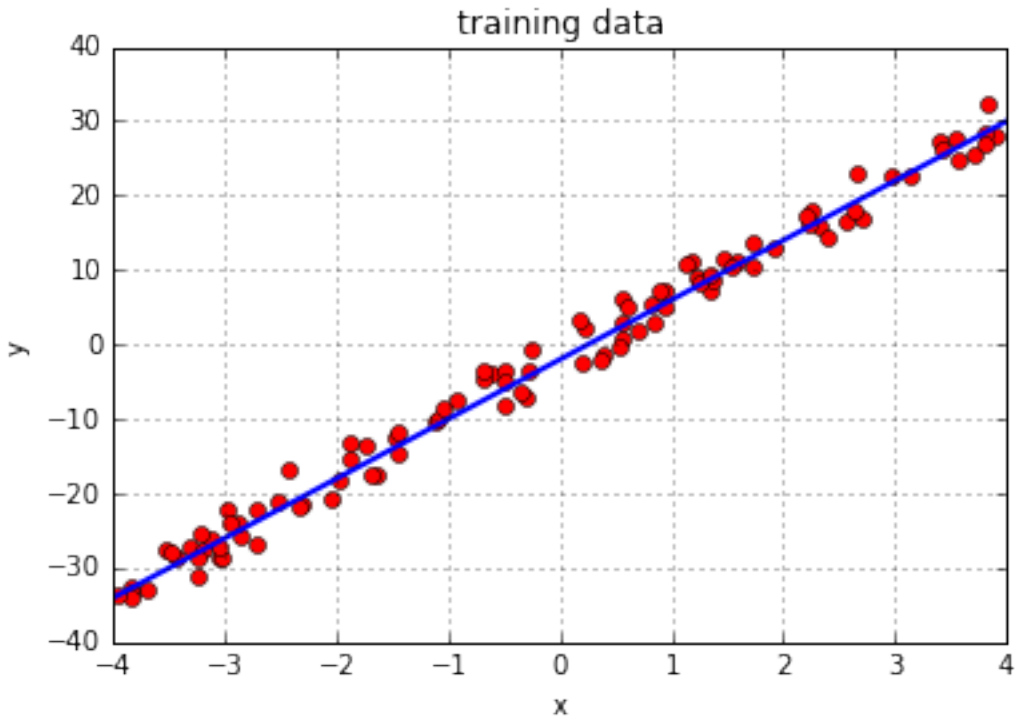
```
In [19]: %matplotlib inline
         import matplotlib.pyplot as plt
         def dataPlot(file, w,title):
             with open(file, 'r') as f:
                 reader = csv.reader(f)
                 for row in reader:
                     plt.plot(float(row[1]), float(row[0]),'o'+'r')
             plt.xlabel("x")
```

```
    plt.ylabel("y")
    x = [-4, 4]
    y = [(i * w[0] + w[1]) for i in x]
    plt.plot(x,y, linewidth=2.0)
    plt.title(title)
    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid()
    plt.show()
```

In [20]:
```
w=[8,-2]
data_generate('training dataset.csv', w, 100)
```

Out[20]: True

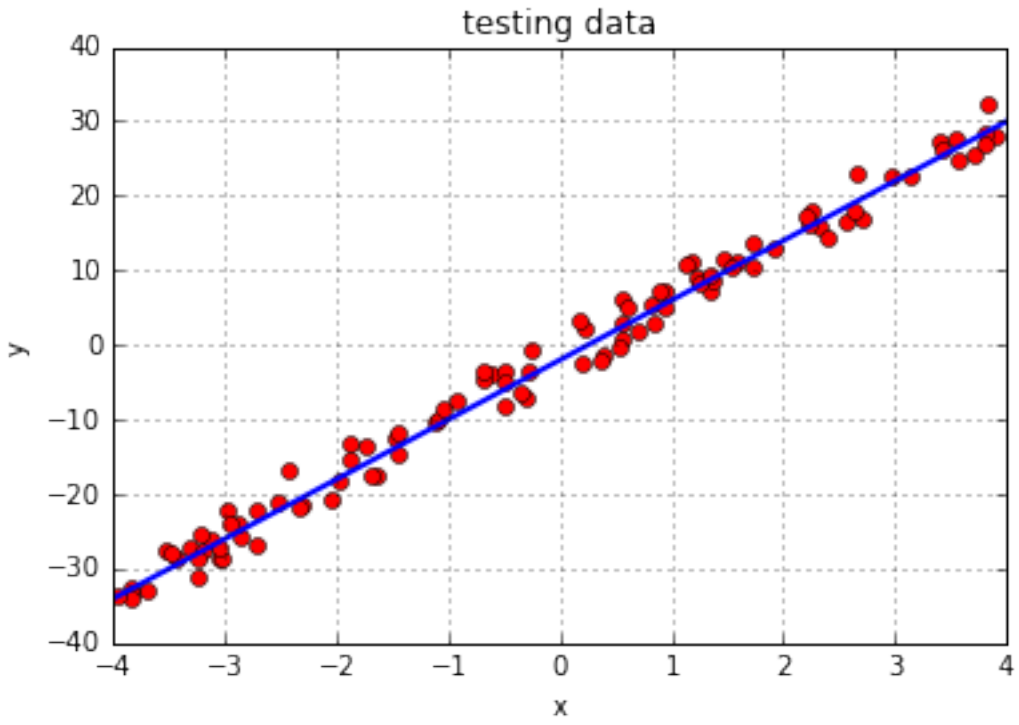
In [21]:
```
dataPlot('training dataset.csv',w,'training data')
```



In [22]:
```
w=[8,-2]
data_generate('testing set.csv', [8,-2], 100)
```

Out[22]: True

In [23]:
```
dataPlot('testing set.csv',w, 'testing data')
```



In [24]:
```
from pyspark.mllib.regression import LabeledPoint, LinearRegressionWithSGD
from numpy import array
from operator import add
from math import sqrt

def parsePoint(line):
    values = [float(x) for x in line.split(',')]
    return LabeledPoint(values[0], values[1:])

def runHW361(trainData, testData, iterations, stepSize):

    trainingdata = sc.textFile(trainData)
    parsedTrainData = trainingdata.map(parsePoint)
    testingdata = sc.textFile(testData)
    parsedTestData = testingdata.map(parsePoint)

    # Load and parse the data
    for ints in iterations:
        MSE = 0

        # Build the model
        model = LinearRegressionWithSGD.train(parsedTrainData, intercept=T
```

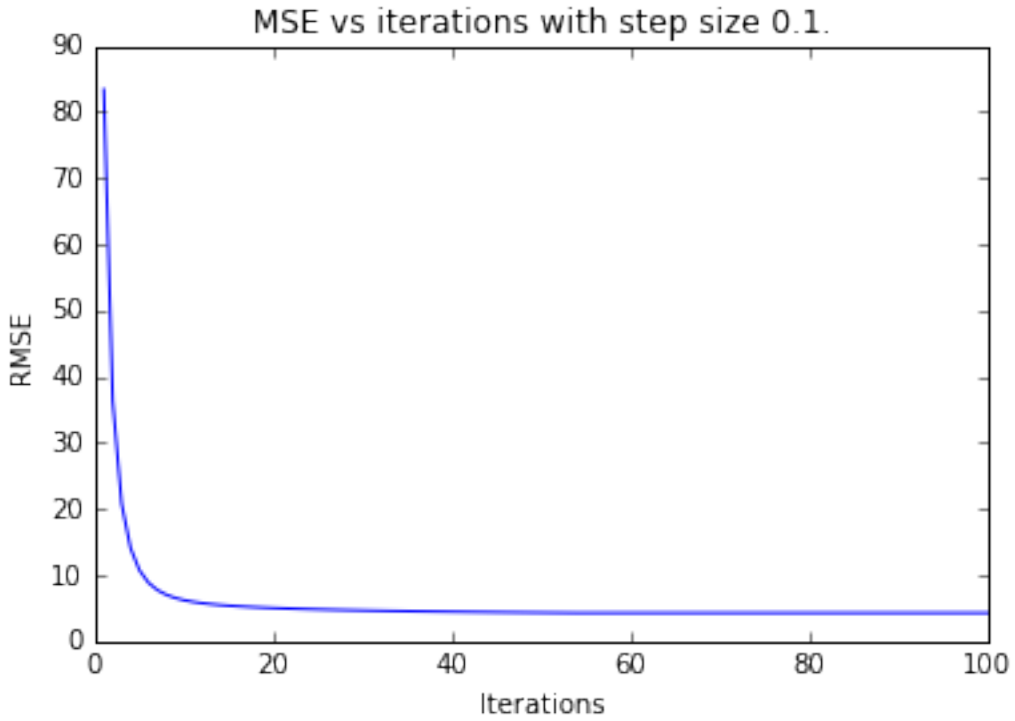
```
            valuesAndPreds = parsedTestData.map(lambda p: (p.label, model.pred

            MSE = valuesAndPreds.map(lambda kv: (kv[0] - kv[1])**2).reduce(lam
            MSE_vs_ints.insert(ints,MSE)
            print("After " + str(ints) + " iterations, the MSE is:  " + str(MS

        print("The model weights are:  "+ str(model) + ".")

        plt.plot(iterations, MSE_vs_ints)
        plt.title("MSE vs iterations with step size " + str(stepSize) + ".")
        plt.xlabel("Iterations")
        plt.ylabel("RMSE")
        plt.show()
```

```
In [25]: MSE_vs_ints = []
         runHW361('training dataset.csv', 'testing set.csv', range(1,101), .1)
```

```
After 1 iterations, the MSE is:  83.3910975477.
After 2 iterations, the MSE is:  36.375472824.
After 3 iterations, the MSE is:  20.7914812562.
After 4 iterations, the MSE is:  14.0837236233.
After 5 iterations, the MSE is:  10.7290134917.
After 6 iterations, the MSE is:  8.86948303561.
After 7 iterations, the MSE is:  7.7546818602.
After 8 iterations, the MSE is:  7.04213131637.
After 9 iterations, the MSE is:  6.56105474437.
After 10 iterations, the MSE is:  6.22028890273.
After 11 iterations, the MSE is:  5.96843334659.
After 12 iterations, the MSE is:  5.77516039275.
After 13 iterations, the MSE is:  5.62186818329.
After 14 iterations, the MSE is:  5.49675335172.
After 15 iterations, the MSE is:  5.39209607543.
After 16 iterations, the MSE is:  5.30270649018.
After 17 iterations, the MSE is:  5.22500597884.
After 18 iterations, the MSE is:  5.1564679349.
After 19 iterations, the MSE is:  5.09526843932.
After 20 iterations, the MSE is:  5.04006291486.
After 21 iterations, the MSE is:  4.9898402615.
After 22 iterations, the MSE is:  4.94382571384.
After 23 iterations, the MSE is:  4.90141496232.
After 24 iterations, the MSE is:  4.86212871158.
After 25 iterations, the MSE is:  4.82558083083.
After 26 iterations, the MSE is:  4.79145568947.
After 27 iterations, the MSE is:  4.75949179422.
After 28 iterations, the MSE is:  4.72946981108.
After 29 iterations, the MSE is:  4.70120367969.
After 30 iterations, the MSE is:  4.67453393678.
After 31 iterations, the MSE is:  4.64932263723.
```

```
After 32 iterations, the MSE is:  4.62544944415.
After 33 iterations, the MSE is:  4.60280858419.
After 34 iterations, the MSE is:  4.58130645029.
After 35 iterations, the MSE is:  4.56085969404.
After 36 iterations, the MSE is:  4.54139369219.
After 37 iterations, the MSE is:  4.52284130186.
After 38 iterations, the MSE is:  4.50514184072.
After 39 iterations, the MSE is:  4.48824024419.
After 40 iterations, the MSE is:  4.47208636298.
After 41 iterations, the MSE is:  4.45663437316.
After 42 iterations, the MSE is:  4.44184227696.
After 43 iterations, the MSE is:  4.42767147746.
After 44 iterations, the MSE is:  4.41408641393.
After 45 iterations, the MSE is:  4.40105424715.
After 46 iterations, the MSE is:  4.38854458649.
After 47 iterations, the MSE is:  4.37652925187.
After 48 iterations, the MSE is:  4.36498206511.
After 49 iterations, the MSE is:  4.35387866632.
After 50 iterations, the MSE is:  4.3431963515.
After 51 iterations, the MSE is:  4.33291392848.
After 52 iterations, the MSE is:  4.32301158853.
After 53 iterations, the MSE is:  4.31347079167.
After 54 iterations, the MSE is:  4.3042741638.
After 55 iterations, the MSE is:  4.3042741638.
After 56 iterations, the MSE is:  4.3042741638.
After 57 iterations, the MSE is:  4.3042741638.
After 58 iterations, the MSE is:  4.3042741638.
After 59 iterations, the MSE is:  4.3042741638.
After 60 iterations, the MSE is:  4.3042741638.
After 61 iterations, the MSE is:  4.3042741638.
After 62 iterations, the MSE is:  4.3042741638.
After 63 iterations, the MSE is:  4.3042741638.
After 64 iterations, the MSE is:  4.3042741638.
After 65 iterations, the MSE is:  4.3042741638.
After 66 iterations, the MSE is:  4.3042741638.
After 67 iterations, the MSE is:  4.3042741638.
After 68 iterations, the MSE is:  4.3042741638.
After 69 iterations, the MSE is:  4.3042741638.
After 70 iterations, the MSE is:  4.3042741638.
After 71 iterations, the MSE is:  4.3042741638.
After 72 iterations, the MSE is:  4.3042741638.
After 73 iterations, the MSE is:  4.3042741638.
After 74 iterations, the MSE is:  4.3042741638.
After 75 iterations, the MSE is:  4.3042741638.
After 76 iterations, the MSE is:  4.3042741638.
After 77 iterations, the MSE is:  4.3042741638.
After 78 iterations, the MSE is:  4.3042741638.
After 79 iterations, the MSE is:  4.3042741638.
```

```
After 80 iterations, the MSE is:  4.3042741638.
After 81 iterations, the MSE is:  4.3042741638.
After 82 iterations, the MSE is:  4.3042741638.
After 83 iterations, the MSE is:  4.3042741638.
After 84 iterations, the MSE is:  4.3042741638.
After 85 iterations, the MSE is:  4.3042741638.
After 86 iterations, the MSE is:  4.3042741638.
After 87 iterations, the MSE is:  4.3042741638.
After 88 iterations, the MSE is:  4.3042741638.
After 89 iterations, the MSE is:  4.3042741638.
After 90 iterations, the MSE is:  4.3042741638.
After 91 iterations, the MSE is:  4.3042741638.
After 92 iterations, the MSE is:  4.3042741638.
After 93 iterations, the MSE is:  4.3042741638.
After 94 iterations, the MSE is:  4.3042741638.
After 95 iterations, the MSE is:  4.3042741638.
After 96 iterations, the MSE is:  4.3042741638.
After 97 iterations, the MSE is:  4.3042741638.
After 98 iterations, the MSE is:  4.3042741638.
After 99 iterations, the MSE is:  4.3042741638.
After 100 iterations, the MSE is:  4.3042741638.
The model weights are:  (weights=[8.01037303194], intercept=-1.0378476468396287).
```



A step size of 1 led to a sawtoothed plot. To ensure constant decrease in error, I reduced the step size to .1, as you are seeing above. Both the graph and the printouts indicate tehat after approximately 54 iterations the error converges to 4.3042741638. So, a good number of iterations could be any one of: * 54 becuase we found this to be the exact number of iterations for this model to converge. * 55 to ensure the model error remained constant (and also because 55 is a nicer number than 54) * 60 to round up to nearest 10.

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 [281]: def linearRegressionGDReg(trainData, testData, wInitial=None, learningRat

                  trainingData = sc.textFile(trainData).map(lambda line: [float(v) for
                  testingData = sc.textFile(testData).map(lambda line: [float(v) for v
                  xtest = testingData.map(lambda x: x[0]).collect()
                  ytest = testingData.map(lambda x: x[1]).collect()
                  featureLen = len(data.take(1)[0])-1

                  n = data.count()
                  if wInitial is None:
                      w = np.random.normal(size=featureLen) # w should be broadcasted
                  else:
                      w = wInitial
                  for i in range(1,iterations+1):
                      wBroadcast = sc.broadcast(w)
                      gradient = data.map(lambda d: -2 * (d[0] - np.dot(wBroadcast.valu
                                  .reduce(lambda a, b: a + b)
                      if regType == "Ridge":
                          wReg = np.square(w)
                          wReg[-1] = 0
                      elif regType == "Lasso":
                          wReg = np.abs(w)
                          wReg[-1] = 0
                      else:
                          wReg = np.zeros(w.shape[0])
                      gradient = gradient + regParam * wReg  #gradient:  GD of Sqaured
                      w = w - learningRate * gradient / n

                      yhat = [a * w[0] for a in xtest] + w[1]
                      MSEvec = np.square(yhat - ytest)
                      MSE = np.sum(MSEvec) / len(MSEvec)

                  return (regParam, w,MSE)
```

2.3.1 Run Ridge Regression for different regParams (10, 5, 1, .1, 0.01, 0.001,.0001) I chose 500 iterations to (all but) ensure convergence.

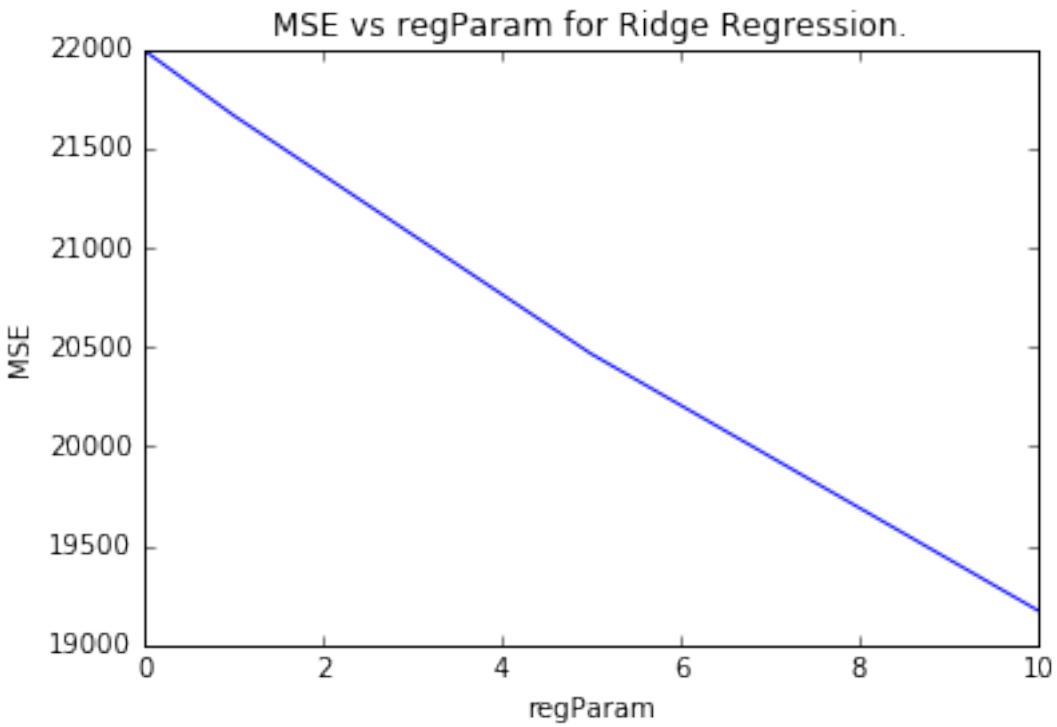
```
In [278]: np.random.seed(400)
          modelRidge_10 = linearRegressionGDReg('training dataset.csv', 'testing se
          modelRidge_5 = linearRegressionGDReg('training dataset.csv', 'testing set
          modelRidge_1 = linearRegressionGDReg('training dataset.csv', 'testing set
          modelRidge_pt1 = linearRegressionGDReg('training dataset.csv', 'testing s
          modelRidge_pt01 = linearRegressionGDReg('training dataset.csv', 'testing
          modelRidge_pt001 = linearRegressionGDReg('training dataset.csv', 'testing
          modelRidge_pt0001 = linearRegressionGDReg('training dataset.csv', 'testin
```

2.3.2 Summarize Ridge regression results

```
In [279]: print(modelRidge_10)
          print(modelRidge_5)
          print(modelRidge_1)
          print(modelRidge_pt1)
          print(modelRidge_pt01)
          print(modelRidge_pt001)
          print(modelRidge_pt0001)
```

```
(10, array([ 7.46100609, -1.73264295]), 19174.962041457329)
(5, array([ 7.70521378, -1.67949132]), 20466.187929249329)
(1, array([ 7.92519786, -1.63161194]), 21665.373333173702)
(0.1, array([ 7.97825086, -1.620065  ]), 21959.690190128964)
(0.01, array([ 7.98363466, -1.61889322]), 21989.668380530107)
(0.001, array([ 7.98417384, -1.61877586]), 21992.671779264259)
(0.0001, array([ 7.98422776, -1.61876413]), 21992.972175051837)
```

```
In [282]: regRidgePM = [modelRidge_10[0], modelRidge_5[0], modelRidge_1[0],modelRid
          regRidgeMSE = [modelRidge_10[2], modelRidge_5[2], modelRidge_1[2],modelRi
          plt.plot(regRidgePM, regRidgeMSE)
          plt.title("MSE vs regParam for Ridge Regression.")
          plt.xlabel("regParam")
          plt.ylabel("MSE")
          plt.show()
```



2.3.3 Run Lasso Regression for different regParams (10, 5, 1, .1, 0.01, 0.001,.0001) I chose 500 iterations to (all but) ensure convergence.

```
In [283]: np.random.seed(400)
          modelLasso_10 = linearRegressionGDReg('training dataset.csv', 'testing se
          modelLasso_5 = linearRegressionGDReg('training dataset.csv', 'testing set
          modelLasso_1 = linearRegressionGDReg('training dataset.csv', 'testing set
          modelLasso_pt1 = linearRegressionGDReg('training dataset.csv', 'testing s
          modelLasso_pt01 = linearRegressionGDReg('training dataset.csv', 'testing
          modelLasso_pt001 = linearRegressionGDReg('training dataset.csv', 'testing
          modelLasso_pt0001 = linearRegressionGDReg('training dataset.csv', 'testin
```

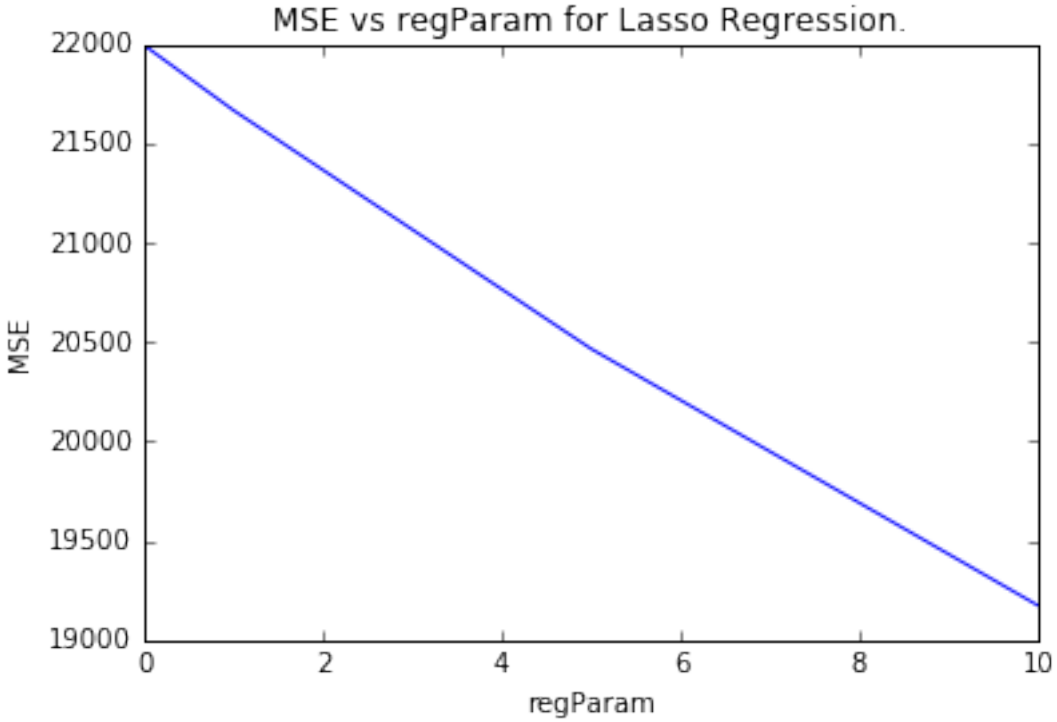
Setting the regParam to 10 shows the minimal MSE when compared to <1, 1, 5.

2.3.4 Summarize Lasso regression results

```
In [285]: print(modelLasso_10)
          print(modelLasso_5)
          print(modelLasso_1)
          print(modelLasso_pt1)
          print(modelLasso_pt01)
          print(modelLasso_pt001)
          print(modelLasso_pt0001)
```

```
(10, array([ 7.90988629, -1.6349445 ]), 21580.80029836332)
(5, array([ 7.94688614, -1.62689151]), 21785.451143800186)
(1, array([ 7.97673617, -1.62039467]), 21951.259760947421)
(0.1, array([ 7.98348336, -1.61892615]), 21988.825655773227)
(0.01, array([ 7.98415871, -1.61877916]), 21992.587510021403)
(0.001, array([ 7.98422625, -1.61876446]), 21992.963748159924)
(0.0001, array([ 7.984233  , -1.61876299]), 21993.001372500985)
```

```
In [286]: regLassoPM = [modelLasso_10[0], modelLasso_5[0], modelLasso_1[0],modelLas
          regLassoMSE = [modelLasso_10[2], modelLasso_5[2], modelLasso_1[2],modelLa
          plt.plot(regRidgePM, regRidgeMSE)
          plt.title("MSE vs regParam for Lasso Regression.")
          plt.xlabel("regParam")
          plt.ylabel("MSE")
          plt.show()
```



Setting the regParam to 10 shows the minimal MSE when compared to <1, 1, 5.
Back to Table of Contents
——- END OF HWK 9 ———