FINISHED

DATASCI W261, Machine Learning at Scale

Assignement: week #11 (https://www.zeppelinhub.com/viewer/notebooks/aHR0cHM6Ly9yYXcuZ2l0aHVi

Lei Yang (mailto:leiyang@berkeley.edu) | Michael Kennedy (mailto:mkennedy@ischool.berkeley.edu) | Natarajan Krishnaswami (mailto:natarajan@krishnaswami.org)

Due: 2016-04-07, 8AM PST

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.
    - 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/E Kmeans.ipynb)
- Please describe your changes in English first, implement, comment your code and highlight your changes:

Answer

- Broadcast variables allows the program to efficiently send a large, read-only value to *all* the worker

nodes for use in one or more Spark operations.

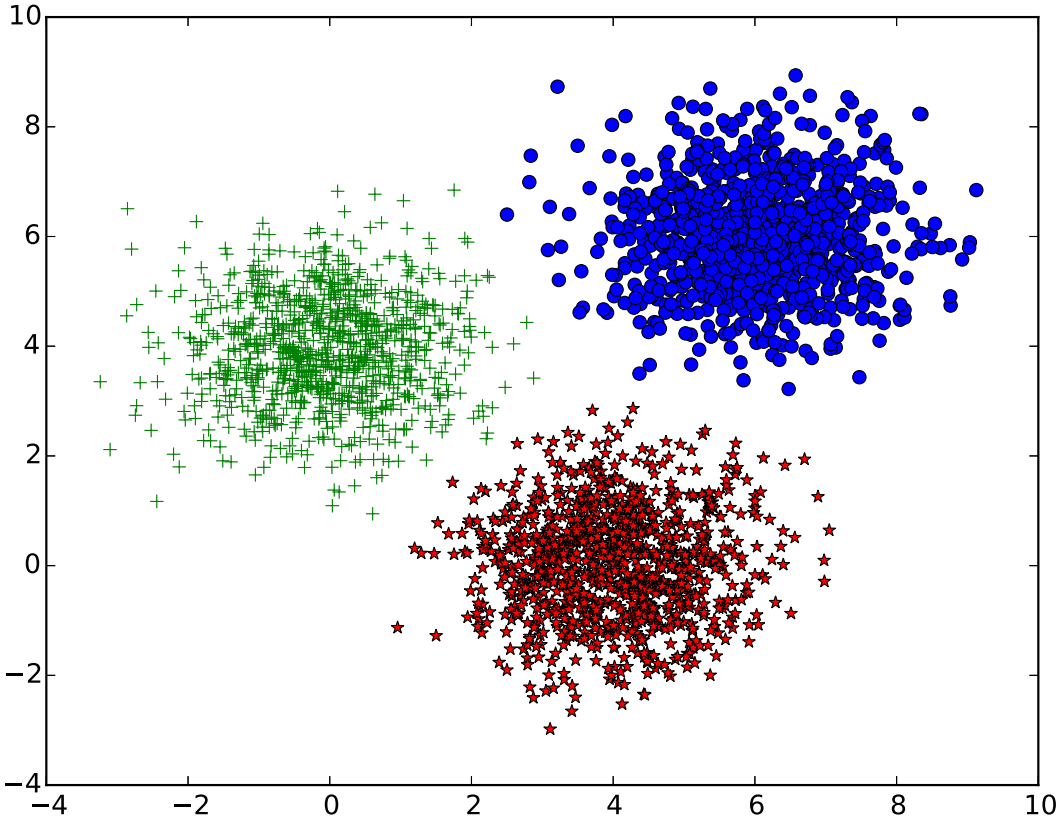
- Caching stores RDD in memory to facilitate reuse of the data, by avoiding regenerating the RDD from the lineage
- Broadcast reduce data transfer over the network, caching reduce data reprocessing and retrieving (from disk)
- **Code changes**:
    - broadcast variable is read-only, it is accessed via the *.value* property of the variable
    - to update the new value, we need to re-broadcast again
    - highlight changes below, after each iteration, reinitialize centers by broadcasting the new value

Took 2 seconds

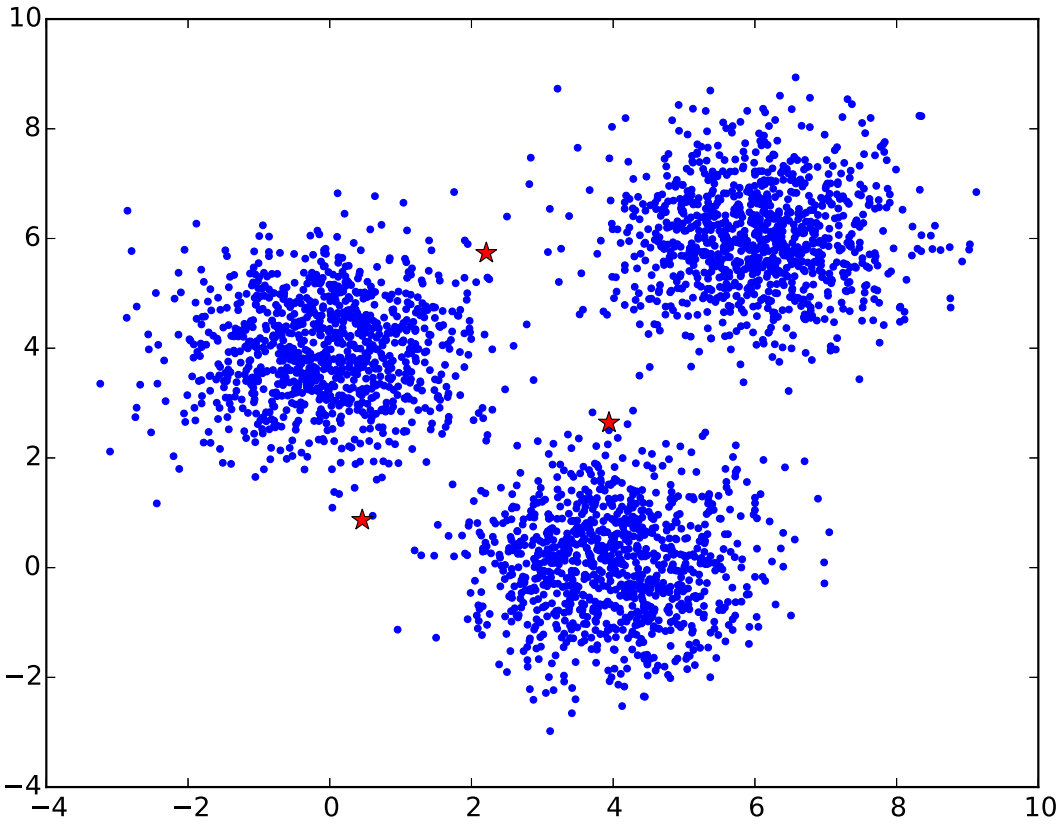
```
%pyspark

# code for inline plotting
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import StringIO

def showMPL(plot, px=600):
    img = StringIO.StringIO()
    plot.savefig(img, format='svg')
    img.seek(0)
    print "%html <div style='width:" + str(px) +"px'>" + img.buf + "</div>"
```

```
%pyspark

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 = ',')

plt.clf()
plt.plot(samples1[:, 0], samples1[:, 1],'*', color = 'red')
plt.plot(samples2[:, 0], samples2[:, 1],'o',color = 'blue')
plt.plot(samples3[:, 0], samples3[:, 1],'+',color = 'green')
showMPL(plt)

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

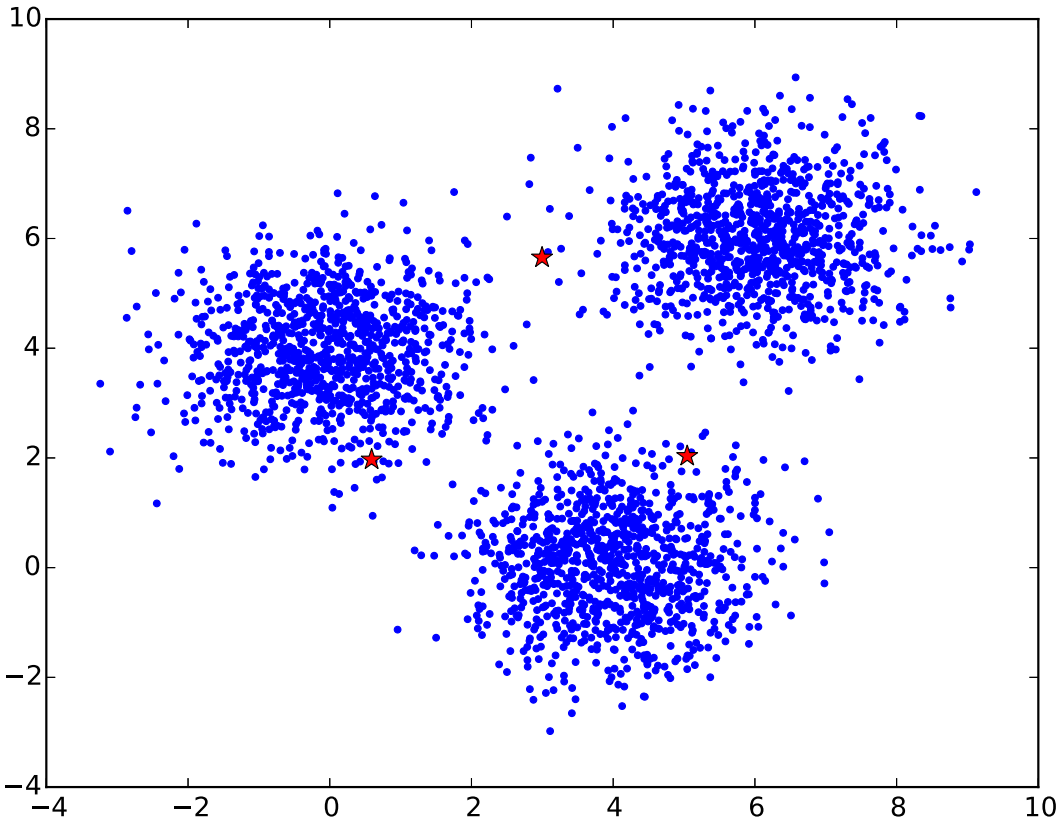
```
#plot centroids and data points for each iteration
def plot_iteration(means):
    plt.clf()
    plt.plot(samples1[:, 0], samples1[:, 1], '.', color = 'blue')
    plt.plot(samples2[:, 0], samples2[:, 1], '.', color = 'blue')
    plt.plot(samples3[:, 0], samples3[:, 1],'.', color = 'blue')
    plt.plot(means[0][0],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


K = 3
# Initialization: initialization of parameter is fixed to show an example
################################## broadcasting variable ##################################
centroids = sc.broadcast(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 = 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
    ################################## access broadcasting value ##########################
    if np.sum(np.absolute(centroids_new-centroids.value))<0.01:
        break
    print "Iteration " + str(iter_num)
    iter_num = iter_num + 1
    ################################## broadcasting new centers ###########################
    centroids = sc.broadcast(centroids_new)
    print centroids.value
    plot_iteration(centroids.value)
print "Final Results:"
print centroids.value
```



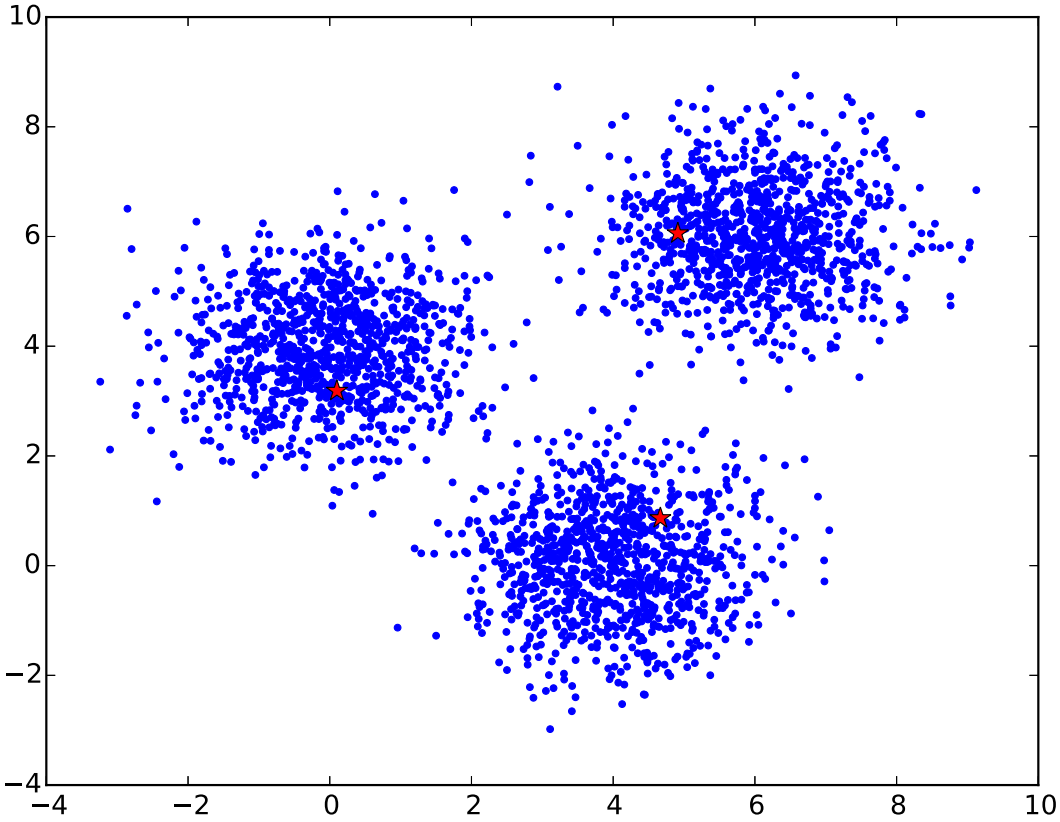
Iteration 0 [[0.45882362 0.8676519] [3.9408185 2.64230366] [2.2093432 5.73550592]] %html



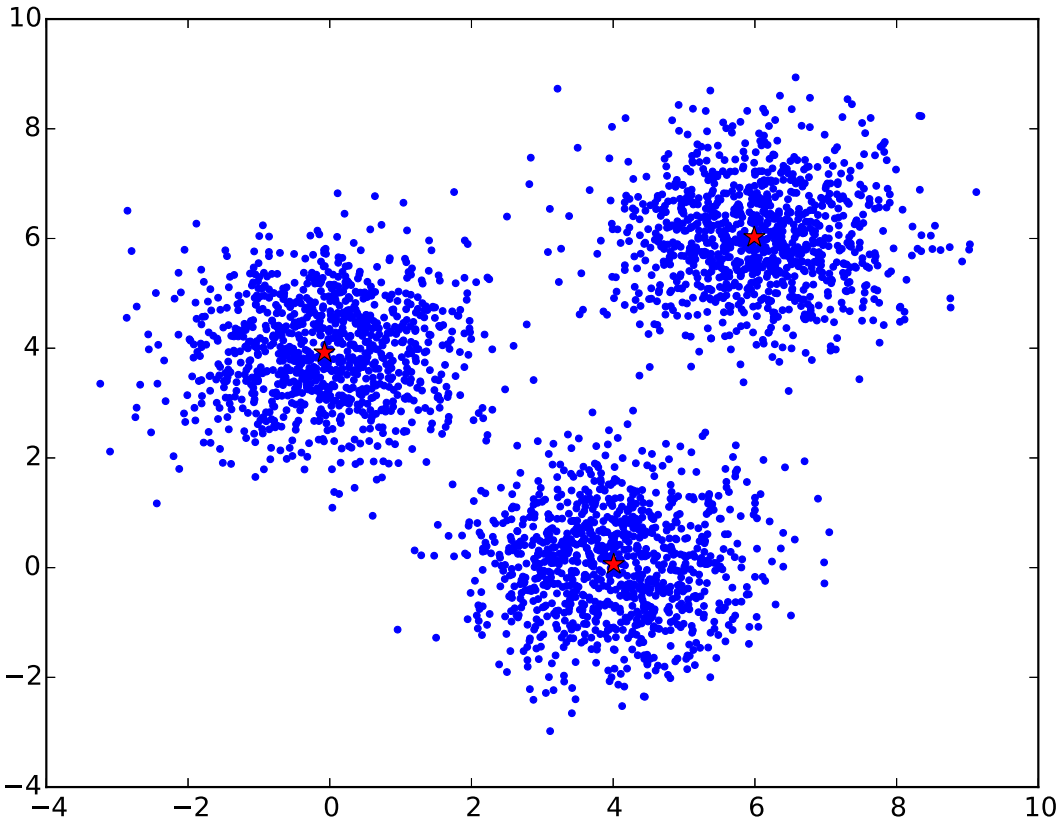
Iteration 1 [[0.59084296 1.9692811] [5.04466739 2.03444504] [2.99651538 5.65213842]] %html



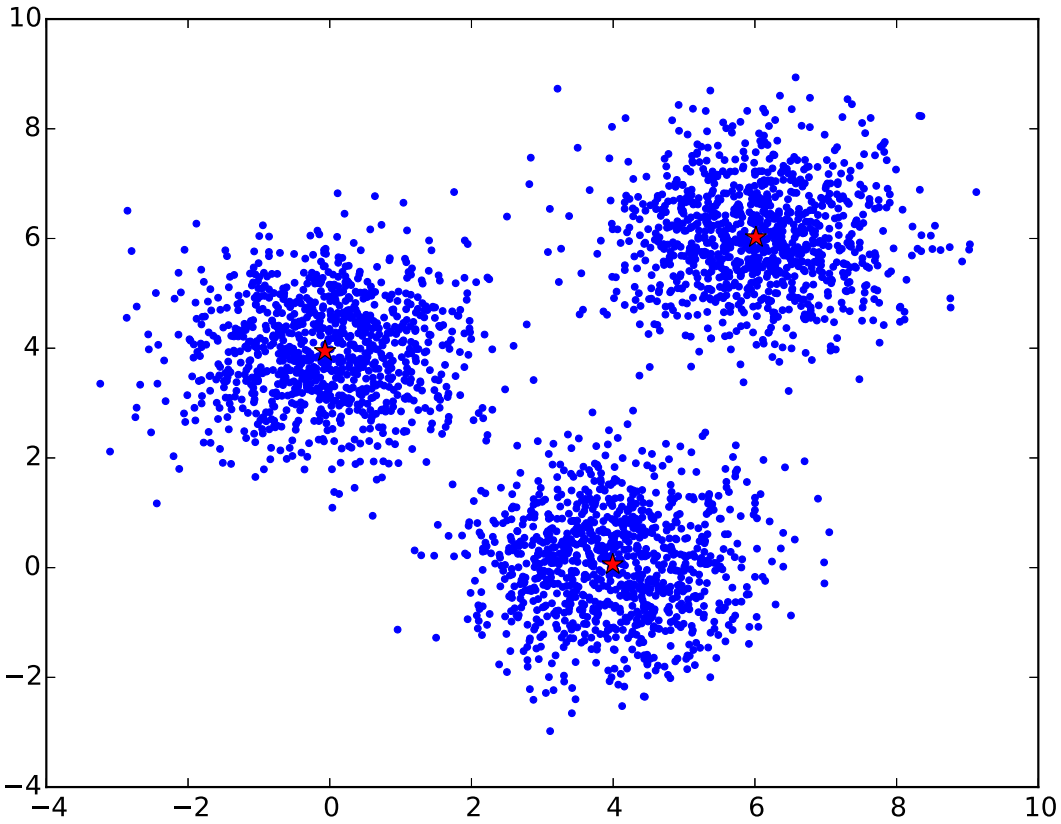
Iteration 2 [[0.10035939 3.18672078] [4.66568528 0.865962] [4.91230796 6.05806013]] %html



Iteration 3 [[-0.07410467 3.92198004] [4.0076407 0.06297751] [5.9911622 6.02410448]] %html



Iteration 4 [[-0.0668399 3.94498952] [3.99305558 0.05908884] [6.01458012 6.02111018]] %html



Final Results: [[-0.0668399 3.94498952] [3.99305558 0.05908884] [6.01458012 6.02111018]]

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).

Answer:

- For binary classification, linear SVM does **NOT** yield the same result as a L2 penalized logistic regression
- SVM aims to find a hyperplane with optimal geometric margin:

$$\begin{aligned} &\max_{\gamma,w,b} \gamma \\ s.t. \quad &y^{(i)}(w^T x^{(i)} + b) \geq \gamma, \quad i = 1, \dots, m, \\ &\|w\| = 1 \end{aligned}$$

- where

$$\gamma = \min_{i=1,\dots,m} \gamma^{(i)}$$

,

- and $\gamma^{(i)}$ is the geometric margin among training data:

$$\gamma^{(i)} = \frac{w^T x^{(i)} + b}{\|w\|} = \left(\frac{w}{\|w\|}\right)^T x^{(i)} + \frac{b}{\|w\|} = y^{(i)} \left(\left(\frac{w}{\|w\|}\right)^T x^{(i)} + \frac{b}{\|w\|} \right)$$

;

- while logistic regression uses a probabilistic interpretation for odds ratio, with $y \in \{0, 1\}$:

$$p(y \mid x; \theta) = (h_\theta(x))^y (1 - h_\theta(x))^{1-y}$$

- where

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

- the weights w are usually obtained with maximum likelihood estimation (MLE), but a L2 penalized loss function with $y \in \{-1, 1\}$, can also be defined as

$$J(w) = \frac{1}{m} \sum_{i=1}^{m} \log(1 + e^{-y^{(i)}(w^T x^{(i)} - b)}) + \lambda \mid w|$$

.

- Similarly, linear SVM does **NOT** yield the same result as a perceptron algorithm either,
- because for perceptron it finds a heuristic hyperplane for a linear separatable dataset without considering margin between clusters:

$$g(z) = g(\theta^T x) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

- and a loss function can be defined as:

$$J(w) = \sum_{i=1}^{m} (g(w^T x_i + b) - y_i)^2$$

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)

Answer:

- The loss term of logistic regression is defined as:

$$J(w) = \frac{1}{m}\sum_{i=1}^{m} \log(1 + e^{-y^{(i)}(w^T x^{(i)} - b)})$$

.

>The03cli>L1 regularization: $||w||_1$

- L2 regularization: $\frac{1}{2}||w||_2^2$
- elastic net regularization: $\alpha||w||_1 + (1-\alpha)\frac{1}{2}||w||_2^2$
- L2-regularized problems are generally easier to solve than L1-regularized due to smoothness. However, L1 regularization can help promote sparsity in weights leading to smaller and more interpretable models, the latter of which can be useful for feature selection. Elastic net is a combination of L1 and L2 regularization. It is not recommended to train models without any regularization, especially when the number of training examples is small. MLlib link (http://spark.apache.org/docs/latest/mllib-linear-methods.html#regularizers)
- The use of a regularization condition is equivalent to doing maximum a posteriori (MAP) estimation, an extension of maximum likelihood.
- Regularization is most commonly done using a squared regularizing function, which is equivalent to placing a zero-mean Gaussian prior distribution on the coefficients.
- For a 2D example, the Gaussian prior is $p(w_1, w_2) = \frac{1}{2\pi\sigma^2}\exp\left(-\frac{w_1^2+w_2^2}{2\sigma^2}\right)$, given that $w_1^2 + w_2^2 = ||w||$, we can see that $L2$ regularization is actually maximum a posteriori estimation with Gaussian prior.
- Similarly, $L1$ regularization is equivalent to Laplacian prior.

HW11.3 Logistic Regression

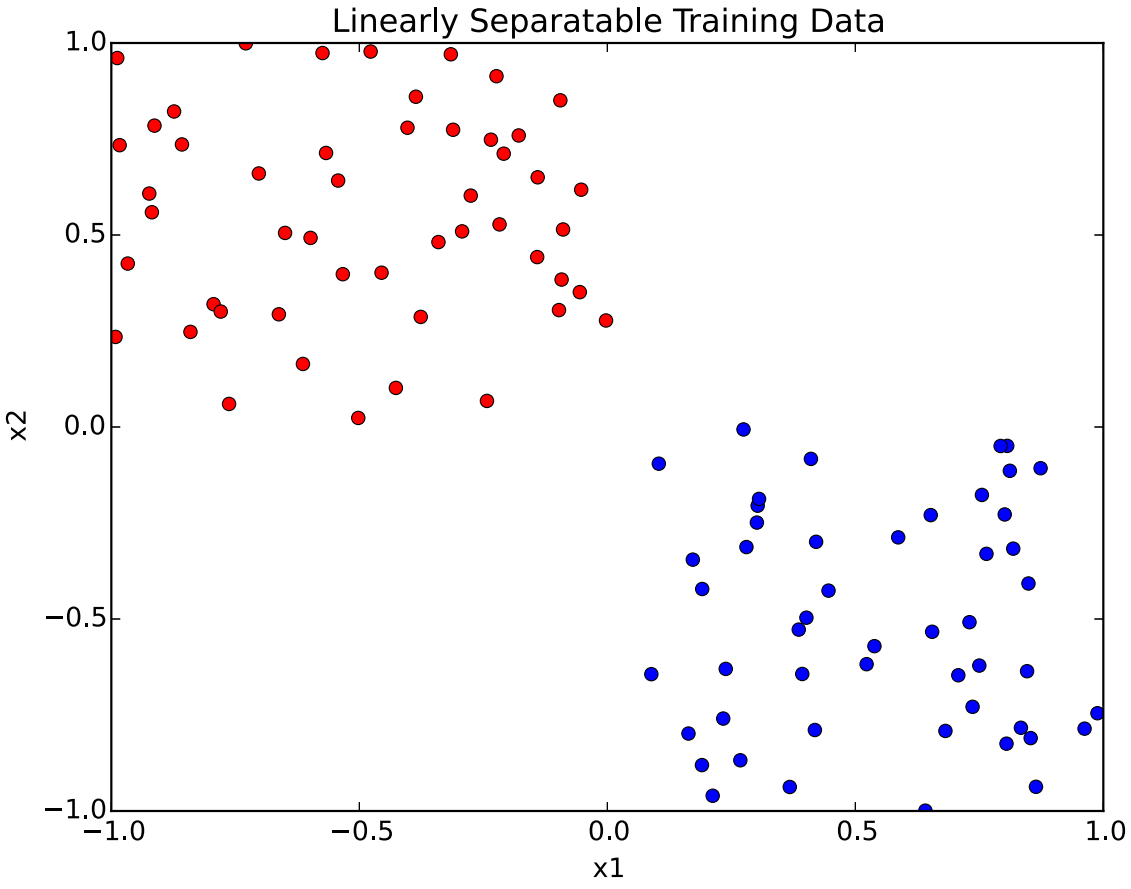
- Generate 2 sets of linearly separable data with 100 data points each using the data generation code provided below and plot each in separate plots. Call one the training set and the other the testing set.

```
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
```

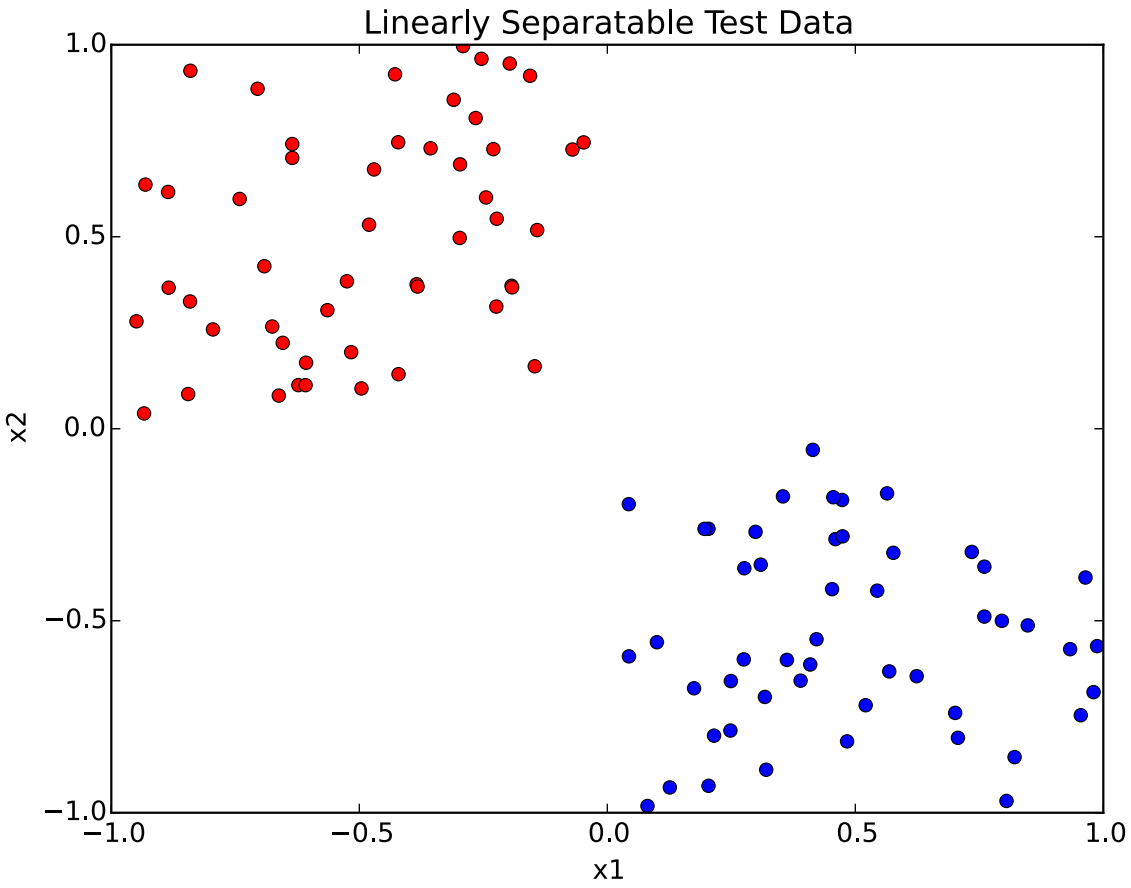
```
%pyspark

import matplotlib.pyplot as plt
from pylab import rand

def generateData(n, separable=True):
    """
    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])
    inputs = np.array(inputs)
    # swap the label randomly, without considering margin - not the best thing to do
    if not separable:
        wrongidx = np.random.choice(range(2*n), np.round(0.1*2*n), replace =False )
        inputs[wrongidx,-1] = -inputs[wrongidx,-1]
    return inputs

def plot_data(data, y, title, w=None, show=True):
```

```
    plt.clf()
    #cols = {1: 'r', 0: 'b'}
    cols = {1: 'r', -1: 'b'}
    for i in range(len(y)):
        plt.plot(data[i,0], data[i,1], cols[y[i]]+'o')
    plt.xlabel("x1")
    plt.ylabel("x2")
    plt.title(title)
    if w is not None:
        x_b = [-1,1]
        y_b = [-(i*w[0]+w[2])/w[1]for i in x_b]
        plt.plot(x_b,y_b)
    plt.grid()
    if show:
        showMPL(plt)

def plot_model(w, spec, style):
    x_b=[-1,1]
    y_b = [-(i*w[0]+w[2])/w[1]for i in x_b]
    plt.plot(x_b, y_b, style, label=spec)
```

```
%pyspark

trainingData = generateData(50)
plot_data(trainingData[:,:-1], trainingData[:,-1], 'Linearly Separatable Training Data')
testData = generateData(50)
plot_data(testData[:,:-1], testData[:,-1], 'Linearly Separatable Test Data')
```

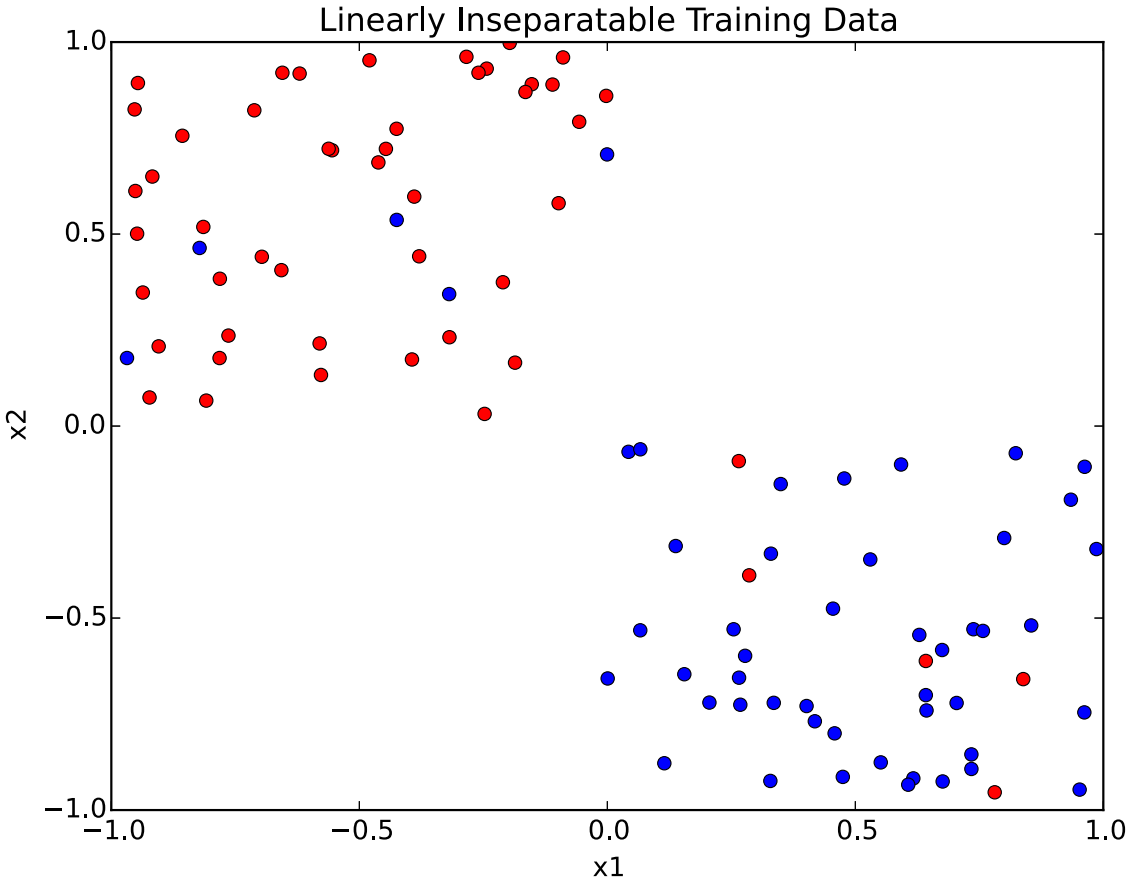


%html

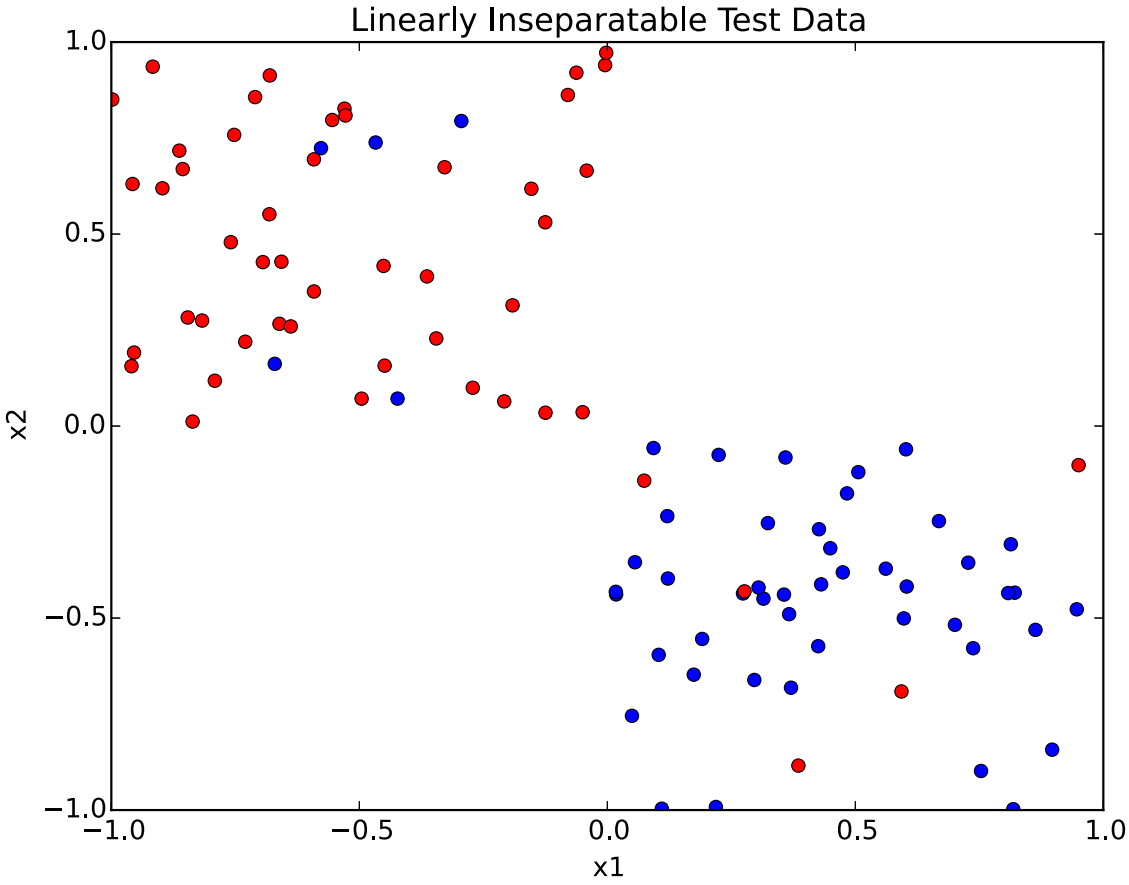


- Modify this data generation code to generating non-linearly separable training and testing datasets
- with approximately 10% of the data falling on the wrong side of the separating hyperplane. Plot the resulting datasets.
- **NOTE**: For the remainder of this problem please use the non-linearly separable training and testing datasets.

```
%pyspark

trainingData = generateData(50, False)
plot_data(trainingData[:,:-1], trainingData[:,-1], 'Linearly Inseparatable Training Data')
testData = generateData(50, False)
plot_data(testData[:,:-1], testData[:,-1], 'Linearly Inseparatable Test Data')
```


Linearly Inseparatable Training Data
1.0
0.5
0.0
−0.5
−1.0
x2
−1.0
−0.5
0.0
0.5
1.0
x1

%html

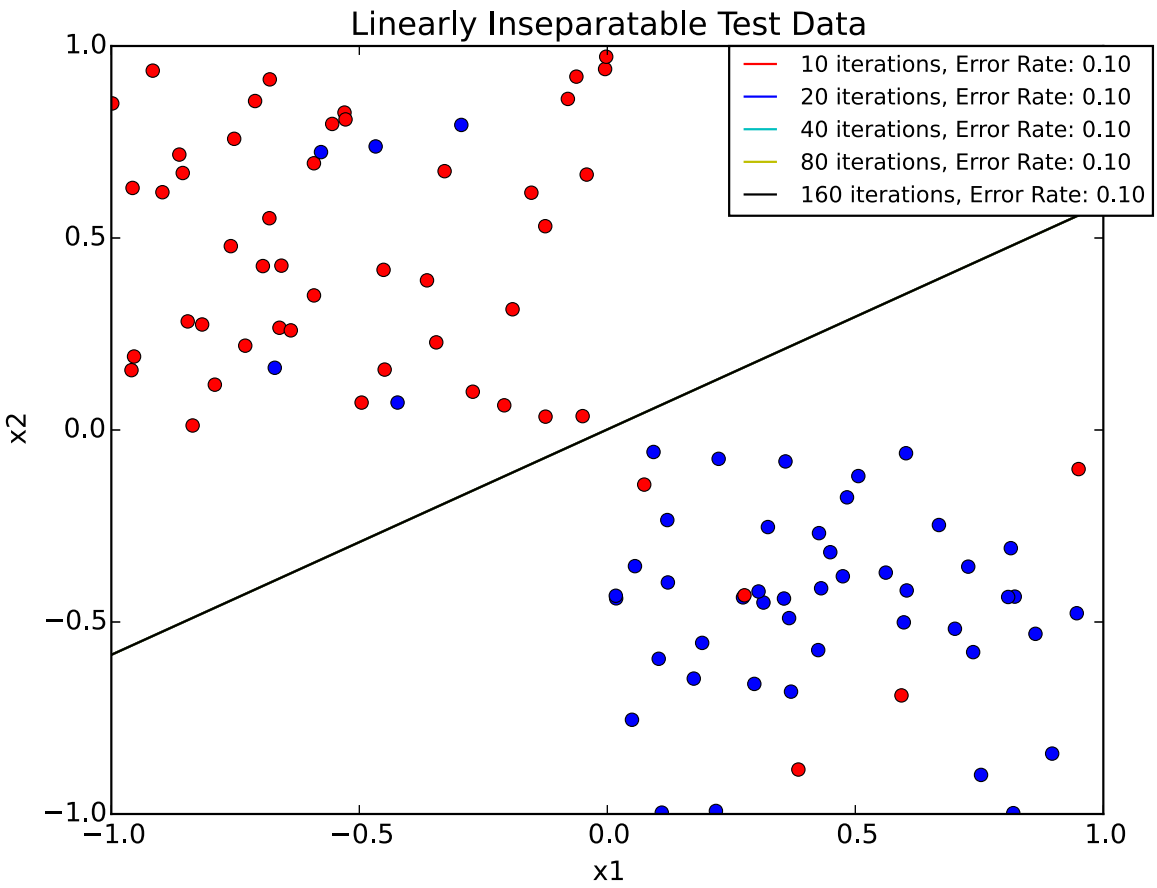


- Using MLLib train up a LASSO logistic regression model with the training dataset and evaluate with the testing set.
- What a good number of iterations for training the logistic regression model? Justify with plots and words.

```
%pyspark

from pyspark.mllib.classification import LogisticRegressionWithLBFGS
from pyspark.mllib.regression import LabeledPoint

# prepare data
def labelPoint(d): return LabeledPoint(d[-1] if d[-1]==1.0 else 0, d[:-1])
data = sc.parallelize(trainingData).map(labelPoint)
test = sc.parallelize(testData).map(labelPoint)

# plot data
plot_data(testData[:,:-1], testData[:,-1], 'Linearly Inseparatable Test Data', None, False)

# train the model with training data
for i, style in zip([10,20,40,80,160], ['r','b','c','y','k']):
    lrm = LogisticRegressionWithLBFGS.train(data, iterations=i, regType='l1', intercept=True
    # predict for testing data
```

```
    labelsAndPreds = test.map(lambda p: (p.label, lrm.predict(p.features)))
    testErr = labelsAndPreds.filter(lambda (v, p): v != p).count() / float(test.count())
    # plot model
    print '%d iterations: weights %s, intercept %.4f, error rate: %.2f' %(i, str(lrm.weights
    plot_model(np.append(lrm.weights, lrm.intercept), '%d iterations, Error Rate: %.2f' %(i,

# show chart
plt.legend(bbox_to_anchor=(1.05, 1), loc=1, fontsize=10,borderaxespad=0.)
```

```
10 iterations: weights [-1.3795346533,2.35057146557], intercept -0.0033, error rate: 0.10
20 iterations: weights [-1.3795346533,2.35057146557], intercept -0.0033, error rate: 0.10
40 iterations: weights [-1.3795346533,2.35057146557], intercept -0.0033, error rate: 0.10
80 iterations: weights [-1.3795346533,2.35057146557], intercept -0.0033, error rate: 0.10
160 iterations: weights [-1.3795346533,2.35057146557], intercept -0.0033, error rate: 0.10
```




```
%md
###Comments
- we can see because the data is linearly inseparable, increasing iteration doesn't change t
- in this case even 10 iteration can yield decent performance
```

- 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):
  - $weight(X) = \frac{1}{\|X\|}$,
  - where $\|X\| = \sqrt{X^T X} = \sqrt{X_1^2 + X_2^2}$
  - Here X is vector made up of X_1 and X_2.
- 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.

Derivation

- loss function:

$$J(w) = \frac{1}{m}\sum_{i=1}^{m} \log\left(1 + \exp(-y^{(i)}\frac{w^T x^{(i)} - b}{\|x^{(i)}\|})\right) + \lambda \mid w|$$

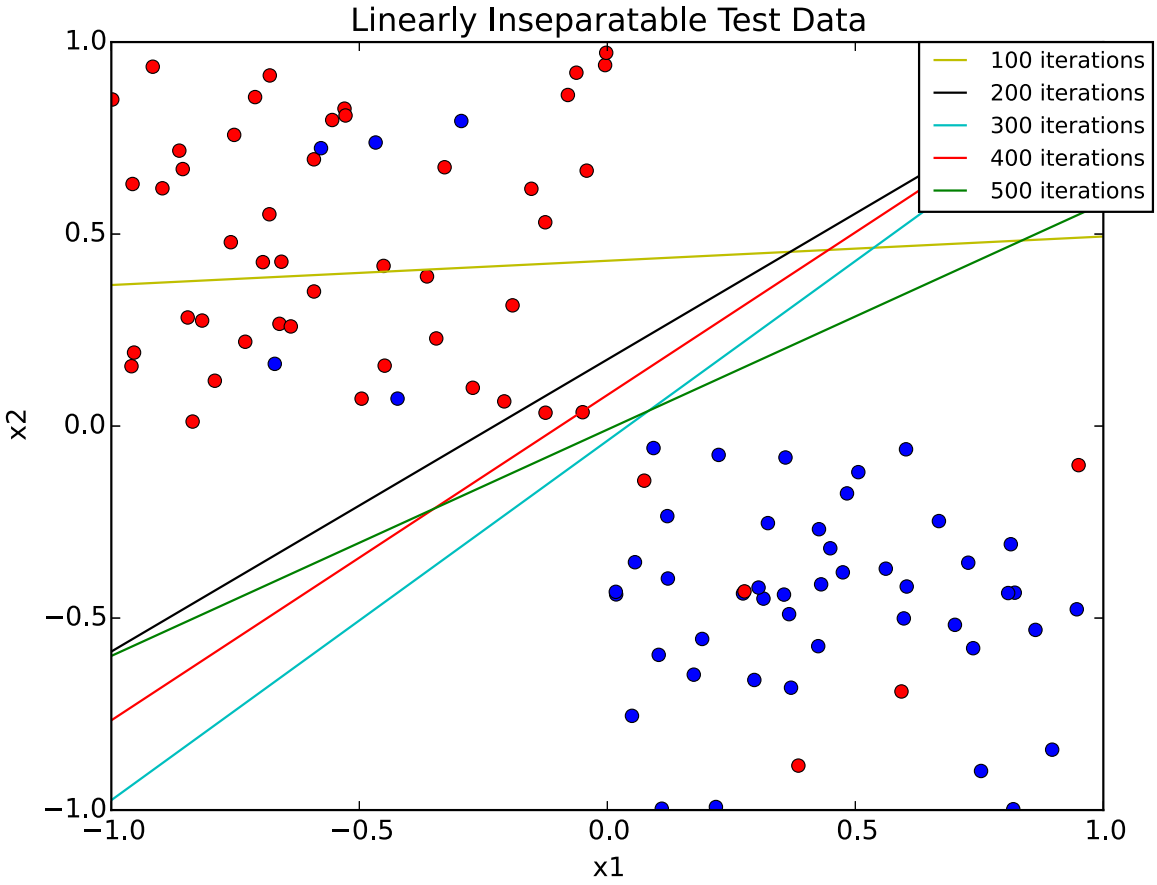
- gradient for one component in the weight vector w_j:

$$\frac{\partial J}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m} -y^{(i)}\left(1 - \frac{1}{1 + \exp\left(-y^{(i)}\frac{w^T x^{(i)} - b}{\|x^{(i)}\|}\right)}\right)\frac{x_j^{(i)}}{\|x^{(i)}\|} + sign(w_j)$$

```
%pyspark

# define functions
def grad(p):
    x, y = p[:-1], p[-1]
    norm = np.sqrt(sum(x**2))
    return -y*(1-1/(1+np.exp(-y*np.dot(x,w)/norm)))*x/norm

def weight_logistic_regression_bgd(eta=0.05,regPara=0.01):
    g = data.map(grad).reduce(lambda a,b: a+b)/m
    wreg = np.sign(w)
    wreg[0] = 0
    return eta*(g+regPara*wreg)
```

```
# add a column of 1 at the front for the intercept
training = np.append(np.ones((trainingData.shape[0],1)), trainingData, axis=1)
test = np.append(np.ones((testData.shape[0],1)), testData, axis=1)

# prepare the data
data = sc.parallelize(training)
test = sc.parallelize(test)

# plot data
plot_data(testData[:,:-1], testData[:,-1], 'Linearly Inseparatable Test Data', None, False)

# define training parameters
d = len(data.first())-1
m = data.count()
n_iter = [100,200,300,400,500]
style = ['y','k','c','r','g']
stopCriteria=0.0001

# simulation
for n, s in zip(n_iter, style):
    w = np.random.normal(size=d)
    n_break = n
    for i in range(n):
        delta = weight_logistic_regression_bgd()
        if sum(abs(delta)) <= stopCriteria*sum(abs(w)):
            print 'break after %d iterations' %(i+1)
            n_break = i+1
            break
        w = w - delta

    # get test error
    testError = test.map(lambda d:(-1 if np.dot(d[:-1], w)<0 else 1) - d[-1]).filter(lambda

    # plot model
    print 'Weigths after %d iterations: %s, test error: %.2f' %(n_break, str(w), testError)
    plot_model(np.append(w[1:],w[0]), '%d iterations' %n_break, s)
```

```
Weigths after 100 iterations: [-0.43810323 -0.06430721  1.01770857], test error: 0.27
Weigths after 200 iterations: [-0.25641846 -1.12548573  1.47996598], test error: 0.12
Weigths after 300 iterations: [ 0.05058665 -1.23700339  1.3216664 ], test error: 0.10
Weigths after 400 iterations: [-0.14973548 -1.56940546  1.85263177], test error: 0.11
Weigths after 500 iterations: [ 0.01997063 -1.27088175  2.15638914], test error: 0.10
```




```
%md

###Comments:
- after 300 iteration, it's overfitting the bad data, test error increases
- MLlib doesn't have an option for weighted logistic regression
```

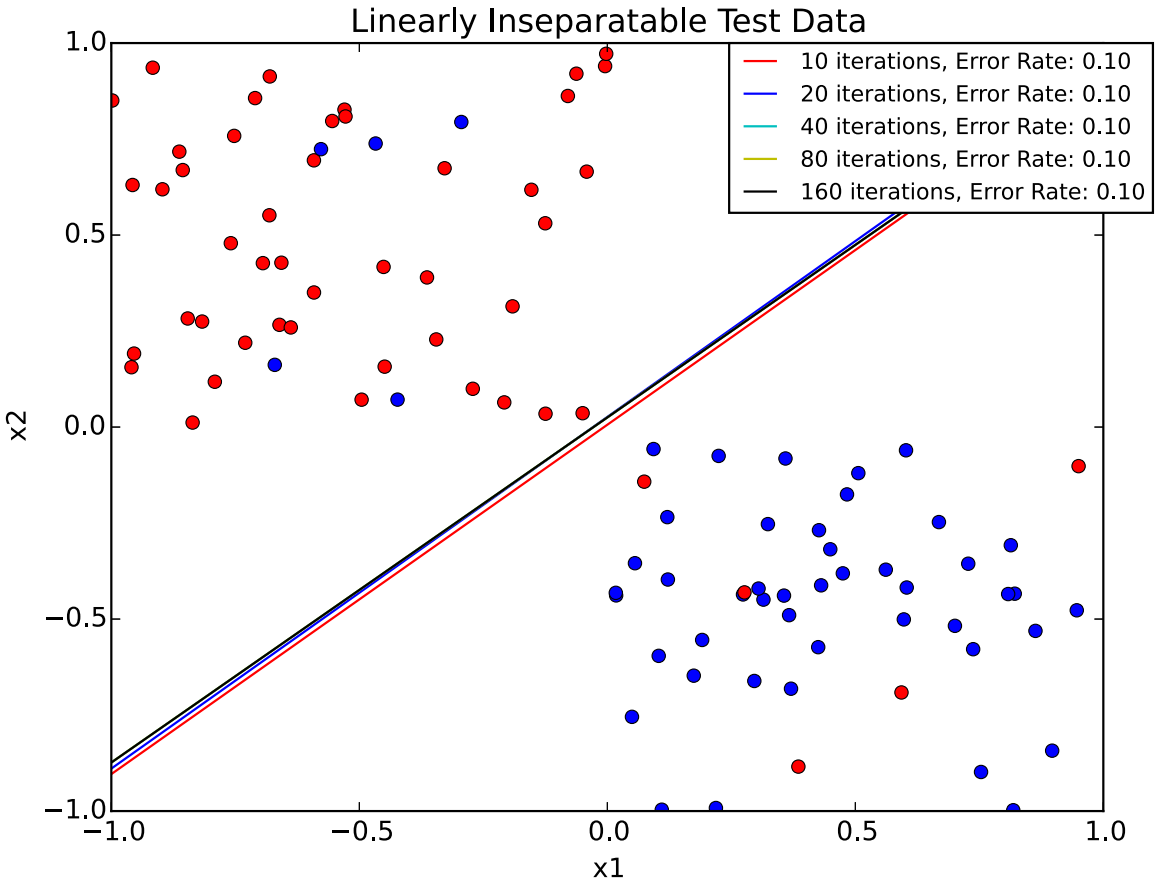
HW11.4 SVMs

- 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.

```
%pyspark

from pyspark.mllib.classification import SVMWithSGD, SVMModel
```

```
from pyspark.mllib.regression import LabeledPoint

# prepare data
def labelPoint(d): return LabeledPoint(1 if d[-1]==1.0 else 0, d[:-1])
data = sc.parallelize(trainingData).map(labelPoint)
test = sc.parallelize(testData).map(labelPoint)

# plot data
plot_data(testData[:,:-1], testData[:,-1], 'Linearly Inseparatable Test Data', None, False)

# train the model with training data
for i, style in zip([10,20,40,80,160], ['r','b','c','y','k']):
    svm = SVMWithSGD.train(data, iterations=i, regType='l2', intercept=True)
    # predict testing data
    labelsAndPreds = test.map(lambda p: (p.label, svm.predict(p.features)))
    testErr = labelsAndPreds.filter(lambda (v, p): v != p).count() / float(test.count())
    # plot model
    print '%d iterations: weights %s, intercept %.4f, error rate: %.2f' %(i, str(svm.weights
    plot_model(np.append(svm.weights, svm.intercept), '%d iterations, Error Rate: %.2f' %(i,

# show chart
```

```
10 iterations: weights [-0.972714806149,1.06918066529], intercept -0.0065, error rate: 0.10
20 iterations: weights [-1.05445506935,1.15142147838], intercept -0.0304, error rate: 0.10
40 iterations: weights [-1.07619552578,1.19762928331], intercept -0.0300, error rate: 0.10
80 iterations: weights [-1.07619552578,1.19762928331], intercept -0.0300, error rate: 0.10
160 iterations: weights [-1.07619552578,1.19762928331], intercept -0.0300, error rate: 0.10
```



Comments:

- those wrong red points are further way from the wrong blue points, thus the hyperplane is pulled toward the blue cluster a bit
- with stochastic gradient descent, convergence is fast, thus increasing iteration doesn't improve model performance much

- 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.

Derivation:

- Objective function:

$$\min_w \frac{\lambda}{2} w^T w + \frac{1}{m} \sum_i \frac{1}{\|x_i\|} (1 - y_i (w^T x_i - b))$$

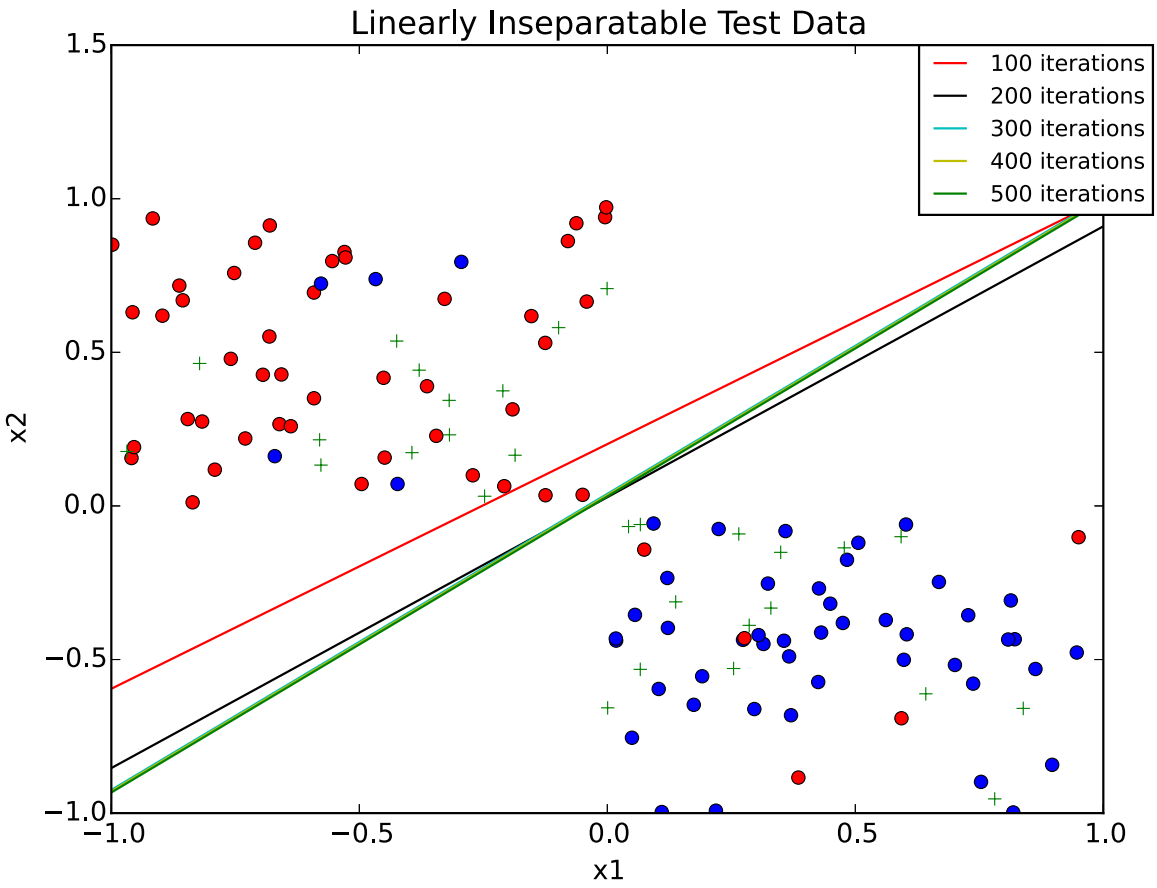
- Gradient from the support vectors:

$$\lambda w + y_i \frac{x_i}{\|x_i\|}, \quad \text{if } y_i (w^T x_i - b)) < 1$$

```
%pyspark

# define functions
def grad(p):
    x, y = p[:-1], p[-1]
    return -y*x/np.sqrt(sum(x**2))

def weight_svm_bgd(sv, eta=0.05, regPara=0.01):
    g = sv.map(grad).reduce(lambda a,b: a+b)/m
    wreg = np.sign(w)
    wreg[0] = 0
    return eta*(g+regPara*wreg)

# add a column of 1 at the front for the intercept
```

```
training = np.append(np.ones((trainingData.shape[0],1)), trainingData, axis=1)
test = np.append(np.ones((testData.shape[0],1)), testData, axis=1)

# prepare the data
data = sc.parallelize(training)
test = sc.parallelize(test)

# plot data
plot_data(testData[:,:-1], testData[:,-1], 'Linearly Inseparatable Test Data', None, False)

# define training parameters
d = len(data.first())-1
m = data.count()
n_iter = [100,200,300,400,500]
style = ['r','k','c','y','g']
stopCriteria=0.001
w_init = np.random.normal(size=d)

# simulation
for n, s in zip(n_iter, style):
    w = w_init
    n_break = n
    for i in range(n):
        # get support vectors
        sv = data.filter(lambda p: p[-1]*(np.dot(w, p[:-1]))<0.1)
        delta = weight_svm_bgd(sv)
        # converge critirion
        if sum(abs(delta)) <= stopCriteria*sum(abs(w)):
            print 'break after %d iterations' %(i+1)
            n_break = i+1
            break
        w = w - delta

    # get test error
    testError = test.map(lambda d:(-1 if np.dot(d[:-1], w)<0 else 1) - d[-1]).filter(lambda

    # plot model
    print 'Weigths after %d iterations: %s, test error: %.2f, support vectors: %d' %(n_break
    plot_model(np.append(w[1:],w[0]), '%d iterations' %n_break, s)

plt.legend(bbox_to_anchor=(1.05, 1), loc=1, fontsize=10,borderaxespad=0.)

# plot support vectors - green +
for p in sv.collect():
    plt.plot(p[1], p[2], 'g+')
```

```
Weigths after 100 iterations: [-0.0482253  -0.19083116  0.23962261], test error: 0.12, suppor
t vectors: 18
Weigths after 200 iterations: [-0.00377884 -0.11725688  0.1329851 ], test error: 0.10, suppor
t vectors: 29
Weigths after 300 iterations: [-0.00485102 -0.1188118   0.12327955], test error: 0.10, suppor
t vectors: 30
Weigths after 400 iterations: [-0.00432991 -0.11887991  0.12335346], test error: 0.10, suppor
t vectors: 30
Weigths after 500 iterations: [-0.0038088  -0.11894801  0.12342736], test error: 0.10, suppor
t vectors: 29
```




```
%md

###Comments:
- all the noisy data point will give bad influence on the model, thus the number of support
- MLlib doesn't have option for weighted soft SVM
```

HW11.5 [OPTIONAL]: Distributed Perceptron algorithm.

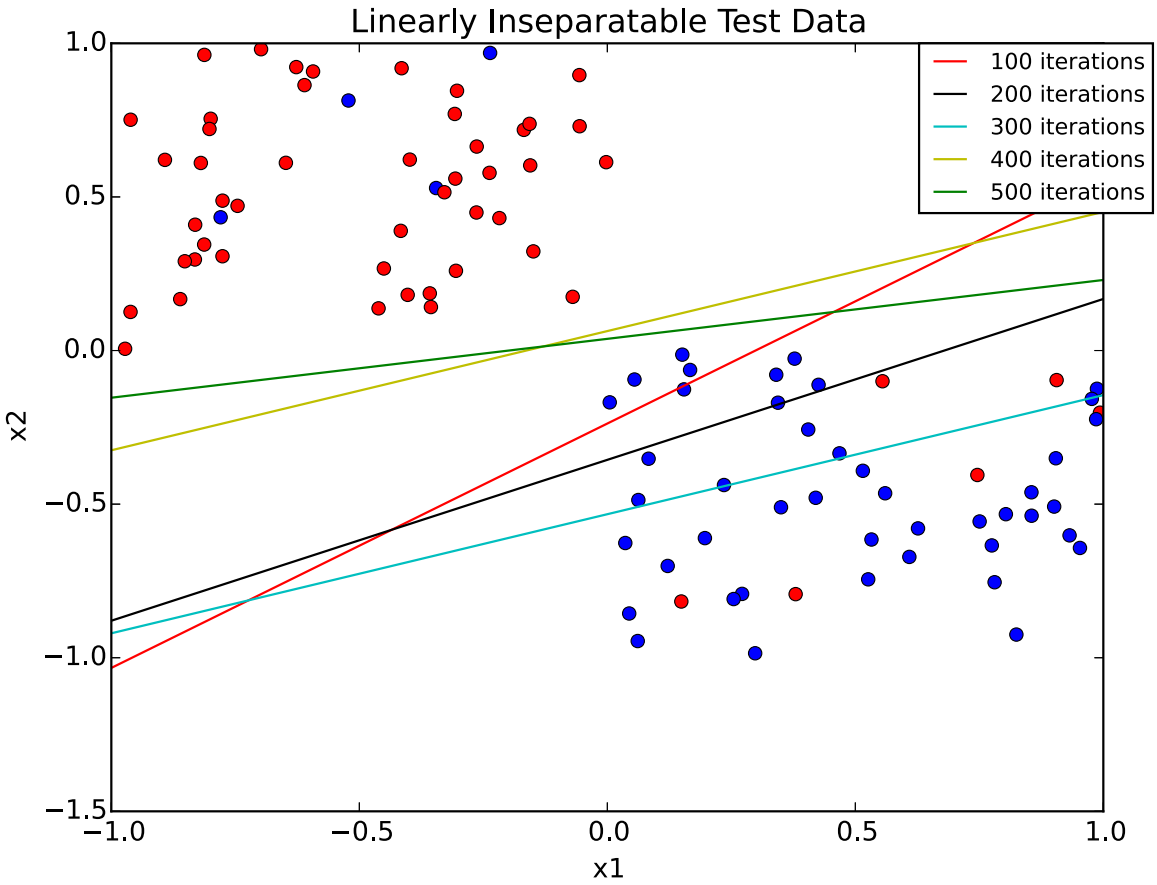
- Using the following papers as background:
  - http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/
  - https://www.dropbox.com/s/a5pdcp0r8ptudgj/gesmundo-tomeh-eacl-2012.pdf?dl=0
  - http://www.slideshare.net/matsubaray/distributed-perceptron
- Implement each of the following flavors of perceptron learning algorithm:
  1. Serial (All Data): This is the classifier returned if trained serially on all the available data. On a single computer for example (Mistake driven)
  2. Serial (Sub Sampling): Shard the data, select one shard randomly and train serially.
  3. Parallel (Parameter Mix): Learn a perceptron locally on each shard: Once learning is complete combine each learnt perceptron using a uniform weighting
  4. Parallel (Iterative Parameter Mix) as described in the above papers.

1. Serial (All Data): This is the classifier returned if trained serially on all the available data. On a single computer for example (Mistake driven)

```
%pyspark

# add a column of 1 at the front for the intercept
training = np.append(np.ones((trainingData.shape[0],1)), trainingData, axis=1)
test = np.append(np.ones((testData.shape[0],1)), testData, axis=1)

test = sc.parallelize(test)

# plot data
plot_data(testData[:,:-1], testData[:,-1], 'Linearly Inseparatable Test Data', None, False)

# define training parameters
d = 3
m = 100
n_iter = [100,200,300,400,500]
style = ['r','k','c','y','g']
stopCriteria=0.001
eta = 0.05

# Perceptron training
for n, s in zip(n_iter, style):
    n_break = n
    w = np.random.normal(size=d) #np.zeros(d)
    # repeat n epochs
    for i in range(n):
        k = 0
        for p in training:
            x, y = p[:-1], p[-1]
            if np.dot(w, x)*y<=0:
                w += eta*y*x
                k += 1
        if k == 0:
            break
            n_break = i


    # get test error
    testError = test.map(lambda d:(-1 if np.dot(d[:-1], w)<0 else 1) - d[-1]).filter(lambda

    # plot model
    print 'Weigths after %d iterations: %s, test error: %.2f' %(n_break, str(w), testError)
    plot_model(np.append(w[1:],w[0]), '%d iterations' %n_break, s)

plt.legend(bbox_to_anchor=(1.05, 1), loc=1, fontsize=10,borderaxespad=0.)
```

```
Weigths after 100 iterations: [ 0.01626408 -0.05444038  0.06843425], test error: 0.14
Weigths after 200 iterations: [ 0.02186778 -0.03218456  0.06145578], test error: 0.19
Weigths after 300 iterations: [ 0.03985791 -0.02900019  0.0748093 ], test error: 0.23
```

```
Weigths after 400 iterations: [-0.00630826 -0.0386543   0.09961648], test error: 0.10
Weigths after 500 iterations: [-0.00175851 -0.00886026  0.04623739], test error: 0.10
```



Comments:

- training process fluctuates a lot, as the data is linearly inseparable, there always be wrong points to add bad influcence to the model

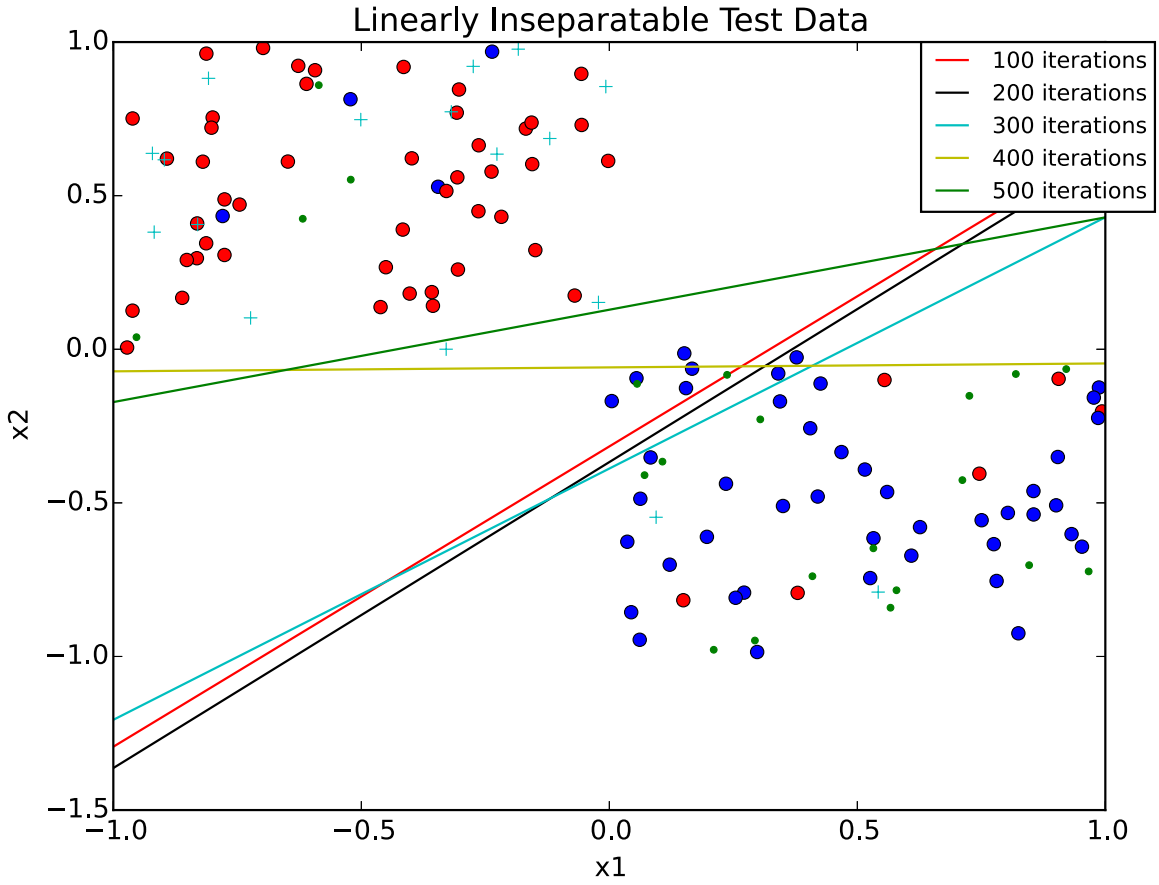
2. Serial (Sub Sampling): Shard the data, select one shard randomly and train serially.

```
%pyspark

# add a column of 1 at the front for the intercept
training = np.append(np.ones((trainingData.shape[0],1)), trainingData, axis=1)
test = np.append(np.ones((testData.shape[0],1)), testData, axis=1)

# prepare data
test = sc.parallelize(test).cache()
r1, r2, r3 = sc.parallelize(training).randomSplit([1,1,1])
trainData = r2.collect()
```

```
# plot data
plot_data(testData[:,:-1], testData[:,-1], 'Linearly Inseparatable Test Data', None, False)

# define training parameters
d = 3
m = 100
n_iter = [100,200,300,400,500]
style = ['r','k','c','y','g']
stopCriteria=0.001
eta = 0.05

# Perceptron training
for n, s in zip(n_iter, style):
    n_break = n
    w = np.random.normal(size=d) #np.zeros(d)
    # repeat n epochs
    for i in range(n):
        k = 0
        for p in trainData:
            x, y = p[:-1], p[-1]
            if np.dot(w, x)*y<=0:
                w += eta*y*x
                k += 1
        if k == 0:
            break
            n_break = i


    # get test error
    testError = test.map(lambda d:(-1 if np.dot(d[:-1], w)<0 else 1) - d[-1]).filter(lambda

    # plot model
    print 'Weigths after %d iterations: %s, test error: %.2f' %(n_break, str(w), testError)
    plot_model(np.append(w[1:],w[0]), '%d iterations' %n_break, s)

plt.legend(bbox_to_anchor=(1.05, 1), loc=1, fontsize=10,borderaxespad=0.)

# plot training datas - green . or +
for p in trainData:
    plt.plot(p[1], p[2], 'g.' if p[-1]==-1 else 'c+')
```

```
Weigths after 100 iterations: [ 0.01151002 -0.03557326  0.03639122], test error: 0.15
Weigths after 200 iterations: [ 0.01222528 -0.03315505  0.03328887], test error: 0.15
Weigths after 300 iterations: [ 0.02103369 -0.04430109  0.05416563], test error: 0.17
Weigths after 400 iterations: [ 0.00392051 -0.00085143  0.06644665], test error: 0.12
Weigths after 500 iterations: [-0.01205304 -0.02815246  0.09364889], test error: 0.10
```

```
%pyspark

showMPL(plt)
```



Comments:

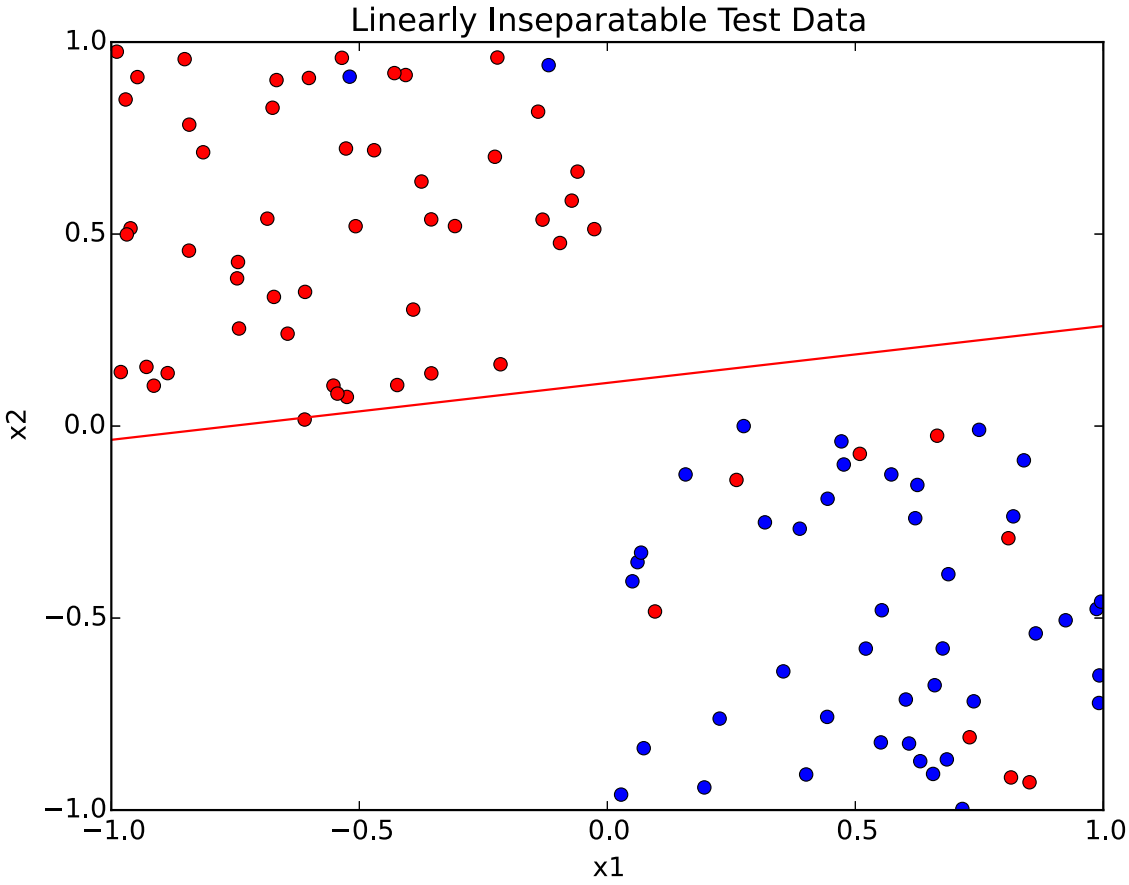
- depends on the partition of the data, model differs dramatically
- again, there will always be noisy data, so the test error can reach zero

3. Parallel (Parameter Mix): Learn a perceptron locally on each shard: Once learning is complete combine each learnt perceptron using a uniform weighting

```
%pyspark

# add a column of 1 at the front for the intercept
training = np.append(np.ones((trainingData.shape[0],1)), trainingData, axis=1)
test = np.append(np.ones((testData.shape[0],1)), testData, axis=1)

# prepare data, shuffle it to 3 partitions
test = sc.parallelize(test).cache()
train = sc.parallelize(training).coalesce(3, shuffle=True).cache()

# plot data
plot_data(testData[:,:-1], testData[:,-1], 'Linearly Inseparatable Test Data', None, False)
```

```
# define training parameters
d = 3
m = 100
eta = 0.05

# Perceptron training function
def localTrain(iterator):
    w = np.random.normal(size=d) #np.zeros(d)
    # repeat n epochs
    for i in range(100):
        k = 0
        for p in iterator:
            x, y = p[:-1], p[-1]
            if np.dot(w, x)*y<=0:
                w += eta*y*x
                k += 1
        if k == 0:
            break
    return [w]

# train perceptron on each partition
weights = train.mapPartitions(localTrain).collect()
w = np.mean(weights, axis=0)

# get test error
testError = test.map(lambda d:(-1 if np.dot(d[:-1], w)<0 else 1) - d[-1]).filter(lambda d: d

# plot model
print 'Weigths: %s, test error: %.2f' %(str(w), testError)
plot_model(np.append(w[1:],w[0]), 'parameter mixing', 'r')
```

```
Weigths: [-0.06131624 -0.08085764  0.54553357], test error: 0.11
```



Comments:

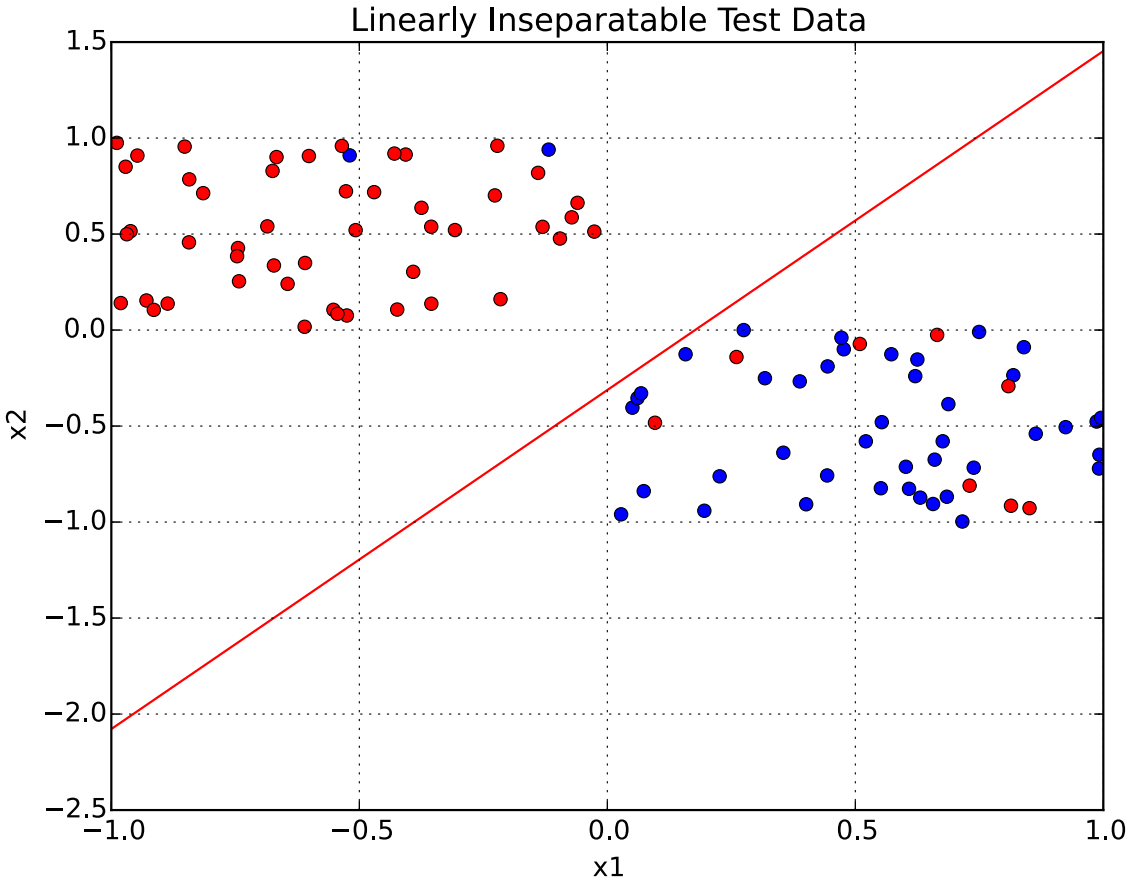
- model depends on the shuffling of sharding, must have balanced training data in order for good results

4. Parallel (Iterative Parameter Mix) as described in the above papers.

```
%pyspark

# add a column of 1 at the front for the intercept
training = np.append(np.ones((trainingData.shape[0],1)), trainingData, axis=1)
test = np.append(np.ones((testData.shape[0],1)), testData, axis=1)

# prepare data, shuffle it to 3 partitions
test = sc.parallelize(test).cache()
train = sc.parallelize(training).coalesce(3, shuffle=True).cache()

# plot data
plot_data(testData[:,:-1], testData[:,-1], 'Linearly Inseparatable Test Data', None, False)

# define training parameters
d = 3
```

```
m = 100
eta = 0.05
w = np.random.normal(size=d) #np.zeros(d)

# Perceptron training function
def localTrain(iterator):
    # really should use broadcast variable, just for simplicity here
    global w
    # repeat n epochs
    for i in range(100):
        k = 0
        for p in iterator:
            x, y = p[:-1], p[-1]
            if np.dot(w, x)*y<=0:
                w += eta*y*x
                k += 1
        if k == 0:
            break
    return [w]

# iterative parameter mixing
for i in range(10):
    weights = train.mapPartitions(localTrain).collect()
    w = np.mean(weights, axis=0)

# get test error
testError = test.map(lambda d:(-1 if np.dot(d[:-1], w)<0 else 1) - d[-1]).filter(lambda d: d

# plot model
print 'Weigths: %s, test error: %.2f' %(str(w), testError)
plot_model(np.append(w[1:],w[0]), 'parameter mixing', 'r')
```

```
Weigths: [ 0.01590157 -0.08990141  0.05091974], test error: 0.10
```



HW11.6 [OPTIONAL: consider doing this in a group] Evalution of perceptron algorihtms on PennTreeBank POS corpus

- Reproduce the experiments reported in the following paper:
  - HadoopPerceptron: a Toolkit for Distributed Perceptron Training and Prediction with MapReduce Andrea Gesmundo and Nadi Tomeh
  - http://www.aclweb.org/anthology/E12-2020
- These experiments focus on the prediction accuracy on a part-of-speech (POS) task using the PennTreeBank corpus. They use sections 0-18 of the Wall Street Journal for training, and sections 22-24 for testing.

HW11.7 [OPTIONAL: consider doing this in a group] Kernal Adatron

- Implement the Kernal Adatron in Spark (contact Jimi for details)