## HU Extension Assignment 06 E63 Big Data Analytics

### Handed out: 03/03/2017 Due by 9:30 AM EST on Saturday, 03/11/2017

All code has been executed using Jupyter Notebooks. All my code and comment are in blue color.

**Problem 1.** Attached file auto\_mpg\_original.csv contains a set of data on automobile characteristics and fuel consumption. File auto\_mpg\_description.csv contains the description of the data. Import data into Spark.

Instantiate spark context and loaded file into a RDD

|  |
| --- |
| from pyspark import SparkContext, SparkConf  conf = SparkConf().setAppName("Spark Count")  sc = SparkContext(conf=conf)  path = "file:///home/jovyan/work/auto\_mpg\_original.csv"  raw\_data = sc.textFile(path)  records = raw\_data.map(lambda x: x.split(","))  records.cache()  num\_data = records.count()  first = records.first()  print "First data row: %s" %first  print "Total number of rows %u" %num\_data |

Randomly select 10-20% of you data for testing and use remaining data for training.

|  |
| --- |
| records\_with\_idx = records.zipWithIndex().map(lambda (k, v): (v, k))  test\_data\_idx = records\_with\_idx.sample(False, 0.2, 42)  training\_data\_idx = records\_with\_idx.subtractByKey(test\_data\_idx)  test\_data = test\_data\_idx.map(lambda (idx, p) : p)  training\_data = training\_data\_idx.map(lambda (idx, p) : p) |

Look initially at two variables: the horsepower and the displacement. Treat displacement as a feature and horsepower as the target variable (label). Use MLlib linear regression to identify the model for the relationship.

|  |
| --- |
| from pyspark.mllib.regression import LinearRegressionWithSGD  from pyspark.mllib.regression import LabeledPoint  import numpy as np  test\_dt = test\_data.filter(lambda v: v[3]<>"NA").map(lambda r:  LabeledPoint(float(r[3]),np.array(r[2:3])))  data\_dt = training\_data.filter(lambda v: v[3]<>"NA").map(lambda r:  LabeledPoint(float(r[3]),np.array(r[2:3])))  linear\_model = LinearRegressionWithSGD.train(data\_dt, iterations=200,step=0.000001, intercept=False) |

Use the test data to illustrate accuracy of the linear regression model and its ability to predict the relationship.

|  |
| --- |
| true\_vs\_predicted = test\_dt.map(lambda p: (p.label, linear\_model.predict(p.features)))  print "Linear Model predictions: " + str(true\_vs\_predicted.take(10)) |

Next table shows model result comparing actual values with those provided by the model

|  |
| --- |
| Linear Model predictions: [(150.0, 92.347297046995223), (150.0, 88.281692774486004), (190.0, 113.2561190198998), (175.0, 111.2233168836452), (160.0, 98.736103760938292), (140.0, 87.700892164127538), (215.0, 104.54410986452289), (193.0, 88.281692774486004), (90.0, 40.656042725092234), (175.0, 116.1601220716921)] |

Calculate two standard measures of model accuracy.

|  |
| --- |
| def squared\_error(actual, pred):  return (pred - actual)\*\*2  def abs\_error(actual, pred):  return np.abs(pred - actual)  mse = true\_vs\_predicted.map(lambda (t, p): squared\_error(t, p)).mean()  mae = true\_vs\_predicted.map(lambda (t, p): abs\_error(t, p)).mean()  print "Mean Square Error (MSE): %f" %mse  print "Mean Absolute Error (MAE): %f" %mae |

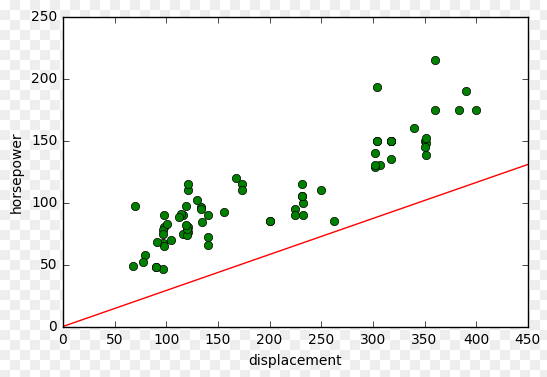
Error measure results obtained comparing actual data to what predicted by the model

|  |
| --- |
| Mean Square Error (MSE): 2680.795431  Mean Absolute Error (MAE): 48.550393 |

Create a diagram using any technique of convenience to presents the model (straight line), and the original test data. Please label your axes and use different colors for original data and predicted data.

|  |
| --- |
| import matplotlib.pyplot as plt  x\_model=[0, 450]  y\_model = [linear\_model.intercept, linear\_model.intercept + 450 \* linear\_model.weights[0]]  plt.plot(x\_model, y\_model, color='red')  print x\_model  plt.xlabel('displacement')  plt.ylabel('horsepower')  x\_real = test\_data.map(lambda v: float(v[2]))  y\_real = test\_data.map(lambda v: float(v[3]))  plt.plot (x\_real.collect(), y\_real.collect(), "go")  plt.show() |

Plotting result:



**Problem 2**. Consider the entire data set. Regard mpg as the target variable and all other variables as features. Please note that some of those are categorical variables. Identify categorical variables and use 1-of-k binary encoding for those variables.

|  |
| --- |
| def get\_mapping(rdd, idx):  return rdd.map(lambda fields: fields[idx]).distinct().zipWithIndex().collectAsMap()  data\_ord = records.filter(lambda v: v[3]<>"NA" and v[0]<>"NA")\  .map(lambda v: (v[1],v[6],v[7],v[8],v[2],v[3],v[4],v[5],v[0]))  print data\_ord.take(1)  mappings = [get\_mapping(data\_ord, i) for i in range(0,4)]  cat\_len = sum(map(len, mappings))  num\_len = num\_len = len(data\_ord.first()[4:8])  total\_len = num\_len + cat\_len  def extract\_features(record):  cat\_vec = np.zeros(cat\_len)  i = 0  step = 0  for field in record[0:4]:  m = mappings[i]  idx = m[field]  cat\_vec[idx + step] = 1  i = i + 1  step = step + len(m)  num\_vec = np.array([float(field) for field in record[4:8]])  return np.concatenate((cat\_vec, num\_vec))    def extract\_label(record):  return float(record[8])  data\_bin = data\_ord.map(lambda r: LabeledPoint(extract\_label(r),extract\_features(r)))  print data\_bin.take(1)  first\_point = data\_bin.first()  print "Label: " + str(first\_point.label)  print "Linear Model feature vector:\n" + str(first\_point.features)  print "Linear Model feature vector length: " + str(len(first\_point. features)) |

Result of previous code execution:

|  |
| --- |
| [LabeledPoint(18.0, [1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,307.0,130.0,3504.0,12.0])]  Label: 18.0  Linear Model feature vector:  [1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,307.0,130.0,3504.0,12.0]  Linear Model feature vector length: 63 |

Train your model using LinearRegressionSGD method. Use test data to assess quality of prediction for mpg variable.

|  |
| --- |
| #from the binary encoded DDR, sample 20% for training and rest for test  records\_ord\_with\_idx = data\_bin.zipWithIndex().map(lambda (k, v): (v, k))  test\_data\_ord\_idx = records\_ord\_with\_idx.sample(False, 0.2, 42)  training\_data\_ord\_idx = records\_ord\_with\_idx.subtractByKey(test\_data\_ord\_idx)  test\_data\_ord\_bin = test\_data\_ord\_idx.map(lambda (idx, p) : p)  training\_data\_ord\_bin = training\_data\_ord\_idx.map(lambda (idx, p) : p)  linear\_model\_bin = LinearRegressionWithSGD.train(training\_data\_ord\_bin, iterations=200,step=0.000001, intercept=False)  print linear\_model\_bin |

Linear model approximation as per the intercept and theta values:

|  |
| --- |
| (weights=[-4.21008505334e-05,5.33311742477e-07,7.45910120355e-07,9.47592829671e-05,-4.9269245261e-06,1.41319213125e-06,2.8411482803e-06,-2.9034811825e-07,3.34579127888e-06,-1.07767545496e-05,-3.42232820761e-06,1.32074063019e-06,-2.90821773285e-06,1.65854784981e-05,1.76425234945e-05,1.29541462769e-05,4.42163777876e-06,5.88372002577e-06,-2.0617097041e-05,4.54574306275e-05,2.41703962308e-05,1.53683622298e-07,-2.42449018113e-06,1.89577044782e-06,3.66381074659e-06,1.95919773287e-07,-4.20583232877e-07,7.73583368314e-07,2.51203812411e-06,-1.35701611292e-06,-3.5340634894e-06,3.02179736914e-07,9.68014411037e-07,7.71622345131e-07,-2.42264177469e-06,-6.50318482397e-07,-2.98150849445e-06,1.04631755584e-05,6.9417435362e-07,-3.51798133598e-06,5.27432147954e-06,2.37624519327e-07,0.0,5.8632379819e-07,3.13626941682e-07,8.53350160274e-06,8.32965803335e-07,3.09078522754e-06,-2.18157505799e-06,1.58927098801e-06,8.59637932487e-07,1.14062830764e-05,4.53141310936e-07,-2.90169099133e-07,-3.57516592818e-07,-1.55303529933e-06,1.43526032005e-05,5.37597085769e-08,7.23810861505e-07,-0.006638890404,-0.000441874519887,0.00735800828307,0.000965433013973], intercept=0.0) |

Calculate performance metrics of your model.

|  |
| --- |
| true\_vs\_predicted\_bin = test\_data\_ord\_bin.map(lambda p: (p.label, linear\_model\_bin.predict(p.features)))  mse\_bin = true\_vs\_predicted\_bin.map(lambda (t, p): squared\_error(t, p)).mean()  mae\_bin = true\_vs\_predicted\_bin.map(lambda (t, p): abs\_error(t, p)).mean()  print "Mean Square Error (MSE): %f" %mse\_bin  print "Mean Absolute Error (MAE): %f" %mae\_bin |

And results:

|  |
| --- |
| Mean Square Error (MSE): 150.737088  Mean Absolute Error (MAE): 10.257624 |

**Problem 3**. Repeat the above analysis in Problem 2 with the decision tree method.

First need to create feature vector for the decision tree. I have removed car model to proceed with the exercise, because I was facing some issues using a dictionary instead of the string values themselves

|  |
| --- |
| def get\_mapping\_dt(rdd, idx):  return rdd.map(lambda fields: fields[idx]).distinct().zipWithIndex().collectAsMap()  #convert car string variable to numeric  print "Mapping of first categorical feature column: %s" % get\_mapping(data\_ord, 3)  def extract\_features\_dt(record):  return np.array(map(float, record[0:7]))  def extract\_label\_dt(record):  return float(record[7])    #removed car model from the feauture variables, I could manage to add the dictionary to the RDD  data\_tree\_no\_car\_type = data\_ord.map(lambda v: (v[0],v[1],v[2],v[4],v[5],v[6],v[7],v[8]))  data\_tree = data\_tree\_no\_car\_type.map(lambda r: LabeledPoint(extract\_label\_dt(r), extract\_features\_dt(r)))  first\_point\_tree = data\_tree.first()  print "Decision Tree feature vector: " + str(first\_point\_tree.features)  print "Decision Tree feature vector length: " + str(len(first\_point\_tree.features)) |

Next create the decision tree model using training data and validation using test data

|  |
| --- |
| from pyspark.mllib.tree import DecisionTree  #from the RDD sample 20% for training and rest for test  records\_tree\_with\_idx = data\_tree.zipWithIndex().map(lambda (k, v): (v, k))  test\_tree\_idx = records\_tree\_with\_idx.sample(False, 0.2, 42)  training\_tree\_idx = records\_tree\_with\_idx.subtractByKey(test\_tree\_idx)  test\_tree = test\_tree\_idx.map(lambda (idx, p) : p)  training\_tree = training\_tree\_idx.map(lambda (idx, p) : p)  model\_tree = DecisionTree.trainRegressor(training\_tree,{})  preds\_tree = model\_tree.predict(test\_tree.map(lambda p: p.features))  actual\_tree = test\_tree.map(lambda p: p.label)  true\_vs\_predicted\_tree = actual\_tree.zip(preds\_tree)  print "Decision Tree predictions: " + str(true\_vs\_predicted\_tree.take(5))  print "Decision Tree depth: " + str(model\_tree.depth())  print "Decision Tree number of nodes: " + str(model\_tree.numNodes()) |

Compare quality of the decision tree model and the linear regression technique using Performance metrics.

|  |
| --- |
| mse\_tree = true\_vs\_predicted\_tree.map(lambda (t, p): squared\_error(t, p)).mean()  mae\_tree = true\_vs\_predicted\_tree.map(lambda (t, p): abs\_error(t, p)).mean()  print "Mean Square Error(MSE): Linear Regression %f" %mse\_bin + "| Decision Tree %f" %mse\_tree  print "Mean Absolute Error (MAE): Linear Regression %f" %mae\_bin + "| Decision Tree %f" %mae\_tree |

Result of the previous code that clearly highlights that decision tree is a better approximation than linear regression model

|  |
| --- |
| Mean Square Error(MSE): Linear Regression 150.737088| Decision Tree 13.420670  Mean Absolute Error (MAE): Linear Regression 10.257624| Decision Tree 2.684123 |

**Problem 4.** Now that your code works, investigate the impact of different parameter settings on model performance. Work with the linear regression model. Collect RMSLE

for several values of the number of iterations and plot those values as a graph.

Calculate the RMSL error for the first linear regression model calculated in Problem 1

|  |
| --- |
| import math  def squared\_log\_error(pred, actual):  return (np.log(pred + 1) - np.log(actual + 1))\*\*2  rmsle = np.sqrt(true\_vs\_predicted.map(lambda (t, p): squared\_log\_error(t,p)).mean())  print rmsle |

And the RMSLE result:

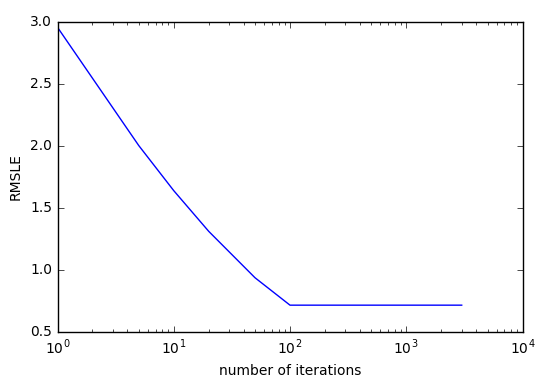
|  |
| --- |
| 0.715926436898 |

Now we recalculate the RMSLE using different parameters to feed the model. Using regularization L2

|  |
| --- |
| def evaluate(train, test, iterations, step, regParam,regType,intercept):  model = LinearRegressionWithSGD.train(train,iterations,step,regParam=regParam, regType=regType, intercept=intercept)  tp = test.map(lambda p: (p.label, model.predict(p.features)))  rmsle = np.sqrt(tp.map(lambda (t, p): squared\_log\_error(t,p)).mean())  return rmsle  train\_rmsle = training\_data.filter(lambda v: v[3]<>"NA").map(lambda r: LabeledPoint(float(r[3]),np.array(r[2:3])))  test\_rmsle = test\_data.filter(lambda v: v[3]<>"NA").map(lambda r: LabeledPoint(float(r[3]),np.array(r[2:3])))    #number of iterations  params = [1, 5, 10, 20, 50, 100, 500, 1000, 2000, 3000]  metrics = [evaluate(train\_rmsle, test\_rmsle, param, 0.000001, 0.0, 'l2',False) for param in params]  plt.plot(params, metrics)  plt.xscale('log')  plt.xlabel('number of iterations')  plt.ylabel('RMSLE') |

Present the number of iterations as the x-axis. Use the log scale. Present RMSLE as the y-axis. For y-axis use the linear scale. You can use any plotting tool including Excel.

Next graph shows the result as per the above code:

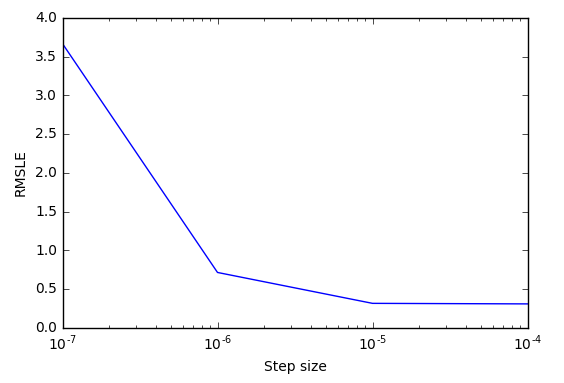


It seems from the previous that the model reaches a point (100 iterations) where increasing the number of iterations does not improve the model, since the error remains the same.

Subsequently, once/if you find the most optimal value of the number of iterations, vary the step size and again plot RMSLE on the y-axis and the logarithm of the step size on the x-axis.

The optimum number if iterations seems to be 100, since more iterations don’t reduce error. Same process for step as a parameter considering 100 iterations

|  |
| --- |
| params\_step = [0.0000001, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1, 1.0]  metrics\_step = [evaluate(train\_rmsle, test\_rmsle, 100, param, 0.0, 'l2',False) for param in params\_step]  plt.plot(params\_step, metrics\_step)  plt.xscale('log')  plt.xlabel('Step value')  plt.ylabel('RMSLE') |



Try to find an optimal step size. In production environment you would go back and forth between those two analyses. We do not expect you to do that here. Also, we do not expect you to sweet this out. Several data points on each graph is fine.

According to the 2 previous analysis the optimum step size corresponds to a step size of 10-5 combined with 100 iterations.