Zeppelin

MIDS-W261-2016-HWK-Week11-DangSeltzer

General Housekeeping

Submission Details

- **Student**: Miki Seltzer, Minhchau Dang
- **Email Address**: miki.seltzer@berkeley.edu, minhchau.dang@berkeley.edu
- **Course**: 2016-0111 DATASCI W261: Machine Learning at Scale
- **Section**: Spring 2016, Section 2
- **Assignment**: Homework 11, Week 11
- **Submission Date**: April 7, 2016

MathJax For LaTeX Notation

```
%angular

<script type="text/javascript" async src="https://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-MML-AM_CHTML"></script>
<script type="text/x-mathjax-config">MathJax.Hub.Config({tex2jax: {inlineMath: [['$','$']]}});</script>
```

Pyspark Plots Without SQL

```
%spark.pyspark

# Code adapted from the following places:
# https://github.com/bernhard-42/Zeppelin-Visualizations/blob/master/Code.md
# http://stackoverflow.com/questions/5314707/matplot-store-image-in-variable

import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import StringIO
import urllib, base64
```

```
def showMPL():
    img = StringIO.StringIO()
    plt.savefig(img, format='png', dpi=72)
    img.seek(0)

    uri = '' + urllib.quote(base64.b64encode(img.buf))
    print "%html <img src='" + uri + "'/>"
    plt.clf()
```

HW11.0 Broadcast Versus Caching In Spark

What Is The Difference Between Broadcasting And Caching Data In Spark?

A broadcast variable (http://spark.apache.org/docs/latest/programming-guide.html#broadcast-variables) ensures that all partitions receive a materialized value in memory without having to bundle this value in a closure, and this value is shared by all partitions on the same node. A cached variable (http://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence) represents elements that have been materialized from an RDD on some partition, but the cache is subject to a least-recently-used (LRU) eviction policy.

Give An Example (In The Context Of Machine Learning) Of Each Mechanism (At A High Level).

One example in the context of machine learning would be the PageRank algorithm. In PageRank, you would want all partitions to recognize the total number of nodes in the graph and any mass associated with dangling nodes so that they can properly apply the teleportation factor. In this case, you would want to broadcast the node count and the dangling node mass to all partitions. At the same time, you would want to avoid re-materializing the web graph in every iteration. Since the entire web graph would not fit in memory, you would instead opt for each partition remembering as much as possible (usually the specific values it materialized), and so you would cache the data.

Review The Following Spark-Notebook-Based Implementation Of K-Means. (Part 1)

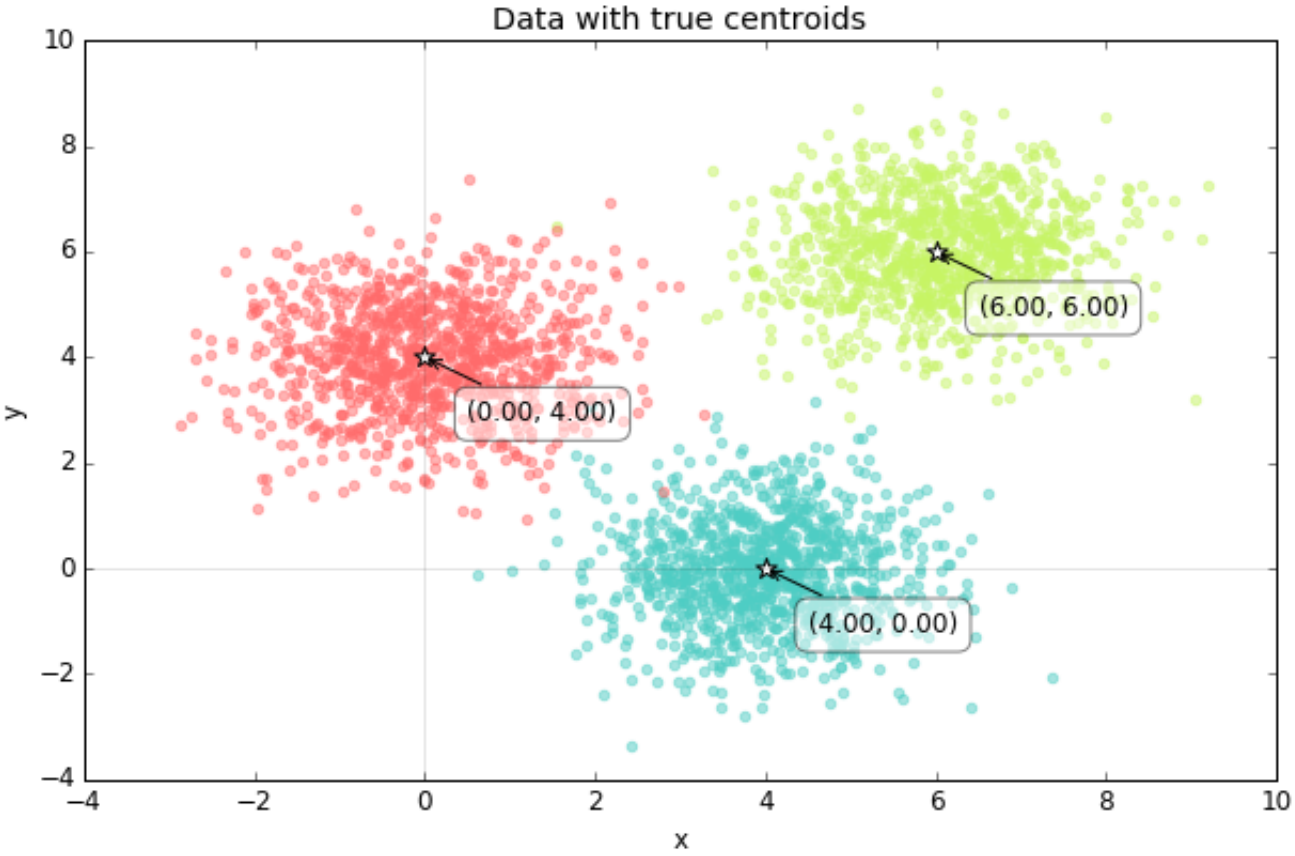
```
%spark.pyspark

# Start with the sample code
# http://nbviewer.jupyter.org/urls/dl.dropbox.com/s/41q9lgyqhy8ed5g/EM-Kmeans.ipynb

# Create the randomized data.

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),]
```

Perform Data Exploration Of The Generated K-Means Data.

```
%spark.pyspark

def plot_samples(title):
    plt.figure(figsize=(10,10/3*2))

    # Plot axis
    plt.axhline(0, color='black', alpha=0.1)
    plt.axvline(0, color='black', alpha=0.1)

    # Plot data points
    plt.scatter(samples1[:, 0], samples1[:, 1], color='#4ECDC4', alpha=0.5)
    plt.scatter(samples2[:, 0], samples2[:, 1], color='#C7F464', alpha=0.5)
    plt.scatter(samples3[:, 0], samples3[:, 1], color='#FF6B6B', alpha=0.5)

    # Label graph
    plt.title(title)
    plt.xlabel('x')
    plt.ylabel('y')

def plot_iteration(before, after):
    centroids_path = np.array([before, after])

    # Plot the transition from one centroid to another as a path
    for i in range(3):
        plt.plot(
            centroids_path[:,i,0], centroids_path[:,i,1], color='grey', alpha=0.9, marker='o', markersize=5,
            markerfacecolor='white', markeredgecolor='black', zorder=2)

def plot_centroids(means):
    # Plot centroids
    plt.scatter(means[0][0], means[0][1], marker='*', s=100, color='white', edgecolor='black')
    plt.scatter(means[1][0], means[1][1], marker='*', s=100, color='white', edgecolor='black')
    plt.scatter(means[2][0], means[2][1], marker='*', s=100, color='white', edgecolor='black')

    # Annotate centroids
    for i in means:
        text = '({:.2f}'.format(i[0]) + ', ' + '{:.2f})'.format(i[1])
        plt.annotate(text,
            xy = i, xytext = (20, -20),
            textcoords = 'offset points', ha = 'left', va = 'top',
            bbox = dict(boxstyle = 'round,pad=0.5', fc = 'white', alpha = 0.5),
            arrowprops = dict(arrowstyle = '->', connectionstyle = 'arc3,rad=0'))
```

```
fig = plt.gcf()
fig.set_size_inches(8, 6)

plot_samples('Data with true centroids')
plot_centroids([[0,4],[6,6],[4,0]])
showMPL()
```



Make The Created K-Means Data Available In HDFS.

```
%sh

hdfs dfs -rm -f -skipTrash data.csv > /dev/null
hdfs dfs -copyFromLocal data.csv
```

Review The Following Spark-Notebook-Based Implementation Of K-Means. (Part 2)

```
%spark.pyspark
```

```
# Calculate which class each data point belongs to

def nearest_centroid(line):
    x = np.array([float(f) for f in line.split(',')])
    closest_centroid_idx = np.sum((x - centroids)**2, axis=1).argmin()
    return (closest_centroid_idx,(x,1))

# Distributed KMeans in Spark

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()

fig = plt.gcf()
fig.set_size_inches(8, 6)

plot_samples('Centroid updates using closures')

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
    plot_iteration(centroids, centroids_new)
    if np.sum(np.absolute(centroids_new-centroids))<0.01:
        break
    centroids = centroids_new

plot_centroids(centroids)
showMPL()
```

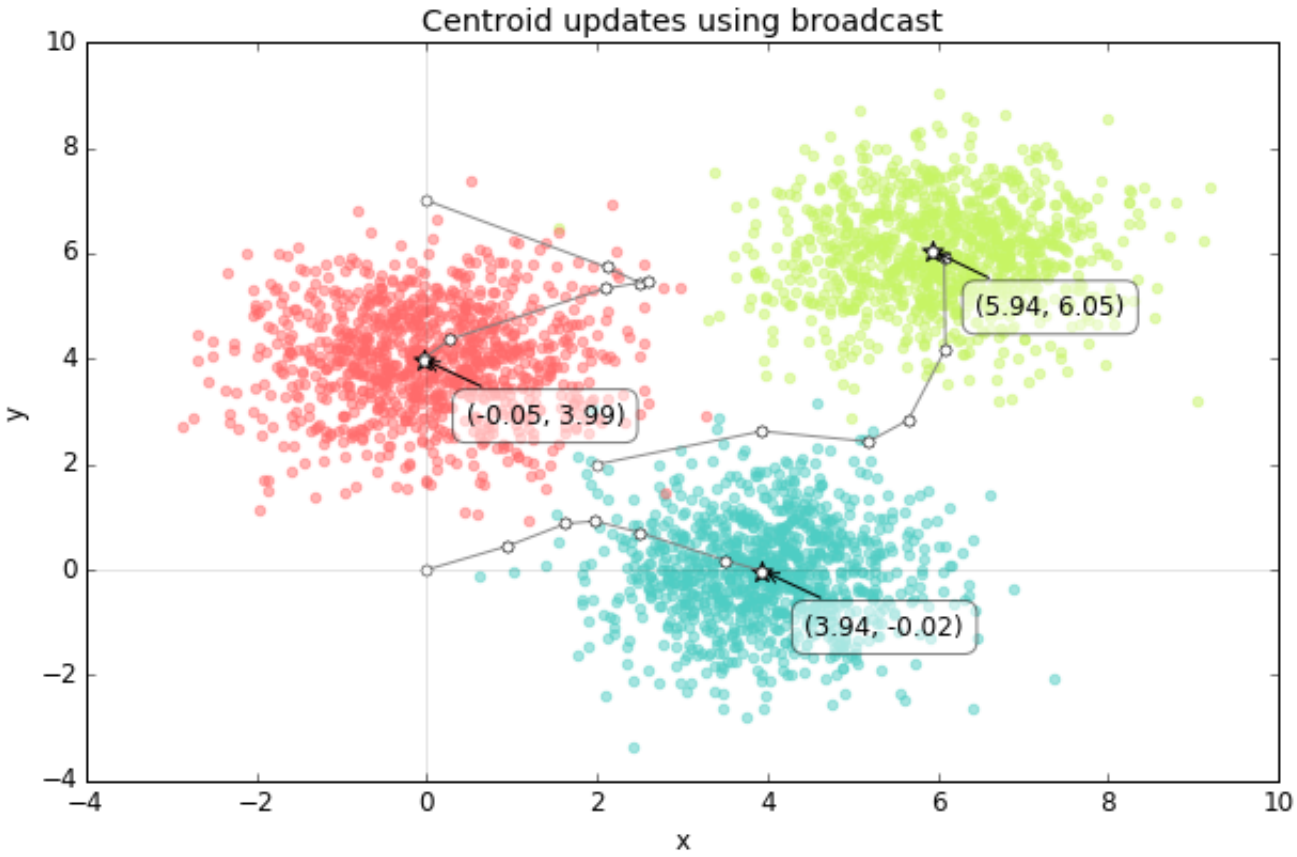
Use The Broadcast Pattern To Make This Implementation More Efficient. Please Describe Your Changes In English First.

Currently, the centroids are transmitted to each partition through a closure. We can instead broadcast the centroids so that the centroids are transmitted once per node rather than once per partition.

Implement, Comment Your Code And Highlight Your Changes.

```
%spark.pyspark

# Calculate which class each data point belongs to

def nearest_centroid(line):
    x = np.array([float(f) for f in line.split(',')])

    # Customization START - access broadcast value
```

```
    closest_centroid_idx = np.sum((x - centroids.value)**2, axis=1).argmin()
    # Customization END

    return (closest_centroid_idx,(x,1))

# Distributed KMeans in Spark

K = 3
# Initialization: initialization of parameter is fixed to show an example

# Customization START - broadcast centroids
centroids = sc.broadcast(np.array([[0.0,0.0],[2.0,2.0],[0.0,7.0]]))
# Customization END

D = sc.textFile("data.csv").cache()

fig = plt.gcf()
fig.set_size_inches(8, 6)

plot_samples('Centroid updates using broadcast')

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

    # Customization START - access broadcast value
    if np.sum(np.absolute(centroids_new-centroids.value))<0.01:
        break
    # Customization END

    # Customization START - access broadcast value
    plot_iteration(centroids.value, centroids_new)
    # Customization END

    # Customization START - broadcast centroids
    centroids = sc.broadcast(centroids_new)
    # Customization END

# Customization START - access broadcast value
plot_centroids(centroids.value)
# Customization END
showMPL()
```



HW11.1 Loss Functions

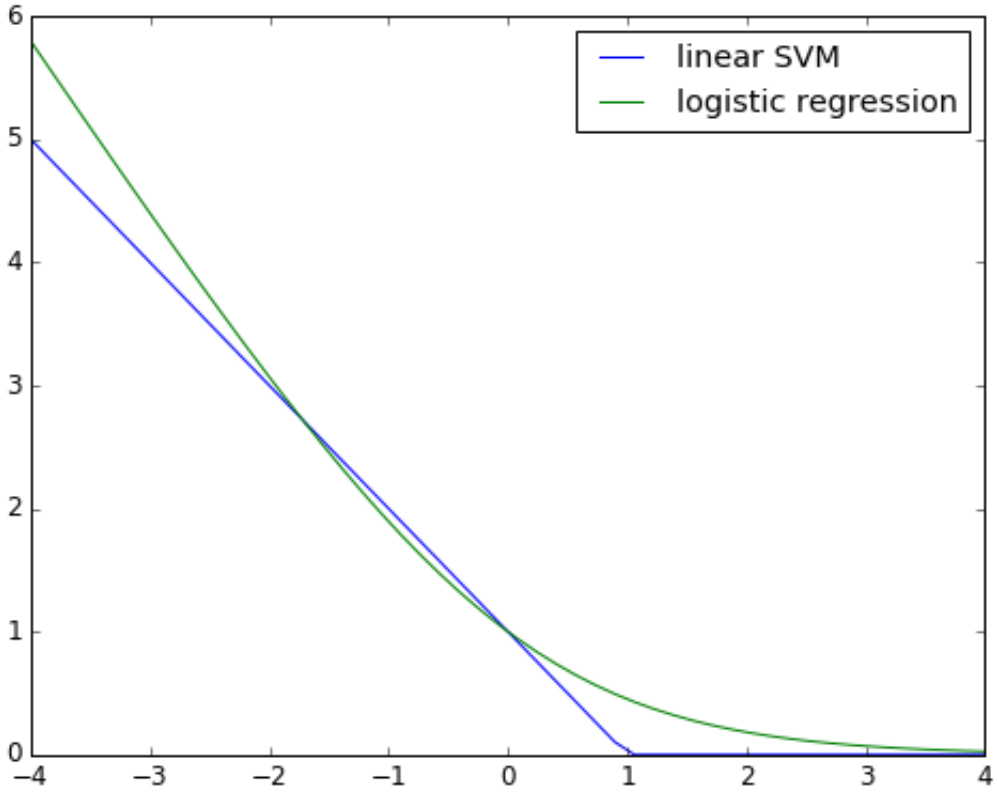
In The Context Of Binary Classification Problems, Does The Linear SVM Learning Algorithm Yield The Same Result As An L2 Penalized Logistic Regression Learning Algorithm?

Logistic regression and linear SVM are minimizing different concepts, where linear SVM is minimizing margin while logistic regression is minimizing log loss (more details below). Therefore, the two can end up with different boundary hyperplanes and thus different models. Therefore, as far as we know, the two are different, though it has been demonstrated that a logistic regression that has both an L1 and L2

penalty can be reduced to a linear SVM (reference (https://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9856)). Unless that situation changes (we are able to convert one problem to the other), that understanding holds.

Loss Function For Linear SVM Vs. Logistic Regression

```
%spark.pyspark

from numpy import exp, log

# Linear SVM is hinge loss, logistic regression is log loss

x = np.linspace(-4, 4)
hinge_loss = [max(0, 1-n) for n in x]
log_loss = log(1 + exp(-x)) / log(2)

# Plot the loss functions on the same plot

fig = plt.gcf()
fig.set_size_inches(8, 6)

plt.plot(x, hinge_loss, label = 'linear SVM')
plt.plot(x, log_loss, label = 'logistic regression')
plt.legend()

showMPL()
```



Separating Surfaces For Linear SVM Vs. Logistic Regression

In binary classification, a linear SVM and logistic regression both have a hyperplane as a decision surface. Logistic regression fits the data as if they were along a continuous sigmoid function, where the separating hyperplane surface occurs where the probability of one of the classes is greater than some threshold (usually 0.5). Linear SVM fits a hyperplane that attempts to separate the data into two classes that lie in either side of the hyperplane, and it chooses the hyperplane that maximizes the margin.

In The Context Of Binary Classification Problems, Does The Linear SVM Learning Algorithm Yield The Same Result As A Perceptron Learning Algorithm?

- If the two classes are linearly separable, the two may yield different models, because the perceptron learning algorithm does not attempt to minimize the margin; rather, it is finished as soon as it finds a weight vector corresponding to any of the hyperplanes that are able to separate the two classes. While the perceptron loss function is similar to the hinge loss function used by linear SVM, there

is no penalty for a correct guess, regardless of the distance from the separating hyperplane (hinge loss penalizes a correct guess that is closer than 1 to the hyperplane).
- If the two classes are not linearly separable, then assuming that we are talking about a hard SVM, then neither converges, which is essentially the same result, though the two may diverge to entirely different weight vectors.

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)?

```
%angular

As noted in the Apache Spark documentation (<a href="http://spark.apache.org/docs/latest/mllib-linear-methods.html#regularizers">reference</a>)

<ul>
<li><strong>L1 penalty (lasso)</strong>: The model is penalized by the L1 norm of the weight vector, $\lVert \mathbf{w} \rVert_1$ (<a href="htt
<li><strong>L2 penalty (ridge)</strong>: The model is penalized by the L2 norm of the weight vector, $\frac{1}{2} \lVert \mathbf{w} \rVert_2^2$
<li><strong>L1 and L2 penalty (elastic net)</strong>: The model is penalized by a linear combination of the L1 norm and the L2 norm of the weigh
/generated/sklearn.linear_model.ElasticNet.html">reference</a>)</li>
</ul>

<script type="text/javascript">
MathJax.Hub.Queue(["Typeset",MathJax.Hub]);
</script>
```

As noted in the Apache Spark documentation (reference (http://spark.apache.org/docs/latest/mllib-linear-methods.html#regularizers)), in the context of logistic regression, there are three flavors of penalized loss functions supported by Spark MLLib.

- **L1 penalty (lasso)**: The model is penalized by the L1 norm of the weight vector, $\lVert \mathbf{w} \rVert_1$ (reference (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html))
- **L2 penalty (ridge)**: The model is penalized by the L2 norm of the weight vector, $\frac{1}{2} \lVert \mathbf{w} \rVert_2^2$ (reference (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Ridge.html))
- **L1 and L2 penalty (elastic net)**: The model is penalized by a linear combination of the L1 norm and the L2 norm of the weight vector, $\alpha \lVert \mathbf{w} \rVert_1 + (1 - \alpha) \frac{1}{2} \lVert \mathbf{w} \rVert_2^2$ (reference (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.ElasticNet.html))

Describe Probabilistic Interpretations Of The L1 And L2 Priors For Penalized Logistic

Regression (HINT: See Synchronous Slides For Week 11 For Details)

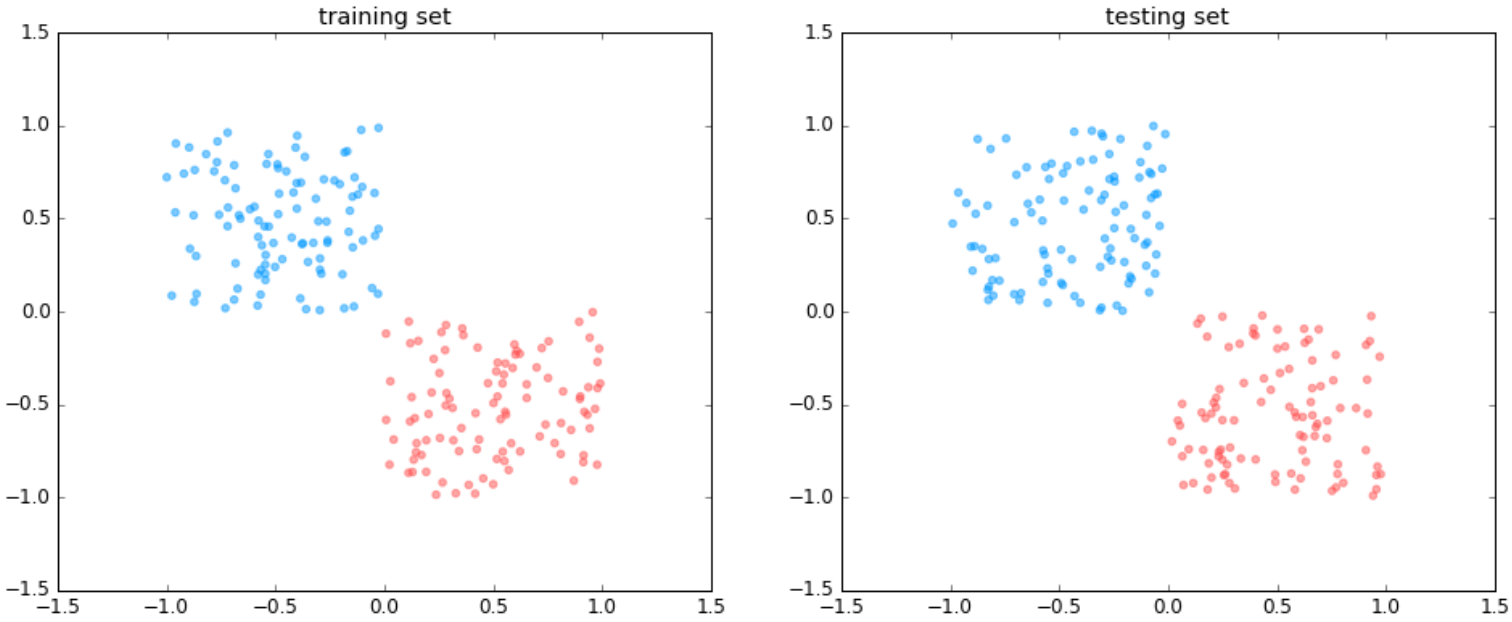
The intuition is described in a Cross Validated post (reference (http://stats.stackexchange.com/questions/163388/l2-regularization-is-equivalent-to-gaussian-prior)), and it is made more obvious from the visualizations of the probability surfaces presented in lecture slides from Brown University (reference (https://cs.brown.edu/courses/archive/2006-2007/cs195-5/lectures/lecture13.pdf)). When using L1 penalties, we recognize that the value that minimizes the L1 norm is the median, which is equivalent to the result for the MLE estimator for a Laplace distribution. When using L2 penalties, we recognize that the value that minimizes the L2 norm is the sample mean, which is equivalent to the result for the MLE estimator for a Gaussian distribution. Therefore, when we look at probabilistic interpretations of the L1 and L2 priors, they can be interpreted as a Laplacian prior and a Gaussian prior, respectively.

HW11.3 Logistic Regression

Generate 2 Sets Of Linearly Separable Data With 100 Data Points Each Using The Data Generation Code Provided Below. Call One The Training Set And The Other The Testing Set. Plot Each In Separate Plots.

```
%spark.pyspark

# Provided data generation code

from numpy.random import rand

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

# Create the training set and testing set

training_set = np.array(generateData(100))
testing_set = np.array(generateData(100))

# Plot the data sets
```

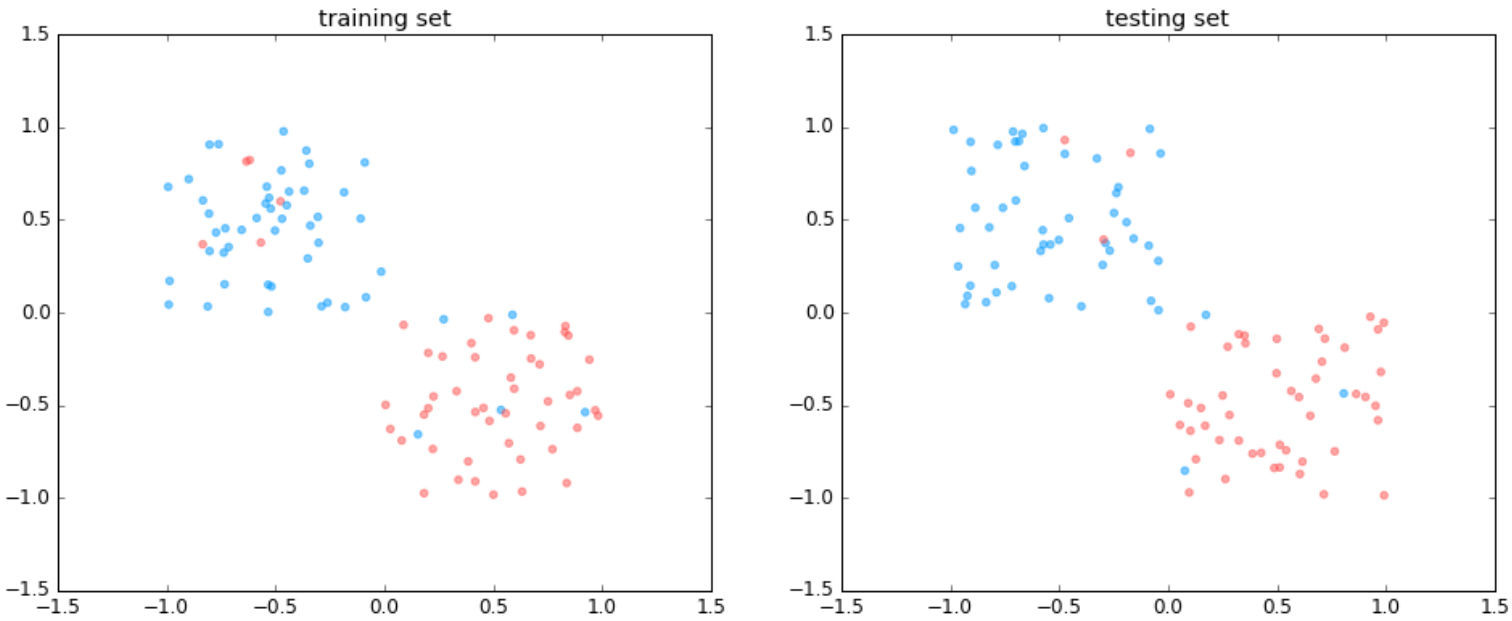
```
def plot_data_set(data_set):
    data_colors = ['#0099ff' if x == 1 else '#ff5050' for x in data_set[:, 2]]
    plt.scatter(data_set[:, 0], data_set[:, 1], color=data_colors, alpha=0.5)

fig = plt.gcf()
fig.set_size_inches(16, 6)

plt.subplot(1, 2, 1)
plt.title('training set')
plot_data_set(training_set)

plt.subplot(1, 2, 2)
plt.title('testing set')
plot_data_set(testing_set)

showMPL()
```



Modify This Data Generation Code To Generate 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 Data Sets.

```
%spark.pyspark

def generateNonSeparableData(n, flip_rate=0.1):
    # Generate separable data, and then flip each element at 10% chance
```

```
    separable_data = generateData(n)
    non_separable_data = [x[:-1] + [-x[-1] if flip <= flip_rate else x[-1]] for x, flip in zip(separable_data, rand(n))]
    return non_separable_data

# Create the training set and testing set

training_set = np.array(generateNonSeparableData(100))
testing_set = np.array(generateNonSeparableData(100))

# Plot the data sets

fig = plt.gcf()
fig.set_size_inches(16, 6)

plt.subplot(1, 2, 1)
plt.title('training set')
plot_data_set(training_set)

plt.subplot(1, 2, 2)
plt.title('testing set')
plot_data_set(testing_set)
```



Using MLLib Train Up A LASSO Logistic Regression Model With The Training Dataset.

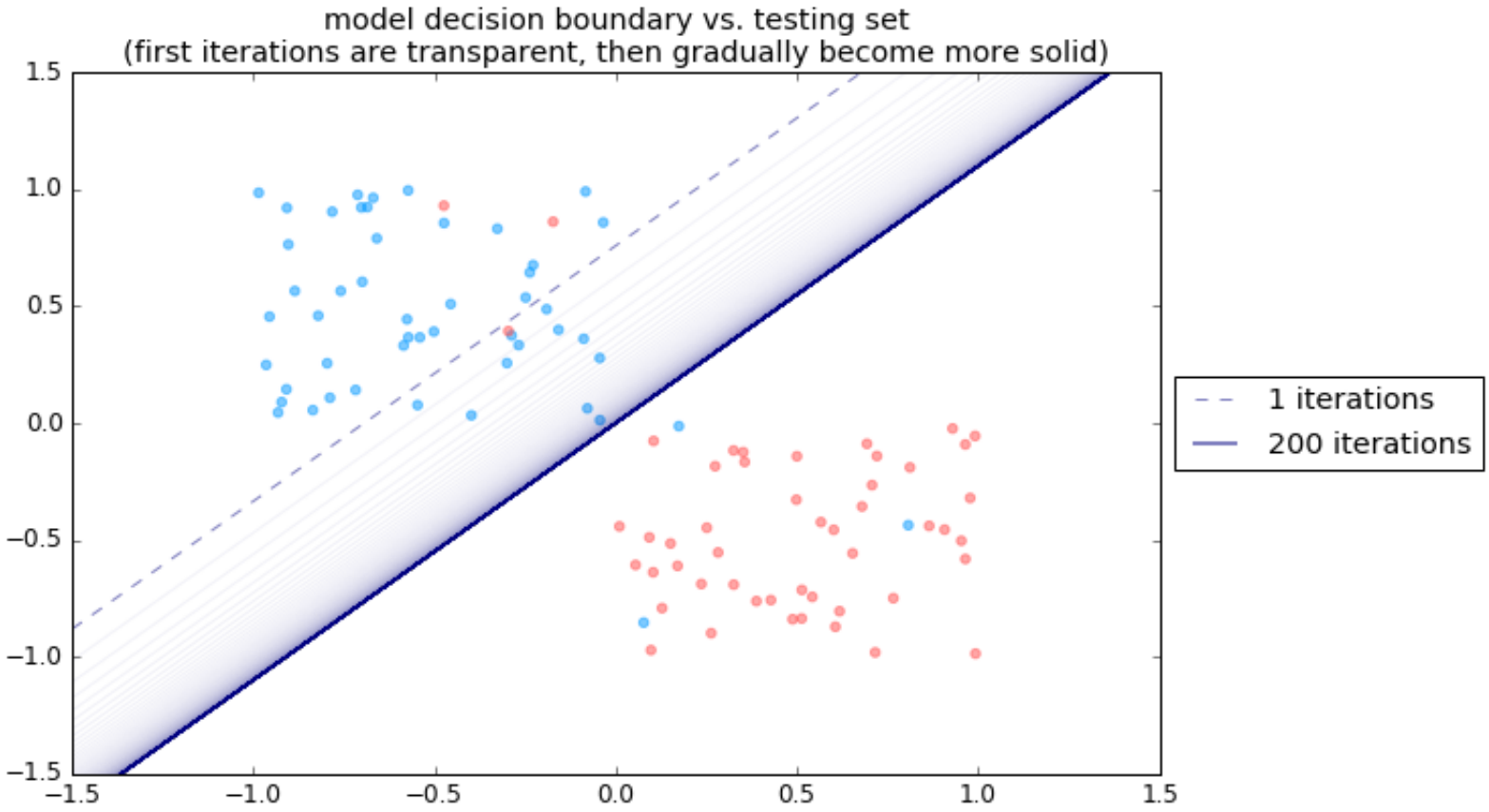
```
%spark.pyspark

import functools
```

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.classification import LogisticRegressionModel, LogisticRegressionWithSGD

# Use the last element as the label, all other elements as the features

def getLabeledPoint(point):
    label = 1 if point[-1] == 1 else 0
    return LabeledPoint(label, point[:-1])

# Create an RDD for the training set and test sets

training_rdd = sc.parallelize(training_set).map(getLabeledPoint).cache()
testing_rdd = sc.parallelize(testing_set).map(getLabeledPoint).cache()

actual_labels_rdd = testing_rdd.map(lambda x: x.label).cache()
testing_features_rdd = testing_rdd.map(lambda x: x.features).cache()

# Define a function to iterate on top of the weights from a previous logistic regression

def mllib_logistic_regression(n):
    model = LogisticRegressionWithSGD.train(data = training_rdd, regType='l1', iterations=n, initialWeights=[0.0, 0.0], intercept=True)
    return list(model.weights) + [model.intercept]

# Initialize the weights and the tracking of the iterations

mllib_iterations = []
mllib_weights = []

# Run iterations

for i in range(200):
    iterations = i + 1
    next_weights = mllib_logistic_regression(iterations)

    mllib_iterations.append(iterations)
    mllib_weights.append(next_weights)
```

Evaluate With The Testing Set.

```
%spark.pyspark

# Plot the weights for an iteration

def plot_weights(iteration_id, model_weights, xlim, ylim, line_style, line_alpha, line_width):
    slope = model_weights[0] / (-1 * model_weights[1])
    intercept = model_weights[2]

    x_values = xlim
    y_values = [x * slope + intercept for x in x_values]
    zip_values = np.array(zip(x_values, y_values))

    if iteration_id is not None:
        plot_label = str(iteration_id) + ' iterations'
    else:
        plot_label = None
```

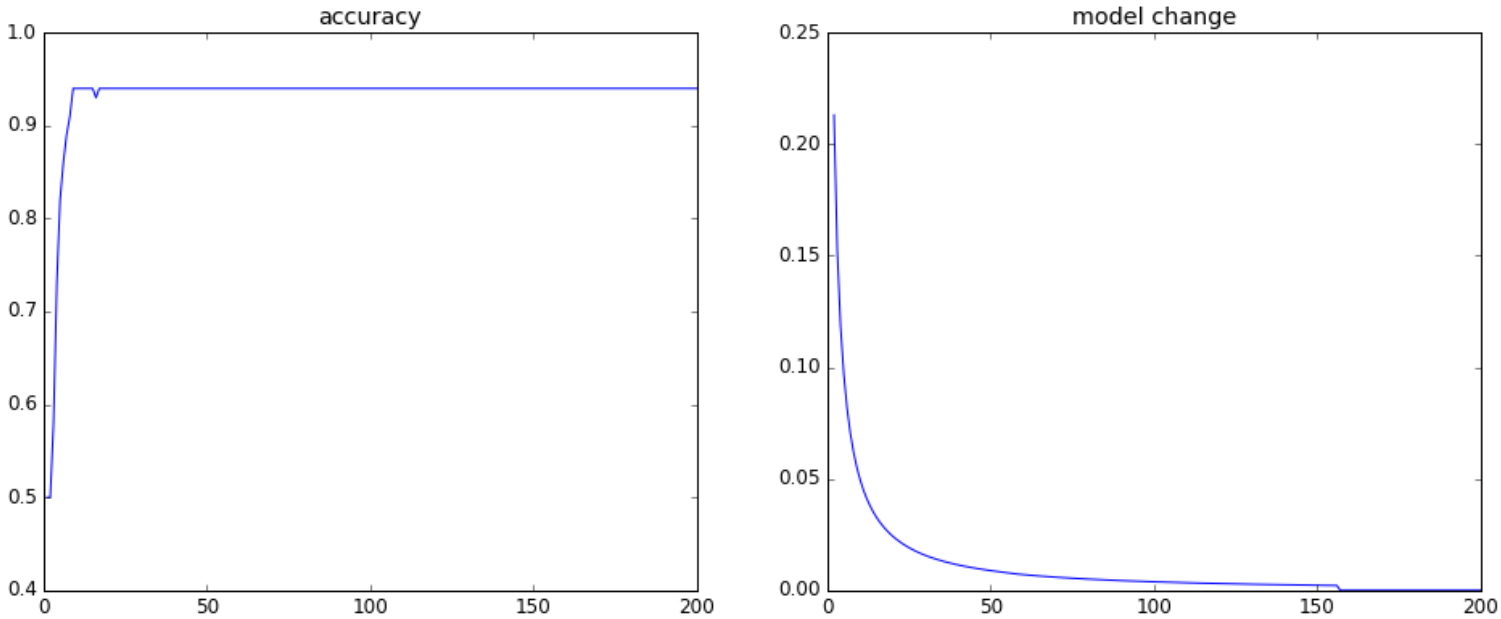
```
    print plot_label
    plt.plot(zip_values[:, 0], zip_values[:, 1], linestyle=line_style, linewidth=line_width, label=plot_label, alpha=line_alpha, color='navy')

# Plot the weights for all iterations

def evaluate_model_progress(iterations, weights, plot_step):

    # Initialize our plot

    fig = plt.gcf()
    fig.set_size_inches(12, 6)

    plt.title('model decision boundary vs. testing set\n(first iterations are transparent, then gradually become more solid)')
    plot_data_set(testing_set)

    axes = plt.gca()

    xlim = axes.get_xlim()
    ylim = axes.get_ylim()

    # Plot each iteration

    plot_weights(iterations[0], weights[0], xlim, ylim, '--', 0.5, 1)

    for i, alpha in zip(range(1, len(iterations)), np.linspace(0.05, 0.2, len(iterations))):
        plot_weights(None, weights[i], xlim, ylim, '-', alpha, 1)

    plot_weights(iterations[-1], weights[-1], xlim, ylim, '-', 0.5, 2)

    # There are a lot of iterations, so add the legend outside the bounding box

    box = axes.get_position()
    axes.set_position([box.x0, box.y0, box.width * 0.8, box.height])

    plt.xlim(xlim[0], xlim[1])
    plt.ylim(ylim[0], ylim[1])

    plt.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    showMPL()

evaluate_model_progress(mllib_iterations, mllib_weights, 10)
```



What Is A Good Number Of Iterations For Training The Logistic Regression Model? Justify With Plots.

```
%spark.pyspark

# Initialize our plot

fig = plt.gcf()
fig.set_size_inches(16, 6)

# Compute accuracy for the given set of weights using the testing set

def get_accuracy(model_weights):
    model = LogisticRegressionModel(model_weights[:-1], model_weights[-1], 2, 2)

    # Compare the predictions with the actual result

    predicted_labels_rdd = model.predict(testing_features_rdd)
```

```
    return predicted_labels_rdd.zip(actual_labels_rdd).map(lambda x: 1.0 if x[0] == x[1] else 0.0).mean()

# Plot our accuracies

mllib_accuracies = [get_accuracy(model_weights) for model_weights in mllib_weights]
plt.subplot(1, 2, 1)
plt.title('accuracy')
plt.plot(mllib_iterations, mllib_accuracies)

# Returns how much the model has changed

def get_model_changes(prev_weights, next_weights):
    weight_change = np.array(prev_weights) - np.array(next_weights)
    return np.linalg.norm(weight_change)

# Plot our model changes

mllib_model_changes = [get_model_changes(prev_weights, next_weights) for prev_weights, next_weights in zip(mllib_weights[:-1], mllib_weights[1:]
plt.subplot(1, 2, 2)
plt.title('model change')
plt.plot(mllib_iterations[1:], mllib_model_changes)

showMPL()
```



What Is A Good Number Of Iterations For Training The Logistic Regression Model? Justify With Words. (Part 1)

```
%spark.pyspark

def show_iteration_row(iteration, iteration_weights, accuracy, model_change):
    print str(iteration) + '\t' + str(accuracy) + '\t' + str(1.0 - accuracy) + '\t' + str(iteration_weights[:-1]) + '\t' + str(iteration_weights

def show_iteration_table(iterations, weights, accuracies, model_changes):
    print '%table iterations\taccuracy\tmisclassification\tweights\tintercept\tmodel change'

    indices = np.arange(9, len(iterations), 10)

    show_iteration_row(iterations[0], weights[0], accuracies[0], None)

    for i in indices:
        show_iteration_row(iterations[i], weights[i], accuracies[i], model_changes[i-1])

    if (len(iterations) - 1) not in indices:
        show_iteration_row(iterations[-1], weights[-1], accuracies[-1], model_changes[-1])

show_iteration_table(mllib_iterations, mllib_weights, mllib_accuracies, mllib_model_changes)
```

iterations	accuracy	misclassification	weights
1	0.5	0.5	[-0.19710158781919551, 0.180255010204728
10	0.94	0.06	[-0.74601879395639048, 0.680333038156884
20	0.94	0.06	[-0.96896940410600252, 0.882645981075286
30	0.94	0.06	[-1.1022143138976559, 1.0033736778739311]
40	0.94	0.06	[-1.1959119587477329, 1.0882097097599344]
50	0.94	0.06	[-1.2672775371303515, 1.1528056518030945]
60	0.94	0.06	[-1.3243203475299195, 1.2044359964671854]
70	0.94	0.06	[-1.3714090708104845, 1.2470858160971376]
80	0.94	0.06	[-1.4112024007085586, 1.2831594178318073]
90	0.94	0.06	[-1.4454377126393521, 1.3142222075967775]
100	0.94	0.06	[-1.4753096982042484, 1.3413505801350416]
110	0.94	0.06	[-1.5016739138565451, 1.3653153696346834]

120	0.94	0.06	[-1.5251632206254797, 1.386686777598676]
130	0.94	0.06	[-1.5462583082532284, 1.4058979174327504]
140	0.94	0.06	[-1.5653324275766591, 1.4232851162025668]
150	0.94	0.06	[-1.5826808667195003, 1.4391144694240368]
160	0.94	0.06	[-1.5923623118764163, 1.4479550742688827]
170	0.94	0.06	[-1.5923623118764163, 1.4479550742688827]
180	0.94	0.06	[-1.5923623118764163, 1.4479550742688827]
190	0.94	0.06	[-1.5923623118764163, 1.4479550742688827]
200	0.94	0.06	[-1.5923623118764163, 1.4479550742688827]

What Is A Good Number Of Iterations For Training The Logistic Regression Model? Justify With Words. (Part 2)

From an accuracy perspective, we can stop after 10 iterations. The model change drops to the 0.01 threshold at around 50 iterations. If we were to use the change in the model as our measure of what would be a good number of iterations, we would stop at 50 iterations.

Derive And Implement In Spark A Weighted LASSO Logistic Regression. Weight Each Example Using The Inverse Vector Length (Euclidean Norm).

```
%angular

$weight(X) = \large{\frac{1}{\lVert X \rVert}}$ where $\lVert X \rVert = \sqrt{X \cdot X} = \sqrt{X_{1}^2 + X_{2}^2}$

<script type="text/javascript">
MathJax.Hub.Queue(["Typeset",MathJax.Hub]);
</script>
```

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

```
%spark.pyspark

# Add bias term and transform points into weighted points
```

```
from collections import namedtuple

Point = namedtuple('Point', 'x y w')

def readPoint(labeledPoint):
    x = labeledPoint.features
    x = np.append(x, 1.0)
    return Point(x, labeledPoint.features[-1], 1 / np.linalg.norm(x))

weighted_training_rdd = training_rdd.map(readPoint).cache()

# Define a function to iterate on top of the weights from a previous logistic regression

def weighted_logistic_regression(previous_weights=None, learning_rate = 0.05, reg_param = 0.01):
    n = weighted_training_rdd.count()
    featureLen = len(weighted_training_rdd.take(1)[0].x)

    # Initialize weights to random
    if previous_weights is None:
        previous_weights = np.random.normal(size=featureLen)

    # Broadcast weights
    w = sc.broadcast(previous_weights)

    # Calculate gradient
    gradient_tuple = weighted_training_rdd.map(lambda p: ((p.w / (1 + np.exp(-p.y * np.dot(w.value, p.x))) - 1) * p.y * np.array(p.x), p.w)).re
    gradient = gradient_tuple[0] / n

    # Lasso regularization
    wReg = np.array([1.0 if x > 0 else 0.0 for x in w.value])
    wReg[-1] = 0 # last value of weight vector is bias term, ignored in regularization

    # Put it all together
    gradient = gradient + reg_param * wReg  # gradient: GD of Sqaured Error + GD of regularized term
    newW = w.value - learning_rate * gradient

    return newW
```

Implement A Convergence Test Of Your Choice To Check For Termination Within Your Training Algorithm. Report How Many Iterations It Took To Converge.

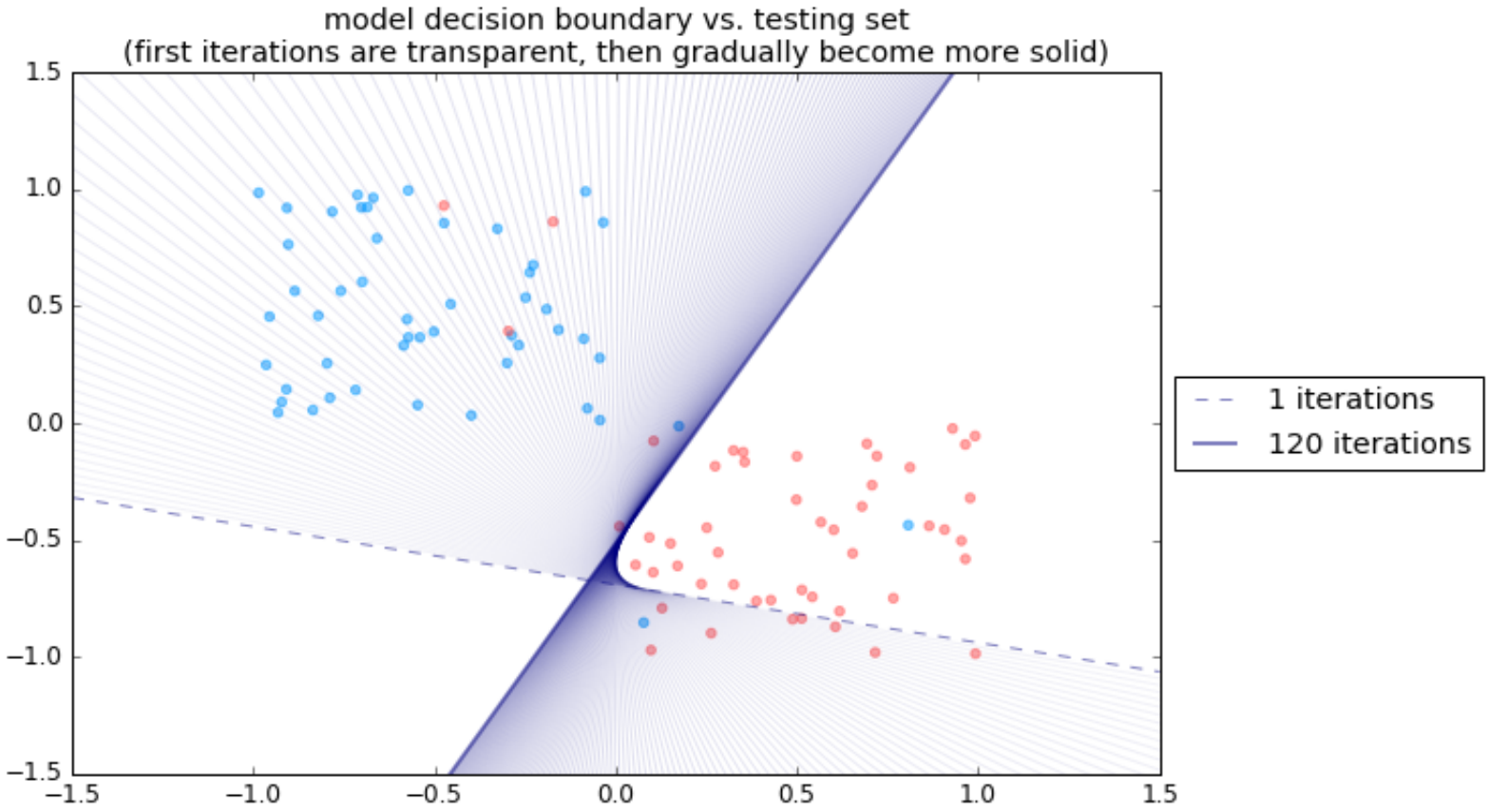
```
%spark.pyspark

# Initialize the weights and the tracking of the iterations

previous_weights = None

normed_iterations = []
normed_weights = []

converged = False

# Run iterations until we have converged. We'll assume convergence is when the change
# in the model is below 0.01. We'll also add a safety net of 500 iterations in case the
```

```
# model cannot converge.

iteration_id = 0

while not converged and iteration_id < 500:
    iteration_id += 1
    next_weights = weighted_logistic_regression(previous_weights)

    normed_iterations.append(iteration_id)
    normed_weights.append(next_weights)

    if iteration_id > 1:
        model_change = get_model_changes(previous_weights, next_weights)
        converged = model_change <= 0.01

    previous_weights = next_weights

if converged:
    print 'Converged in', len(normed_iterations), 'iterations'
else:
```

```
Converged in 120 iterations
```

Evaluate Your Homegrown Weighted LASSO Logistic Regression On The Test Data Set.

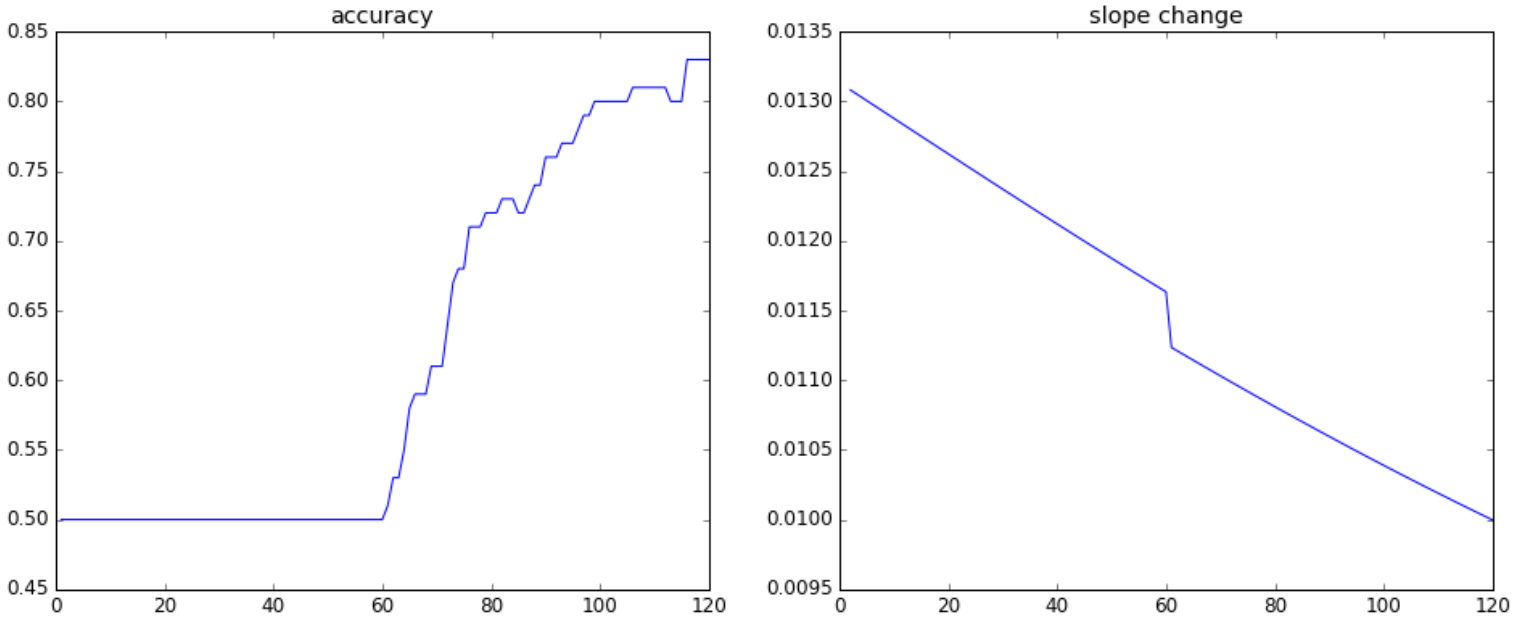
```
%spark.pyspark

evaluate_model_progress(normed_iterations, normed_weights, 10)
```



Report Misclassification Error (1 - Accuracy). (Part 1)

```
%spark.pyspark

# Initialize our plot

fig = plt.gcf()
fig.set_size_inches(16, 6)

# Plot our accuracies

normed_accuracies = [get_accuracy(model_weights) for model_weights in normed_weights]
plt.subplot(1, 2, 1)
plt.title('accuracy')
plt.plot(normed_iterations, normed_accuracies)

# Plot our model changes

normed_model_changes = [get_model_changes(prev_weights, next_weights) for prev_weights, next_weights in zip(normed_weights[:-1], normed_weights
```

```
plt.subplot(1, 2, 2)
plt.title('slope change')
plt.plot(normed_iterations[1:], normed_model_changes)
```



Report Misclassification Error (1 - Accuracy). (Part 2)

```
%spark.pyspark

show_iteration_table(normed_iterations, normed_weights, normed_accuracies, normed_model_changes)
```

iterations	accuracy	misclassification	weights
1	0.5	0.5	[-0.13718988 -0.55257782]
10	0.5	0.5	[-0.21111799 -0.46347625]
20	0.5	0.5	[-0.29179849 -0.36638943]
30	0.5	0.5	[-0.37095375 -0.27130134]

40	0.5	0.5	[-0.4486036 -0.17818652]
50	0.5	0.5	[-0.52477352 -0.087012]
60	0.5	0.5	[-0.59949408 0.0022621]
70	0.61	0.39	[-0.67282029 0.08471152]
80	0.72	0.28	[-0.74481415 0.16542204]
90	0.76	0.24	[-0.81551549 0.24444591]
100	0.8	0.2	[-0.88496627 0.32183831]
110	0.81	0.19	[-0.95320998 0.39765651]
120	0.83	0.17	[-1.02029103 0.4719591]

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.

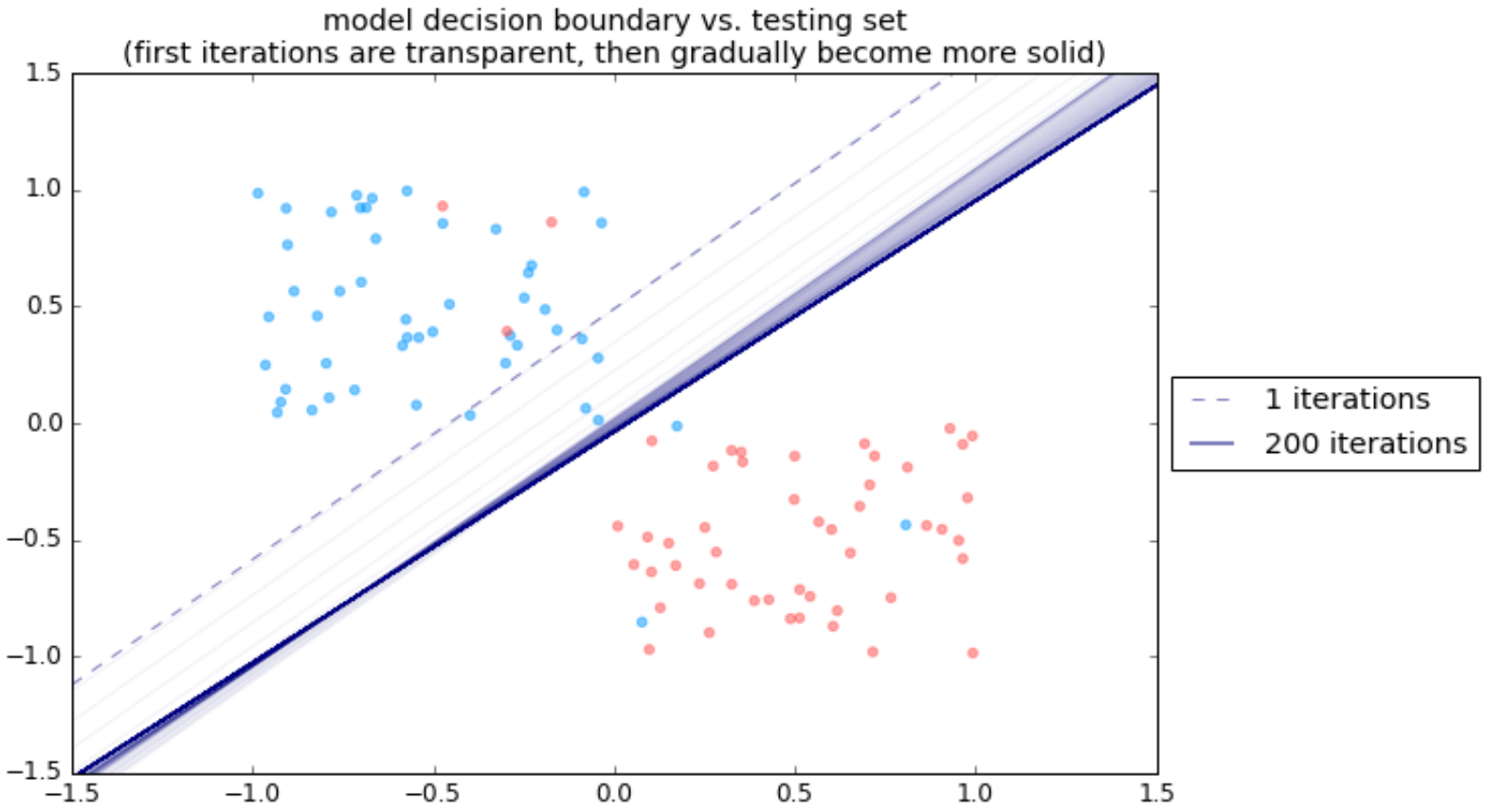
We were unable to identify a built-in weighted LASSO logistic regression implementation.

HW11.4 SVMs

Using MLLib Train Up A Soft SVM Model With The Training Dataset And Evaluate With The Testing Set.

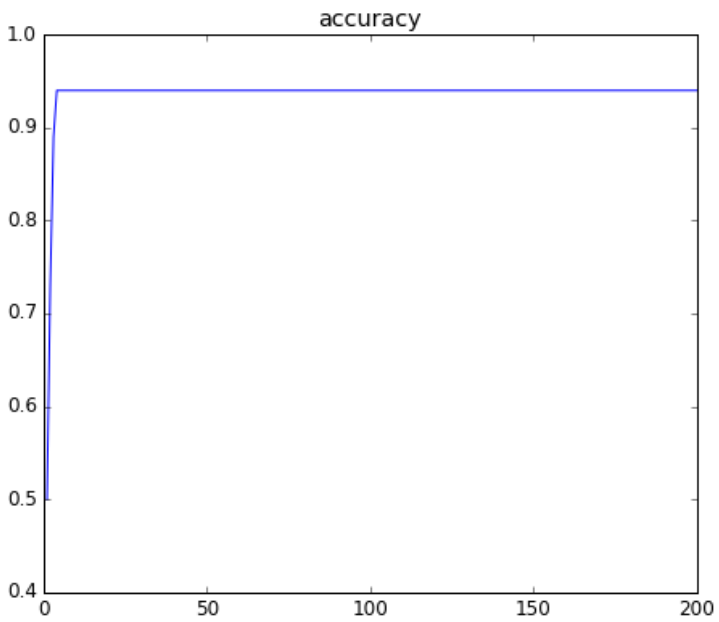
```
%spark.pyspark

from pyspark.mllib.classification import SVMModel, SVMWithSGD

# Define a function to iterate on top of the weights from a previous SVM

def mllib_svm(n):
    model = SVMWithSGD.train(data = training_rdd, regType='l2', iterations=n, initialWeights=[0.0, 0.0], intercept=True)
    return list(model.weights) + [model.intercept]

# Initialize the weights and the tracking of the iterations
```

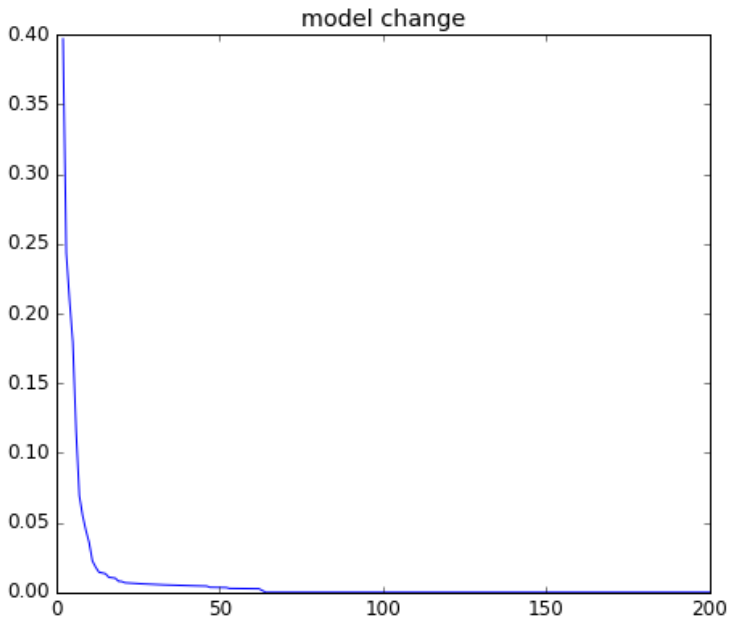
```
previous_weights = [1.0, 1.0]

mllib_svm_iterations = []
mllib_svm_weights = []

# Run iterations

for i in range(200):
    next_weights = mllib_svm(i + 1)

    mllib_svm_iterations.append(i + 1)
    mllib_svm_weights.append(next_weights)

    previous_weights = next_weights

evaluate_model_progress(mllib_svm_iterations, mllib_svm_weights, 10)
```



What Is A Good Number Of Iterations For Training The SVM Model? Justify With Plots.

```
%spark.pyspark

# Initialize our plot

fig = plt.gcf()
fig.set_size_inches(16, 6)

# Compute accuracy for the given set of weights using the testing set

def get_svm_accuracy(model_weights):
    model = SVMModel(model_weights[:-1], model_weights[-1])

    # Compare the predictions with the actual result

    predicted_labels_rdd = model.predict(testing_features_rdd)

    return predicted_labels_rdd.zip(actual_labels_rdd).map(lambda x: 1.0 if x[0] == x[1] else 0.0).mean()

# Plot our accuracies

mllib_svm_accuracies = [get_svm_accuracy(model_weights) for model_weights in mllib_svm_weights]
plt.subplot(1, 2, 1)
plt.title('accuracy')
plt.plot(mllib_svm_iterations, mllib_svm_accuracies)

# Plot our model changes

mllib_svm_model_changes = [get_model_changes(prev_weights, next_weights) for prev_weights, next_weights in zip(mllib_svm_weights[:-1], mllib_svm
plt.subplot(1, 2, 2)
plt.title('model change')
plt.plot(mllib_svm_iterations[1:], mllib_svm_model_changes)

show()
```





What Is A Good Number Of Iterations For Training The SVM Model? Justify With Words. (Part 1)

```
%spark.pyspark

show_iteration_table(mllib_svm_iterations, mllib_svm_weights, mllib_svm_accuracies, mllib_svm_model_changes)
```

iterations	accuracy	misclassification	weights
1	0.5	0.5	[-0.20518880279996701, 0.1908725541811849
10	0.94	0.06	[-1.0896071933494564, 1.0009220610138638]
20	0.94	0.06	[-1.1667658904270148, 1.0938694116764225]
30	0.94	0.06	[-1.1945905323627195, 1.1453892934777339]
40	0.94	0.06	[-1.2176852113909453, 1.1881512142428901]

50	0.94	0.06	[-1.2338857332686504, 1.224587463621017]
60	0.94	0.06	[-1.2408885450645983, 1.2492225264552226]
70	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
80	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
90	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
100	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
110	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
120	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
130	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
140	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
150	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
160	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
170	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
180	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
190	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]
200	0.94	0.06	[-1.2426113969686954, 1.2547689838127196]

What Is A Good Number Of Iterations For Training The SVM Model? Justify With Words. (Part 2)

With the current initial weights, accuracy peaks after 10 iterations, so we will need at least that many iterations. Since the SVM model is actually trying to minimize the margin, the model change is more important than the accuracy. We notice that the model is very stable after 70 iterations, which makes 70 iterations a good candidate for this model.

Derive And Implement In Spark A Weighted Soft Linear SVM Classification Learning

Algorithm.

```
%spark.pyspark

# Define a function to iterate on top of the weights from a previous SVM iteration

def weighted_svm(previous_weights=None, learning_rate = 0.05, reg_param = 0.01):
    n = weighted_training_rdd.count()
    featureLen = len(weighted_training_rdd.take(1)[0].x)

    # Initialize weights to random
    if previous_weights is None:
        previous_weights = np.random.normal(size=featureLen)

    w = sc.broadcast(previous_weights)

    # Find support vectors
    sv = weighted_training_rdd.filter(lambda p: p.y * np.dot(w.value, p.x) < 1)

    if sv.isEmpty():
        return previous_weights

    # Calculate gradient
    gradient_tuple = sv.map(lambda p: (p.w * p.y * np.array(p.x), p.w)).reduce(lambda x, y: (x[0] + y[0], x[1] + y[1]))
    gradient = -gradient_tuple[0] / n

    # Lasso regularization
    wReg = np.array([1.0 if x > 0 else 0.0 for x in w.value])
    wReg[-1] = 0 # last value of weight vector is bias term, ignored in regularization

    wDelta = learning_rate * (gradient + reg_param * wReg)

    newW = w.value - wDelta

    return newW
```

Evaluate Your Homegrown Weighted Soft Linear SVM Classification Learning Algorithm On The Weighted Training Dataset And Test Dataset From HW11.3. Report How Many Iterations It Took To Converge.

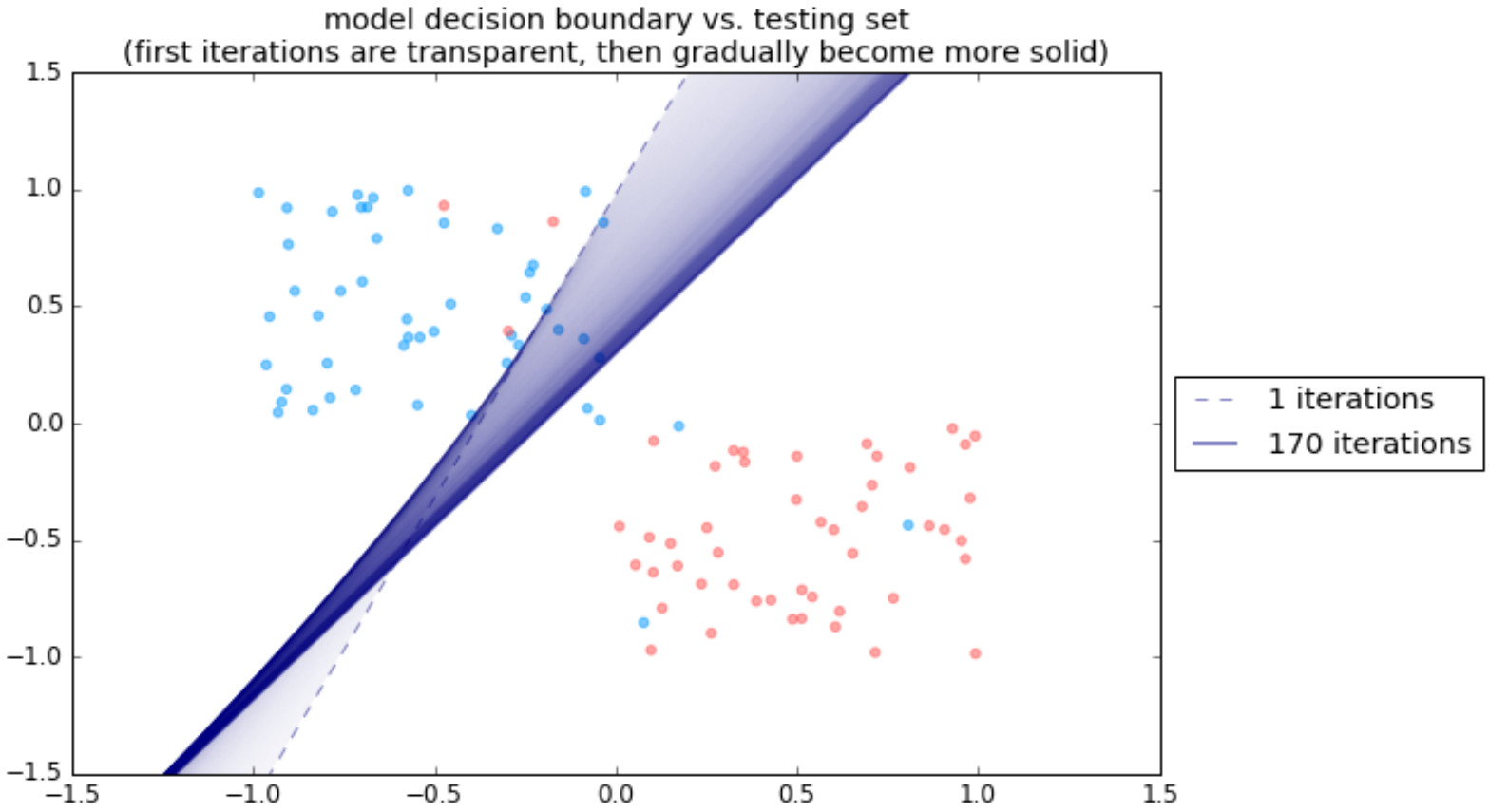
```
%spark.pyspark

# Initialize the weights and the tracking of the iterations

previous_weights = None

normed_svm_iterations = []
normed_svm_weights = []

converged = False

# Run iterations until we have converged. We'll assume convergence is when the change
```

```
# in the model is below 0.001. We'll also add a safety net of 500 iterations in case the
# model cannot converge.

iteration_id = 0

while not converged and iteration_id < 500:
    iteration_id += 1
    next_weights = weighted_svm(previous_weights)

    normed_svm_iterations.append(iteration_id)
    normed_svm_weights.append(next_weights)

    if iteration_id > 1:
        model_change = get_model_changes(previous_weights, next_weights)
        converged = model_change <= 0.005

    previous_weights = next_weights

if converged:
    print 'Converged in', len(normed_svm_iterations), 'iterations'
else:
```

```
Converged in 170 iterations
```

Evaluate Your Homegrown Weighted Linear SVM On The Test Dataset.

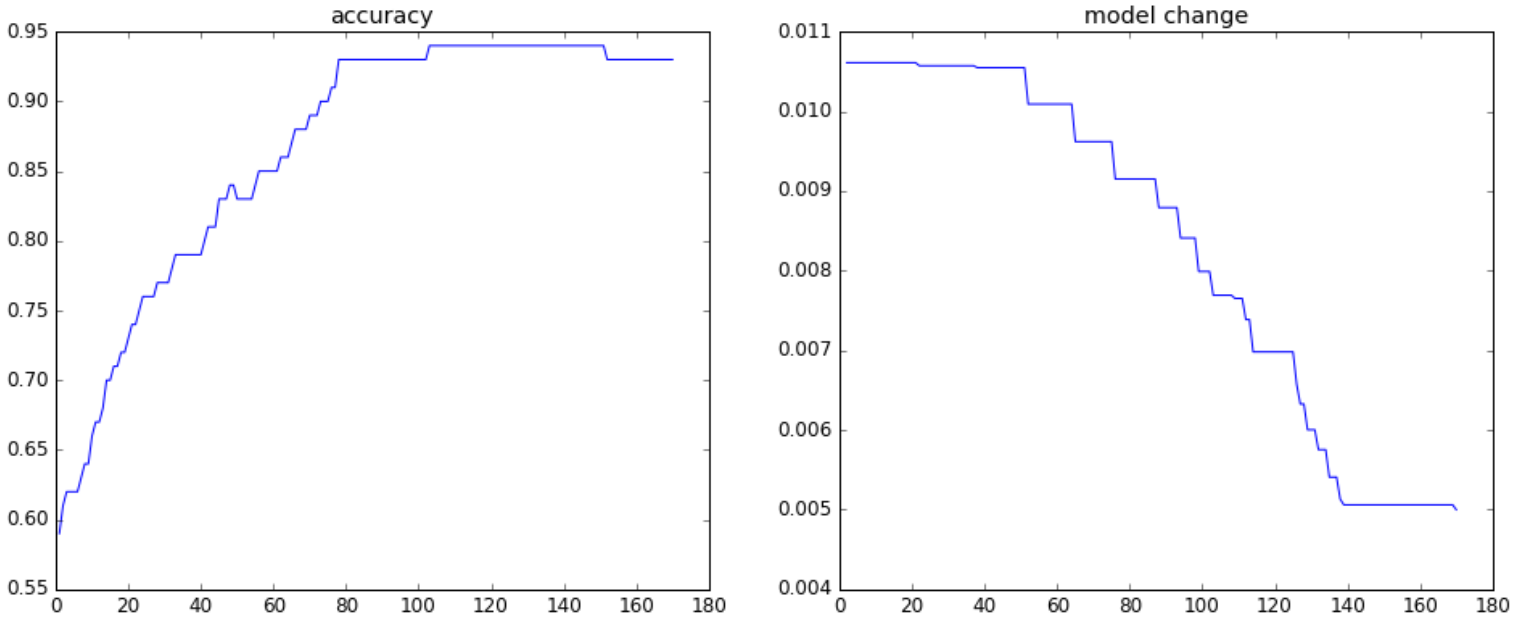
```
%spark.pyspark

evaluate_model_progress(normed_svm_iterations, normed_svm_weights, 10)
```



Report Misclassification Error (1 - Accuracy). (Part 1)

```
%spark.pyspark

# Initialize our plot

fig = plt.gcf()
fig.set_size_inches(16, 6)

# Plot our accuracies

normed_svm_accuracies = [get_svm_accuracy(model_weights) for model_weights in normed_svm_weights]
plt.subplot(1, 2, 1)
plt.title('accuracy')
plt.plot(normed_svm_iterations, normed_svm_accuracies)

# Plot our model changes

normed_svm_model_changes = [get_model_changes(prev_weights, next_weights) for prev_weights, next_weights in zip(normed_svm_weights[:-1], normed_
```

```
plt.subplot(1, 2, 2)
plt.title('model change')
plt.plot(normed_svm_iterations[1:], normed_svm_model_changes)

show()
```



Report Misclassification Error (1 - Accuracy). (Part 2)

```
%spark.pyspark

show_iteration_table(normed_svm_iterations, normed_svm_weights, normed_svm_accuracies, normed_svm_model_changes)
```

iterations	accuracy	misclassification	weights
1	0.59	0.41	[-1.03778932 0.3988391]
10	0.66	0.34	[-1.09469448 0.46011935
20	0.73	0.27	[-1.15792244 0.52820851
30	0.77	0.23	[-1.22010562 0.59517499

40	0.79	0.21	[-1.28185899 0.66161165
50	0.83	0.17	[-1.34288036 0.72710311
60	0.85	0.15	[-1.40172893 0.79020902
70	0.89	0.11	[-1.4591353 0.85121759]
80	0.93	0.07	[-1.5149135 0.90938268]
90	0.93	0.07	[-1.56927983 0.96536796
100	0.93	0.07	[-1.620792 1.0179171]
110	0.94	0.06	[-1.66897178 1.06620081
120	0.94	0.06	[-1.71342733 1.11042362
130	0.94	0.06	[-1.75606784 1.15123521
140	0.94	0.06	[-1.79321755 1.18542805
150	0.94	0.06	[-1.82761117 1.21697889
160	0.93	0.07	[-1.8620048 1.24852974]
170	0.93	0.07	[-1.89627347 1.28001078

How Many Support Vectors Do You End Up With?

```
%spark.pyspark

def get_support_vector_count(weights):
    w = sc.broadcast(weights)
    sv = weighted_training_rdd.filter(lambda p: p.y * np.dot(w.value, p.x) < 1)
    return sv.count()

print get_support_vector_count(normed_svm_weights[-1]), 'support vectors'
```

```
61 support vectors
```

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.

We were unable to identify a built-in weighted soft SVM learner.