## Owen Galvin

## HU Extension Assignment 11 E63 Big Data Analytics

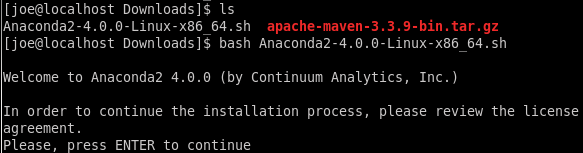
### Handed out: 04/16/2016 Due by 11:30PM EST on Friday, 04/22/2016

Please, describe every step of your work and present all intermediate and final results in a Word document. Please, copy past text version of all essential command and snippets of results into the Word document with explanations of the purpose of those commands. We cannot retype text that is in JPG images. Please, always submit a separate copy of the original, working scripts and/or class files you used. Sometimes we need to run your code and retyping is too costly. Please include in your MS Word document only relevant portions of the console output or output files. Sometime either console output or the result file is too long and including it into the MS Word document makes that document too hard to read. PLEASE DO NOT EMBED files into your MS Word document. For issues and comments visit the class Discussion Board. You are not obliged to use Java or Eclipse. You are welcome to use any language and any IDE of your choice.

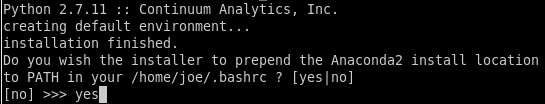
**Problem 1.** Remove the header of the attached Samll\_Car\_Data.csv file and then import it into Spark. Randomly select 10% of you data for testing and use remaining data for training. Look initially at horsepower and displacement. Treat displacement as a feature and horsepower as the target variable. Use MLlib linear regression to identify the model for the relationship. Use test data to illustrate accuracy of your ability to predict the relationship. Create a diagram using D3 which presents the model (straight line), original test data and predictions of your analysis. Please label your axes and use different colors for original data and predicted data.

**Solution:**

**Using the CentOS 6.7 VM created in an earlier assignment, first install the Anaconda2 4.0 package I downloaded, which includes Python 2.7, same as the package used in the lecture pdf. Install that using the bash command. Wind up going with default/recommended path of /home/joe/anaconda2 to avoid any permissions errors.**



**Answer yes to prepend the Anaconda2 path to PATH in .bashrc. My googling indicates this might allow being able to skip putting into .bash\_profile.**



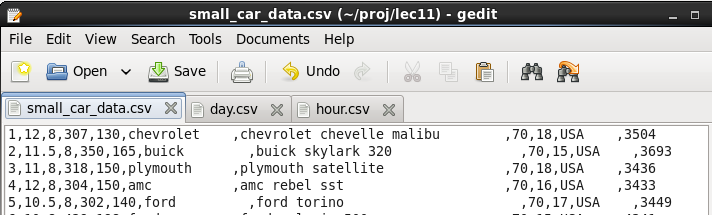
**Update .bash\_profile with necessary IPython export**

|  |  |
| --- | --- |
|  | **Though I later change the IPTYHON\_OPTS and do most of the testing and ad-hoc work in browser via Jupyter notebook.** |

**After saving, source the edited .bash\_profile.**



**Dataset = small\_car\_data.csv on my VM, with header removed, easiest to do manually in gpedit. And delete extraneous empty line at the end. Also, change the NaN Horsepower value for 1982 AMC Concord to a real value, 82 according to my internet research.**



**As mentioned earlier, most of the initial development work was created in Jupyter/IPython notebook.**

|  |
| --- |
|  |

**But the final code being submitted will be a single python file named a11\_p1.py. The first part, below is straightforward. The standard import to get access to pyspark and then for LabeledPoint and LinearRegressionWithSGD modules. Also import numpy with standard np alias. The series of defined functions are taken directly from the lecture pdf so there isn’t much reason to discuss them, suffice to say each is used in the creating measures of the model’s accuracy. The evaluate() function will be accessed multiple times in order to determine optimal iteration and step parameters – quite possible RMSLE is not best measure of model’s accuracy given the source data but it is good enough for this purpose.**

|  |
| --- |
| from pyspark import SparkConf, SparkContext  from pyspark.mllib.regression import LabeledPoint  from pyspark.mllib.regression import LinearRegressionWithSGD  import numpy as np  def squared\_log\_error(pred, actual):  return (np.log(pred + 1) - np.log(actual + 1))\*\*2  def squared\_error(actual, pred):  return (pred - actual)\*\*2  def abs\_error(actual, pred):  return np.abs(pred - actual)  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 |

**Create the standard conf & sc variables used in previous pyspark assignments. The .csv is located in a local file path on my VM and is loaded in as the raw\_data RDD. The records RDD is the result of splitting each line on comma in order to create a list of values for each row. I also remove the whitespace, which won’t matter to the model but makes for easier troubleshooting. Pull out the first member of the RDD and display to make sure we are on the right track. I’ve included a screenshot of that print, where we can see everything looks good, encoded as a list of Unicode strings.**

|  |
| --- |
| conf = SparkConf().setMaster('local').setAppName('pyAssign11\_Simple')  sc = SparkContext(conf = conf)  file\_path = 'file:////home/joe/proj/lec11/small\_car\_data.csv'  raw\_data = sc.textFile(file\_path)  records = raw\_data.map(lambda x: [cell.strip() for cell in x.split(',')])  first = records.first()  print first |
|  |

**Now create the primary data RDD, which will consist of a list of LabeledPoint() objects. Each LabeledPoint() has a Label value = index 4 of the record/csv = horsepower, the target value we are trying to predict. The second value is a vector of features, in this case only the single feature = index 3 of the record/csv = displacement. The commented out line works just as well, whatever implicit conversion necessary taking place behind the scenes, but the live line is more like from the lecture and no doubt better for larger datasets. Call .cache on the RDD as it will be accessed multiple times. Display the first 5 members of the data RDD and see the series of LabeldPoint objects, with label for hp and single-member array of features.**

|  |
| --- |
| #data = records.map(lambda r: LabeledPoint(float(r[4]), [r[3]] ))  data = records.map(lambda r: LabeledPoint(float(r[4]), np.array( [float(r[3])] )))  data.cache()  print data.take(5) |
|  |

**The following lines map to what was in the lecture pdf, except that I use .13 as the “fraction” value for the test dataset size. I wasn’t able to research .sample() method of RDD much but in the end .13 is what returned 10 records, i.e. 10% of 100, given a seed = 63. And then we wind up with 90 records in train\_data and 10 in test\_data**

|  |
| --- |
| data\_with\_idx = data.zipWithIndex().map(lambda (k, v): (v, k))  test = data\_with\_idx.sample(False, 0.13, 63) #.13 WAS working to get 10..., for 10% ..., 63 is the seed  train = data\_with\_idx.subtractByKey(test)  train\_data = train.map(lambda (idx, p): p) #train\_size = train\_data.count() = 90  test\_data = test.map(lambda (idx, p) : p) #test\_size = test\_data.count() = 10 |

**Below illustrates how I came up with good step and iteration values, by passing in a series of values. For step\_params, I first came up with a value that returned at least a reasonable value and then went up and down from there. In the end, 0.00001 gave the best Root Mean Squared Logarithmic Error value, as returned by the evaluate() error value. Similar with iteration selection, doing more than 20 didn’t seem to produce any benefit.**

|  |
| --- |
| #step\_params = [0.0000001,0.000001,0.00001,0.0001,0.0001]  #step\_metrics = [evaluate(train\_data, test\_data, 10, step\_param, 0.0, 'l2',False) for step\_param in step\_params]  #print step\_metrics #-> [3.4679800804243932, 1.5228864259213952, 0.28894714450593106, nan, nan]  #iter\_params = [1,5,10,20,50,100] #20 good enough = 0.27639320673299744  #iter\_metrics = [evaluate(train\_data, test\_data, iter\_param, 0.00001, 0.0, 'l2',False) for iter\_param in iter\_params]  #print iter\_metrics #-> [0.75783871278320125, 0.33747797400340473, 0.28894714450593106, 0.27639320673299744, 0.27639320673299744, 0.27639320673299744] |

**Next bit covers the creation of the linear model along with the true\_vs\_predicted RDD from the lecture pdf. The latter can then be submitted to the various pdf functions created earlier in order to get various measures of the model’s accuracy. I also create a tvp\_with\_input RDD that will hold the first/only feature, i.e. displacement, followed by the actual measurement and then also the prediction. Most of the code below is directly from the lecture pdf, accuracy measure printouts will be included in the main terminal screenshot in a few paragraphs.**

|  |
| --- |
| linear\_model = LinearRegressionWithSGD.train(train\_data, iterations=20, step=0.00001, intercept=False) #, regType='l1')  true\_vs\_predicted = test\_data.map(lambda p: (p.label, linear\_model.predict(p.features)))  tvp\_with\_input = test\_data.map(lambda p: (p.features[0], p.label, linear\_model.predict(p.features)))  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()  rmsle=np.sqrt(true\_vs\_predicted.map(lambda(t,p):squared\_log\_error(t,p)).mean())  print "Linear Model - Mean Squared Error: %2.4f" % mse  print "Linear Model - Mean Absolute Error: %2.4f" % mae  print "Linear Model - Root Mean Squared Log Error: %2.4f" % rmsle |

**The last part of the code basically prints the contents of the tvp\_with\_input RDD I had created, so that I have a dataset to use in D3. The printout includes a header row and comma delimiters and since it is only 10 rows, I do a simply copy and paste into a file in order to create D3 data, which along with the linear\_model.weights value is discussed later.**

|  |
| --- |
| #print true\_vs\_predicted.take(2) #[(130.0, 151.90443654598366), (165.0, 173.18095371691948)]  tvp = tvp\_with\_input.collect()  lec11\_p1\_csv = 'displacement,actual,predicted\r\n' + '\r\n'.join(['{0},{1},{2:.4f}'.format(tup[0], tup[1], tup[2]) for tup in tvp])  print lec11\_p1\_csv  print linear\_model.weights |

**To execute the code, pass in to spark-submit while in directory holding the a11p1\_simple.py file. Below includes full display of all output.**

|  |
| --- |
|  |

**To create the dataset for the D3 visualization, I simply copied the screen output delivered by printing the lec11\_p1\_csv variable. Looks like below, will also be included in the .zip upload as lec11\_p1.csv.**

|  |
| --- |
| displacement,actual,predicted  455.0,225.0,223.7321  200.0,85.0,98.3438  318.0,210.0,156.3666  250.0,105.0,122.9297  91.0,53.0,44.7464  97.0,75.0,47.6967  112.0,88.0,55.0725  151.0,90.0,74.2496  91.0,67.0,44.7464  232.0,112.0,114.0788 |

**I used the bubble chart solution from Assignment09 as a template, so things will look similar to that viz in certain respects. Following will review the javascript code in the html file as it pertains to this solution, beginning with the three part d3.csv() function:**

1. **Path to the csv data above**
2. **The accessor function, for any data object creation and manipulation – none needed here**
3. **The callback function, where everything important happens once the csv is loaded. Everything following will take place in here**

**The callback contents begin with some hard-coded settings for width, height etc., same used for previous assignment. The scales are different though, beginning with the fact that both are simple/linear. And of course the number ranges are different, leading to different domain ranges. Set xMin & xMax variables here because I’ll need them later when creating the line for the linear model.**

|  |
| --- |
| d3.csv('../lec11\_p1.csv', function (d) {  return d;  }, function (data) {  var width = 1150, height = 555,  margin = { top: 20, right: 20, bottom: 20, left: 70 };  //uncomment below for scaled-down version, easier to work with on smaller screen  //width = 750; height = 400;  //horizontal/GDP scale  var xMin = 50, xMax=500;  var xScale = d3.scale.linear()  .domain([xMin, xMax])  .range([50, width]);  var yScale = d3.scale.linear()  .domain([25, 250])  .range([height,40]) |

**The x and y axis are set with the applicable xScale/yScale just created above. The tick values are encoded using reasonable values given the data – perhaps the default would have been good enough.**

|  |
| --- |
| var xAxis = d3.svg.axis().scale(xScale).orient('bottom')  .tickValues([100,200,300,400,500]);  var yAxis = d3.svg.axis().scale(yScale).orient('left')  .tickValues([50, 100, 150, 200, 250]); |

**Make a box frame around the entire viz and then add in the text labels for the axes, independent variable = Displacement on x axis and Horsepower on y.**

|  |
| --- |
| var svg = d3.select('body').append('svg')  .attr('width', width + margin.right + margin.left)  .attr('height', height + margin.top + margin.bottom)  .attr('class', 'frame');  svg.append("text").attr("x", 25).attr("y", 30)  .text("Displacement").attr('class', 'axis-label');  svg.append("text").attr("x", 45).attr("y", height + 35)  .text("Horsepower").attr('class', 'axis-label'); |

**The below is a bunch of hard coded values that will create a rectangle in the upper left and within place text descriptors along with circles of correct color to match the scatterplot**

|  |
| --- |
| //legend  svg.append("rect").attr("x", 80).attr("y", 80)  .attr("height", 50).attr("width",100).attr('class', 'legend');  svg.append("text").attr("x", 100).attr("y", 100)  .text("Actual").attr('class', 'axis-label');  svg.append("text").attr("x", 100).attr("y", 120)  .text("Predicted").attr('class', 'axis-label');  svg.append("circle").attr("cx", 90).attr("cy", 95)  .attr('r', 5).attr('class', 'actual');  svg.append("circle").attr("cx", 90).attr("cy", 115)  .attr('r', 5).attr('class', 'predicted'); |

**Create circle objects with radius = 5, based off of the x and y data within the csv, where the x value = displacement is the same for both sets of dots but different for actual vs. predicted. The below is for the actual/true values that match the displacement. The css will color them in as light blue. Hovering over the dot will display the values behind each dot/circle.**

|  |
| --- |
| var dot\_actual = svg.selectAll('circle-actual')  .data(data).enter().append('circle')  .attr('class', function (d) {  return 'actual';  });  dot\_actual.attr('cx', function (d) {  return xScale(d.displacement);  })  .attr('cy', function (d) {  return yScale(d.actual);  })  .attr('r', function (d) {  return 5;  })  .append('title')  .text(function (d) {  return Math.round(d.displacement) + ' displacement, actual hp: ' + Math.round(d.actual);  }); |

**Same but for values as predicted by the linear model, will be in red.**

|  |
| --- |
| var dot\_pred = svg.selectAll('circle-predicted')  .data(data).enter().append('circle')  .attr('class', function (d) {  return 'predicted';  });  dot\_pred.attr('cx', function (d) {  return xScale(d.displacement);  })  .attr('cy', function (d) {  return yScale(d.predicted);  })  .attr('r', function (d) {  return 5;  })  .append('title')  .text(function (d) {  return Math.round(d.displacement) + ' displacement, predicted hp: ' + Math.round(d.predicted);  }); |

**To create the line I first determined the approximate slope by taking x & y values from two points + some quick and dirty algebra to come up with a slope of around 0.49. I figured the linear\_model object from python would have the info encoded somewhere, though I couldn’t find much documentation. Still, with only a couple of members available, it appears that .weights had what I needed – apparently an array and for simple regression only had one value = 0.491718981427. Plugging that into x values resulted in predicted y’s, so multiplied it by the minimum domain value from the xScale to get beginning y1 on lower left &by xMax to get ending y2 upper right of svg. Combined with xMin/xMax as the x1/x2 values, get the beginning and ending points in the line that D3 draws.**

|  |
| --- |
| //some quick & dirty algebra gave me a slope = 0.49  //linear\_model.weights = [0.491718981427]  var model\_weights = 0.491718981427;  svg.append("line").attr("x1", xScale(xMin)).attr("y1", yScale(xMin \* model\_weights))  .attr("x2", xScale(xMax)).attr("y2", yScale(xMax \* model\_weights))  .attr('class', 'model'); |

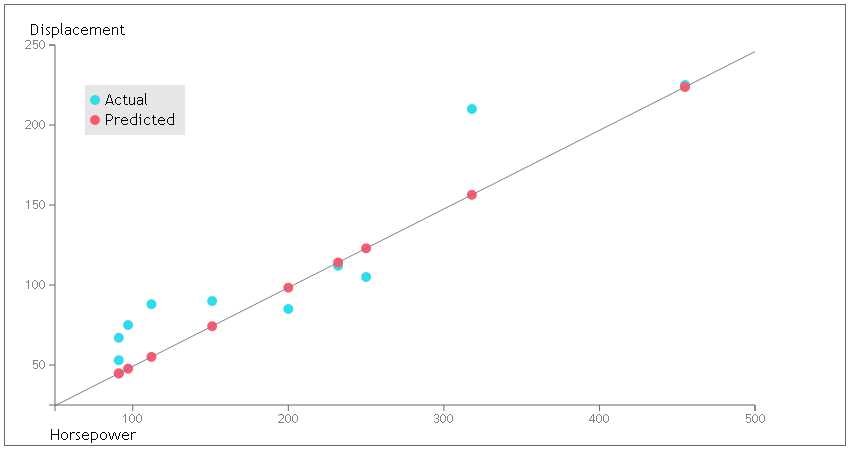
**Last bit of javascript is what draws the x & y axes onto the primary svg canvas, where xAxis & yAxis objects were created earlier. I found they were not being displayed when drawn earlier but work fine when done at the end, here.**

|  |
| --- |
| //keep these at end  svg.append('g')  .attr('class', 'axis')  .attr('transform', 'translate(0,' + height + ')')  .call(xAxis);  svg.append('g')  .attr('class', 'axis')  .attr('transform', 'translate(50,0)')  .call(yAxis);  }); |

**As with Assigment 9, I’ll let the inline comments in the css (contents of lec11\_p1.css) below describe what each section is for.**

|  |
| --- |
| /\*simple border around edge of primary svg\*/  .frame {  border: 1px solid dimgray;  }  /\*the actual x/y axes lines \*/  .axis {  fill: none;  stroke: dimgray;  stroke-width: 1;  }  /\*affects the tick mark values, i.e. 100,200,300, etc. on x axis\*/  .axis text {  font-family: 'Trebuchet MS', 'Lucida Sans Unicode', 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;  font-size: small;  fill: gray;  stroke: none;  }  /\*same with the axes text labels, e.g. 'Horsepower' \*/  text.axis-label {  fill: black;  font-size: medium;  font-family: 'Trebuchet MS', 'Lucida Sans Unicode', 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;  }  /\*predicted value dots\*/  circle.predicted {  fill: #FF5872; /\*red\*/  }  /\*actual/true value dots\*/  circle.actual {  fill: #30DCEC; /\*blue\*/  }  /\*square around legend\*/  rect.legend {  opacity: 0.10;  }  /\*linear model straight line\*/  line.model {  stroke: gray;  stroke-width: 1;  } |

**Here is the D3 output.**



**Problem 2**. Treat: cylinders, displacement, manufacturer, model\_year, origin and weight as features and use linear regression to predict two target variable: horsepower and acceleration. Please note that some of those are categorical variables. Use test data to assess quality of prediction for both target variables. Which of two target variables is easier to predict, in the sense that predicted values differ less from the original values.

**Solution:**

**This solution is pretty much review of code + screen output. Some of the code is the same from Problem 1 and of course same as from lecture pdf and I will attempt to point out all differences.**

**Same imports as before, get\_mapping is directly from pdf, where it creates a list of dict objects, with each dict holding the contents of a specific category - unique string values as keys and the index of the key in the ordering of categories as value, so each string gets a unique numerical value assigned. The extract\_features is very similar to lecture pdf, with two changes to adapt for the non-contiguous categorical and numerical variables. In terms of the categories there is a cat\_idx list of index values (defined later) related to the categorical variables in the csv. Through a list comprehension create a list of those, same thing is done later on with numerical variables, which likewise refer to a num\_idx (list of csv/data index values). As with elsewhere in the program, various variables are crudely defined at global scope, refinement will have to wait.**

|  |
| --- |
| from pyspark import SparkConf, SparkContext  from pyspark.mllib.regression import LabeledPoint  from pyspark.mllib.regression import LinearRegressionWithSGD  import numpy as np  def get\_mapping(rdd, idx):  return rdd.map(lambda fields: fields[idx]).distinct().zipWithIndex().collectAsMap()  def extract\_features(record):  cat\_vec = np.zeros(cat\_len)  i = 0  step = 0  #for field in record[2:9]:  fields = [record[j] for j in cat\_idx]  for field in fields:  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[10:13]])  num\_vec = np.array([float(record[j]) for j in num\_idx])  return np.concatenate((cat\_vec, num\_vec)) |

**There are two variants to the extract\_label() function introduced in lecture, one that casts the 2nd column in the record, acceleration, as a float and returns that as the target variable/label for the LabeledPoint. Then of course extract\_label\_hp does same for horsepower target variable. The next three functions, for measuring model accuracy, are same as from lecture & problem 1. The evaluate() is also something seen before in Problem 1. I’ll included commented out lines that represent the results of the evaluate() tests.**

|  |
| --- |
| def extract\_label\_acc(record):  return float(record[1])    def extract\_label\_hp(record):  return float(record[4])    def squared\_log\_error(pred, actual):  return (np.log(pred + 1) - np.log(actual + 1))\*\*2  def squared\_error(actual, pred):  return (pred - actual)\*\*2  def abs\_error(actual, pred):  return np.abs(pred - actual)  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 |

**The guts of the evaluate\_final function were seen in Problem 1 but here they have been consolidated so as to be callable for each of the target variables. The first set of lines are essentially straight from lecture, though with a different split for the sample. As described in Problem 1 I found a fraction=0.13 param w/seed = 63 gave me the 90/10 split I wanted. A linear model is created based on the 90-member train\_data RDD, created as subset of passed in data RDD of LabeledPoints specific to the given target variable. The remaining lines in the function, in regards to judging accuracy of the model, are from lecture pdf, making sure to create a true vs. predicted based passing the test\_data RDD to the linear\_model that was built off of the training set. I did add one additional print line, for Root Mean Squared Error, which is the measurement I’ve heard of before and the one that may make most sense to review as a general snapshot of model accuracy.**

|  |
| --- |
| def evaluate\_final(description, data):  data\_with\_idx = data.zipWithIndex().map(lambda (k, v): (v, k))  test = data\_with\_idx.sample(False, 0.13, 63) #.13 WAS working to get 10..., for 10% ..., 63 is the seed  train = data\_with\_idx.subtractByKey(test)  train\_data = train.map(lambda (idx, p): p) #train\_size = train\_data.count()  test\_data = test.map(lambda (idx, p) : p) #test\_size = test\_data.count()  #no need to adjust iter & step values for each dependent variable since they shared the same optimal values  linear\_model = LinearRegressionWithSGD.train(train\_data, iterations=5, step=0.0000001, intercept=False)  true\_vs\_predicted = test\_data.map(lambda p: (p.label, linear\_model.predict(p.features)))  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()  rmsle=np.sqrt(true\_vs\_predicted.map(lambda(t,p):squared\_log\_error(t,p)).mean())    print '{0} Linear Model - Mean Squared Error: {1:2.4f}'.format(description, mse)  print '{0} Linear Model - Root Mean Squared Error: {1:2.4f}'.format(description, np.sqrt(mse))  print '{0} Linear Model - Mean Absolute Error: {1:2.4f}'.format(description, mae)  print '{0} Linear Model - Root Mean Squared Log Error: {1:2.4f}'.format(description, rmsle) |

**Now for some of the run-code, first few lines are no different vs. Problem 1, with a new AppName.**

|  |
| --- |
| conf = SparkConf().setMaster('local').setAppName('pyAssign11\_Multiple')  sc = SparkContext(conf = conf)  file\_path = 'file:////home/joe/proj/lec11/small\_car\_data.csv'  raw\_data = sc.textFile(file\_path)  records = raw\_data.map(lambda x: [cell.strip() for cell in x.split(',')])  records.cache()  first = records.first() |

**Next are the definitions for the list of indexes that correspond to the explanatory variables. Putting Manufacturer & Origin as categorical is necessary as they are strings. I have a little experience with linear regression and my gut instinct was to put cylinders and Model\_Year as categories. To begin with there are only three of each, not enough to represent a true continuous variable. And then they just feel like categories, 4 cylinder vs. V6 vs. V8… what if the dataset contained a 1982 Mazda RX-7? - that car had a rotary engine, w/o any cylinders. Similar thinking for model year gas-guzzling tanks vs. early gas-crisis vs. adaptation to gas-crisis. Anyway, moving on…**

**Displacement and Weight are the only two continuous variables in each model. The rest of the lines below are from lecture pdf and mostly serve as a sanity check here, to make sure things are going as expected – that includes a numlen = the total of distinct counts for each of the categorical explanatory variables. Related screen output is below.**

|  |
| --- |
| cat\_idx = [2,5,7,9] #Cylinders, Manufacturer, Model\_Year, Origin  num\_idx = [3,10] #Displacement, Weight  mappings = [get\_mapping(records, i) for i in cat\_idx]  cat\_len = sum(map(len, mappings))  num\_len = sum(len(records.first()[i]) for i in num\_idx)  #total\_len = num\_len + cat\_len  print records.first()  print 'Feature vector length for categorical features: %d' % cat\_len  #40 cat\_len = distinct counts for each column: Mfg: 28, Origin: 6, Year: 3, Cylinders: 3 -- 28+6+3+3=40  print 'Feature vector length for numerical features: %d' % num\_len  print 'Total feature vector length: %d' % total\_len |
|  |

**Here we create two data RDDs of (LabeledPoint, [FeatureVector]), one that uses the Acceleration value as the Label/target variable and the other that uses Horsepower. Then some debug info is printed out for each. Raw data is the same of course, Label values for each RDD is printed, and finally info re the feature vector, which would be the same for both RDDs.**

|  |
| --- |
| data\_acc = records.map(lambda r: LabeledPoint(extract\_label\_acc(r), extract\_features(r)))  data\_hp = records.map(lambda r: LabeledPoint(extract\_label\_hp(r), extract\_features(r)))  first\_point\_acc = data\_acc.first()  first\_point\_hp = data\_hp.first()  print 'Raw data: ' + str(first)  print 'Acc Label: ' + str(first\_point\_acc.label)  print 'HP Label: ' + str(first\_point\_hp.label)  print 'Linear Model feature vector:\n' + str(first\_point\_acc.features)  print 'Linear Model feature vector length: ' + str(len(first\_point\_acc.features)) |
|  |

**Finally we pass in each of the RDDs to evaluate\_final(), which will generate linear model objects for each and print out relevant measurements regarding accuracy of the models.**

|  |
| --- |
| evaluate\_final('ACCELERATION',data\_acc)  evaluate\_final('HORSEPOWER',data\_hp) |

**The last bit is present as in Problem 1, where I wasn’t sure how else to display. Suffice to say that both models wound up optimizing on the same iteration and step values.**

|  |
| --- |
| #--------------------------------------------------------  #coming up with best step/iteration values for each model  #each returns differing rmsle for submitted step/iter values as expected for diff generated models  #but in the end step = 0.0000001 and iterations = 5 seems to work best for both acceleration and horsepower  ## dependent var = horsepower  #step\_params = [0.00000001,0.00000005,0.00000008,0.0000001,0.0000005,0.000001]  #step\_metrics = [evaluate(train\_data, test\_data, 10, step\_param, 0.0, 'l2',False) for step\_param in step\_params]  #print step\_metrics #[1.1006039362999183, 0.26902311044451516, 0.25271535442823323, 0.25005936125107098, nan, nan]  ##use step = 0.0000001, same as for acceleration    #iter\_params = [1,5,10,20,50,100] #5 good enough, same as with acceleration  #iter\_metrics = [evaluate(train\_data, test\_data, iter\_param, 0.0000001, 0.0, 'l2',False) for iter\_param in iter\_params]  #print iter\_metrics #[0.25022360622846102, 0.25005936125107098, 0.25005936125107098, 0.25005936125107098, 0.25005936125107098, 0.25005936125107098]  ### dependent var = acceleration  #step\_params = [0.00000001,0.00000005,0.00000008,0.0000001,0.0000005,0.000001]  #step\_metrics = [evaluate(train\_data, test\_data, 10, step\_param, 0.0, 'l2',False) for step\_param in step\_params]  #print step\_metrics #[1.7312475116799362, 0.68034979567703358, 0.46920254015648477, 0.41748612595024526, nan, nan]  ##use step = 0.0000001    #iter\_params = [1,5,10,20,50,100] #5 good enough = 0.41748612595024526  #iter\_metrics = [evaluate(train\_data, test\_data, iter\_param, 0.0000001, 0.0, 'l2',False) for iter\_param in iter\_params]  #print iter\_metrics #[0.4189442995988179, 0.41748612595024526, 0.41748612595024526, 0.41748612595024526, 0.41748612595024526, 0.41748612595024526] |

**Here is the full screen output of .py being run via spark-submit.**

|  |
| --- |
|  |

**I don’t have enough of a background in statistics/Spark MLLib knowledge to judge which accuracy measurement is most appropriate for this dataset and use of LinearRegressionWithSGD module but I think RMSE is the most intuitive~~. There we see that predicting Acceleration results in less of an error on average~~…. Ok, changed my mind, when I’ve dealt with regression before it was to come up with best feature selection etc. when composing a model, comparing various models that were trying to predict the same target variable. Seeing as these variables are being measured in entirely different units, at a minimum need to compare against the average for each variable (I know there are other scaling things that can be done but this should be good enough). If that is avg=15 for Acceleration, an RMSE of 5.88 is actually quite large. And RMSE of 23.3 is pretty good for Horsepower considering the dataset average = 112. So I’ll say Horsepower is easier to predict, a simple plot might make the answer obvious.**

**Either way I was going to skip RMSLE since I think it is only used in specific circumstances.**

**I did create some histograms in Jupyter/IPython to see that Acceleration at least had a reasonably normal distribution, Horsepower not so much. Trying to log transform the data was way too ambitious for this assignment given the overall amount of work.**

|  |
| --- |
| **Histogram re distribution of Acceleration values** |
|  |

|  |
| --- |
| **Horsepower histogram** |
|  |

**Problem 3**. Repeat above analysis with decision tree method. Compare predicting ability/quality of this technique with that of the linear regression.

**Solution:**

**As with Problem 2, begin with a code review, try to concentrate only on differences vs. Problem 2 and/or the lecture pdf.**

**Mostly standard imports and function defs that we’ve seen before. Of course importing module for DecisionTree model generation instead of LinearRegression. The extract\_features\_dt() is simpler, in that we don’t need to do the binary vector padding for decision trees and instead wind up with what is essentially a basic dense vector. Create the two category vs. number vector/lists separately since the categories need to be transformed into number equivalents by taking index values from the mappings list of dict objects. Additionally, it looks like a standard python list object was good enough for numpy concatenate and didn’t need to turn into a numpy vector beforehand.**

|  |
| --- |
| from pyspark import SparkConf, SparkContext  from pyspark.mllib.regression import LabeledPoint  from pyspark.mllib.tree import DecisionTree  import numpy as np  def get\_mapping(rdd, idx):  return rdd.map(lambda fields: fields[idx]).distinct().zipWithIndex().collectAsMap()  def extract\_features\_dt(record):  cat\_lst = []  fields = [record[j] for j in cat\_idx]  for idx,field in enumerate(fields):  cat\_lst.append(mappings[idx][field])  #cat\_vec = np.array([float(j) for j in cat\_lst]) #don't need it, np.concatenate takes raw lst    num\_idx = [3,10]  num\_vec = np.array([float(record[j]) for j in num\_idx])  return np.concatenate((cat\_lst, num\_vec))  def extract\_label\_acc(record):  return float(record[1])    def extract\_label\_hp(record):  return float(record[4])  def squared\_log\_error(pred, actual):  return (np.log(pred + 1) - np.log(actual + 1))\*\*2  def squared\_error(actual, pred):  return (pred - actual)\*\*2  def abs\_error(actual, pred):  return np.abs(pred - actual) |

**The primary evaluate\_final() acts as it did in Problem 2, doing the actual work. The first difference is that the function takes maxDepth and maxBins params since unlike the Linear models these two optimized on different param values for Acceleration target variable vs. Horsepower. The dt\_model now holds the decision tree model, created by passing in the training data along with the depth/bin values specific to the target variable. Everything else appears quite similar to equivalent function in Problem 1.**

|  |
| --- |
| def evaluate\_final(description, data, maxDepth, maxBins):  data\_with\_idx = data.zipWithIndex().map(lambda (k, v): (v, k))  test = data\_with\_idx.sample(False, 0.13, 63) #.13 WAS working to get 10..., for 10% ..., 63 is the seed  train = data\_with\_idx.subtractByKey(test)  train\_data = train.map(lambda (idx, p): p) #train\_size = train\_data.count()  test\_data = test.map(lambda (idx, p) : p) #test\_size = test\_data.count()  dt\_model = DecisionTree.trainRegressor(train\_data,{}, maxDepth=maxDepth, maxBins=maxBins)  preds = dt\_model.predict(test\_data.map(lambda p: p.features))  actual = test\_data.map(lambda p: p.label)  true\_vs\_predicted\_dt = actual.zip(preds)  print '\r\n-------- ' + description + ' ---------'  print "Decision Tree predictions: " + str(true\_vs\_predicted\_dt.take(5))  print "Decision Tree depth: " + str(dt\_model.depth())  print "Decision Tree number of nodes: " + str(dt\_model.numNodes())    mse\_dt = true\_vs\_predicted\_dt.map(lambda (t, p): squared\_error(t, p)).mean()  mae\_dt = true\_vs\_predicted\_dt.map(lambda (t, p): abs\_error(t, p)).mean()  rmsle\_dt = np.sqrt(true\_vs\_predicted\_dt.map(lambda (t, p): squared\_log\_error(t, p)).mean())    print 'Decision Tree -Mean Squared Error: {0:2.4f}'.format(mse\_dt)  print 'Decision Tree -Root Mean Squared Error: {0:2.4f}'.format(np.sqrt(mse\_dt))  print 'Decision Tree -Mean Absolute Error: {0:2.4f}'.format(mae\_dt)  print 'Decision Tree -Root Mean Squared Log Error: {0:2.4f}'.format(rmsle\_dt) |

**Everything except the last two lines is essentially the same vs. Problem 2. I kept the mappings object in there with print out because it helped me to get a handle as how that was working in terms of the category encoding.**

|  |
| --- |
| conf = SparkConf().setMaster('local').setAppName('pyAssign11\_DecisionTree')  sc = SparkContext(conf = conf)  file\_path = 'file:////home/joe/proj/lec11/small\_car\_data.csv'  raw\_data = sc.textFile(file\_path)  records = raw\_data.map(lambda x: [cell.strip() for cell in x.split(',')])  records.cache()  first = records.first()  cat\_idx = [2,5,7,9] #Cylinders, Displacement, Manufacturer, Model\_Year, Origin  num\_idx = [3,10] #Displacement, Weight  mappings = [get\_mapping(records, i) for i in cat\_idx]  print 'mappings: ' + str(mappings) |

**Create an RDD for predicting acceleration given the feature array, followed by one for predicting horsepower. Some debug printout to make sure everything is on track. Note that the vector length for decision tree will be = the number of explanatory variables, without needing the more complicated binary vectors of LinearRegressionWithSGD object.**

|  |
| --- |
| data\_dt\_acc = records.map(lambda r: LabeledPoint(extract\_label\_acc(r),extract\_features\_dt(r)))  data\_dt\_hp = records.map(lambda r: LabeledPoint(extract\_label\_hp(r),extract\_features\_dt(r)))  first\_point\_dt = data\_dt\_acc.first()  print "Decision Tree feature vector: " + str(first\_point\_dt.features)  print "Decision Tree feature vector length: " + str(len(first\_point\_dt.features)) |

**These final two lines are simply calling evaluate\_final, which will do all the heavy lifting given the applicable arguments for depth and bin count.**

|  |
| --- |
| #def evaluate\_final(description, data, maxDepth, maxBins):  evaluate\_final('ACCELERATION', data\_dt\_acc, maxDepth=5, maxBins=32)  evaluate\_final('HORSEPOWER', data\_dt\_hp, maxDepth=3, maxBins=8) |

**At the bottom of the .py file are several commented out lines that show the code used to determine the best depth/bin for each of the target variables. At the bottom of each block of code are a set of lines that can be used to draw a plot in Jupyter notebook, I plan to include that final output at the very end of the Word document**

|  |
| --- |
| #===============================================================================  # #######################  # #Acceleration, depth = 5  # train\_data\_dt, test\_data\_dt = create\_train\_test(data\_dt\_acc)  # depth\_params = [1, 2, 3, 4, 5, 10, 20]  # metrics = [evaluate\_dt(train\_data\_dt, test\_data\_dt, param, 32) for param in depth\_params]  #  # %matplotlib inline  # import matplotlib  # from pylab import plot  # print metrics #[0.092841717862485865, 0.089265498562083476, 0.11480498057663967, 0.10559226478908185, 0.088691632155098635, 0.13995172893623051, 0.13995172893623051]  # plot(depth\_params, metrics)  # fig = matplotlib.pyplot.gcf()  #  # #######################  # #Horsepower, depth = 3  # train\_data\_dt, test\_data\_dt = create\_train\_test(data\_dt\_hp)  # depth\_params = [1, 2, 3, 4, 5, 10, 20]  # metrics = [evaluate\_dt(train\_data\_dt, test\_data\_dt, param, 32) for param in depth\_params]  #  # %matplotlib inline  # import matplotlib  # from pylab import plot  # print metrics #[0.23444495934071619, 0.17013551304201496, 0.14229474701127939, 0.16098453079652439, 0.17897035995486035, 0.18641182106302645, 0.18603726827029393]  # plot(depth\_params, metrics)  # fig = matplotlib.pyplot.gcf()  #  # #######################  # #Acceleration, bins = 32  # train\_data\_dt, test\_data\_dt = create\_train\_test(data\_dt\_acc)  # bin\_params = [2, 4, 8, 16, 32, 64, 100]  # metrics = [evaluate\_dt(train\_data\_dt, test\_data\_dt, 5, param) for param in bin\_params]  # print metrics #[0.089726588821365555, 0.16188926584413466, 0.10092368423911695, 0.101790771551064, 0.088691632155098635, 0.1326161778087083, 0.1326161778087083]  #  # %matplotlib inline  # import matplotlib  # from pylab import plot  # plot(bin\_params, metrics)  # fig = matplotlib.pyplot.gcf()  #  # #######################  # #Horsepower, bins = 8  # train\_data\_dt, test\_data\_dt = create\_train\_test(data\_dt\_hp)  # bin\_params = [2, 4, 8, 16, 32, 64, 100]  # metrics = [evaluate\_dt(train\_data\_dt, test\_data\_dt, 5, param) for param in bin\_params]  # print metrics  #  # %matplotlib inline  # import matplotlib  # from pylab import plot  # plot(bin\_params, metrics)  # fig = matplotlib.pyplot.gcf()  #=============================================================================== |

**Below screenshot displays the result of passing the a11p3\_tree.py file on to spark**

|  |
| --- |
|  |

**I was going to compare the Root Mean Squared Error values in order to judge the two prediction approaches, linear regression vs. decision trees. Displaying the linear regression output again, we can see the Acceleration model had an RMSE of 5.88, as compared to 1.47 the decision tree was returning. Have to give that round to decision tree since the error measurement was lower. Linear model for Horsepower returned RMSE = 38.65, while the decision tree had 20.53, so again here also the decision tree is apparently a superior predictor.**

|  |
| --- |
|  |

**------**

**Below are screenshots showing the Jupyter output of the commented out evaluate() code.**

