Zeppelin

MIDS-W261-2015-HWK-Week11-Hamlin-Thomas-Baek-Danish

FINISHED

default

DATASCI W261: Machine Learning at Scale

Nick Hamlin (nickhamlin@gmail.com)
Tigi Thomas (tgthomas@berkeley.edu)
Rock Baek (rockb1017@gmail.com)
Hussein Danish (husseindanish@gmail.com)
Time of Submission: 12:45 AM EST, Thursday, April 7, 2016
W261-3, Spring 2016
Week 11 Homework

Submission Notes:

- For each problem, we've included a summary of the question as posed in the instructions. In many cases, we have not included the full text to keep the final submission as uncluttered as possible. For reference, we've included a link to the original instructions in the “Useful Reference” below.

Useful Links

- Original Assignment (https://www.dropbox.com/s/rkxw455jj8ntqxy/MIDS-MLS-HW-11.txt?dl=0)
- This notebook on Zeppelin Viewer (https://www.zeppelinhub.com/viewer/notebooks/aHR0cHM6Ly9yYXcuZ2l0aHVidXNlcmNvbnR
- Logistic Regression Notebook (http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/3e1bsvpvqbf97u1/LogisticRegression-Notebook.ipynb)
- SVM in Spark Notebook (http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/dm2l73iznde7y4f/SVM-Notebook-Linear-Kernel-2015-06-19.ipynb)

Took 0 seconds (outdated)

FINISHED

HW 11.0 - Broadcast versus Caching in Spark

What is the difference between broadcasting and caching data in Spark? Give an example (in the context of machine learning) of each mechanism (at a highlevel). Feel free to cut and paste code examples from the lectures to support your answer.

Caching (or persisting) is a key operation in Spark. When you cache an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset (or datasets derived from it). This allows future actions to be much faster (often by more than 10x). Caching is a key tool for iterative algorithms and fast interactive use.

Meanwhile, **broadcast** variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node a copy of a large input dataset in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

If we were dealing with large input that was not broadcast, it would have to be serialized and shipped to each mapper task on each node for each task. This could become a very resource intensive process and using broadcast will save us lots of bandwidth. In KMeans for example, if we have 900 gigs of data (assuming 1 gig chunks) then we ship each mapper and centroids data to each executor 900 times. However, if we have 5 worker nodes then we we broadcast centroids we only ship the data 5 times.

In the linear regression example below (from the lecture), the weight vectors are broadcast for every iteration. Therefore, every worker node will have a copy of the weight vectors rather than having to send it for each executor task.
Took 0 seconds (outdated)

READY

```
%pyspark
import numpy as np

def linearRegressionGD(data, wInitial=None, learningRate=0.05, iterations=50):
    featureLen = len(data.take(1)[0])-1
    n = data.count()
    if wInitial is None:  #start learning from a random vector
        w = np.random.normal(size=featureLen) # w should be broadcasted if it is large
    else:                 #start from provided vector
        w = wInitial
    for i in range(iterations):
        wBroadcast = sc.broadcast(w) #make available in memory as read-only to the exec
        gradient = data.map(lambda d: -2 * (d[0] - np.dot(wBroadcast.value, d[1:])) * n
                    .reduce(lambda a, b: a + b)
        w = w - learningRate * gradient/n
    return w
```

FINISHED

An example of the .cache() function can be seen below. Our data would be read from a file, parsed and then cached into memory. If we were to apply actions or transformations (i.e. a map or filter operation) on this data, it would be available in memory rather than having to read it from disk.

Took 1 seconds (outdated)

READY

```
%pyspark
data = sc.textFile('data.csv').map(lambda line: [float(v) for v in line.split(',')]+[1.
# data [[-1.2062354302179288, 0.390508031418598, 1.0],
#       [13.573772421743222, 1.7215149309793558, 1.0],
#       [5.508180948046128, 0.822107008573151, 1.0], .......]
linearRegressionGD(data)
```

FINISHED

Review this Spark-notebook-based implementation of KMeans (http://nbviewer.ipython.org/urls/dl.dropbox.com/s/41q9lgyqhy8ed5g/EM-Kmeans.ipynb) and use the broadcast pattern to make this implementation more efficient. Please describe your changes in English first, implement, comment your code and highlight your changes:

In the KMeans code, centroids are shipped to each mapper as part of the call to the nearest_centroids() function. If we have very large centroids then this can be a very resource intensive process. Currently, centroids is a global variable that is accessed through the nearest_centroids() function, which is called in the mapper for each iteration of our KMeans process. Given that centroids do not change during an iteration, we will want to broadcast these for each iteration so they can be made use of inside the nearest_centroids() function. As such, there are two changes to make to the code for this KMeans implementation. These changes will be similar in nature to those shown in the above code for linear regression. Firstly, we will broadcast the centroids at the start of each iteration. Secondly, we will make use of the broadcast values inside the nearest_centroids() function (called by the mapper). The changes would thus look as follows:
Took 1 seconds (outdated)

READY

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

#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 - centroidsBroadcast.value)**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()

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

#D = sc.textFile("./data.csv").cache()
D = sc.parallelize(data).cache()
iter_num = 0
for i in range(10):
    centroidsBroadcast = sc.broadcast(centroids) # HERE WE BROADCAST THE CENTROIDS IN O
    res = D.map(nearest_centroid).reduceByKey(lambda x,y : (x[0]+y[0],x[1]+y[1])).colle
    #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])  #divide by cluster size
    if np.sum(np.absolute(centroids_new-centroids))<0.01:
        break
    print "Iteration" + str(iter_num)
    iter_num = iter_num + 1
    centroids = centroids_new
    print centroids
    plot_iteration(centroids)
print "Final Results:"
```

FINISHED

HW 11.1 - Loss Functions

In the context of binary classification problems, does the linear SVM learning algorithm yield the same result as a L2 penalized logistic regesssion learning algorithm? In your reponse, please discuss the loss functions, and the learnt models, and separating surfaces between the two classes.

Logistic regression and linear SVM are both models used for classifcation problems.

- Logistic regression works by optimizing the log likelihood, which are used to transform the class assignment based on conditional probabilities. The separator here is based on a logit function.
- SVMs work by finding the maximal margin hyperplane. The seperator here (hyperplane) is fit based on support vectors, which represent the edge of each class.

Some of the key differences here are that logistic regression fits data points as if they were along a continuous function. This could pose problems if our data is around P=0.5 in the function. Meanwhile SVMs attempt to fit a hyperplane by separating the two classes of data. This could pose problems if our classes are not separable. Based on this, the results between the SVM learning algorithm and the L2 penalized logistic regression learning algorithm will not be the same. However, given that we are considering L2 penalized logistic regression, this will mitigate some of the effects of classifying along the margin that we would typically face with logistic regression.

In the context of binary classification problems, does the linear SVM learning algorithm yield the same result as a perceptron learning algorithm?

As mentioned above, SVMs try to find the maximal margin hyperplane that separates our classes. Meanwhile, the perceptron will find one of many hyperplanes separating our classes. If we consider a 2d example, the SVM while find the line with the maximal margins whereas the perceptron will find one of many lines that will separate both classes but the line found will depend on the order in which the data points are processed. As such, when it comes to a classification problem, the SVM will likely have better peformance given that the hyperplane that it finds is optimized compared to that of a perceptron.

Took 1 seconds (outdated)

FINISHED

HW11.2 Gradient descent

In the context of logistic regression describe and define three flavors of penalized loss functions. Are these all supported in Spark MLLib (include online references to support your answers)?

A penalized loss function is achieved by adding a regularization term to to the loss in order to prevent overfitting. Some of the most common methods for achieving this in the context of logistic regression are:

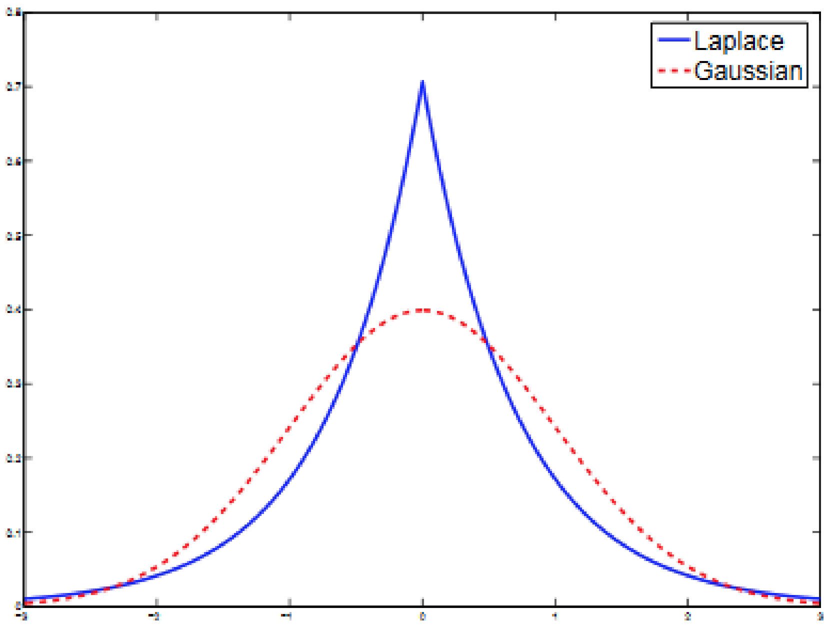
1. **L1 regularization.** The regularization term applied here is |w|
2. **L2 regularization.** The regularization term applied here is w^2
3. **Elastic-net regularization.** This is a combination of L1 and L2 regularization and the term applied is (a/2)w^2 + (1-a)|w|

All three of these regularization terms are supported in Spark MLlib as seen in the online documentation here: http://spark.apache.org/docs/latest/mllib-linear-methods.html#regularizers (http://spark.apache.org/docs/latest/mllib-linear-methods.html#regularizers)

Describe probabilitic interpretations of the L1 and L2 priors for penalized logistic regression (HINT: see synchronous slides for week 11 for details)

As seen in the synchronous slides, the L1 norm is a Manhattan distance and the L2 norm is a

Euclidean distance. When comparing L1 and L2 regularization in the context of penalty terms, L1 regularization can be visualized as concentric diamonds from the center point whereas L2 regularization can be visualized as concentric circles from the center point. Considering this in more probabilistic terms, we can view L1 as setting a Laplacean prior on the regression coefficients and picking a MAP hypothesis. Similarly, L2 can be viewed as setting a Gaussian prior. Here, L1 drives **w** to zero more efficiently as can be seen in this graph:



.This has the practical effect of driving less-important feature weights to zero more rapidly than in L2 regularization

Took 1 seconds (outdated)

FINISHED

HW11.3 Logistic Regression

HW 11.3 Generating and plotting test data

Generate 2 sets of linearly separable data with 100 data points each using the data generation code provided below and plot each in separate plots. Call one the training set and the other the testing set. Modify this data generation code to generating non-linearly separable training and testing datasets (with approximately 10% of the data falling on the wrong side of the separating hyperplane. Plot the resulting datasets. For the remainder of this problem please use the non-linearly separable training and testing datasets.

We start by using the function provided to create linearly separable test data, then dealing with plotting overhead in Zeppelin so we can display our results.

Took 1 seconds (outdated)

FINISHED

```
%pyspark

from numpy.random import rand #This line wasn't included in the original code, but is r
```

```
#data generation code
def generateData(n):
    """
    generates a 2D linearly separable dataset with n samples.
    The third element of the sample is the label
    """
    xb = (rand(n)*2-1)/2-0.5
    yb = (rand(n)*2-1)/2+0.5
    xr = (rand(n)*2-1)/2+0.5
    yr = (rand(n)*2-1)/2-0.5
    inputs = []
    for i in range(len(xb)):
        inputs.append([xb[i],yb[i],1])
        inputs.append([xr[i],yr[i],-1])
    return inputs

#Generate training and test datasets
train=generateData(100)
test=generateData(100)
```

Took 0 seconds (outdated)

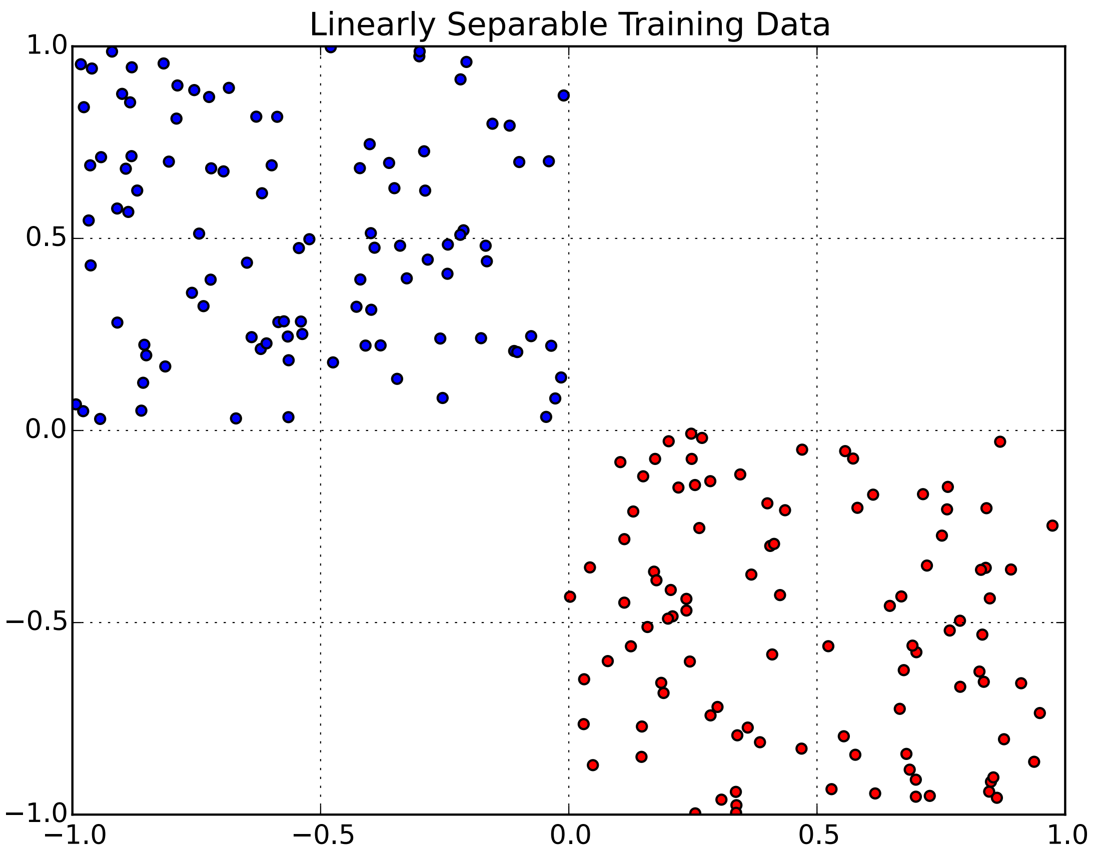
FINISHED

```
%pyspark

import matplotlib
import matplotlib.pylab as plt
matplotlib.use('Agg')
import StringIO

def showMPL(plot):
    """Helper function necessary for printing in Zeppelin"""
    img = StringIO.StringIO()
    plot.savefig(img, format='svg')
    img.seek(0)
    print "%html <div style='width:600px; height=100px'>" + img.buf + "</div>"

def plot_data(data,title):
    """
    Helper function to plot our test or training data.
    We've defined it this way for easy recycling later
    """
    plt.figure( figsize=(8, 6))
    plt.clf()

    train_array=np.array(data)

    # Plot the training points
    colors = list(np.select([train_array[:,2] == -1, train_array[:,2] == 1 ], ['r', 'b'

    plt.scatter(train_array[:, 0], train_array[:, 1], c=colors)
    plt.xlim(-1, 1)
    plt.ylim(-1, 1)
    plt.xticks()
    plt.yticks()
    plt.grid()
    plt.title(title)
```

Took 0 seconds (outdated)

FINISHED

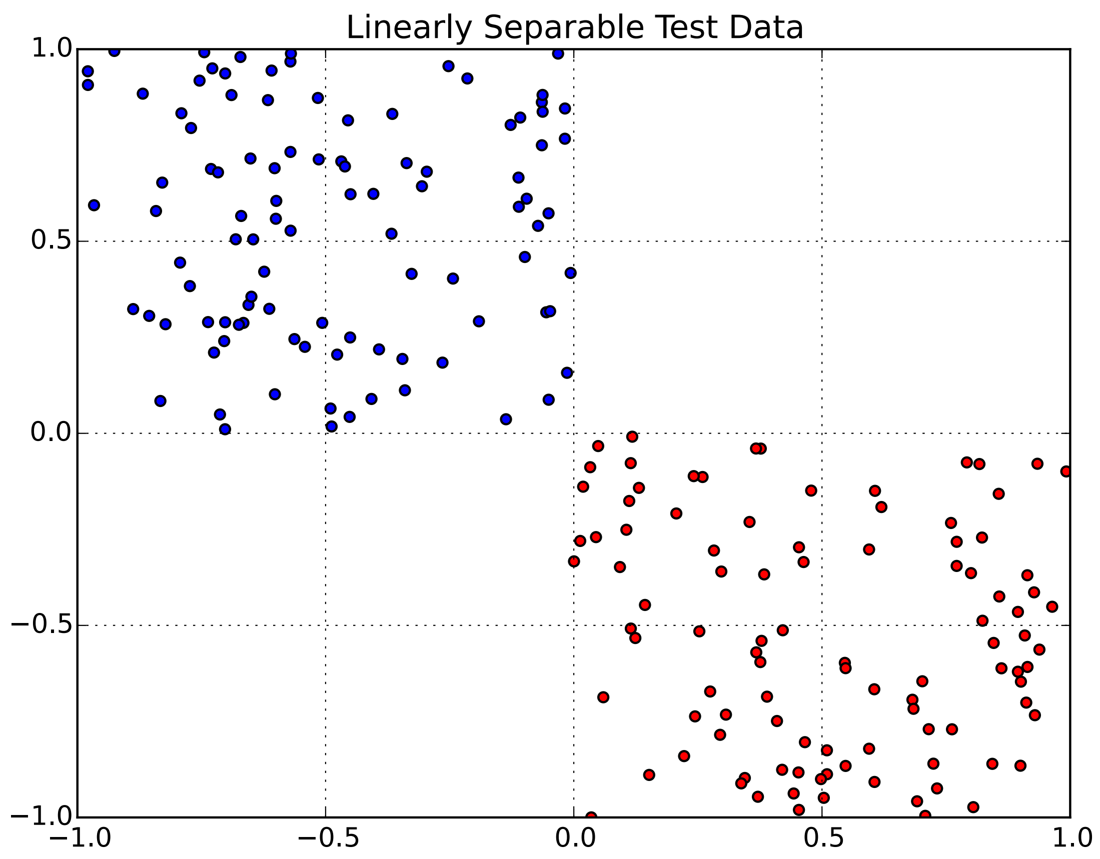
```
%pyspark
plot_data(train, "Linearly Separable Training Data")
```



Took 0 seconds

FINISHED

```
%pyspark
#Plot Test Data

plot_data(test, "Linearly Separable Test Data")
```



Took 1 seconds

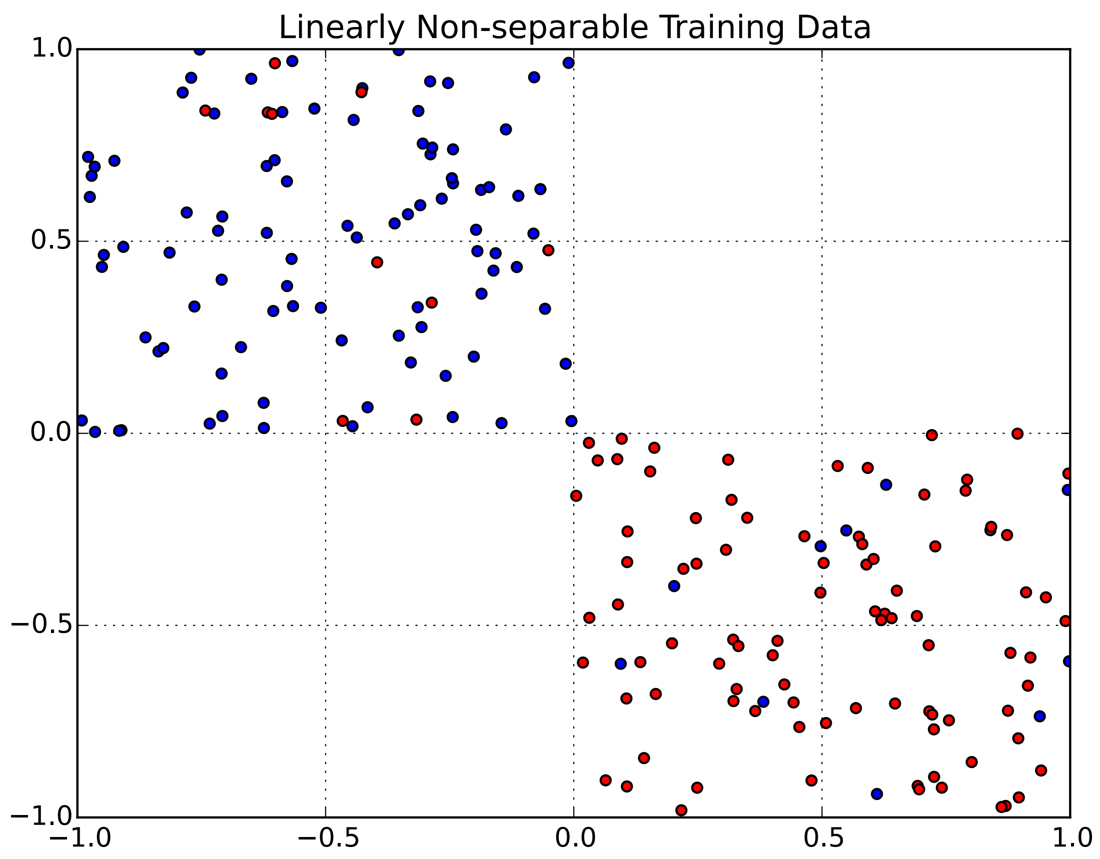
FINISHED

```
%pyspark
def generateInseparableData(n):
    """
    generates a 2D NOT linearly separable dataset with n samples.
    The third element of the sample is the label
    """
    #Take 10% of the data and manually switch its label to the opposite
    xb = np.concatenate([(rand(n*0.9)*2-1)/2-0.5,(rand(n*0.1)*2-1)/2+0.5],axis=0)
    yb = np.concatenate([(rand(n*0.9)*2-1)/2+0.5,(rand(n*0.1)*2-1)/2-0.5],axis=0)
    xr = np.concatenate([(rand(n*0.9)*2-1)/2+0.5,(rand(n*0.1)*2-1)/2-0.5],axis=0)
    yr = np.concatenate([(rand(n*0.9)*2-1)/2-0.5,(rand(n*0.1)*2-1)/2+0.5],axis=0)
    inputs = []
    for i in range(len(xb)):
        inputs.append([xb[i],yb[i],1])
        inputs.append([xr[i],yr[i],-1])
    return inputs

ntrain=generateInseparableData(100)
ntest=generateInseparableData(100)
```

Took 0 seconds (outdated)

FINISHED

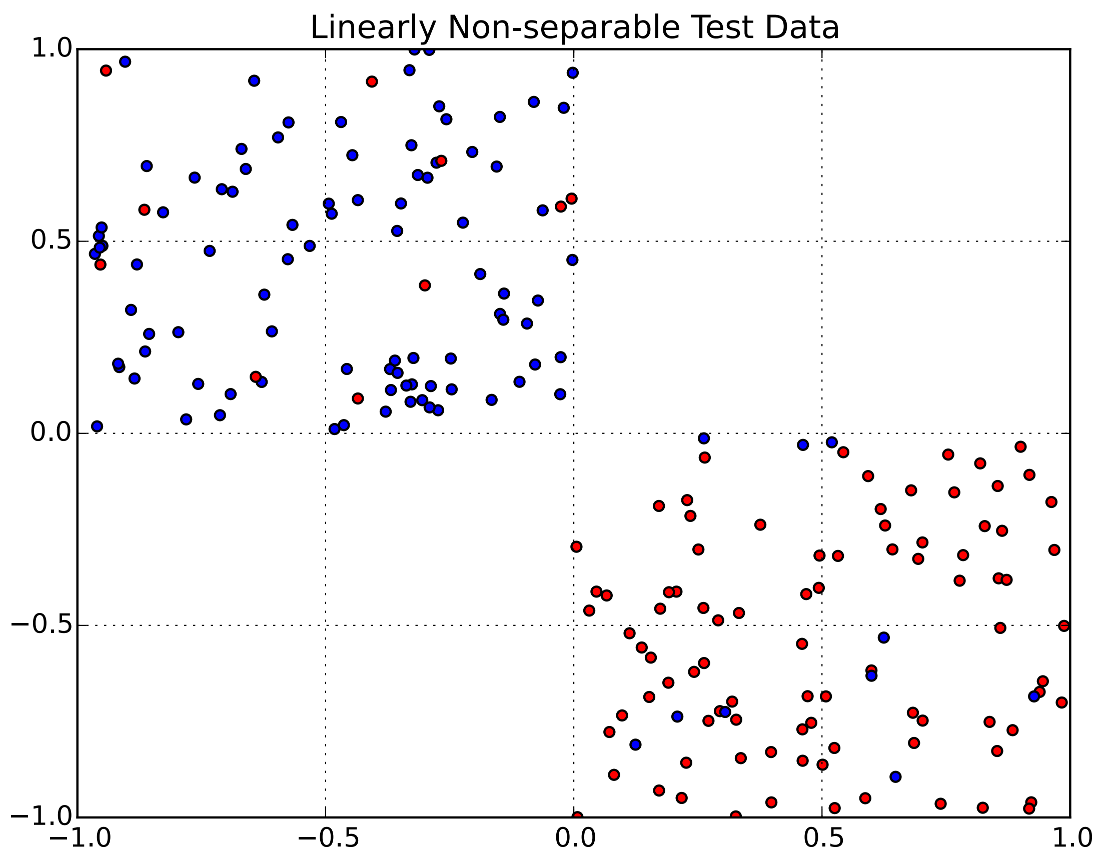
```
%pyspark
plot_data(ntrain,"Linearly Non-separable Training Data")
```



Took 1 seconds

FINISHED

```
%pyspark
plot_data(ntest,"Linearly Non-separable Test Data")
```



Took 0 seconds

FINISHED

HW 11.3 - Lasso Logistic Regression

Using MLLib train up a Lasso logistic regression model with the training dataset and evaluate with the testing set. What a good number of iterations for training the logistic regression model? Justify with plots and words.

Took 1 seconds (outdated)

FINISHED

```
%pyspark
from __future__ import division
from pyspark.mllib.classification import LogisticRegressionWithSGD, LogisticRegressionM
from pyspark.mllib.regression import LabeledPoint

def parsePoint(line):
    """
    Create LabeledPoint object out of an array
    """
    label=line[2]
    if label==-1: #Convert -1 labels to 0s
        return LabeledPoint(0, line[:2])
    else:
        return LabeledPoint(1, line[:2])
```

```
def test_prediction(labeled_point):
    """
    Given a labeled point and (implicitly) a model,
    check if the point has been classified correctly
    """
    # print labeled_point
    # print model.weights
    result=np.dot(labeled_point.features,model.weights)
    # print result
    if result>0:
        pred=1
    else:
        pred=0
    # print pred
    actual=labeled_point.label
    if actual==pred:
        return 1
    else:
        return 0


# Load and parse data
data = sc.parallelize(ntrain)
parsedData = data.map(parsePoint)
total_estimated=parsedData.count()

# Setup plot needs and iteration ranges
x1 = [-1, 1]
iterations=[1,2,3,4,5,10,20,50,100,200,300]
line_colors=['b','g','r','c','m','y','k','burlywood','chartreuse','lightblue','violet']

plt.figure( figsize=(8, 6))
plt.clf()

# Main loop
for k,iteration in enumerate(iterations):

    # Build and evaluate the model
    model= LogisticRegressionWithSGD.train(parsedData, iterations=iteration, regType='l
    total_correct=parsedData.map(test_prediction).sum()
    error_rate=1-total_correct/total_estimated

    # Plot results
    x2 = [-(i * model.weights[0] + model.intercept) / model.weights[1] for i in x1]
    plt.plot(x1, x2, 'b', label="Iter: "+str(iteration), linewidth=1, color=line_colors

    # Display output
    print "Iteration "+str(iteration)+":"
    print model
    print "Error rate: "+str(error_rate)
    print ""


# Add general plot content (to be displayed below)
colors = list(np.select([test_data[:,2] == -1, test_data[:,2] == 1 ], ['r', 'b']))
```

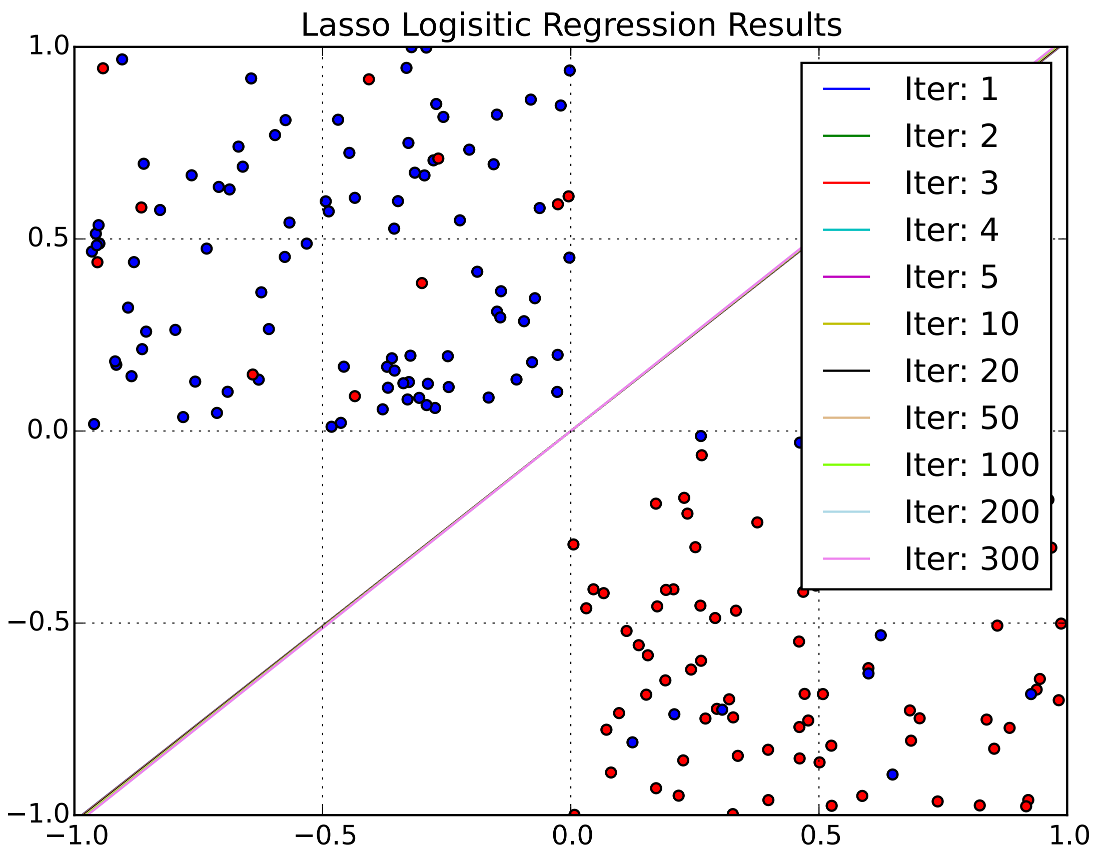
```
plt.scatter(test_data[:, 0], test_data[:, 1], c=colors)
plt.xlim(-1, 1)
plt.ylim(-1, 1)
plt.xticks()
plt.yticks()
plt.grid()
plt.title("Lasso Logisitic Regression Results")
plt.legend()
```

```
Iteration 1:
(weights=[-0.189305171257,0.186408104718], intercept=0.0)
Error rate: 0.1

Iteration 2:
(weights=[-0.303712910666,0.298929151994], intercept=0.0)
Error rate: 0.1

Iteration 3:
(weights=[-0.387738041835,0.381494412792], intercept=0.0)
Error rate: 0.1

Iteration 4:
(weights=[-0.454683319537,0.447226774797], intercept=0.0)
Error rate: 0.1

Iteration 5:
(weights=[-0.510512266553,0.50200743494], intercept=0.0)
Error rate: 0.1

Iteration 10:
(weights=[-0.702453903863,0.690050475319], intercept=0.0)
Error rate: 0.1

Iteration 20:
(weights=[-0.91361363234,0.896253809971], intercept=0.0)
Error rate: 0.1

Iteration 50:
(weights=[-1.19832282066,1.17258405075], intercept=0.0)
Error rate: 0.1

Iteration 100:
(weights=[-1.39815088557,1.36449525788], intercept=0.0)
Error rate: 0.1

Iteration 200:
(weights=[-1.51317924369,1.47353454637], intercept=0.0)
Error rate: 0.1

Iteration 300:
(weights=[-1.51317924369,1.47353454637], intercept=0.0)
Error rate: 0.1
```

Took 10 seconds (outdated)

FINISHED

```
%pyspark
showMPL(plt)
plt.close()
```



Took 1 seconds

FINISHED

It's difficult to see because of how the lines overlap, but it's clear that the model converges to an acceptible level in less than 100 iterations. We do see some minor adjustments being made during the initial iterations (as demonstrated by the visible distinctions in the separating hyperplanes towards the bottom left corner of the plot), but these changes are not dramatic enough to lead to a difference in the classification decisions that the model makes.
Took 0 seconds (outdated)

FINISHED

HW 11.3 - Weighted Lasso Logistic Regression

Derive and implement in Spark a weighted LASSO logistic regression. Implement a convergence test of your choice to check for termination within your training algorithm. Weight each example using the inverse vector length (Euclidean norm):
Evaluate your homegrown weighted LASSO logistic regression on the test dataset. Report misclassification error (1 - Accuracy) and how many iterations does it took to converge.
Took 0 seconds (outdated)

FINISHED

```
%pyspark
def logisticRegressionGDRegWeighted(data, wInitial=None, learningRate=0.05, iterations=
    n = data.count()
    w = wInitial
    for i in range(iterations):
        wBroadcast = sc.broadcast(w)

        #Original Unweighted Version
        #gradient = data.map(lambda d: (1 / (1 + np.exp(-d[0]*np.dot(wBroadcast.value,
                    #.reduce(lambda a, b: a + b)

        #Weighted Version
        gradient = data.map(lambda d: (1 / (1 + np.exp(-d[0]*np.dot(wBroadcast.value, d
                    * (1/(np.sqrt(np.sum(np.square(d[1:])))))) \
                    .reduce(lambda a, b: a + b)

        #Regularization
        if regType == "Ridge":
            wReg = w * 1
            wReg[-1] = 0 #last value of weight vector is bias term, ignored in regulari
        elif regType == "Lasso":
            wReg = w * 1
            wReg[-1] = 0 #last value of weight vector is bias term, ignored in regulari
            wReg = np.sign(wReg)
        else:
            wReg = np.zeros(w.shape[0])

        #Calculate the gradient and step
        gradient = gradient + regParam * wReg  #gradient:  GD of Sqaured Error+ GD of r
        wdelta=learningRate * gradient / n

        #Check if model converged
        if sum(abs(wdelta))<0.005:
            print "Converged after {0} iterations".format(str(i))
            break
        else:
            w = w - wdelta
    return w
```

Took 1 seconds (outdated)

FINISHED

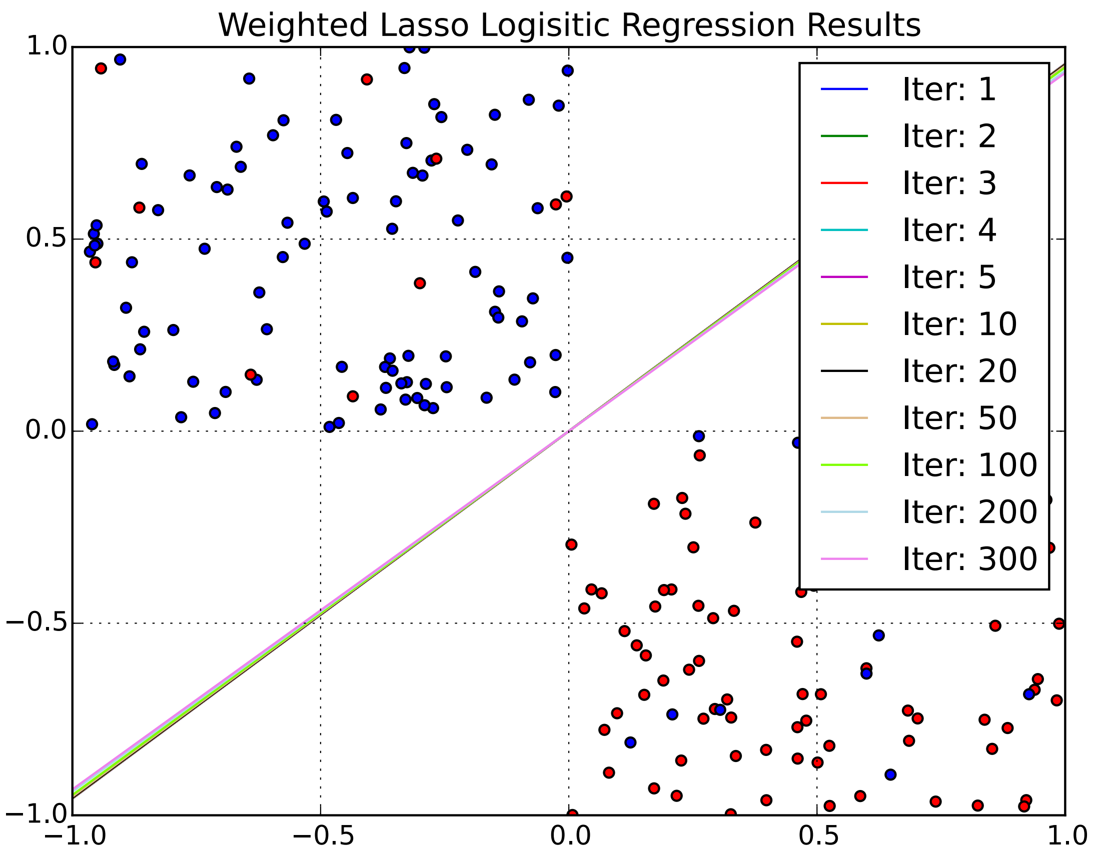
```
%pyspark
from __future__ import division
from pyspark.mllib.regression import LabeledPoint


def parsePointForHomegrown(line):
    """
    Refactor data as we've built it to play nice with the
    notebook LR implementation
    """
    label=line[2]
    if label==-1: #Convert -1 labels to 0s
        output=[0]+line[0:2]
    else:
        output=[1]+line[0:2]
    return output


def test_homegrown_prediction(labeled_point):
    """
    Given a labeled point and (implicitly) a model,
    check if the point has been classified correctly
    """
    # print labeled_point
    # print model.weights
    result=np.dot(labeled_point[1:2],model[0:1])
    # print result
    if result>0:
        pred=1
    else:
        pred=0
    # print pred
    actual=labeled_point[0]
    if actual==pred:
        return 1
    else:
        return 0


# Load and parse data
data = sc.parallelize(ntrain)
parsedData2 = data.map(parsePointForHomegrown)
total_estimated=parsedData.count()

# Setup plot needs and iteration ranges
x1 = [-1, 1]
iterations=[1,2,3,4,5,10,20,50,100,200,300]
line_colors=['b','g','r','c','m','y','k','burlywood','chartreuse','lightblue','vi
plt.figure( figsize=(8, 6))
plt.clf()

# Main loop
for k, iteration in enumerate(iterations):

    # Build and evaluate the model
    model= logisticRegressionGDRegWeighted(parsedData2, iterations=iteration, reg
    total_correct=parsedData2.map(test_homegrown_prediction).sum()
    error_rate=1-total_correct/total_estimated

    # Plot results
    x2 = [-(i * model[0]) / model[1] for i in x1]
    plt.plot(x1, x2, 'b', label="Iter: "+str(iteration), linewidth=1,color=line_c

    # Display output
    print "Iteration "+str(iteration)+":"
    print model
    print "Error rate: "+str(error_rate)
    print ""

# Add general plot content (to be displayed below)
colors = list(np.select([test_data[:,2] == -1, test_data[:,2] == 1 ], ['r', 'b'])
plt.scatter(test_data[:, 0], test_data[:, 1], c=colors)
plt.xlim(-1, 1)
plt.ylim(-1, 1)
plt.xticks()
plt.yticks()
plt.grid()
plt.title("Weighted Lasso Logisitic Regression Results")
plt.legend()
```

```
Iteration 1:
[-0.00626106  0.00654756]
Error rate: 0.1

Iteration 2:
[-0.01249361  0.0130691 ]
Error rate: 0.1

Iteration 3:
[-0.01870026  0.01956473]
Error rate: 0.1

Iteration 4:
[-0.02488112  0.02603455]
Error rate: 0.1

Iteration 5:
[-0.03103631  0.03247867]
Error rate: 0.1

Iteration 10:
[-0.06143079  0.06431756]
Error rate: 0.1

Iteration 20:
[-0.12035614  0.12612992]
Error rate: 0.1

Iteration 50:
[-0.28320841  0.29762247]
Error rate: 0.1

Iteration 100:
[-0.5145312   0.54324118]
Error rate: 0.1

Iteration 200:
[-0.86609031  0.9228474 ]
Error rate: 0.1

Converged after 271 iterations
Iteration 300:
[-1.05360652  1.12978846]
Error rate: 0.1
```

Took 50 seconds

FINISHED

```
%pyspark
showMPL(plt)
plt.close()
```



Took 1 seconds

FINISHED

Here we see similar results with our weighted model as we did for the unweighted model in MLlib. Using an automated convergence check, we find that our model converges after 271 iterations. The misclassification rate remains at a constant 0.1 throughout. This is a result of how we've generated our train/test data, in that if a hyperplane as a positive slope and passes through the origin, it's basically guaranteed to misclassify the 10% of points we deliberately miscategorized to create the linearly inseparable data.

Took 1 seconds (outdated)

FINISHED

HW 11.3 - Weighted Lasso Logistic Regression in MLlib

Does Spark MLLib have a weighted LASSO logistic regression implementation. If so use it and report your findings on the weighted training set and test set.

As per the MLLib documentation (http://spark.apache.org/docs/latest/mllib-linear-methods.html), MLLlib does not have a weighted Lasso Logistic Regression implementation as of version 1.6.1

Took 1 seconds (outdated)

FINISHED

HW11.4 SVMs

HW 11.4 MLlib Soft SVM Implementation

Use the non-linearly separable training and testing datasets from HW11.3 in this problem. Using MLLib train up a soft SVM model with the training dataset and evaluate with the testing set. What is a good number of iterations for training the SVM model? Justify with plots and words.

Took 1 seconds (outdated)

FINISHED

```
%pyspark
from __future__ import division
from pyspark.mllib.classification import SVMWithSGD
from pyspark.mllib.regression import LabeledPoint


def parsePoint(line):
    """
    Create LabeledPoint object out of an array
    """
    label=line[2]
    if label==-1: #Convert -1 labels to 0s
        return LabeledPoint(0, line[:2])
    else:
        return LabeledPoint(1, line[:2])

def test_prediction(labeled_point):
    """
    Given a labeled point and (implicitly) a model,
    check if the point has been classified correctly
    """
    # print labeled_point
    # print model.weights
    result=np.dot(labeled_point.features,model.weights)
    # print result
    if result>0:
        pred=1
    else:
        pred=0
    # print pred
    actual=labeled_point.label
    if actual==pred:
        return 1
    else:
        return 0

# Load and parse data
data = sc.parallelize(ntrain)
parsedData = data.map(parsePoint)
total_estimated=parsedData.count()

# Set up plotting overhead
x1 = [-1, 1]
iterations=[1,2,3,4,5,10,20,50,100]
line_colors=['b','g','r','c','m','y','k','burlywood','chartreuse','lightblue','violet']
plt.figure( figsize=(8, 6))
plt.clf()

for k,iteration in enumerate(iterations):
    # Build the model
    model= SVMWithSGD.train(parsedData, iterations=iteration, regType='l1', initialWeig
    # print iteration
    # print model
    x2 = [-(i * model.weights[0] + model.intercept) / model.weights[1] for i in x1]
    plt.plot(x1, x2, 'b', label="Iter: "+str(iteration), linewidth=1, color=line_colors

    total_correct=parsedData.map(test_prediction).sum()
    error_rate=1-total_correct/total_estimated


    # Display output
    print "Iteration "+str(iteration)+":"
    print model
    print "Error rate: "+str(error_rate)
    print ""


colors = list(np.select([test_data[:,2] == -1, test_data[:,2] == 1 ], ['r', 'b']))
plt.scatter(test_data[:, 0], test_data[:, 1], c=colors)
plt.xlim(-1, 1)
plt.ylim(-1, 1)
plt.xticks()
plt.yticks()
plt.grid()
plt.title("Soft SVM Results")
plt.legend()
```

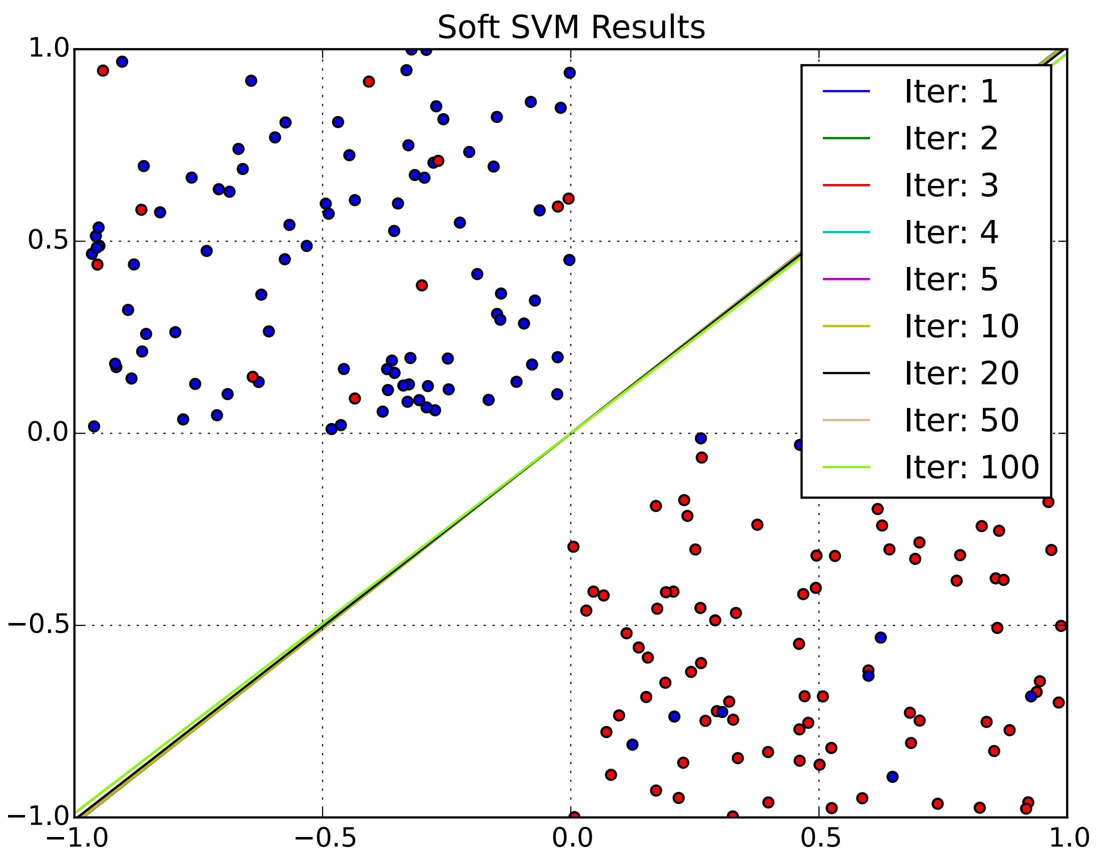
```
Iteration 1:
(weights=[-0.388610342514,0.382816209435], intercept=0.0)
Error rate: 0.1

Iteration 2:
(weights=[-0.663399350944,0.653508147075], intercept=0.0)
Error rate: 0.1

Iteration 3:
(weights=[-0.832972381786,0.818513513901], intercept=0.0)
Error rate: 0.1

Iteration 4:
(weights=[-0.927581800984,0.911843368735], intercept=0.0)
```

```
Error rate: 0.1

Iteration 5:
(weights=[-0.987953179318,0.973963382662], intercept=0.0)
Error rate: 0.1

Iteration 10:
(weights=[-1.12581831518,1.10886511946], intercept=0.0)
Error rate: 0.1

Iteration 20:
(weights=[-1.18182356389,1.17466468268], intercept=0.0)
Error rate: 0.1

Iteration 50:
(weights=[-1.210829499,1.22433759535], intercept=0.0)
Error rate: 0.1

Iteration 100:
(weights=[-1.210829499,1.22433759535], intercept=0.0)
Error rate: 0.1
```

Took 4 seconds

FINISHED

```
%pyspark
showMPL(plt)
plt.close()
```



Took 1 seconds

FINISHED

Here we see that the weights stop changing and the model convergest between 50-100 iterations. As shown in the plot above, the separating hyperplane shifts more in early iterations than it did with our logistic regression implementations, but the model still converges to a good solution. Also as before, we see a constant 10% misclassification rate, which makes sense based on our data generation process.

Took 1 seconds (outdated)

FINISHED

HW 11.4 - Homegrown Weighted Soft SVM Implementation

Derive and Implement in Spark a weighted soft linear svm classification learning algorithm. Evaluate your homegrown weighted soft linear svm classification learning algorithm on the weighted training dataset and test dataset from HW11.3. Report misclassification error (1 - Accuracy) and how many iterations does it took to converge? How many support vectors do you end up?

Took 0 seconds (outdated)

FINISHED

```
%pyspark

```

```
def SVM_GD_SPARK(dataRDD,w=None,eta=0.01,iter_num=1000,regPara=0.01,stopCriteria=(

    for i in range(iter_num):
        #label * margin
        sv = dataRDD.filter(lambda x:x[0]*np.dot(w,np.append(x[1:],1))<1)

        # converged as no more updates possible
        if sv.isEmpty():
            break

        # hinge loss component of gradient y*x and sum up
        #gradient: avg hinge loss

        #UNWEIGHTED VERSION
        # g = -sv.map(lambda x:x[0]*np.append(x[1:],1)).reduce(lambda x,y:x+y)/dat

        #WEIGHTED VERSION
        g = -sv.map(lambda x:x[0]*np.append(x[1:],1)* (1/(np.sqrt(np.sum(np.square

        wreg = w*1   #temp copy of weight vector
        wreg[-1] = 0 #last value of weight vector is bias term; ignore in regular
        wdelta =eta*(g+np.multiply(regPara,wreg))
        w=np.array(w)

        # converged as updates to weight vector are small
        if sum(abs(wdelta))<=stopCriteria*sum(abs(w)):
            print "Converged after {0} iterations".format(str(i))
            break

        w = w - wdelta

    #Get number of support vectors
    num_sv=sv.count()
    print "Number of Support Vectors: {0}".format(str(num_sv))
    return w
```

Took 0 seconds

FINISHED

```
%pyspark

from __future__ import division

def parsePointForHomegrown(line):
    """
    Refactor data as we've built it to play nice with the
    notebook LR implementation
    """
    label=line[2]
    if label==-1: #Convert -1 labels to 0s
        output=[0]+line[0:2]
    else:
        output=[1]+line[0:2]
    return output


def test_homegrown_prediction(labeled_point):
    """
    Given a labeled point and (implicitly) a model,
    check if the point has been classified correctly
    """
    # print labeled_point
    # print model.weights
    result=np.dot(labeled_point[1:2],model[0:1])
    # print result
    if result>0:
        pred=1
    else:
        pred=0
    # print pred
    actual=labeled_point[0]
    if actual==pred:
        return 1
    else:
        return 0


# Load and parse data
data = sc.parallelize(ntrain)
parsedData2 = data.map(parsePointForHomegrown).cache()
total_estimated=parsedData.count()


# Setup plot needs and iteration ranges
x1 = [-1, 1]
iterations=[1,2,5,10,20,50,100,200,300,400]
line_colors=['b','g','r','c','m','y','k','burlywood','chartreuse','lightblue','violet']
plt.figure( figsize=(8, 6))
plt.clf()

# Main loop

for k,iteration in enumerate(iterations):

    # Build and evaluate the model
    model= SVM_GD_SPARK(parsedData2, iter_num=iteration ,w=[0,0,0],eta=0.01, regPara=0.
    total_correct=parsedData2.map(test_homegrown_prediction).sum()
    error_rate=1-total_correct/total_estimated

    # Plot results
    x2 = [-(i * model[0]) / model[1] for i in x1]
    plt.plot(x1, x2, 'b', label="Iter: "+str(iteration), linewidth=1,color=line_colors[

    # Display output
    print "Iteration "+str(iteration)+":"
    print model
    print "Error rate: "+str(error_rate)
    print ""


# Add general plot content (to be displayed below)
colors = list(np.select([test_data[:,2] == -1, test_data[:,2] == 1 ], ['r', 'b']))
plt.scatter(test_data[:, 0], test_data[:, 1], c=colors)
plt.xlim(-1, 1)
plt.ylim(-1, 1)
plt.xticks()
plt.yticks()
plt.grid()
plt.title("Weighted Soft SVM Classification Results")
plt.legend()
```

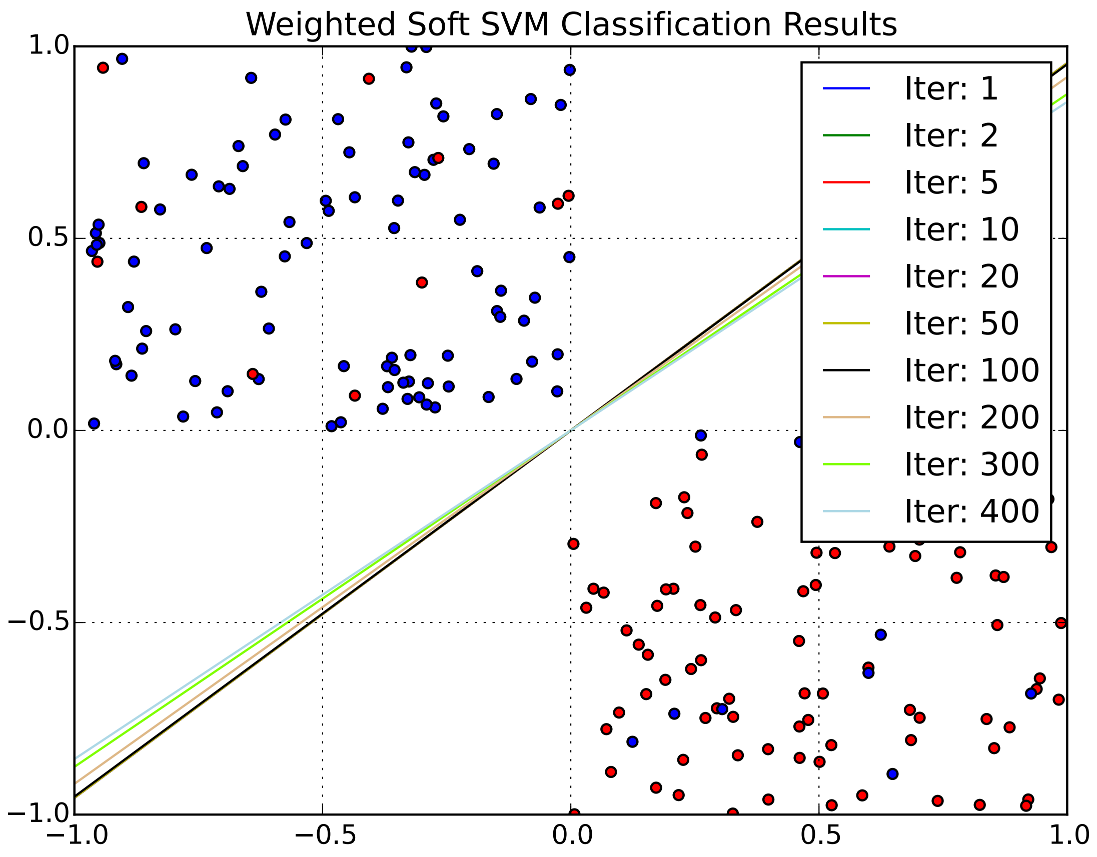
```
Number of Support Vectors: 200
Iteration 1:
[-0.00250442  0.00261902  0.0091748 ]
Error rate: 0.1

Number of Support Vectors: 200
Iteration 2:
[-0.0050086   0.00523779  0.01834961]
Error rate: 0.1

Number of Support Vectors: 200
Iteration 5:
[-0.01251961  0.0130925   0.04587402]
Error rate: 0.1

Number of Support Vectors: 200
Iteration 10:
[-0.02503296  0.02617846  0.09174804]
Error rate: 0.1

Number of Support Vectors: 200
Iteration 20:
[-0.0500409   0.05233076  0.18349608]
Error rate: 0.1

Number of Support Vectors: 200
Iteration 50:
[-0.12491484  0.13063092  0.4587402 ]
Error rate: 0.1

Number of Support Vectors: 127
Iteration 100:
[-0.21575647  0.22632572  0.8611923 ]
Error rate: 0.1

Number of Support Vectors: 110
Iteration 200:
[-0.18453874  0.20058721  1.03193755]
Error rate: 0.1

Number of Support Vectors: 109
Iteration 300:
[-0.14786286  0.16884563  1.09995816]
Error rate: 0.1

Converged after 332 iterations
Number of Support Vectors: 105
```

```
Iteration 400:
[-0.13762896  0.16087891  1.11623814]
Error rate: 0.1
```

Took 163 seconds (outdated)

FINISHED

```
%pyspark
showMPL(plt)
plt.close()
```



Took 1 seconds

FINISHED

Here we see that our model takes 332 iterations to converge to a solution based on 105 support vectors. In addition, we continue to see the same 10% misclassification rate we've seen elsewhere based on the deliberate noise we introduced into the dataset early in 11.3.
Took 0 seconds (outdated)

FINISHED

HW 11.4 - Weighted Soft SVM in MLlib

Does Spark MLLib have a weighted soft SVM learner. If so use it and report your findings on the weighted training set and test set.
Again, as per the MLLib documentation (http://spark.apache.org/docs/latest/mllib-linear-

methods.html), MLLib does not have a weighted soft SVM implementation as of version 1.6.1

Took 0 seconds (outdated)

READY