HW11, DATSCI W261

Team: Kuan Lin, Alejandro J. Rojas, Ricardo Barrera
Emails: kuanlin@ischool.berkeley.edu, ale@ischool.berkeley.edu, ricardofrank@ischool.berkeley.edu
Time of Initial Submission: 8:00 AM PST, Thursday, April 7, 2016
W261-1, Spring 2016 Week 11 Homework
Took 0 seconds. (outdated)

HW11.0 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.

Review the following Spark-notebook-based implementation of KMeans 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:

Notebook
https://www.dropbox.com/s/41q9lgyqhy8ed5g/EM-Kmeans.ipynb?dl=0

Notebook via NBViewer
http://nbviewer.ipython.org/urls/dl.dropbox.com/s/41q9lgyqhy8ed5g/EM-Kmeans.ipynb
Took 0 seconds. (outdated)

```
%pyspark
# zeppelin interop to matplotlib
import StringIO
import matplotlib.pyplot as plt
def show(p):
    img = StringIO.StringIO()
    p.savefig(img, format='svg')
    img.seek(0)
    print "%html <div style='width:600px'>" + img.buf + "</div>"
    p.clf()
```

Took 99 seconds.

Caching vs. Broadcasting

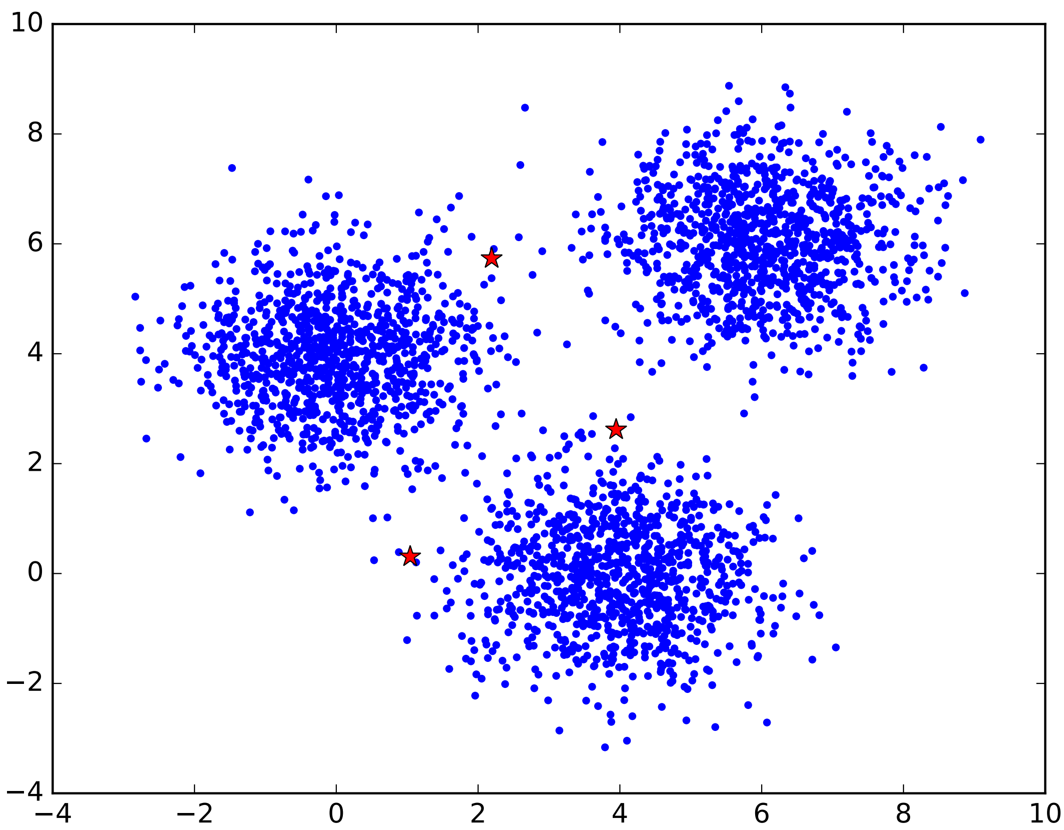
Caching data in Sapark is useful to keep in memory data that you need to process multiple times. An example of it would be cacg=hing a training dataset. On the other hand boradcasting is implemented whem you need to let worker nodes the status of a specific variable. For example when running logistic regression you need to broadcast the value of the weights after each iteration so that worker nodes can process gradient descent using the most current weights.
To make the K-Means code more efficient we will boradcast the values of the centroids of each cluster so that the worker nodes get that info on their own memory
Took 0 seconds. (outdated)

Broadcasted K-Mean

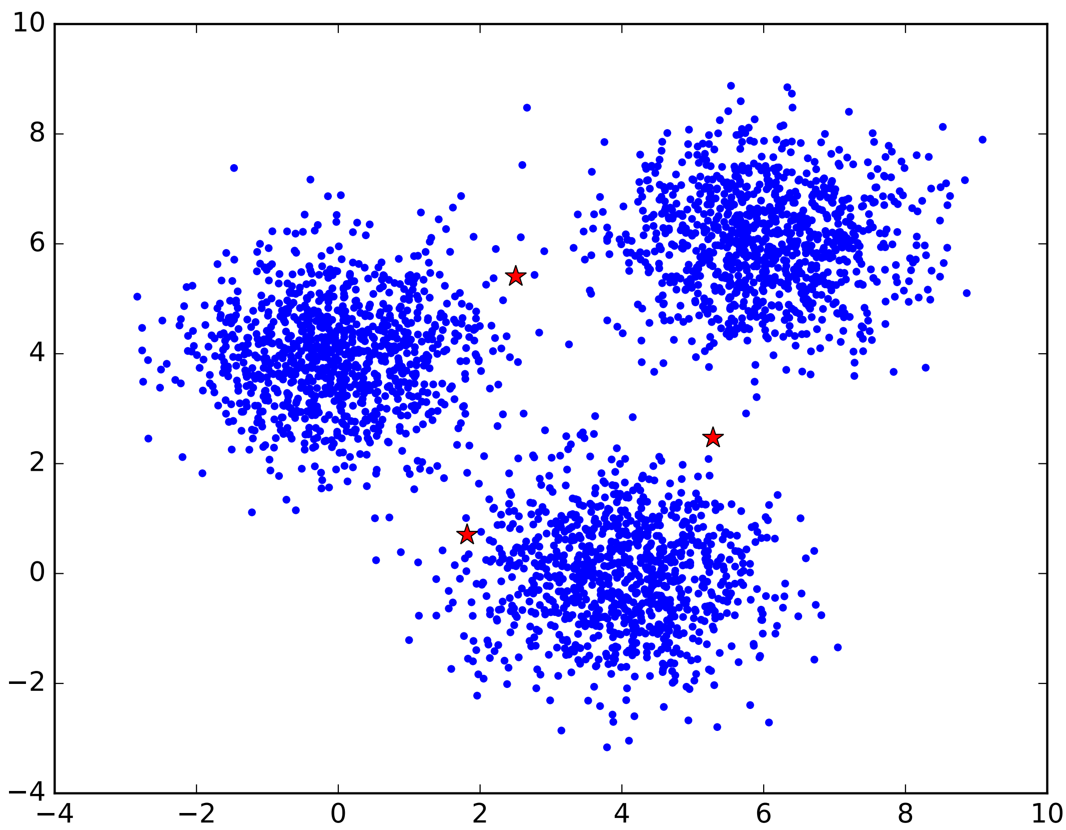
Took 0 seconds. (outdated)

```
%pyspark
# generate data
import numpy as np
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/data.csv',data,delimiter = ',')

import pylab

# 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_broadcasted.value)**2, axis=1).argmin() # use
    return (closest_centroid_idx,(x,1))

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()
    show(pylab.plt)

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]])
```

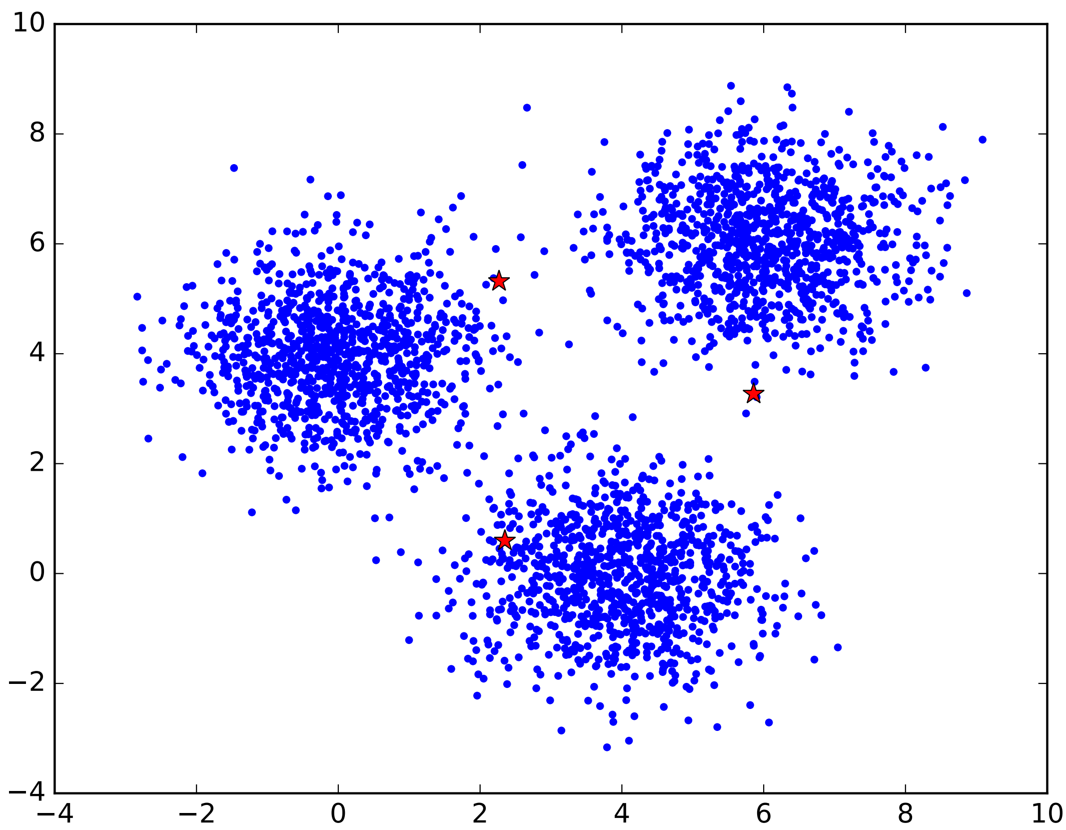
```
D = sc.textFile("file:///data/data.csv").cache()
iter_num = 0
for i in range(10):
    centroids_broadcasted = sc.broadcast(centroids) # broadcast centroids before any distribute
    res = D.map(nearest_centroid).reduceByKey(lambda x,y : (x[0]+y[0],x[1]+y[1])).collect()
    res = sorted(res,key = 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:"
print centroids
```



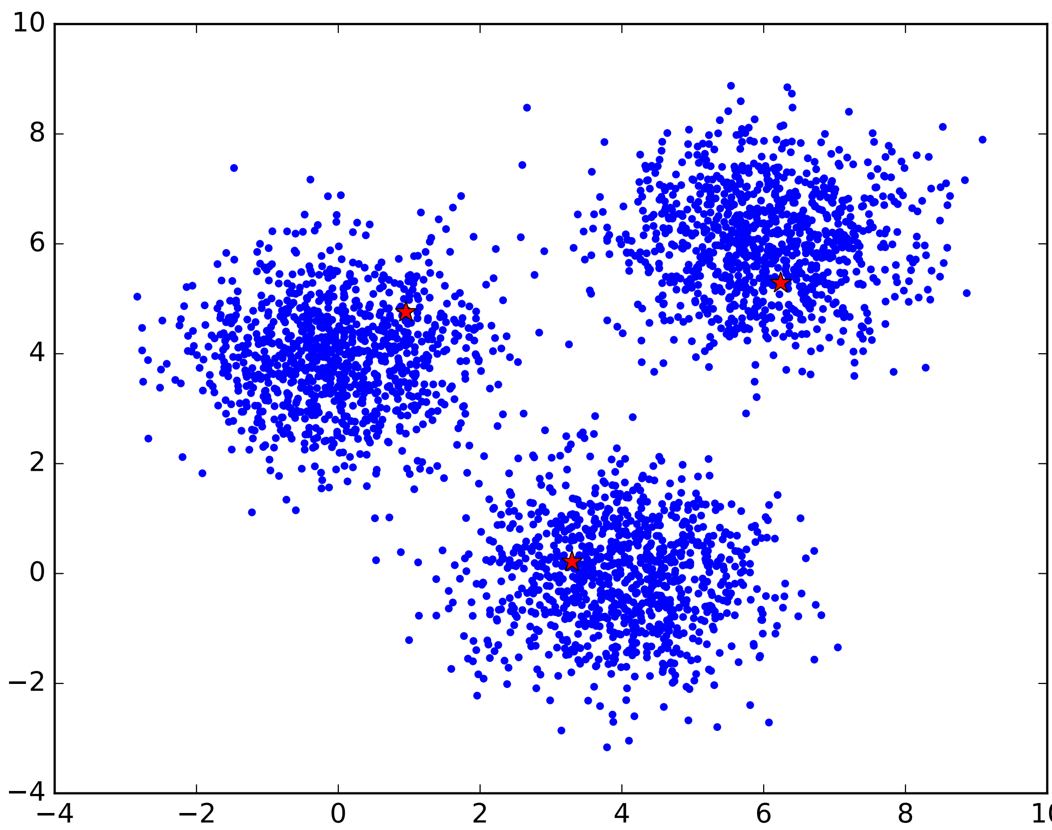
Iteration1 [[1.81718093 0.70375309] [5.28602178 2.46978436] [2.50336223 5.407373]] %html



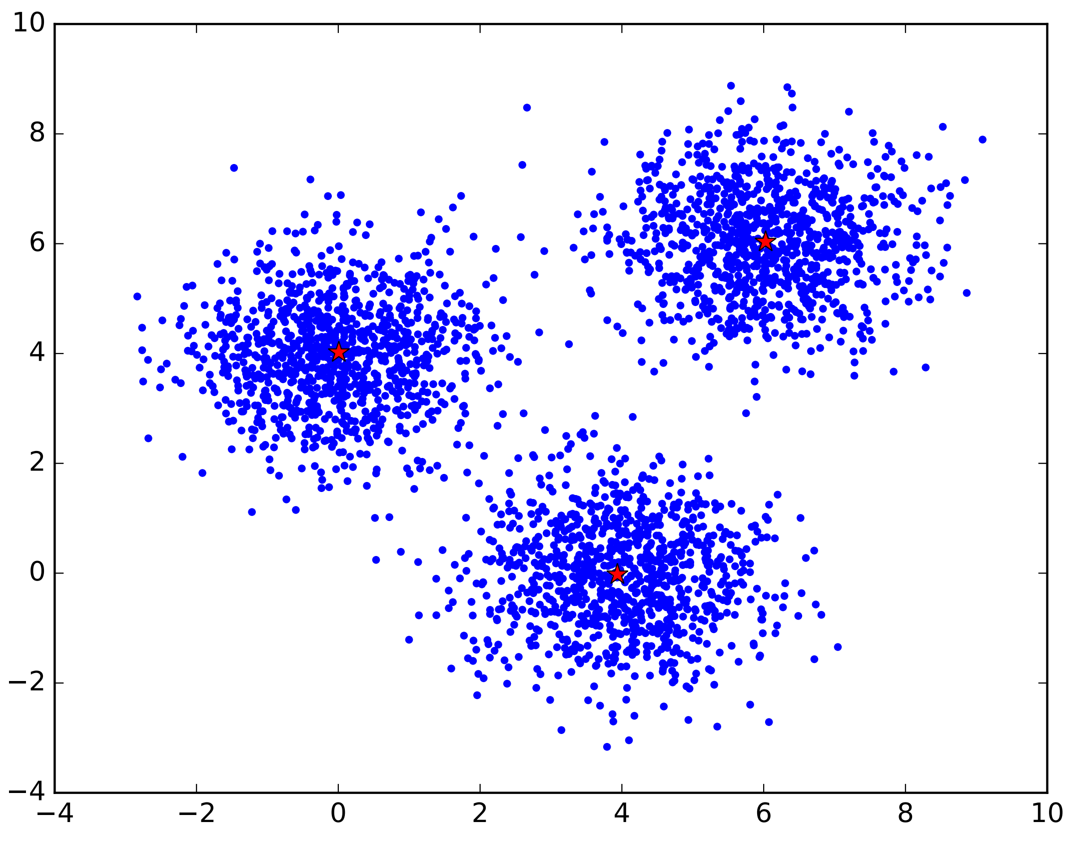
Iteration2 [[2.34995127 0.59701223] [5.8580947 3.27051641] [2.26729464 5.31927826]] %html



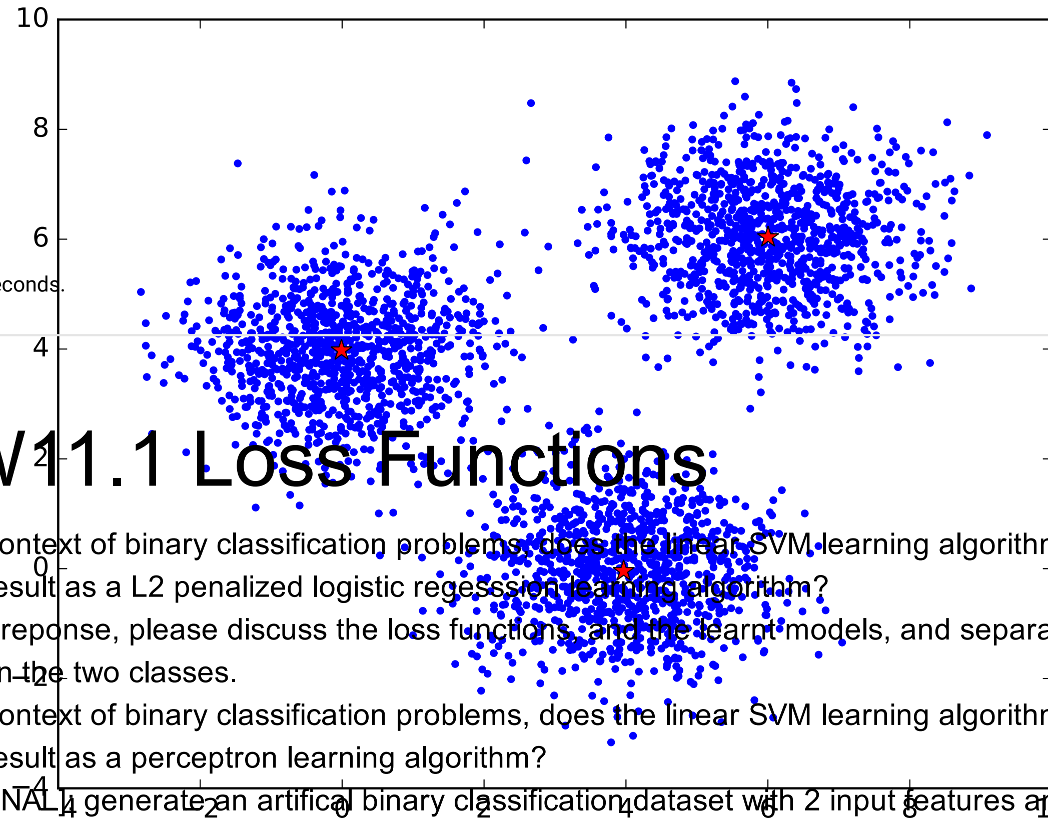
Iteration3 [[3.29638299 0.21308633] [6.23980831 5.28776201] [0.95424176 4.7580034]] %html



Iteration4 [[3.93649784 -0.02283234] [6.027216 6.03508886] [0.0076037 4.02187443]] %html



Iteration5 [[3.966843 -0.0419858] [6.00188246 6.0371927] [-0.00836659 3.97631336]] %html



Took 14 seconds.

HW11.1 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.
In the context of binary classification problems, does the linear SVM learning algorithm yield the same result as a perceptron learning algorithm?
[OPTIONAL]: generate an artifical binary classification dataset with 2 input features and plot the learnt separating surface for both a linear SVM and for logistic regression. Comment on the learnt surfaces. Please feel free to do this in Python (no need to use Spark).

Final Results: [[3.966843 -0.0419858] [6.00188246 6.0371927] [-0.00836659 3.97631336]]

Took 2 seconds. (outdated)

In the context of binary clasification problems, linear SVM should yield similar result as a L2-penalized logistic regression, as both algorithm pushes to have some positive margin.
However, the SVM loss function does not attempt to further distinquish non support vectors, that is,

data points with margins greater than 1. This is different from logistic regression which will preferentially push for more margin if possible.
Linear SVM will not likely yield the same result as a perceptron learning algorithm. The perceptron algorithm does not provide further possibility for gradient descent as long as the data point is classified correctly, even with only a very small margin.
Took 6 seconds. (outdated)

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)?
Descibe probabilitic interpretations of the L1 and L2 priors for penalized logistic regression (HINT: see synchronous slides for week 11 for details)
Took 2 seconds. (outdated)

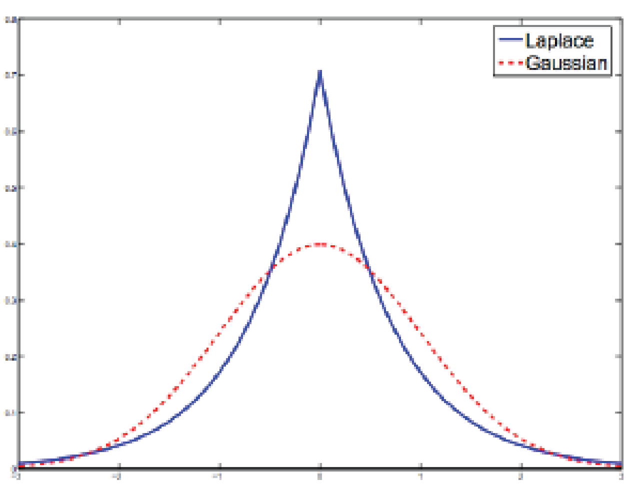
In the context of logistic regression, the three flavors of penalized terms are:

- L1 Reg, which penalizes for sum of absolute weights:
  $$l_{reg}(w) = \lambda \sum \|w_i\|$$
- L2 Reg, penalizes sum of squared weights:
  $$l_{reg}(w) = \lambda \sum w_i^2$$
- Elastic Net, penalizes a linear combination of L1 and L2 norms:
  $$\alpha \cdot \|w\| + (1-\alpha) \cdot 1/2 \cdot \|w\|^2$$

All of the above three regularization methods are supported by spark.mllib:
http://spark.apache.org/docs/latest/mllib-linear-methods.html#regularizers

Probablisitic interpretation of L1 and L2 priors:

L1 regularization can be interpreted as using Laplace distribution as the prior distribution for the model weights, where as L2 regularization can be interpreted as using gaussian distribution as the prior distribution for the model weights.



The Laplace distribution has more density closer to mean (usually zero in most settings) in comparison to the gaussian distribution, and therefore L1 regularization will tend to push the model weights toward zero.
Took 12 seconds. (outdated)

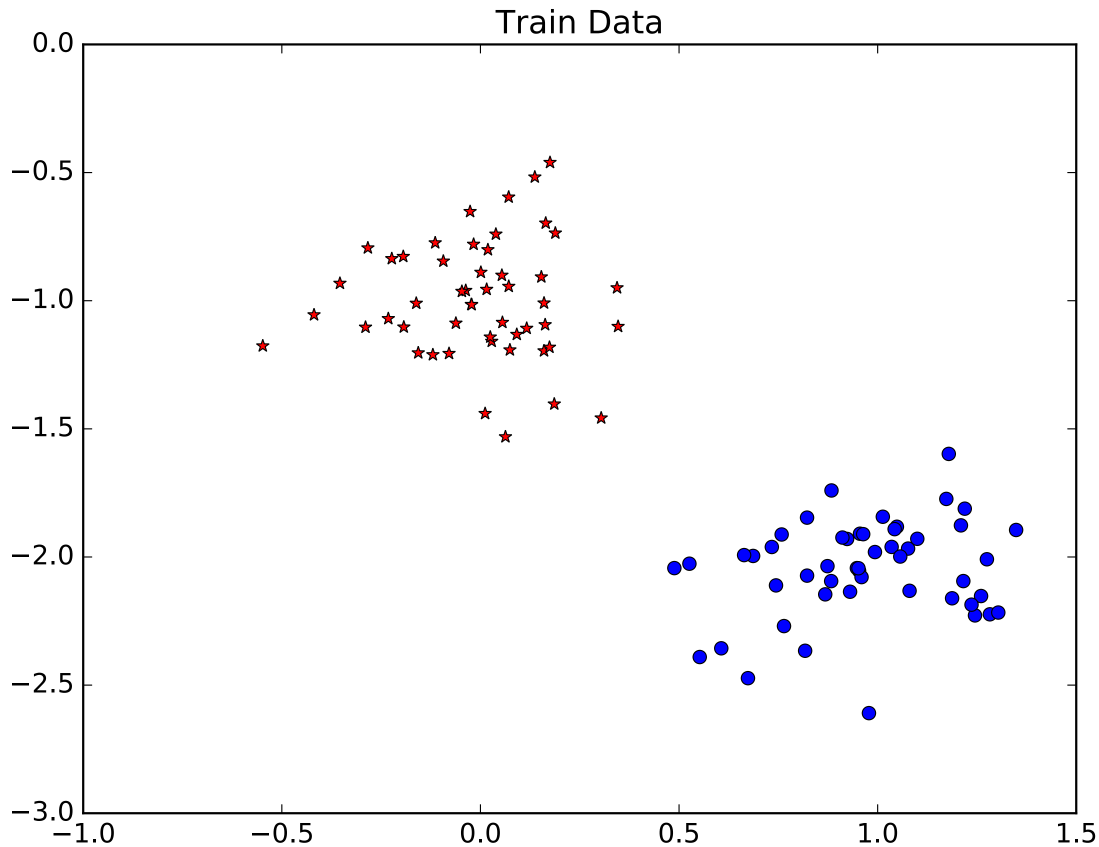
HW11.3 Logistic Regression

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.
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
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.
NOTE: For the remainder of this problem please use the non-linearly separable training and testing datasets.
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.
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 the above training dataset as follows: Weight each example 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 X1 and X2.
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.
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.
Took 7 seconds. (outdated)

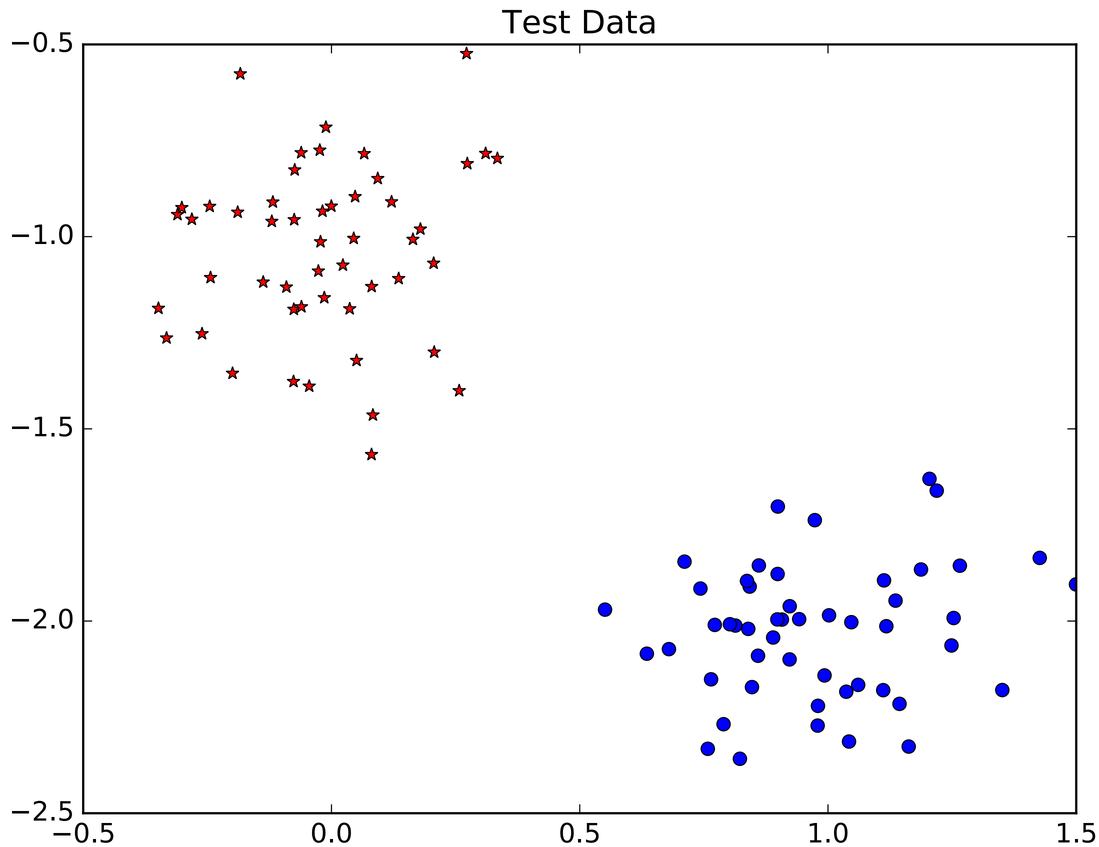
```
%pyspark
import numpy as np
np.random.seed(0)
def generateData(n):
    """
    generates a 2D linearly separable dataset with n samples.
    The third element of the sample is the label
    """
    xb = (np.random.normal(1,0.2,n)*2-1)/2-0.5
    yb = (np.random.normal(-1,0.2,n)*2-1)/2+0.5
    xr = (np.random.normal(1,0.2,n)*2-1)/2+0.5
    yr = (np.random.normal(-1,0.2,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
```

Took 2 seconds.

```
%pyspark
import pylab

# generate 100 linearly seperatable data and plot them
data_lin_seperable_train = generateData(50) # train data
data_lin_seperable_test = generateData(50) # test data

pylab.plot([d[0] for d in data_lin_seperable_train if d[2]==1], [d[1] for d in data_lin_seperab
pylab.plot([d[0] for d in data_lin_seperable_train if d[2]==-1], [d[1] for d in data_lin_sepera
pylab.plt.title("Train Data")
#pylab.show()
show(pylab.plt)

pylab.plot([d[0] for d in data_lin_seperable_test if d[2]==1], [d[1] for d in data_lin_seperabl
pylab.plot([d[0] for d in data_lin_seperable_test if d[2]==-1], [d[1] for d in data_lin_seperab
pylab.plt.title("Test Data")
#pylab.show()
show(pylab.plt)
```



%html



Took 12 seconds.

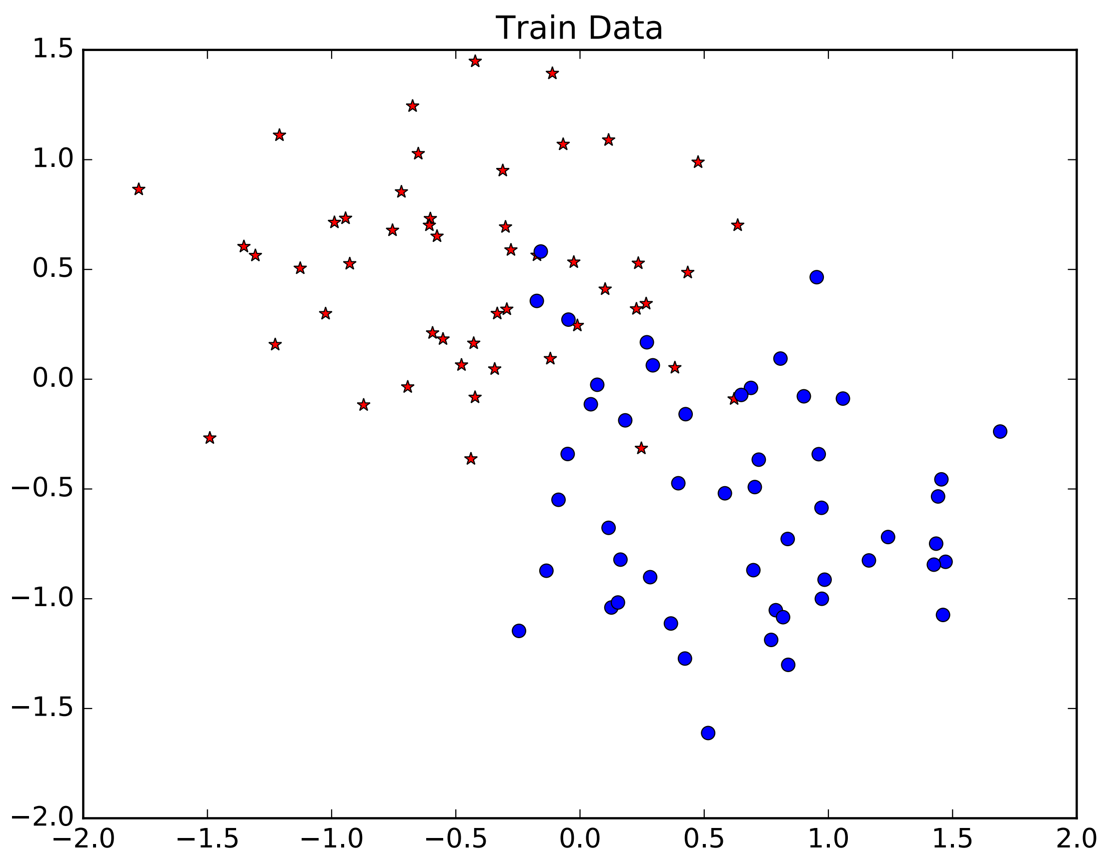
Modify this data generation code to generating non-linearly separable training and testing datasets (with approximately 10% of the data falling on the wrong side of the separating hyperplane. Plot the resulting datasets

Took 6 seconds. (outdated)

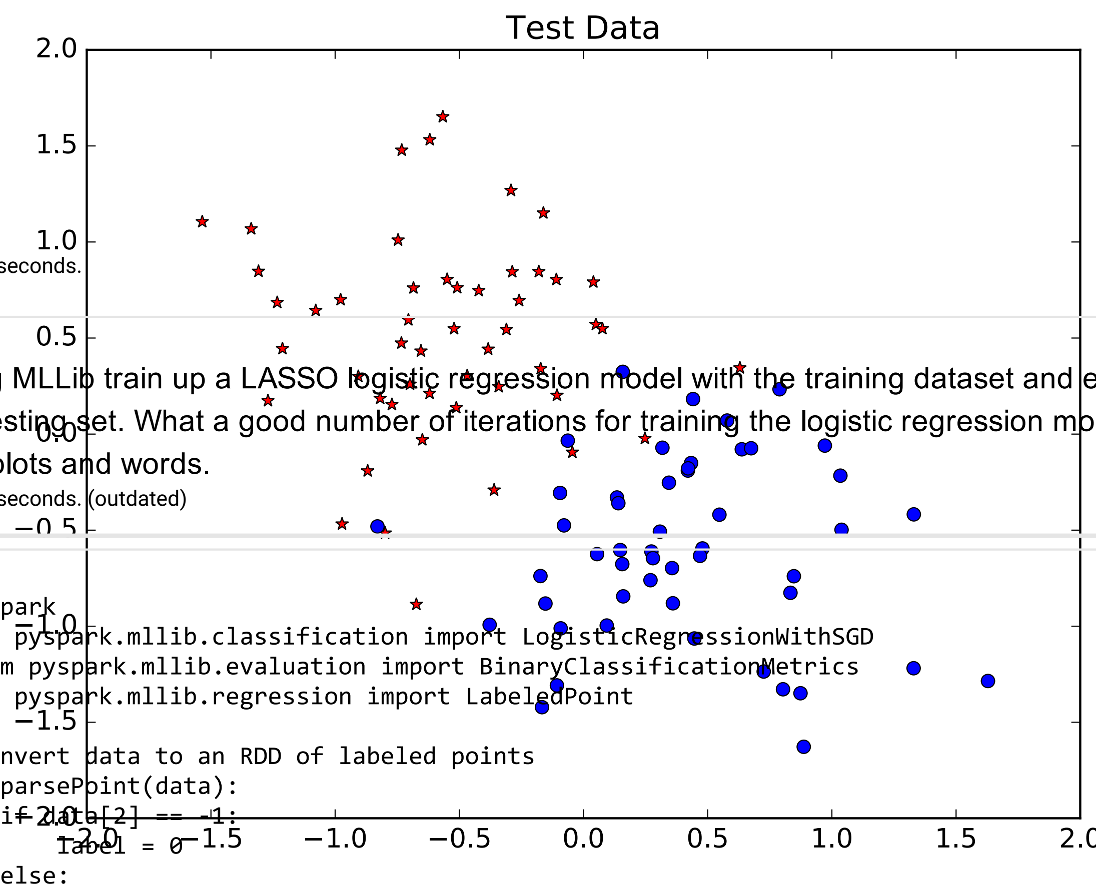
```
%pyspark
import numpy as np
np.random.seed(0)
def generateData2(n):
    """
    non-linearly seperable data
    """
    xb = np.random.normal(0,0.5,n)-0.5
    yb = np.random.normal(0,0.5,n)+0.5
    xr = np.random.normal(0,0.5,n)+0.5
    yr = np.random.normal(0,0.5,n)-0.5
    inputs = []
    for i in range(len(xb)):
        inputs.append([xb[i],yb[i],1])
        inputs.append([xr[i],yr[i],-1])
    return inputs
```

Took 3 seconds.

```
%pyspark
import pylab

# generate 100 linearly seperatable data and plot them
data_lin_inseperable_train = generateData2(50) # train data
data_lin_inseperable_test = generateData2(50) # test data

pylab.plot([d[0] for d in data_lin_inseperable_train if d[2]==1], [d[1] for d in data_lin_insep
pylab.plot([d[0] for d in data_lin_inseperable_train if d[2]==-1], [d[1] for d in data_lin_inse
pylab.plt.title("Train Data")
#pylab.show()
show(pylab.plt)

pylab.plot([d[0] for d in data_lin_inseperable_test if d[2]==1], [d[1] for d in data_lin_insepe
pylab.plot([d[0] for d in data_lin_inseperable_test if d[2]==-1], [d[1] for d in data_lin_insep
pylab.plt.title("Test Data")
#pylab.show()
show(pylab.plt)
```



%html



Took 4 seconds.

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 7 seconds. (outdated)

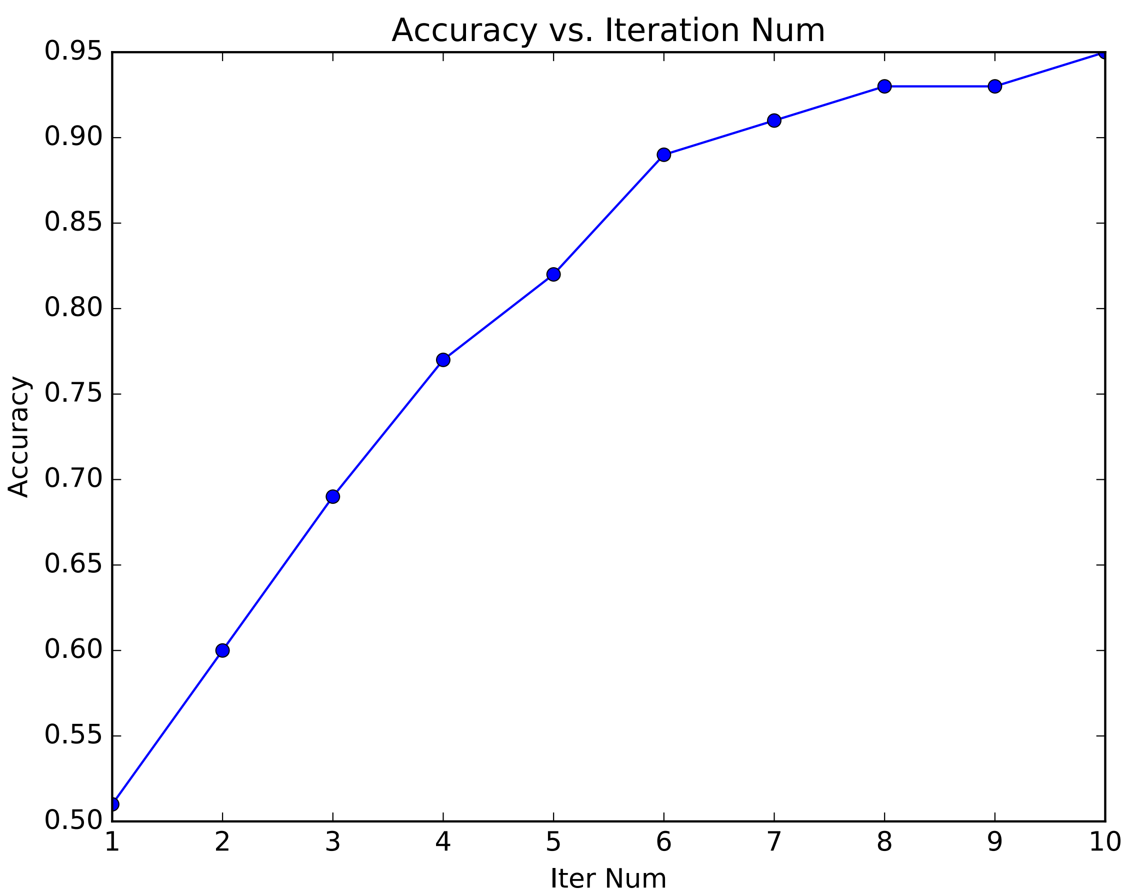
```
%pyspark
from pyspark.mllib.classification import LogisticRegressionWithSGD
#from pyspark.mllib.evaluation import BinaryClassificationMetrics
from pyspark.mllib.regression import LabeledPoint

# convert data to an RDD of labeled points
def parsePoint(data):
    if data[2] == -1:
        label = 0
    else:
        label = 1
    return LabeledPoint(label, [data[0], data[1]])

train_data = sc.parallelize(data_lin_inseperable_train).map(parsePoint).cache()
test_data = sc.parallelize(data_lin_inseperable_test).map(parsePoint).cache()

test_metric_results = []
for iter_n in range(1,11):
    model = LogisticRegressionWithSGD.train(data=train_data, regType="l1", intercept=True,
                        miniBatchFraction=1.0, iterations=iter_n) # lasso regression
```

```
    accuracy = test_data.map(lambda lp: ((float(model.predict(lp.features))==lp.label), 1)).rec
    accuracy = 1.0*accuracy[0]/accuracy[1]
    test_metric_results.append((iter_n,accuracy))
    print "Iter %s | Accuracy:%.4f" %(iter_n, accuracy)
```

```
Iter 1 | Accuracy:0.5100
Iter 2 | Accuracy:0.6000
Iter 3 | Accuracy:0.6900
Iter 4 | Accuracy:0.7700
Iter 5 | Accuracy:0.8200
Iter 6 | Accuracy:0.8900
Iter 7 | Accuracy:0.9100
Iter 8 | Accuracy:0.9300
Iter 9 | Accuracy:0.9300
Iter 10 | Accuracy:0.9500
```

Took 46 seconds.

```
%pyspark
import matplotlib.pyplot as plt

plt.plot([d[0] for d in test_metric_results], [d[1] for d in test_metric_results], '-o')
plt.title("Accuracy vs. Iteration Num")
plt.xlabel("Iter Num")
plt.ylabel("Accuracy")
show(plt)
```



Took 5 seconds.

Based on accuracy, the SGD implementation of LogisticRegression with L1 regularization seems to begin to level-out when iteration is about 8 or more.
Took 7 seconds. (outdated)

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 the above training dataset as follows: Weight each example 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 X1 and X2.
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.
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.
Took 7 seconds. (outdated)

```
%pyspark
import numpy as np

def readPoint(data):
    label = data[2]
    x = [data[0], data[1], 1.0]  #add bias term
    return (x, label)

def vectorWeight(v1, v2):
    weight = 1.0/((v1**2+v2**2)**0.5)
    if weight < 0.1:
        weight = 0.1
    elif weight > 10:
        weight = 10
    return weight

def WeightedlogisticRegressionGD(data, wInitial=None, learningRate=0.05, iterations=10, regPara
    featureLen = len(data.take(1)[0][0])
    #total_weight = data.count()
    total_weight = data.map(lambda p: vectorWeight(p[0][0], p[0][1])).reduce(lambda a,b: a+b)
    if wInitial is None:
        w = np.random.normal(size=featureLen) # w should be broadcasted if it is large
    else:
        w = wInitial
    for i in range(iterations):
        #print "Iteration %s"%(i+1)
        wBroadcast = sc.broadcast(w)
        gradient = data.map(lambda p: vectorWeight(p[0][0], p[0][1])*(1 / (1 + np.exp(-p[1]*np.
                    .reduce(lambda a, b: a + b)
        #gradient = data.map(lambda p: (1 / (1 + np.exp(-p[1]*np.dot(wBroadcast.value, p[0])))-
        #            .reduce(lambda a, b: a + b)
        if regType == "Ridge":
            wReg = w * 1
            wReg[-1] = 0 #last value of weight vector is bias term, ignored in regularization
        elif regType == "Lasso":
```

```
            wReg = w * 1
            wReg[-1] = 0 #last value of weight vector is bias term, ignored in regularization
            wReg = (wReg>0).astype(int) * 2-1
        else:
            wReg = np.zeros(w.shape[0])
        gradient = gradient + regParam * wReg  #gradient:  GD of Sqaured Error+ GD of regulari:
        wdelta = learningRate * gradient / total_weight # scale by total weight

        if sum(abs(wdelta))<=stopCriteria*sum(abs(w)): #Convergence condition
            print "convergenced reached at iteration %"%(i+1)
            break
        #w = w - learningRate * gradient / n
        w = w - wdelta
        #print w
    print "total iterations: %s"%(i+1)
    return w
```

Took 5 seconds.

```
%pyspark
train_data = sc.parallelize(data_lin_inseperable_train).map(readPoint).cache()
test_data = sc.parallelize(data_lin_inseperable_test).map(readPoint).cache()

w = WeightedlogisticRegressionGD(train_data, regType="Lasso", stopCriteria=0.001, iterations=5(
print "final weight:%s"%str(w)
```

```
total iterations: 50
final weight:[ 0.48698643  1.39042164 -0.08542408]
```

Took 14 seconds.

```
%pyspark
# evaluate on test set

def isAccurate(x, w, label):
    score = 1.0/(1.0 + np.exp(-1.0*np.dot(w,x)))
    print "label|score: %s | %.4f"%(label, score)
    if (label == 1 and score >= 0.5) or (label == -1 and score < 0.5):
        return 1
    else:
        return 0

def LogisticRegAccuracy(data, w):
    wBroadcast = sc.broadcast(w)
    result = test_data.map(lambda p: (isAccurate(p[0], wBroadcast.value, p[1]), 1)).reduce(laml
    return result[0]*1.0/result[1]

accuracy = LogisticRegAccuracy(test_data, w)
print "Misclassification Error Rate: %s"%(str(100-round(accuracy*100, 4))+"%")
```

```
Misclassification Error Rate: 23.0%
```

Took 5 seconds.

HW11.4 SVMs

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.
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?
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.

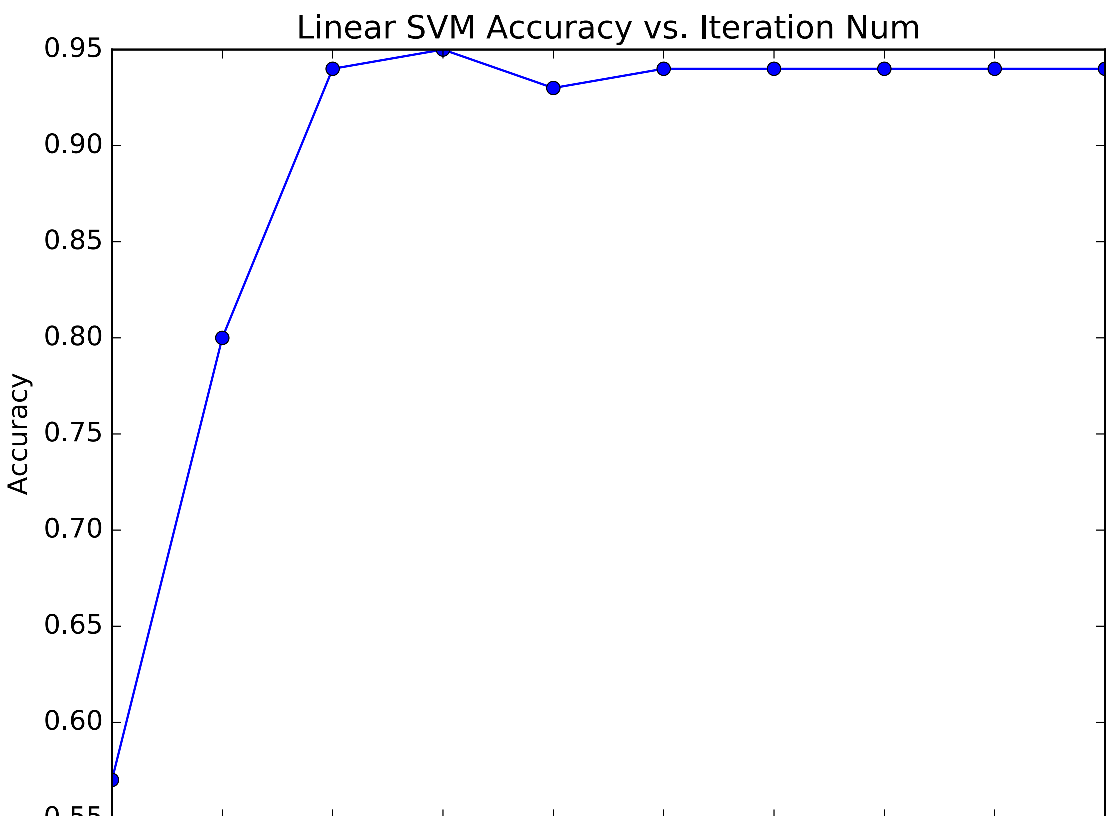
Took 6 seconds. (outdated)

```
%pyspark

# using mllib

from pyspark.mllib.classification import SVMWithSGD
#from pyspark.mllib.evaluation import BinaryClassificationMetrics
from pyspark.mllib.regression import LabeledPoint

# convert data to an RDD of labeled points
def parsePoint(data):
    if data[2] == -1:
        label = 0
    else:
        label = 1
    return LabeledPoint(label, [data[0], data[1]])

train_data = sc.parallelize(data_lin_inseperable_train).map(parsePoint).cache()
test_data = sc.parallelize(data_lin_inseperable_test).map(parsePoint).cache()

test_metric_results = []
for iter_n in range(1,11):
    model = SVMWithSGD.train(data=train_data, regType="l1", intercept=True,
                             miniBatchFraction=1.0, iterations=iter_n) # lasso, same parameters
    accuracy = test_data.map(lambda lp: ((float(model.predict(lp.features))==lp.label), 1)).rec
    accuracy = 1.0*accuracy[0]/accuracy[1]
    test_metric_results.append((iter_n,accuracy))
    print "Iter %s | Accuracy:%.4f" %(iter_n, accuracy)
```

```
Iter 1 | Accuracy:0.5700
Iter 2 | Accuracy:0.8000
Iter 3 | Accuracy:0.9400
Iter 4 | Accuracy:0.9500
Iter 5 | Accuracy:0.9300
Iter 6 | Accuracy:0.9400
Iter 7 | Accuracy:0.9400
Iter 8 | Accuracy:0.9400
Iter 9 | Accuracy:0.9400
Iter 10 | Accuracy:0.9400
```

Took 24 seconds.

```
%pyspark
import matplotlib.pyplot as plt

plt.plot([d[0] for d in test_metric_results], [d[1] for d in test_metric_results], '-o')
plt.title("Linear SVM Accuracy vs. Iteration Num")
plt.xlabel("Iter Num")
plt.ylabel("Accuracy")
show(plt)
```



ZeppelinHub (/)Viewer (/viewer)

Took 6 seconds.

MIDS-W261-2016_HWK11-Week11-Team5.ipynb

For this particular dataset LinearSVM seems to converge very fast, plateauing at 94% accuracy after 3 iterations. This is much faster than logistic regression.

Took 8 seconds. (outdated)

```
%pyspark

# home-grown SVM code

def vectorWeight(v1, v2):
    weight = 1.0/((v1**2+v2**2)**0.5)
    if weight < 0.1:
        weight = 0.1
    elif weight > 10:
        weight = 10
    return weight

def WeightedSVMRegressionGD(data, wInitial=None, learningRate=0.05, iterations=100, regParam=0.
    featureLen = len(data.take(1)[0][0])
    total_weight = data.map(lambda p: vectorWeight(p[0][0], p[0][1])).reduce(lambda a,b: a+b)
    if wInitial is None:
        w = np.random.normal(size=featureLen) # w should be broadcasted if it is large
    else:
        w = wInitial

    for i in range(iterations):
        wBroadcast = sc.broadcast(w)

        sv = data.filter(lambda p: p[1]*np.dot(wBroadcast.value, p[0]) < 1).cache() # find the

        if sv.isEmpty(): # no more support vectors
            break

        # calculate weighted gradients (hinge loss)
        gradient = -1.0*sv.map(lambda p: vectorWeight(p[0][0], p[0][1])*p[1]*np.array(p[0])).re

        if regType == "Ridge":
            wReg = w * 1
            wReg[-1] = 0 #last value of weight vector is bias term, ignored in regularization
        elif regType == "Lasso":
            wReg = w * 1
            wReg[-1] = 0 #last value of weight vector is bias term, ignored in regularization
            wReg = (wReg>0).astype(int) * 2-1
        else:
            wReg = np.zeros(w.shape[0])

        gradient = gradient + regParam * wReg  #gradient:  GD of Sqaured Error+ GD of regulari
        wdelta = learningRate * gradient

        if sum(abs(wdelta))<=stopCriteria*sum(abs(w)): #Convergence condition
            print "convergenced reached at iteration %"%(i+1)
            break
        w = w - wdelta

        #print "iteration %s"%i
        #print w

    return (w, i+1, sv.count()) # learned weights, total iters, support vector counts
```

Took 6 seconds.

```
%pyspark
train_data = sc.parallelize(data_lin_inseperable_train).map(readPoint).cache()
test_data = sc.parallelize(data_lin_inseperable_test).map(readPoint).cache()
w = WeightedSVMRegressionGD(train_data, regType="Lasso", stopCriteria=0.001, iterations=50, reg
print "final weight:%s | total iters:%s | SV counts: %s"%(str(w[0]), str(w[1]), str(w[2]))

# eval with test data

def isAccurateSVM(x, w, label):
    score = label*np.dot(w, x)
    print "label|score: %s | %.4f"%(label, score)
    if score >= 0:
        return 1
    else:
        return 0

def SVMRegAccuracy(data, w):
    wBroadcast = sc.broadcast(w)
    result = data.map(lambda p: (isAccurateSVM(p[0], wBroadcast.value, p[1]), 1)).reduce(lambda
    return result[0]*1.0/result[1]

accuracy = SVMRegAccuracy(test_data, w[0])
print "Misclassification Error Rate: %s"%(str(100-round(accuracy*100, 4))+"%")
```

```
final weight:[-0.97073657  1.40745858  0.15961314] | total iters:50 | SV counts: 42
Misclassification Error Rate: 5.0%
```

Took 20 seconds.

11.5 [OPTIONAL] Distributed Perceptron algorithm

Using the following papers as background:
http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en//pubs/

https://www.dropbox.com/s/a5pdcp0r8ptudgj/gesmundo-tomeh-eacl-2012.pdf?dl=0

http://www.slideshare.net/matsubaray/distributed-perceptron
Implement each of the following flavors of perceptron learning algorithm:

- Serial (All Data): This is the classifier returned if trained serially on all the available data. On a single computer for example (Mistake driven)
- Serial (Sub Sampling): Shard the data, select one shard randomly and train serially.
- Parallel (Parameter Mix): Learn a perceptron locally on each shard: Once learning is

complete combine each learnt percepton using a uniform weighting
- Parallel (Iterative Parameter Mix) as described in the above papers.

Took 6 seconds. (outdated)

```
%pyspark

# perceptron gradient descent calculation

def perceptronSubGradientCalc(data, w, regParam, feature_size):
    #gradient = data.map(lambda p: -p[1]*np.array(p[0]) if (p[1]*np.dot(w.value,p[0]))<0 else r
    incorrect = data.filter(lambda p: np.dot(w.value,p[0])*p[1] < 0)
    if incorrect.isEmpty():
        gradient = np.zeros(feature_size)
    else:
        gradient = incorrect.map(lambda p: -p[1]*np.array(p[0])).reduce(lambda a,b: a+b)
    wreg = np.array(w.value)  # L2 reg
    wreg[-1] = 0 # don't count the bias term in reg
    return gradient/data.count() + regParam*wreg

def perceptronAccuracy(data, w):
    correct_count = data.map(lambda p: 1 if np.dot(w.value, p[0])*p[1] > 0 else 0).reduce(lambc
    return 1.0*correct_count/data.count()
```

Took 6 seconds.

```
%pyspark

# train serially with all data

def perceptronGDSerialAll(data, wInitial=None, nShards=1, nIter=10, stopCriteria=0.001,
                 learningRate=0.05, regParam=0.01):
    feature_size = len(data.take(1)[0][0])
    if wInitial is None:
        w = np.random.normal(size=feature_size)
    else:
        w = wInitial
    #model_data = data.randomSplit([1.0/nShards]*nShards) # split data into shards
    for i in range(nIter):
        wb = sc.broadcast(w) # broadcast weight
        wdelta = learningRate*perceptronSubGradientCalc(data, wb, regParam, feature_size)
        if sum(abs(wdelta))<=stopCriteria*sum(abs(w)):
            break
        w = w - wdelta
    return w

w = perceptronGDSerialAll(train_data, nIter=50)
print "Train serially on all data"
print "learned weight: %s" % str(w)
print "Accuracy: %s" % perceptronAccuracy(test_data, sc.broadcast(w))
```

```
Train serially on all data
learned weight: [-0.36552607  0.00169171  0.05655527]
Accuracy: 0.83
```

Took 21 seconds.

```
%pyspark

# train serially on random shards

def perceptronGDSerialShards(data, wInitial=None, nShards=1, nIter=10, stopCriteria=0.001,
                 learningRate=0.05, regParam=0.01):
    feature_size = len(data.take(1)[0][0])
    if wInitial is None:
        w = np.random.normal(size=feature_size)
    else:
        w = wInitial
    for shared_data in data.randomSplit([1.0/nShards]*nShards): # split data into shards
        for i in range(nIter):
            wb = sc.broadcast(w) # broadcast weight
            wdelta = learningRate*perceptronSubGradientCalc(shared_data, wb, regParam, feature_
            if sum(abs(wdelta))<=stopCriteria*sum(abs(w)):
                break
            w = w - wdelta
    return w

w = perceptronGDSerialShards(train_data, nShards=4, nIter=10)
print "Train serially on random shards"
print "learned weight: %s" % str(w)
print "Accuracy: %s" % perceptronAccuracy(test_data, sc.broadcast(w))
```

```
Train serially on random shards
learned weight: [ 0.01615768  0.14727661 -0.00092179]
Accuracy: 0.87
```

Took 19 seconds.

```
%pyspark

# non-iterative parameter mixing

def percetronTrainLocal(data, w, regParam, feature_size, learningRate, nIter):
    # train with local data instead of RDD
    weight = np.array(w.value)
    for i in range(nIter):
        gradient = sum(map(lambda p: -p[1]*np.array(p[0]) if (p[1]*np.dot(weight,p[0]))<0 else
        wreg = weight*1  # L2 reg
        wreg[-1] = 0 # don't count the bias term in reg
        weight -= learningRate*(gradient/len(data) + regParam*wreg)
    return weight

def perceptronGDSerialParaMix(data, wInitial=None, nShards=1, nIter=10, learningRate=0.05, regP
    feature_size = len(data.take(1)[0][0])
    if wInitial is None:
        w = np.random.normal(size=feature_size)
    else:
        w = wInitial
    wb = sc.broadcast(w)
    model_data = sc.parallelize([d.collect() for d in data.randomSplit([1.0/nShards]*nShards)]
    learned_w = model_data.map(lambda data: percetronTrainLocal(data, wb, regParam, feature_si
    return sum(learned_w)/len(learned_w) # take uniform average of the learned weights
```

```
w = perceptronGDSerialParaMix(train_data, nShards=4, nIter=50)
print "non-iterative parameter mixing"
print "learned weight: %s" % str(w)
print "Accuracy: %s" % perceptronAccuracy(test_data, sc.broadcast(w))
```

```
non-iterative parameter mixing
learned weight: [ 0.64234314  0.12919467  0.64654108]
Accuracy: 0.43
```

Took 8 seconds.

```
%pyspark

# Iterative Param Mixing

def perceptronGDSerialIterParaMix(data, wInitial=None, nShards=1, nIter=10, learningRate=0.05,
    feature_size = len(data.take(1)[0][0])
    if wInitial is None:
        w = np.random.normal(size=feature_size)
    else:
        w = wInitial
    model_data = sc.parallelize([d.collect() for d in data.randomSplit([1.0/nShards]*nShards)]
    for i in range(nIter):
        wb = sc.broadcast(w)
        # train only 1 epoch and then do param mixing
        learned_w = model_data.map(lambda data: percetronTrainLocal(data, wb, regParam, feature
        w = sum(learned_w)/len(learned_w) # take uniform average of the learned weights
    return w

w = perceptronGDSerialIterParaMix(train_data, nShards=4, nIter=50)
print "Iterative Param Mixing"
print "learned weight: %s" % str(w)
print "Accuracy: %s" % perceptronAccuracy(test_data, sc.broadcast(w))
```

```
Iterative Param Mixing
learned weight: [-0.72533034  0.15420804  0.05242582]
Accuracy: 0.91
```

Took 13 seconds.