MIDS W261 HW10

HW 10.0: Short answer questions

What is Apache Spark and how is it different to Apache Hadoop? ¶

Apache Spark is an optimized engine that supports general execution graphs over an RDD, or Resilient Distributed Data objects. Spark is 100x faster than Hadoop in memory and 10x faster on disk. It also requires 2-5 times less code to execute the same job than Hadoop.

Fill in the blanks:

Spark API consists of interfaces to develop applications based on it in Java, ... languages (list languages).

Java, Scala, Python, and R

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

Mesos and YARN are the resource managers usually used with Spark

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

An RDD is a Resilient Distributed Data object. RDDs support transformations and actions. An example of creating an RDD is:

```
# set up the reading of CSV data from a file
dataRDD = sc.textFile('data.csv')

# convert each line into a data point
arrayRDD = dataRDD.map(lambda line: [float(v) for v in line.split
(',')])

# bring the first data point of the array back to the driver
print "First data point is: ", arrayRDD.take(1)
```

What is lazy evaluation and give an intuitive example of lazy evaluation and comment on the massive computational savings to be had from lazy evaluation.

Lazy evaluation is the concept that something isn't computed until it's required. RDD transformations are lazily evaluated and are not realized until an action requires them to be. An example of lazy evaluation is reading data from disk or a database. The code describes the read of the data from the device or database, but the actual act of reading the data doesn't happen when that code is encountered in the execution flow of the program. Instead the data is read when some action is requested that requires the

HW 10.1:

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 as a the example document to run an experiment. Report the following: provide a sorted list of tokens in decreasing order of frequency of occurrence.

In [1]:
```
%load_ext autoreload
%autoreload 2
```

In [2]:
```
text_file = sc.textFile("MIDS-MLS-HW-10.txt") #get text file

#use flatmap and then map each word to tuple (word, 1)
#reduce adding up keys and then switch position (count, word) to so
rt
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b) \
             .map(lambda (x,y): (y,x)) \
             .sortByKey(False)

counts.collect()
```

Out[2]:

```
[(108, u''),
 (46, u'the'),
 (23, u'and'),
 (18, u'in'),
 (17, u'of'),
 (12, u'a'),
 (11, u'for'),
 (9, u'code'),
 (9, u'to'),
 (8, u'is'),
 (8, u'='),
 (8, u'data'),
 (7, u'#'),
 (7, u'with'),
 (7, u'this'),
 (7, u'Using'),
 (7, u'your'),
 (7, u'on'),
 (6, u'==='),
 (6, u'HW'),
 (6, u'KMeans'),
 (5, u'from'),
 (5, u'as'),
 (4, u'What'),
 (4, u'Sum'),
 (4, u'Comment'),
 (4, u'Squared'),
 (4, u'==HW'),
 (4, u'each'),
 (4, u'linear'),
 (4, u'example'),
 (4, u'clusters'),
 (4, u'Set'),
 (3, u'words'),
 (3, u'Spark'),
 (3, u'+'),
 (3, u'available'),
 (3, u'lazy'),
 (3, u'100'),
 (3, u'training'),
 (3, u'count'),
 (3, u'Please'),
 (3, u'following'),
 (3, u'report'),
 (3, u'model'),
 (3, u'Errors'),
 (3, u'results'),
 (3, u'using'),
 (3, u'Within'),
 (3, u'===HW'),
 (3, u'it'),
 (3, u'import'),
 (3, u'after'),
 (3, u'plot'),
```

```
 (3, u'an'),
 (3, u'regression'),
 (3, u'document'),
 (3, u'provided'),
 (3, u'x'),
 (2, u'-----------------------'),
 (2, u'homework'),
 (2, u'notebook:'),
 (2, u'(one'),
 (2, u'evaluation'),
 (2, u'--'),
 (2, u'iterations'),
 (2, u'list'),
 (2, u'run'),
 (2, u'plots.'),
 (2, u'per'),
 (2, u'Report'),
 (2, u'https://www.dropbox.com/s/q85t0ytb9apggnh/kmeans_data.txt?d
l=0'),
 (2, u'here'),
 (2, u'clusters.'),
 (2, u'RIDGE'),
 (2, u'10.6.1'),
 (2, u'Apache'),
 (2, u'LASSO'),
 (2, u'word'),
 (2, u'set.'),
 (2, u'HW10.3.'),
 (2, u'regression.'),
 (2, u'NOTE'),
 (2, u'how'),
 (2, u'kmeans_data.txt'),
 (2, u'In'),
 (2, u'set'),
 (2, u'testing'),
 (2, u'iterations,'),
 (2, u'between'),
 (2, u'be'),
 (2, u'found'),
 (2, u'points'),
 (2, u'(OPTIONAL)'),
 (2, u'via'),
 (2, u'or'),
 (2, u'findings.'),
 (2, u'one'),
 (2, u'Explain.'),
 (2, u'differences'),
 (2, u'"myModelPath")'),
 (2, u'up'),
 (2, u'Generate'),
 (2, u'HW10.3'),
 (2, u'vector'),
 (2, u'any'),
 (2, u'that'),
```

```
 (2, u'repeat'),
 (2, u'decreasing'),
 (2, u'order'),
 (2, u'Fill'),
 (1, u'y)'),
 (1, u'all'),
 (1, u'weight(X)='),
 (1, u'10.2:'),
 (1, u'not'),
 (1, u'(cluster'),
 (1, u'consists'),
 (1, u'based'),
 (1, u'parameters'),
 (1, u'error(point):'),
 (1, u'(Euclidean'),
 (1, u'Modify'),
 (1, u'3'),
 (1, u'Here'),
 (1, u'languages'),
 (1, u'snippet'),
 (1, u'SQRT(X.X)='),
 (1, u'2,'),
 (1, u'weeks'),
 (1, u'submissions'),
 (1, u'Load'),
 (1, u'return'),
 (1, u'runs=10,'),
 (1, u'KMeans,'),
 (1, u'load'),
 (1, u'experiment.'),
 (1, u'homeworks'),
 (1, u'compute'),
 (1, u'sc.textFile("kmeans_data.txt")'),
 (1, u'bringing'),
 (1, u'runs'),
 (1, u'resource'),
 (1, u'intuitive'),
 (1, u'questions==='),
 (1, u'array([float(x)'),
 (1, u'frequency.'),
 (1, u'(point'),
 (1, u'1/||X||,'),
 (1, u'(or'),
 (1, u'SPECIAL'),
 (1, u'Learning'),
 (1, u'KMeansModel.load(sc,'),
 (1, u'where'),
 (1, u'initializationMode="random")'),
 (1, u'generation'),
 (1, u'begin'),
 (1, u'(homegrown'),
 (1, u'provided).'),
 (1, u'10.4:'),
 (1, u'Assignments"'),
```

```
 (1, u'sqrt(sum([x**2'),
 (1, u'please'),
 (1, u'labeled'),
 (1, u'cell'),
 (1, u'blanks:'),
 (1, u'above'),
 (1, u'"Teams'),
 (1, u'at:'),
 (1, u'interfaces'),
 (1, u'math'),
 (1, u'HW10'),
 (1, u'sameModel'),
 (1, u'This'),
 (1, u'W261'),
 (1, u'ISVC):'),
 (1, u'modify'),
 (1, u'20'),
 (1, u'fun'),
 (1, u'group'),
 (1, u'Weight'),
 (1, u'column'),
 (1, u'completing'),
 (1, u'implementation'),
 (1, u'03/15/2016'),
 (1, u'follows:'),
 (1, u'UC'),
 (1, u'comment'),
 (1, u'letters'),
 (1, u'?????'),
 (1, u'=================='),
 (1, u'distributed'),
 (1, u'done'),
 (1, u'10.6:'),
 (1, u'DATSCI'),
 (1, u'array'),
 (1, u'==================END'),
 (1, u'https://www.dropbox.com/s/3nsthvp8g2rrrdh/EM-Kmeans.ipynb?d
l=0'),
 (1, u'clustering'),
 (1, u'use'),
 (1, u'findings'),
 (1, u'submit'),
 (1, u'HW10.5.'),
 (1, u'(a-z)'),
 (1, u'error(point)).reduce(lambda'),
 (1, u'Homeworks'),
 (1, u'numpy'),
 (1, u'X2^2)'),
 (1, u'sort'),
 (1, u'form'),
 (1, u'occurrence.'),
 (1, u'========================='),
 (1, u'forward'),
 (1, u'manner.'),
```

```
 (1, u'(using'),
 (1, u'had'),
 (1, u'case'),
 (1, u'10.0:'),
 (1, u'Run'),
 (1, u'MLlib-centric'),
 (1, u'V1.3'),
 (1, u'============================================='),
 (1, u'See'),
 (1, u"')]))"),
 (1, u'def'),
 (1, u'model?'),
 (1, u'links'),
 (1, u'parse'),
 (1, u'single'),
 (1, u'Call'),
 (1, u'tab'),
 (1, u'KMeans.train(parsedData,'),
 (1, u'WSSSE'),
 (1, u'different'),
 (1, u'data)'),
 (1, u'x,'),
 (1, u'provide'),
 (1, u'-'),
 (1, u'length'),
 (1, u'sqrt'),
 (1, u'write'),
 (1, u'answer'),
 (1, u'resulting'),
 (1, u'program.'),
 (1, u'Machine'),
 (1,
  u'https://docs.google.com/spreadsheets/d/1ncFQl5Tovn-16slD8mYjP_
nzMTPSfiGeLLzW8v_sMjg/edit?usp=sharing'),
 (1,
  u'==============================================================
============='),
 (1, u'driver'),
 (1, u'Download'),
 (1, u'SQRT(X1^2'),
 (1, u'Berkeley,'),
 (1, u'develop'),
 (1, u'data.map(lambda'),
 (1, u'tune'),
 (1, u'documents).'),
 (1, u'tokens'),
 (1, u'lower'),
 (1, u'Mesos'),
 (1, u'......'),
 (1, u'point:'),
 (1, u'savings'),
 (1, u'1'),
 (1, u'algorithms'),
 (1, u'pyspark.mllib.clustering'),
```

```
 (1, u'hyper'),
 (1, u'show'),
 (1, u'text'),
 (1, u'Error'),
 (1, u'follow'),
 (1, u'"Gradient'),
 (1, u'Final'),
 (1, u"line.split('"),
 (1, u'inverse'),
 (1, u'number'),
 (1, u'Linear'),
 (1, u"MLlib's"),
 (1, u'going'),
 (1, u'team)'),
 (1, u'10.5:'),
 (1, u'#10==='),
 (1, u'X2.'),
 (1, u'do'),
 (1, u'good'),
 (1, u'get'),
 (1, u'evaluate'),
 (1, u'Spark,'),
 (1, u'Kmean'),
 (1, u'framework'),
 (1, u'made'),
 (1, u'carefully.'),
 (1, u'sorted'),
 (1, u'dataset'),
 (1, u'instructions'),
 (1, u'follows'),
 (1,
  u'https://www.dropbox.com/s/atzqkc0p1eajuz6/LinearRegression-Not
ebook-Challenge.ipynb?dl=0'),
 (1, u'server'),
 (1, u'API'),
 (1, u'either'),
 (1, u'output'),
 (1, u'hundred)'),
 (1, u'Again'),
 (1, u'often'),
 (1, u'ASSIGNMENT'),
 (1, u'experiments'),
 (1, u'located'),
 (1, u'frequency'),
 (1, u'Build'),
 (1, u'X'),
 (1, u'measure'),
 (1, u'progress'),
 (1, u'Save'),
 (1, u'NOTE:'),
 (1, u"MLLib's"),
 (1, u'parsedData'),
 (1, u'following:'),
 (1, u'10.1.1'),
```

```
 (1, u'Evaluate'),
 (1, u'separate'),
 (1, u'find'),
 (1, u'weighted'),
 (1, u'Regression'),
 (1, u'Short'),
 (1, u'creating'),
 (1, u'team'),
 (1, u'"'),
 (1, u'str(WSSSE))'),
 (1, u'snippet:'),
 (1, u'Hadoop?'),
 (1, u'10.6.2'),
 (1, u'by'),
 (1, u'Justify'),
 (1, u'clusters.centers[clusters.predict(point)]'),
 (1, u'languages).'),
 (1, u'la'),
 (1, u'KMEans'),
 (1, u'massive'),
 (1,
  u'https://docs.google.com/forms/d/1ZOr9RnIe_A06AcZDB6K1mJN4vrLeS
mS2PD6Xm3eOiis/viewform?usp=send_form'),
 (1, u'maxIterations=10,'),
 (1, u'first'),
 (1, u'computational'),
 (1, u'assignments'),
 (1, u'Team'),
 (1, u'Scale'),
 (1, u'computing'),
 (1, u'DropBox'),
 (1, u'center)]))'),
 (1, u'LinearRegressionWithSGD'),
 (1, u'are'),
 (1, u'management'),
 (1, u'plots'),
 (1, u'give'),
 (1, u'appears'),
 (1, u'INSTURCTIONS'),
 (1, u'2'),
 (1, u'parsedData.map(lambda'),
 (1, u'Plot'),
 (1, u'thru'),
 (1, u'more'),
 (1, u'provide,'),
 (1, u'iteration,'),
 (1, u'||X||'),
 (1, u'back'),
 (1, u'RDD'),
 (1, u'norm):'),
 (1, u'10.1:'),
 (1, u'===MIDS'),
 (1, u'(and'),
 (1, u'10'),
```

```
 (1, u'clusters.save(sc,'),
 (1, u'cluster'),
 (1, u'can'),
 (1, u'iterations.'),
 (1, u'KMeansModel'),
 (1, u'Your'),
 (1, u'(list'),
 (1, u'exercise.'),
 (1, u'evaluation.'),
 (1, u'X1'),
 (1, u'print("Within'),
 (1, u'MLLib'),
 (1, u'words.'),
 (1, u'at'),
 (1, u'experiements'),
 (1, u'work'),
 (1, u'Then'),
 (1, u'data.'),
 (1, u'line:'),
 (1, u'Java,'),
 (1, u'instance'),
 (1, u'other'),
 (1, u'(regularization)".'),
 (1, u'blanks'),
 (1, u'assignment'),
 (1, u'DATA:'),
 (1, u'10.3:'),
 (1, u'y:'),
 (1, u'applications'),
 (1, u'notebook'),
 (1, u'such'),
 (1, u'descent'),
 (1, u'center'),
 (1, u'sets'),
 (1, u'element'),
 (1, u'train'),
 (1, u'code)'),
 (1, u'HW10.4.')]
```

HW 10.1.1

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 [3]:
```
import re
text_file = sc.textFile("MIDS-MLS-HW-10.txt")
#repeat above filtering for regular expressions that only start wit
h lowercase letters
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .filter(lambda token: re.match(r'^[a-z]+', token)) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b) \
             .map(lambda (x,y): (y,x)) \
             .sortByKey(False)

counts.collect()
```

Out[3]:

```
[(46, u'the'),
 (23, u'and'),
 (18, u'in'),
 (17, u'of'),
 (12, u'a'),
 (11, u'for'),
 (9, u'code'),
 (9, u'to'),
 (8, u'is'),
 (8, u'data'),
 (7, u'with'),
 (7, u'this'),
 (7, u'on'),
 (7, u'your'),
 (5, u'from'),
 (5, u'as'),
 (4, u'clusters'),
 (4, u'each'),
 (4, u'linear'),
 (4, u'example'),
 (3, u'count'),
 (3, u'words'),
 (3, u'report'),
 (3, u'available'),
 (3, u'lazy'),
 (3, u'following'),
 (3, u'training'),
 (3, u'model'),
 (3, u'results'),
 (3, u'using'),
 (3, u'x'),
 (3, u'import'),
 (3, u'plot'),
 (3, u'it'),
 (3, u'an'),
 (3, u'regression'),
 (3, u'document'),
 (3, u'provided'),
 (3, u'after'),
 (2, u'homework'),
 (2, u'notebook:'),
 (2, u'evaluation'),
 (2, u'list'),
 (2, u'run'),
 (2, u'regression.'),
 (2, u'per'),
 (2, u'https://www.dropbox.com/s/q85t0ytb9apggnh/kmeans_data.txt?d
l=0'),
 (2, u'here'),
 (2, u'iterations'),
 (2, u'word'),
 (2, u'set.'),
 (2, u'clusters.'),
 (2, u'plots.'),
```

```
(2, u'findings.'),
(2, u'set'),
(2, u'testing'),
(2, u'iterations,'),
(2, u'between'),
(2, u'be'),
(2, u'found'),
(2, u'how'),
(2, u'via'),
(2, u'or'),
(2, u'one'),
(2, u'that'),
(2, u'differences'),
(2, u'up'),
(2, u'vector'),
(2, u'any'),
(2, u'kmeans_data.txt'),
(2, u'repeat'),
(2, u'points'),
(2, u'decreasing'),
(2, u'order'),
(1, u'y)'),
(1, u'sameModel'),
(1, u'all'),
(1, u'homeworks'),
(1, u'consists'),
(1, u'snippet'),
(1, u'based'),
(1, u'parameters'),
(1, u'error(point):'),
(1, u'had'),
(1, u'languages'),
(1, u'weeks'),
(1, u'submissions'),
(1, u'return'),
(1, u'runs=10,'),
(1, u'load'),
(1, u'runs'),
(1, u'experiment.'),
(1, u'compute'),
(1, u'sc.textFile("kmeans_data.txt")'),
(1, u'bringing'),
(1, u'point:'),
(1, u'resource'),
(1, u'questions==='),
(1, u'array([float(x)'),
(1, u'frequency.'),
(1, u'where'),
(1, u'initializationMode="random")'),
(1, u'generation'),
(1, u'provided).'),
(1, u'sqrt(sum([x**2'),
(1, u'please'),
(1, u'cell'),
```

```
 (1, u'blanks:'),
 (1, u'above'),
 (1, u'at:'),
 (1, u'math'),
 (1, u'interfaces'),
 (1, u'modify'),
 (1, u'not'),
 (1, u'group'),
 (1, u'column'),
 (1, u'completing'),
 (1, u'implementation'),
 (1, u'length'),
 (1, u'resulting'),
 (1, u'follows:'),
 (1, u'pyspark.mllib.clustering'),
 (1, u'comment'),
 (1, u'letters'),
 (1, u'distributed'),
 (1, u'done'),
 (1, u'array'),
 (1, u'https://www.dropbox.com/s/3nsthvp8g2rrrdh/EM-Kmeans.ipynb?d
l=0'),
 (1, u'clustering'),
 (1, u'use'),
 (1, u'findings'),
 (1, u'submit'),
 (1, u'error(point)).reduce(lambda'),
 (1, u'forward'),
 (1, u'numpy'),
 (1, u'sort'),
 (1, u'form'),
 (1, u'occurrence.'),
 (1, u'manner.'),
 (1, u'case'),
 (1, u'intuitive'),
 (1, u'single'),
 (1, u'labeled'),
 (1, u'fun'),
 (1, u'def'),
 (1, u'model?'),
 (1, u'links'),
 (1, u'parse'),
 (1, u'tab'),
 (1, u'different'),
 (1, u'develop'),
 (1, u'x,'),
 (1, u'provide'),
 (1, u'weight(X)='),
 (1, u'sqrt'),
 (1, u'write'),
 (1, u'answer'),
 (1, u'program.'),
 (1, u'begin'),
 (1,
```

```
  u'https://docs.google.com/spreadsheets/d/1ncFQl5Tovn-16slD8mYjP_
nzMTPSfiGeLLzW8v_sMjg/edit?usp=sharing'),
 (1, u'driver'),
 (1, u'data)'),
 (1, u'data.map(lambda'),
 (1, u'tune'),
 (1, u'documents).'),
 (1, u'tokens'),
 (1, u'lower'),
 (1, u'savings'),
 (1, u'algorithms'),
 (1, u'hyper'),
 (1, u'show'),
 (1, u'text'),
 (1, u'experiments'),
 (1, u'follow'),
 (1, u'progress'),
 (1, u'find'),
 (1, u"line.split('"),
 (1, u'inverse'),
 (1, u'instance'),
 (1, u'going'),
 (1, u'team)'),
 (1, u'do'),
 (1, u'good'),
 (1, u'get'),
 (1, u'evaluate'),
 (1, u'framework'),
 (1, u'carefully.'),
 (1, u'sorted'),
 (1, u'dataset'),
 (1, u'instructions'),
 (1, u'evaluation.'),
 (1, u'follows'),
 (1,
  u'https://www.dropbox.com/s/atzqkc0p1eajuz6/LinearRegression-Not
ebook-Challenge.ipynb?dl=0'),
 (1, u'server'),
 (1, u'assignments'),
 (1, u'y:'),
 (1, u'either'),
 (1, u'often'),
 (1, u'parsedData.map(lambda'),
 (1, u'back'),
 (1, u'clusters.save(sc,'),
 (1, u'frequency'),
 (1, u'are'),
 (1, u'measure'),
 (1, u'experiements'),
 (1, u'parsedData'),
 (1, u'following:'),
 (1, u'iterations.'),
 (1, u'print("Within'),
 (1, u'separate'),
```

```
 (1, u'creating'),
 (1, u'team'),
 (1, u'str(WSSSE))'),
 (1, u'snippet:'),
 (1, u'plots'),
 (1, u'by'),
 (1, u'languages).'),
 (1, u'la'),
 (1, u'massive'),
 (1,
  u'https://docs.google.com/forms/d/1ZOr9RnIe_A06AcZDB6K1mJN4vrLeS
mS2PD6Xm3eOiis/viewform?usp=send_form'),
 (1, u'maxIterations=10,'),
 (1, u'first'),
 (1, u'computational'),
 (1, u'number'),
 (1, u'weighted'),
 (1, u'center)]))'),
 (1, u'management'),
 (1, u'computing'),
 (1, u'appears'),
 (1, u'thru'),
 (1, u'more'),
 (1, u'provide,'),
 (1, u'iteration,'),
 (1, u'train'),
 (1, u'made'),
 (1, u'cluster'),
 (1, u'work'),
 (1, u'can'),
 (1, u'norm):'),
 (1, u'clusters.centers[clusters.predict(point)]'),
 (1, u'hundred)'),
 (1, u'exercise.'),
 (1, u'give'),
 (1, u'words.'),
 (1, u'at'),
 (1, u'data.'),
 (1, u'line:'),
 (1, u'output'),
 (1, u'located'),
 (1, u'other'),
 (1, u'blanks'),
 (1, u'assignment'),
 (1, u'applications'),
 (1, u'notebook'),
 (1, u'such'),
 (1, u'descent'),
 (1, u'center'),
 (1, u'sets'),
 (1, u'element'),
 (1, u'code)')]
```

HW 10.2: KMeans a la MLLib

Using the following MLlib-centric KMeans code snippet:

In [19]:
```
from pyspark.mllib.clustering import KMeans, KMeansModel
from numpy import array
from math import sqrt
from datetime import datetime

# Load and parse the data
# NOTE  kmeans_data.txt is available here
#          https://www.dropbox.com/s/q85t0ytb9apggnh/kmeans_data.tx
t?dl=0
data = sc.textFile("kmeans_data.txt")
parsedData = data.map(lambda line: array([float(x) for x in line.sp
lit(' ')]))
```

In [26]:
```
def model(iterations):
    startTime = datetime.now()
    # Build the model (cluster the data)
    clusters = KMeans.train(parsedData, 2, maxIterations=iteration
s,
            initializationMode="random")

    print 'Cluster centers: ',clusters.clusterCenters,'\n'

    # Evaluate clustering by computing Within Set Sum of Squared Er
rors
    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(lambd
a x, y: x + y)
    print("Within Set Sum of Squared Error = " + str(WSSSE))

    print 'Execution in ', datetime.now() - startTime

# Save and load model
#clusters.save(sc, "myModelPath")
#sameModel = KMeansModel.load(sc, "myModelPath")
```

Run this code snippet and list the clusters that you find and compute the Within Set Sum of Squared Errors for the found clusters. Comment on your findings.

** NOTE: kmeans_data.txt is available here (https://www.dropbox.com/s/q85t0ytb9apggnh/kmeans_data.txt?dl=0)

In [27]:
```
model(1)
```

```
Cluster centers:  [array([ 3.68,  3.68,  3.68]), array([ 9.2,  9.
2,  9.2])]

Within Set Sum of Squared Error = 19.1218409156
Execution in  0:00:00.082860
```

In [28]:
```
model(2)
```

```
Cluster centers:  [array([ 0.1,  0.1,  0.1]), array([ 9.1,  9.1,
9.1])]

Within Set Sum of Squared Error = 0.692820323028
Execution in  0:00:00.093207
```

In [29]:
```
model(3)
```

```
Cluster centers:  [array([ 0.1,  0.1,  0.1]), array([ 9.1,  9.1,
9.1])]

Within Set Sum of Squared Error = 0.692820323028
Execution in  0:00:00.092482
```

With such a simple data set the model converges rapidly in just 2 iterations. More interesting is the execution time of less than .1 second.

HW 10.3:

Download the [KMeans notebook (https://www.dropbox.com/s/3nsthvp8g2rrrdh/EM-Kmeans.ipynb?dl=0)](https://www.dropbox.com/s/3nsthvp8g2rrrdh/EM-Kmeans.ipynb?dl=0)

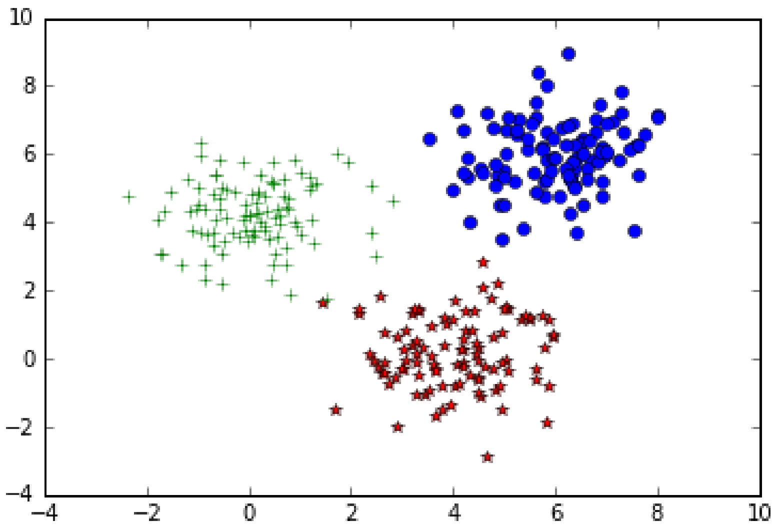
Generate 3 clusters with 100 (one hundred) data points per cluster (using the code provided). Plot the data. Then run MLlib's Kmean implementation on this data and report your results as follows:

- 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. Comment on the progress of this measure as the KMEans algorithms runs for more iterations

In [32]:
```
%matplotlib inline
import numpy as np
import pylab
import json
size1 = size2 = size3 = 100 #set size

#get samples from multivariate distribution
samples1 = np.random.multivariate_normal([4, 0], [[1, 0],[0, 1]], s
ize1)
data = samples1
samples2 = np.random.multivariate_normal([6, 6], [[1, 0],[0, 1]], s
ize2)
data = np.append(data,samples2, axis=0)
samples3 = np.random.multivariate_normal([0, 4], [[1, 0],[0, 1]], s
ize3)
data = np.append(data,samples3, axis=0)
# Randomlize data
data = data[np.random.permutation(size1+size2+size3),]
np.savetxt('data.csv',data,delimiter = ',')
```

```
/Users/rcordell/Documents/MIDS/W261/W261env/lib/python2.7/site-pac
kages/matplotlib/font_manager.py:273: UserWarning: Matplotlib is b
uilding the font cache using fc-list. This may take a moment.
  warnings.warn('Matplotlib is building the font cache using fc-li
st. This may take a moment.')
```

In [33]:
```
#plot
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()
```

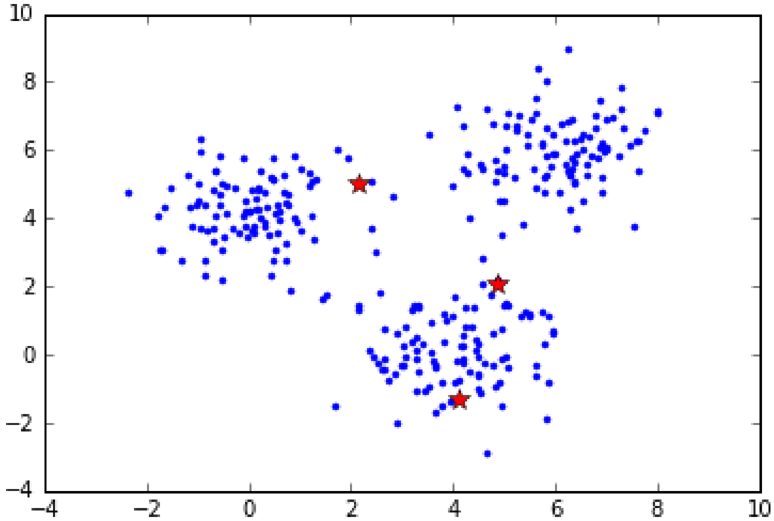


In [34]:
```
from pyspark.mllib.clustering import KMeans, KMeansModel
from numpy import array
from math import sqrt

# Load and parse the data
# NOTE  kmeans_data.txt is available here
#          https://www.dropbox.com/s/q85t0ytb9apggnh/kmeans_data.tx
t?dl=0
data = sc.textFile('data.csv')
parsedData = data.map(lambda line: array([float(x) for x in line.sp
lit(',')]))
```

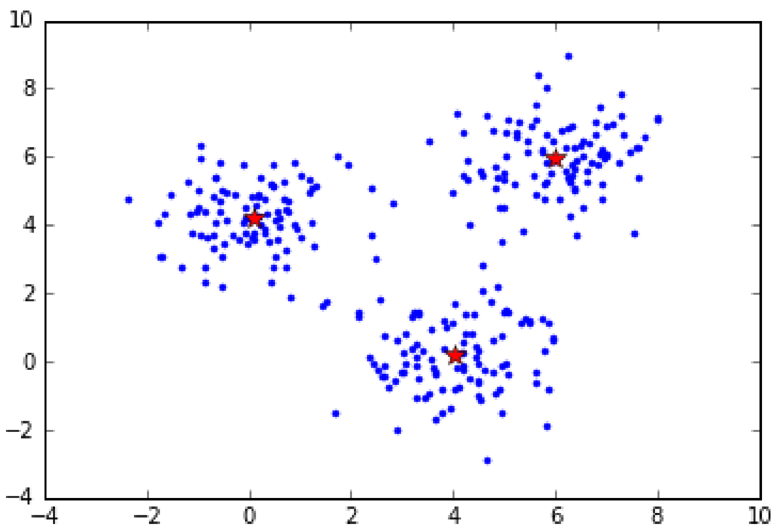
In [35]:
```
#plot centroids and data points for each iteration
def plot_iteration(means):
    pylab.plot(samples1[:, 0], samples1[:, 1], '.', color = 'blue')
    pylab.plot(samples2[:, 0], samples2[:, 1], '.', color = 'blue')
    pylab.plot(samples3[:, 0], samples3[:, 1],'.', color = 'blue')
    pylab.plot(means[0][0], 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()

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

In [36]:
```
for i in [1,10,20,100]:
    # Build the model (cluster the data)
    clusters = KMeans.train(parsedData, 3, maxIterations=i,
            initializationMode="random")

    print "\n{0} Iterations:\n".format(i)
    print 'Cluster centers: \n'
    for c in clusters.clusterCenters:
        print '\t{0:2.8f},{1:2.8f}'.format(c[0],c[1])

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

    plot_iteration(clusters.clusterCenters)
```

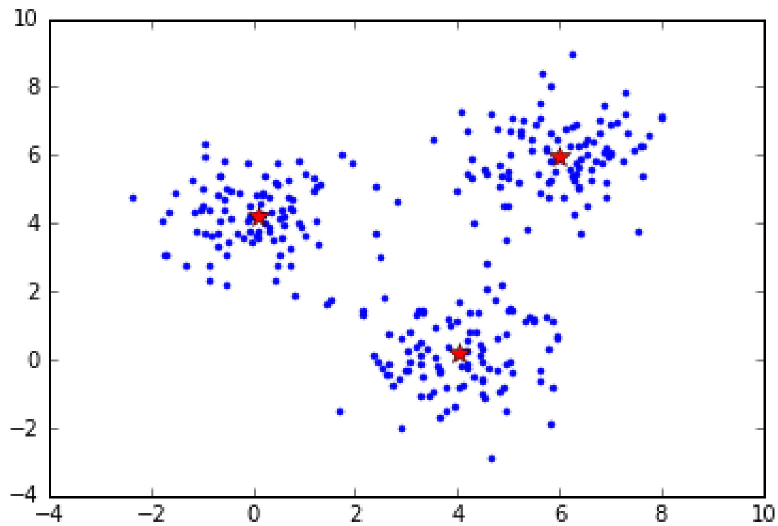
```
1 Iterations:

Cluster centers:

        4.11640627,-1.32534773
        2.15521012,5.03511735
        4.87101304,2.09547679
Within Set Sum of Squared Error = 758.441062452
```



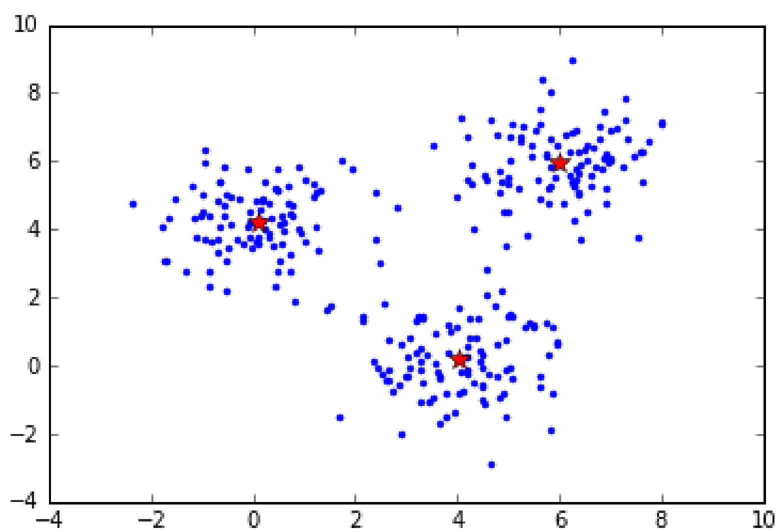

```
10 Iterations:

Cluster centers:

        4.04916143,0.16387724
        6.00008142,5.97948160
        0.09506530,4.21958732
Within Set Sum of Squared Error = 375.885589126
```



```
20 Iterations:

Cluster centers:

        6.00008142,5.97948160
        4.04916143,0.16387724
        0.09506530,4.21958732
Within Set Sum of Squared Error = 375.885589126
```




```
100 Iterations:

Cluster centers:

        6.00008142,5.97948160
        4.04916143,0.16387724
        0.09506530,4.21958732
Within Set Sum of Squared Error = 375.885589126
```



The Within Set Sum of Squared Error converges to its final value within the first 10 iterations. There is no further improvement after the 10th iteration.

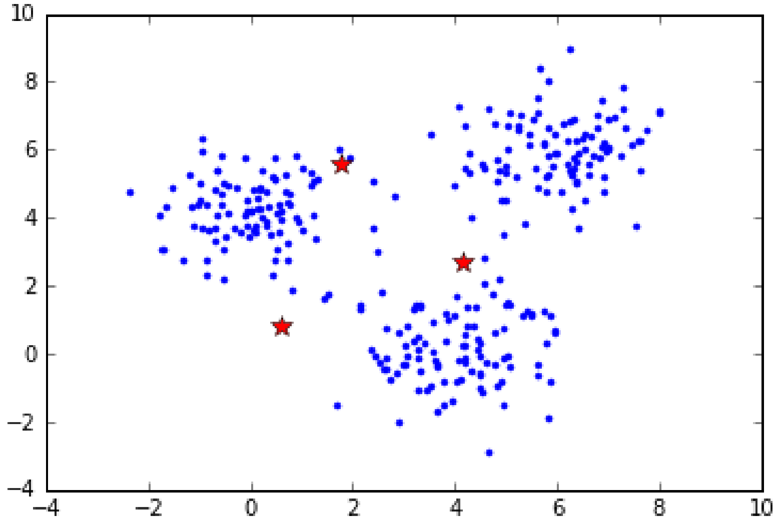
HW 10.4:

Using the KMeans code (homegrown code) provided repeat the experiments in HW10.3. Comment on any differences between the results in HW10.3 and HW10.4. Explain.

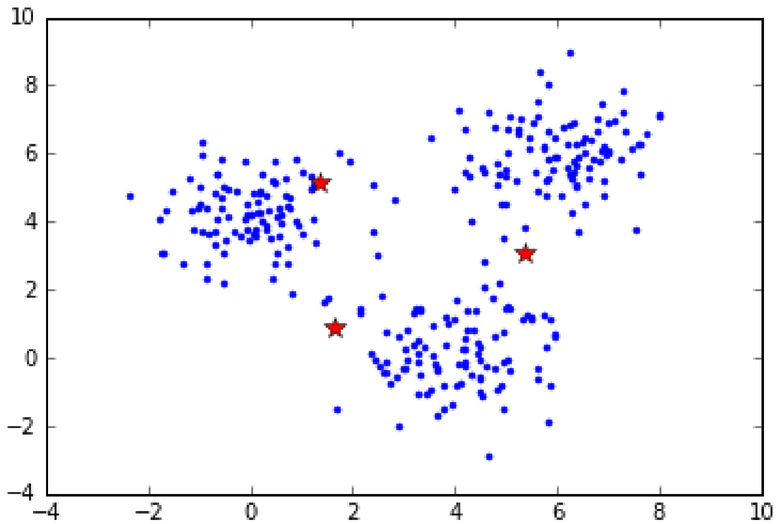
In [37]:
```
import numpy as np

#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).argmi
n()
    return (closest_centroid_idx,(x,1))

#plot centroids and data points for each iteration
def plot_iteration(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()
```

In [38]:
```
K = 3
# Initialization: initialization of parameter is fixed to show an e
xample
centroids = np.array([[0.0,0.0],[2.0,2.0],[0.0,7.0]])

D = sc.textFile("./data.csv").cache()
iter_num = 0
for i in range(10):
    res = D.map(nearest_centroid).reduceByKey(lambda x,y : (x[0]+y
[0],x[1]+y[1])).collect()
    #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_new = np.array([x[1][0]/x[1][1] for x in res])  #divi
de by cluster size
    if np.sum(np.absolute(centroids_new-centroids))<0.001:
        break
    print "Iteration" + str(iter_num)
    iter_num = iter_num + 1
    centroids = centroids_new
    print centroids
    plot_iteration(centroids)
print "Final Results:"
print centroids
```

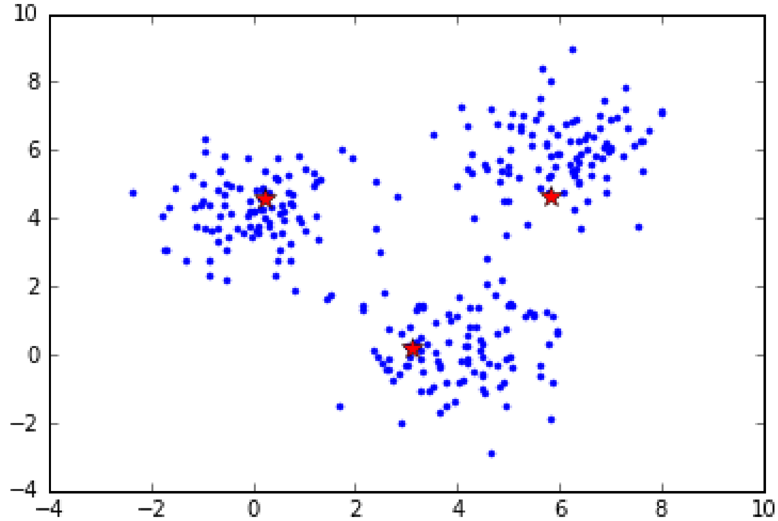
```
Iteration0
[[ 0.59467168  0.8063804 ]
 [ 4.17348094  2.72179691]
 [ 1.752546    5.58021358]]
```



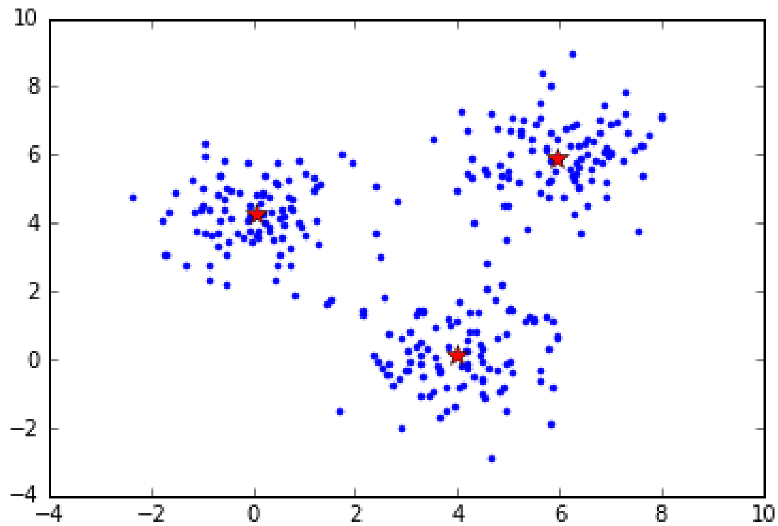
```
Iteration1
[[ 1.6269627   0.90948466]
 [ 5.36290595  3.09860347]
 [ 1.33618468  5.13344031]]
```



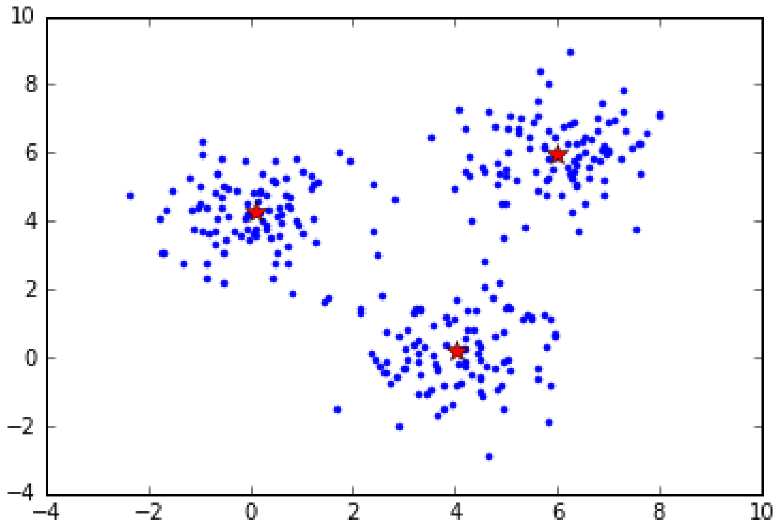
```
Iteration2
[[ 3.10663118  0.20929783]
 [ 5.83174418  4.65918365]
 [ 0.23081098  4.5494637 ]]
```



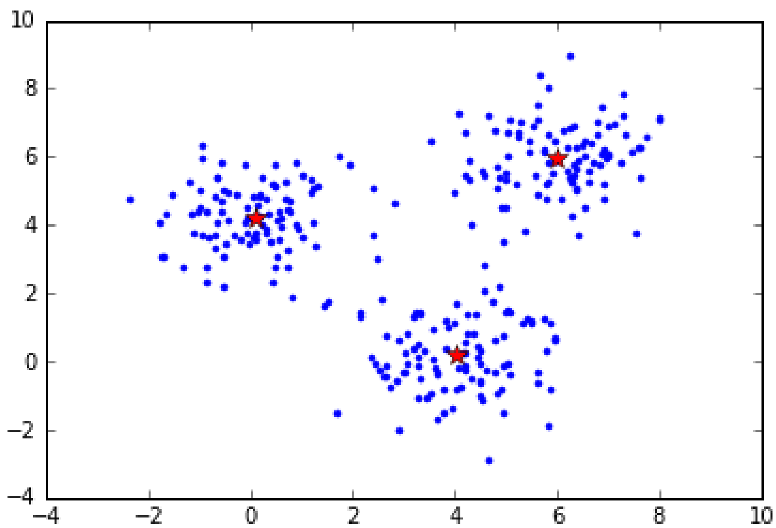
```
Iteration3
[[ 3.98348644  0.14719222]
 [ 5.97517883  5.9116773 ]
 [ 0.06710426  4.27057789]]
```



```
Iteration4
[[ 4.02301719  0.17839295]
 [ 6.00008142  5.9794816 ]
 [ 0.08166858  4.24562871]]
```



```
Iteration5
[[ 4.04916143  0.16387724]
 [ 6.00008142  5.9794816 ]
 [ 0.0950653   4.21958732]]
```



```
Final Results:
[[ 4.04916143  0.16387724]
 [ 6.00008142  5.9794816 ]
 [ 0.0950653   4.21958732]]
```

The homegrown algorithm seems to converge less quickly to the cluster centers than the MLLib code in 10.3 when judged by looking at the plots. In 10.3 the centers move to very near the cluster centers by the 2nd iteration while it takes an extra iteration or so in the homegrown code. However, this is most likely a result of initial cluster center initialization.

HW 10.5: (OPTIONAL)

Using the KMeans code (homegrown code) provided modify it to do a weighted KMeans and repeat the experiements in HW10.3. Comment on any differences between the results in HW10.3 and HW10.5. Explain.

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

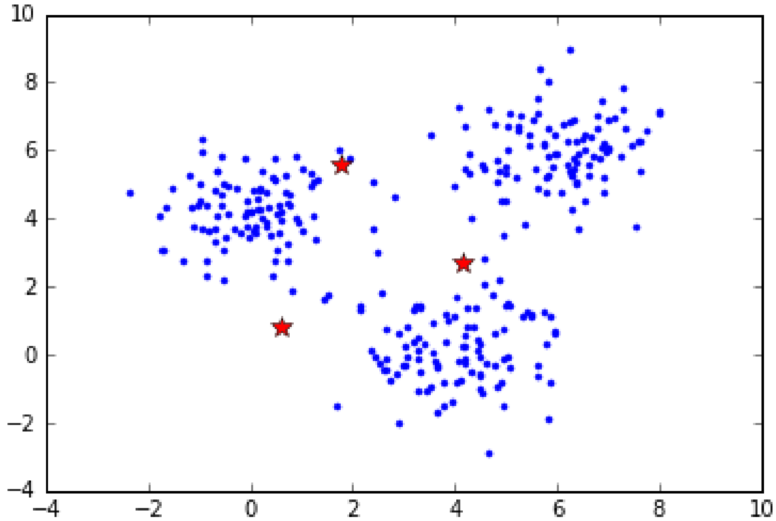
$weight(X) = \frac{1}{||X||}$, where $||X|| = \sqrt{X.X} = \sqrt{(X_1^2 + X_2^2)}$

Here X is vector made up of X_1 and X_2.

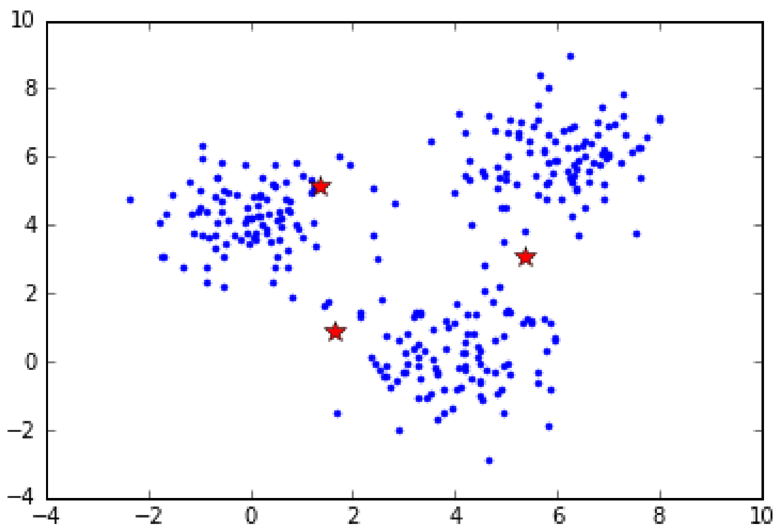
In [92]:
```
import numpy as np

#Calculate which class each data point belongs to
def nearest_centroid(line):
    x = np.array([float(f) for f in line.split(',')])
    # the weight is calculated as (1/np.linalg.norm(x))
    closest_centroid_idx = \
        np.sum((1.0/np.linalg.norm(x))*(x - centroids)**2, axis=1).
argmin()
    return (closest_centroid_idx,(x,1))

#plot centroids and data points for each iteration
def plot_iteration(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()
```

In [39]:
```
K = 3
# Initialization: initialization of parameter is fixed to show an e
xample
centroids = np.array([[0.0,0.0],[2.0,2.0],[0.0,7.0]])

D = sc.textFile("./data.csv").cache()
iter_num = 0
for i in range(10):
    res = D.map(nearest_centroid).reduceByKey(lambda x,y : (x[0]+y
[0],x[1]+y[1])).collect()
    #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_new = np.array([x[1][0]/x[1][1] for x in res])  #divi
de by cluster size
    if np.sum(np.absolute(centroids_new-centroids))<0.001:
        break
    print "Iteration" + str(iter_num)
    iter_num = iter_num + 1
    centroids = centroids_new
    print centroids
    plot_iteration(centroids)
print "Final Results:"
print centroids
```

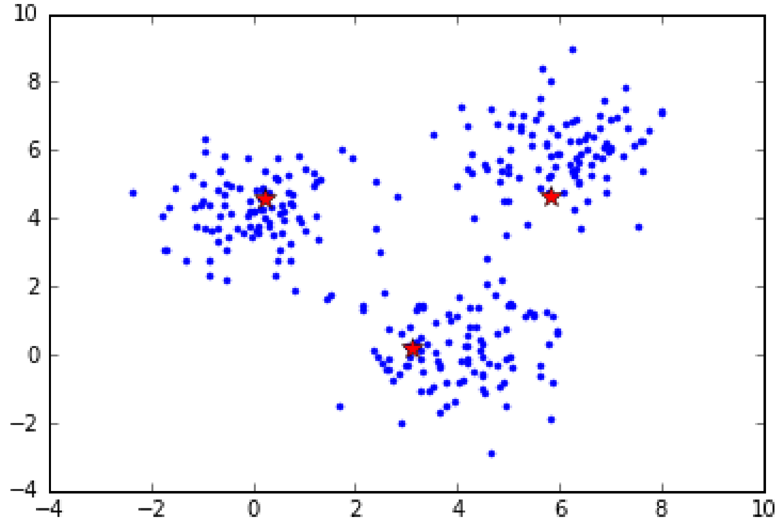
```
Iteration0
[[ 0.59467168  0.8063804 ]
 [ 4.17348094  2.72179691]
 [ 1.752546    5.58021358]]
```



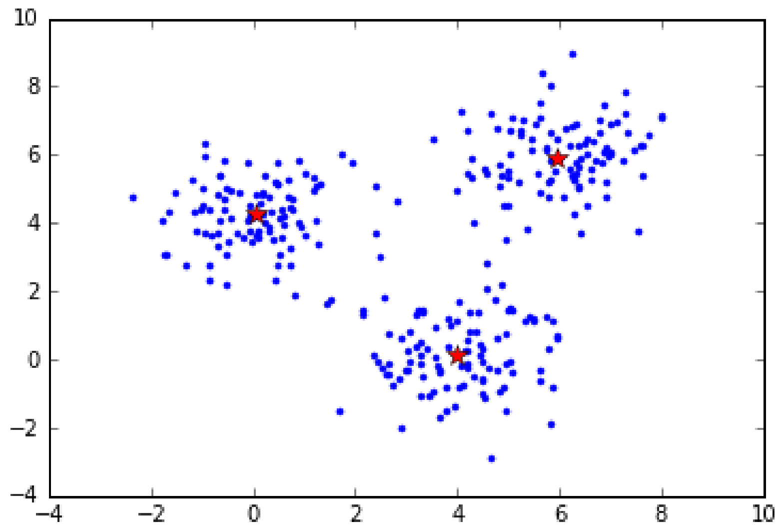

```
Iteration1
[[ 1.6269627   0.90948466]
 [ 5.36290595  3.09860347]
 [ 1.33618468  5.13344031]]
```



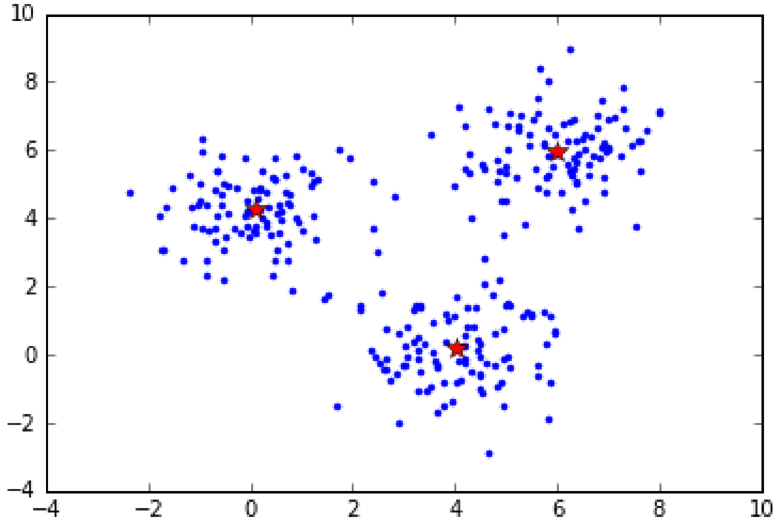

```
Iteration2
[[ 3.10663118  0.20929783]
 [ 5.83174418  4.65918365]
 [ 0.23081098  4.5494637 ]]
```



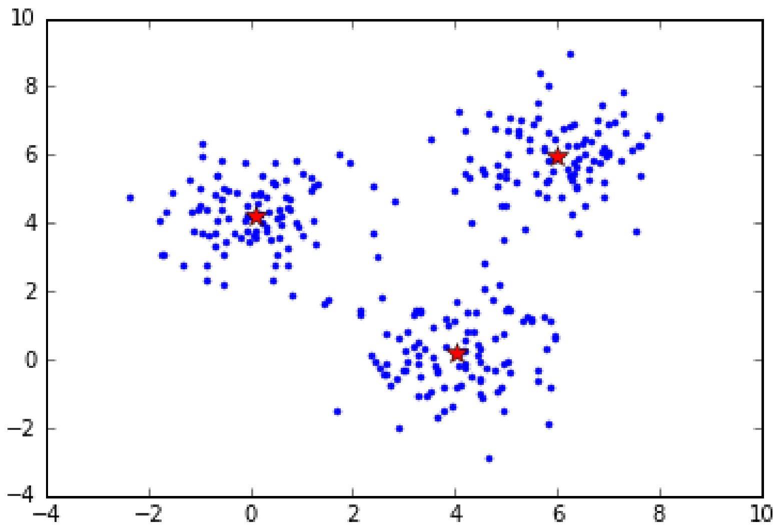
```
Iteration3
[[ 3.98348644  0.14719222]
 [ 5.97517883  5.9116773 ]
 [ 0.06710426  4.27057789]]
```



```
Iteration4
[[ 4.02301719  0.17839295]
 [ 6.00008142  5.9794816 ]
 [ 0.08166858  4.24562871]]
```




```
Iteration5
[[ 4.04916143  0.16387724]
 [ 6.00008142  5.9794816 ]
 [ 0.0950653   4.21958732]]
```




```
Final Results:
[[ 4.04916143  0.16387724]
 [ 6.00008142  5.9794816 ]
 [ 0.0950653   4.21958732]]
```

HW 10.6: Linear Regression (OPTIONAL)

HW 10.6.1

Using the following linear regression notebook:

https://www.dropbox.com/s/atzqkc0p1eajuz6/LinearRegression-Notebook-Challenge.ipynb?dl=0 (https://www.dropbox.com/s/atzqkc0p1eajuz6/LinearRegression-Notebook-Challenge.ipynb?dl=0)

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 and words.

Data Generation

In [62]:
```
import numpy as np
from pyspark.mllib.regression import LabeledPoint, LinearRegression
WithSGD
import csv
def data_generate(fileName, w=[0,0], size=100, seed=0):
    # accept an integer random seed as a parameter to facilitate
    # the generation of difference data sets
    np.random.seed(seed)
    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,'wb') as f:
        writer = csv.writer(f)
        for row in data:
            writer.writerow(row)
    return True
```

Generate a test data set and a train data set. Here we use the same data generation function and parameters twice to generate the train and test data.

In [63]:
```
w = [8,-2]
data_generate('data_train.csv', w, 100, 123)
data_generate('data_test.csv', w, 100, 456)
```

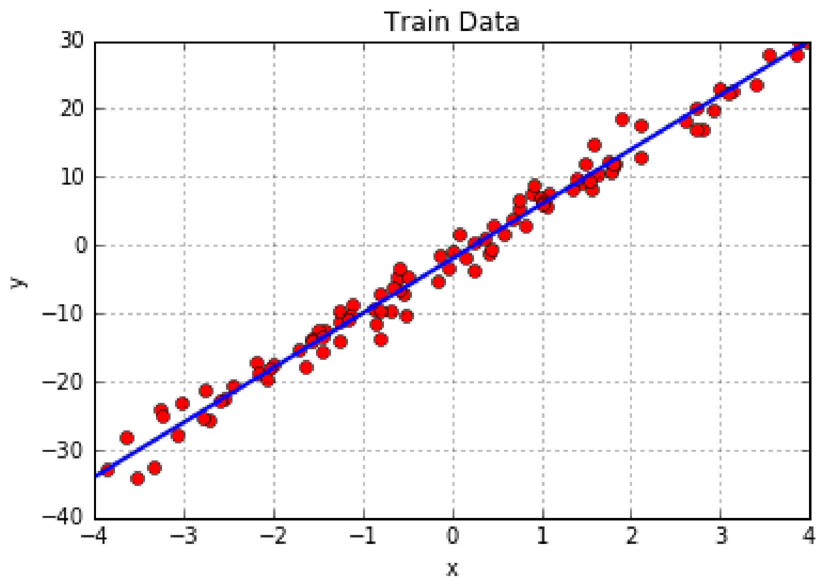
Out[63]: True

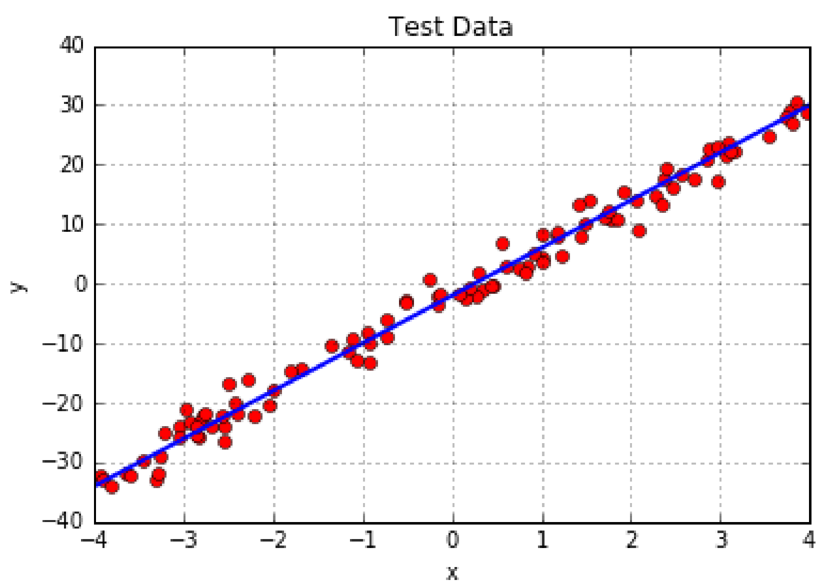
The true $y = 8x - 2$.

Data Visualization

In [64]:
```
%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")
    plt.title(title)
    x = [-4, 4]
    y = [(i * w[0] + w[1]) for i in x]
    plt.plot(x,y, linewidth=2.0, )
    plt.grid()
    plt.show()
```

In [65]:
```
dataPlot('data_train.csv',w, 'Train Data')
dataPlot('data_test.csv',w, 'Test Data')
```





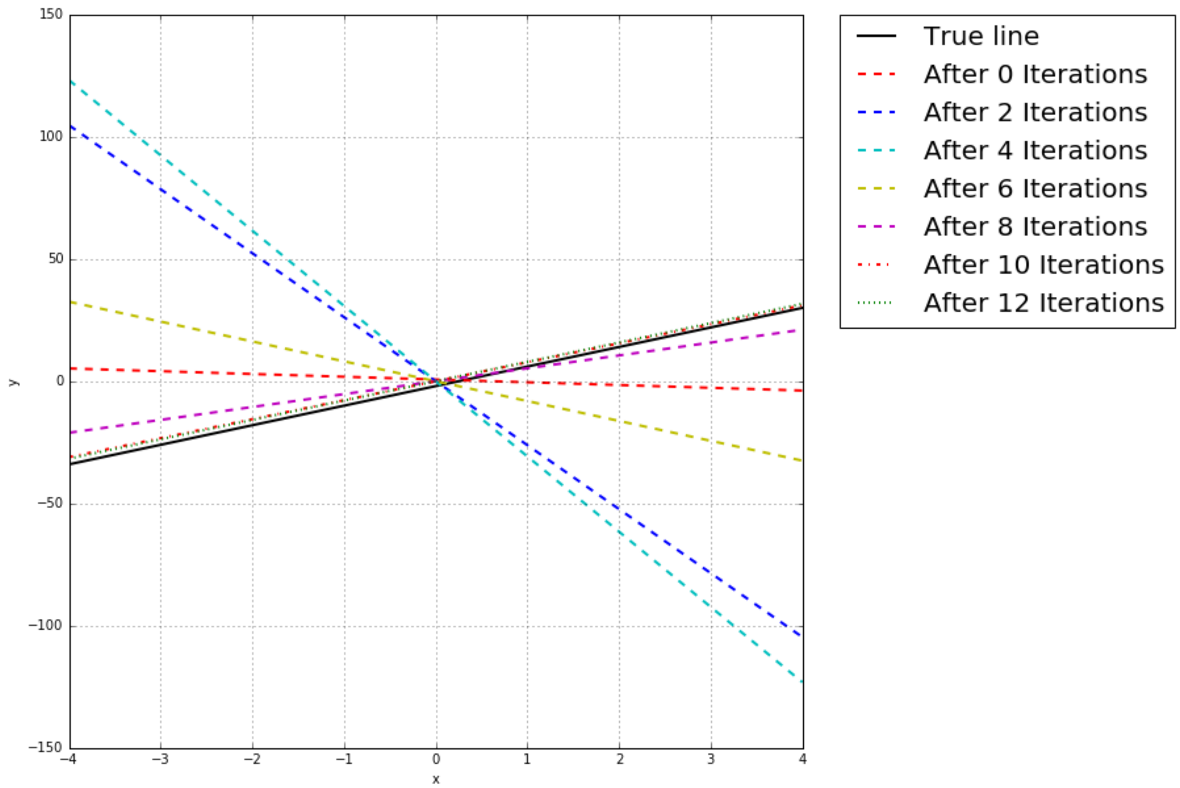
In [56]:

```
def iterationsPlot(testFile, trainFile, w):
    x = [-4, 4]

    y = [(i * w[0] + w[1]) for i in x]

    plt.figure(figsize=(10,10))
    plt.plot(x, y, 'k', label="True line", linewidth=2.0)

    test_data = sc.textFile(testFile). \
        map(lambda line: [float(v) for v in line.split(',')]).cache
()

    train_data = sc.textFile(trainFile). \
        map(lambda line: [float(v) for v in line.split(',')]). \
        map(lambda point: LabeledPoint(point[0],[point[1]])).cache
()

    n = train_data.count()

    np.random.seed(400)
    w = np.random.normal(0,1,2)
    y = [(i * w[0] + w[1]) for i in x]
    plt.plot(x, y, 'r--', label="After 0 Iterations", linewidth=2.
0)

    squared_error = test_data.map(lambda d: (d[0] - np.dot(w, [d
[1],1.0]))**2).reduce(lambda a, b: a + b)
    print "Mean Squared Error after 0 iterations: " + str(squared_e
rror/n)

    lrm = LinearRegressionWithSGD.train(train_data, iterations=2,
        initialWeights=array([1.0]))

    y = [lrm.predict([i]) for i in x]
    plt.plot(x, y, 'b--', label="After 2 Iterations", linewidth=2.
0)
    squared_error = test_data.map(lambda d: (d[0] - lrm.predict([d
[1]]))**2).reduce(lambda a, b: a + b)
    print "Mean Squared Error after 2 iterations: " + str(squared_e
rror/n)

    lrm = LinearRegressionWithSGD.train(train_data, iterations=4,
        initialWeights=array([1.0]))

    y = [lrm.predict([i]) for i in x]
    plt.plot(x, y, 'c--', label="After 4 Iterations", linewidth=2.
0)
    squared_error = test_data.map(lambda d: (d[0] - lrm.predict([d
[1]]))**2).reduce(lambda a, b: a + b)
    print "Mean Squared Error after 4 iterations: " + str(squared_e
rror/n)

    lrm = LinearRegressionWithSGD.train(train_data, iterations=6,
        initialWeights=array([1.0]))
```

```
    y = [lrm.predict([i]) for i in x]
    plt.plot(x, y, 'y--', label="After 6 Iterations", linewidth=2.
0)
    squared_error = test_data.map(lambda d: (d[0] - lrm.predict([d
[1]]))**2).reduce(lambda a, b: a + b)
    print "Mean Squared Error after 6 iterations: " + str(squared_e
rror/n)

    lrm = LinearRegressionWithSGD.train(train_data, iterations=8,
        initialWeights=array([1.0]))

    y = [lrm.predict([i]) for i in x]
    plt.plot(x, y, 'm--', label="After 8 Iterations", linewidth=2.
0)
    squared_error = test_data.map(lambda d: (d[0] - lrm.predict([d
[1]]))**2).reduce(lambda a, b: a + b)
    print "Mean Squared Error after 8 iterations: " + str(squared_e
rror/n)

    lrm = LinearRegressionWithSGD.train(train_data, iterations=10,
        initialWeights=array([1.0]))

    y = [lrm.predict([i]) for i in x]
    plt.plot(x, y, 'r-.', label="After 10 Iterations", linewidth=2.
0)
    squared_error = test_data.map(lambda d: (d[0] - lrm.predict([d
[1]]))**2).reduce(lambda a, b: a + b)
    print "Mean Squared Error after 10 iterations: " + str(squared_
error/n)

    lrm = LinearRegressionWithSGD.train(train_data, iterations=12,
        initialWeights=array([1.0]))

    y = [lrm.predict([i]) for i in x]
    plt.plot(x, y, 'g:', label="After 12 Iterations", linewidth=2.
0)
    squared_error = test_data.map(lambda d: (d[0] - lrm.predict([d
[1]]))**2).reduce(lambda a, b: a + b)
    print "Mean Squared Error after 12 iterations: " + str(squared_
error/n)



    plt.legend(bbox_to_anchor=(1.05, 1), loc=2, fontsize=20, border
axespad=0.)
    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid()
    plt.show()
```

In [57]:
```
iterationsPlot('data_test.csv', 'data_train.csv',w = [8,-2])
```

```
Mean Squared Error after 0 iterations: 449.275905508
Mean Squared Error after 2 iterations: 6222.20739503
Mean Squared Error after 4 iterations: 8019.21717965
Mean Squared Error after 6 iterations: 1383.28058198
Mean Squared Error after 8 iterations: 45.5641860988
Mean Squared Error after 10 iterations: 8.72309274795
Mean Squared Error after 12 iterations: 8.62919896004
```



The MLlib linear regression module converges on the test set after training on the training set for about 10 iterations. There doesn't seem to be much improvement after the 10th iteration.

HW 10.6.2

In the notebook provided, in the cell labeled "Gradient descent (regularization)".

Fill in the blanks and get this code to work for LASS0 and RIDGE linear regression.

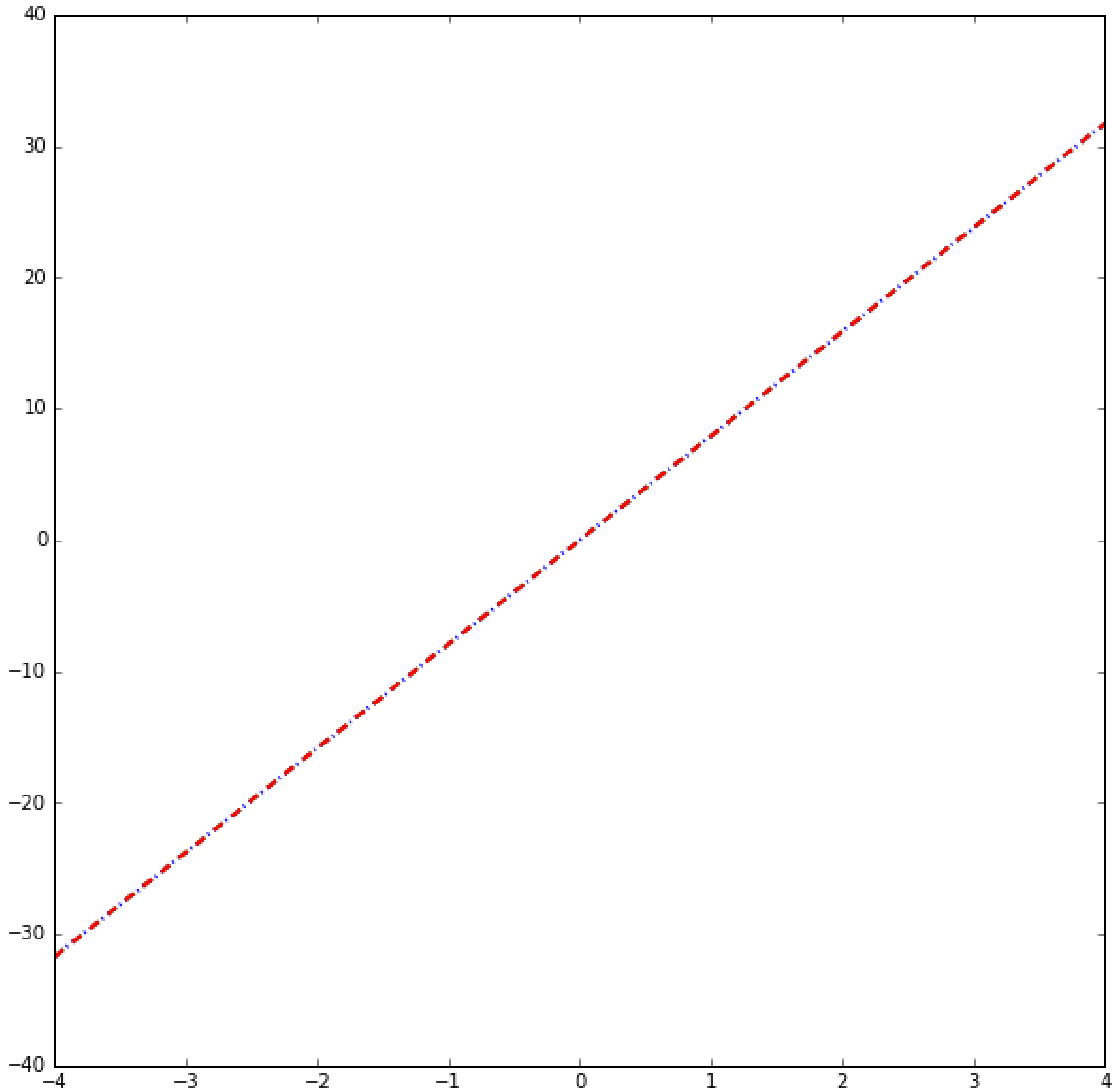
Using the data from 10.6.1 tune the hyper parameters of your LASS0 and RIDGE regression. Report your findings with words and plots.

In [103]:
```
def linearRegressionGDReg(data, wInitial=None, learningRate=0.05, i
terations=50, regParam=0.01, regType=None):
    featureLen = len(data.take(1)[0])-1
    n = data.count()
    if wInitial is None:
        w = np.random.normal(size=featureLen) # w should be broadca
sted if it is large
    else:
        w = wInitial
    for i in range(iterations):
        wBroadcast = sc.broadcast(w)
        gradient = data.map(lambda d: -2 * (d[0] - np.dot(wBroadcas
t.value, d[1:])) * np.array(d[1:])) \
                    .reduce(lambda a, b: a + b)
        if regType == "Ridge":
            #ridge is sum(parameters**2)
            wReg = np.square(w)

        elif regType == "Lasso":
            #lasso is sum(abs(parameters))
            wReg = np.abs(w)
        else:
            wReg = np.zeros(w.shape[0])
        gradient = gradient + regParam * wReg  #gradient:  GD of Sq
aured Error+ GD of regularized term
        w = w - learningRate * gradient / n
    return w
```

In [131]:
```
data = sc.textFile("./data_train.csv"). \
    map(lambda line: [float(v) for v in line.split(',')]).cache()

np.random.seed(400)
plt.figure(figsize=(10,10))

beta_hat = linearRegressionGDReg(data, iterations=50, regParam=0.1,
regType="")
print 'SGD ',beta_hat
y = [beta_hat[0]*i for i in x]
plt.plot(x, y, 'g--', label="", linewidth=2.0)

beta_hat = linearRegressionGDReg(data, iterations=50, regParam=0.0
1, regType="Ridge")
print 'SGD Ridge ',beta_hat
y = [beta_hat[0]*i for i in x]
plt.plot(x, y, 'b:', label="", linewidth=2.0)

beta_hat = linearRegressionGDReg(data, iterations=50, regParam=0.0
1, regType="Lasso")
print 'SGD Lasso ',beta_hat
y = [beta_hat[0]*i for i in x]
plt.plot(x, y, 'r--', label="", linewidth=2.0)
```

```
SGD  [ 7.92701601]
SGD Ridge  [ 7.92620088]
SGD Lasso  [ 7.92691316]
```

Out[131]: [<matplotlib.lines.Line2D at 0x114610e10>]



In [106]:

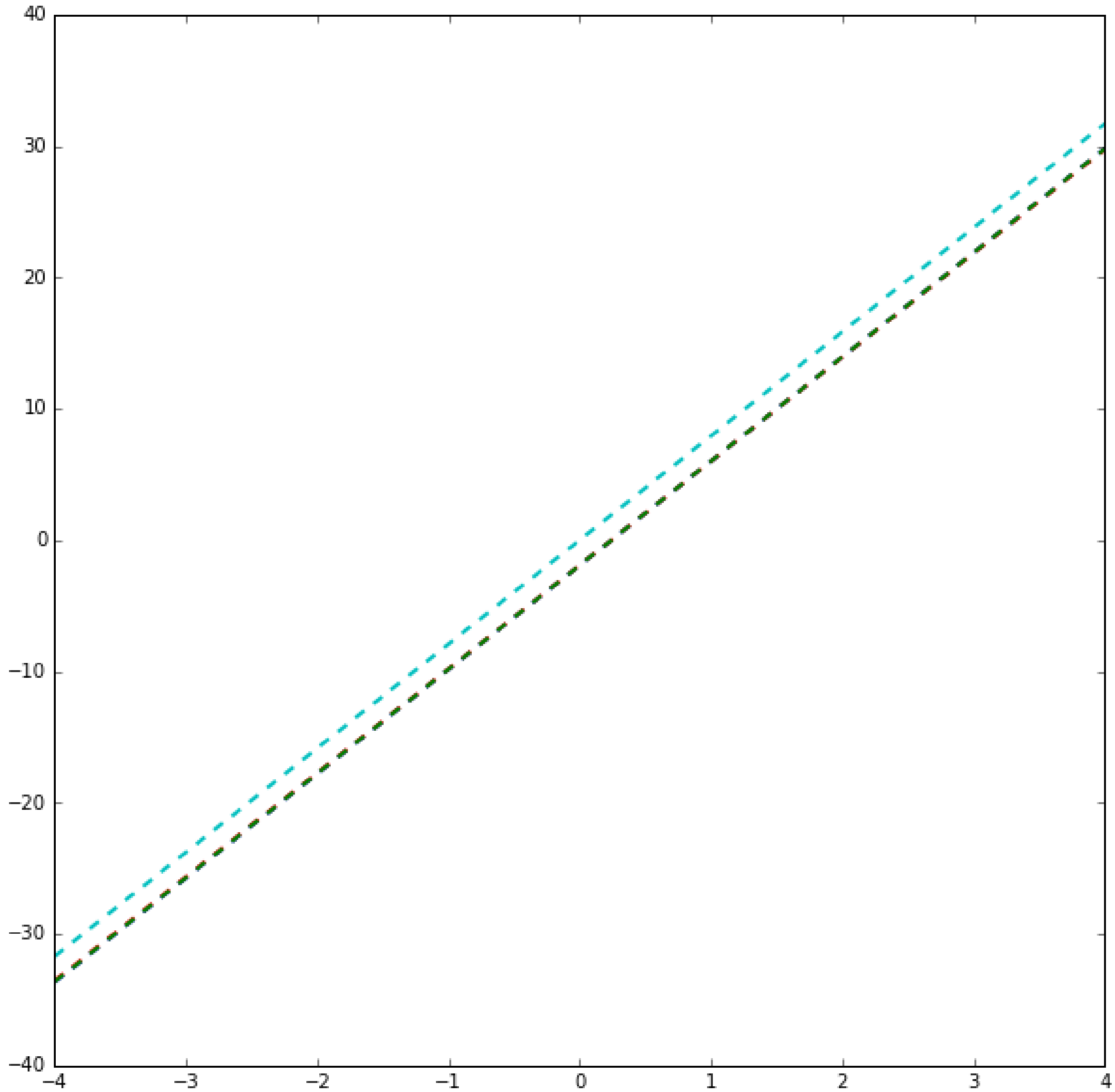
```
np.random.seed(400)
linearRegressionGDReg(data, iterations=50, regParam=0.1, regType="Lasso")
```

Out[106]: array([7.92598764])

In [128]:
```
from pyspark.mllib.regression import LabeledPoint, LinearRegression
WithSGD, LinearRegressionModel

# Load and parse the data
def parsePoint(line):
    values = [float(x) for x in line.split(',')]
    return LabeledPoint(values[0], values[1:])

data = sc.textFile("data_train.csv")
parsedData = data.map(parsePoint)

x = [-4, 4]
plt.figure(figsize=(10,10))

# Build the model
model = LinearRegressionWithSGD.train(parsedData, intercept=True, i
terations=50)
y = [model.predict([i]) for i in x]
plt.plot(x, y, 'b--', label="Linear Regression SGD", linewidth=2.0)
print model

model_ridge = LinearRegressionWithSGD.train(parsedData, regType="l
2", regParam=0.01, intercept=True)
y = [model_ridge.predict([i]) for i in x]
plt.plot(x, y, 'r--', label="Linear Regression Ridge", linewidth=2.
0)
print model_ridge

model_lasso = LinearRegressionWithSGD.train(parsedData, regType="l
1", regParam=0.01, intercept=True)
y = [model_lasso.predict([i]) for i in x]
plt.plot(x, y, 'g--', label="Linear Regression Lasso", linewidth=2.
0)
print model_lasso

# Plot the homegrown SGD Lasso on top of the MLLib SGD plots
y = [beta_hat[0]*i for i in x]
plt.plot(x, y, 'c--', label="", linewidth=2.0)
```

```
(weights=[7.93297544657], intercept=-1.9291504553545304)
(weights=[7.91242675855], intercept=-1.909852379336442)
(weights=[7.93035046639], intercept=-1.9191237277230135)
```

```
Out[128]: [<matplotlib.lines.Line2D at 0x114825c90>]
```



The "homegrown" stochastic gradient descent (SGD) algorithm converges more closely between the 3 different regularization types, but does not produce an intercept estimate. The coefficients between them are slightly different but not enough so that you can tell by eye in the graph.

The MLLib SGD algorithms have slight differences between them with the same number of iterations as can be see in the graph, mainly the Ridge regression is very different from the Lasso no OLS regularization. Modifying the regularization weight hyperparameter from 0.1 to 0.01 helps to minimize the difference.

```
In [ ]:
```