BDAA-34717-Fall 2015

Final Project

John Ternent

**Problem Statement**

One of the recent areas within the Big Data X-Informatics space that has gotten a lot of attention and has been very fluid in recent months is the area of machine learning and predictive analytics on a Big Data stack. To date, most data scientists have had to sample and/or aggregate data from the big data ecosystem, download it into a desktop application such as R or SAS, and conduct data mining and predictive analytics on local machines.

The challenges to date have primarily centered around the parallelization of machine learning algorithms. Many of them either only had closed-form solutions, which requires solving sets of simultaneous linear equations, consequently requiring the entire matrix to be held in-memory (or at least locally, sacrificing performance to swap), or required so many iterations that computational complexity made distributed processing infeasible.

In recent years, new projects have emerged that allow the execution of machine learning algorithms directly in the cluster. The first in open source was Apache Mahout, which ran on top of the classical MapReduce Hadoop implementation. In recent months, a machine learning library called MLLib has emerged atop the Spark distributed processing framework. Because Spark relies on an in-memory directed acyclic graph model, iteration is up to 100x (according to spark.apache.org) faster than traditional MapReduce.

The objective of this project, then, is to configure a Spark/MLLib virtual environment, load some suitable data sets, and exercise a subset of the machine learning algorithms available in Spark MLLib.

Notation below includes 🖐 to highlight a crash, bug, or opportunity for improvement. ☞ signals a non-standard addition or workaround.

**Software Environment**

Using Spark and MLLib has gotten significantly easier in recent months. All the Hadoop distribution providers now incorporate Spark into their Sandbox configuration virtual machines, so the code can be executed on any of the vendor sandboxes. For this exercise, the only addition was the iPython Notebook system which makes for easier and more reproducible content. The instructions to configure iPython notebook for the Hortonworks Sandbox can be found at <http://hortonworks.com/hadoop-tutorial/using-ipython-notebook-with-apache-spark/> and are attached as an appendix to this document.

An alternative option is to use the Vagrant virtual machine management system to download a pre-configured sandbox for Spark vendor Databricks’s Spark massive online courses. A sample Vagrant configuration file is included in the appendix as well.

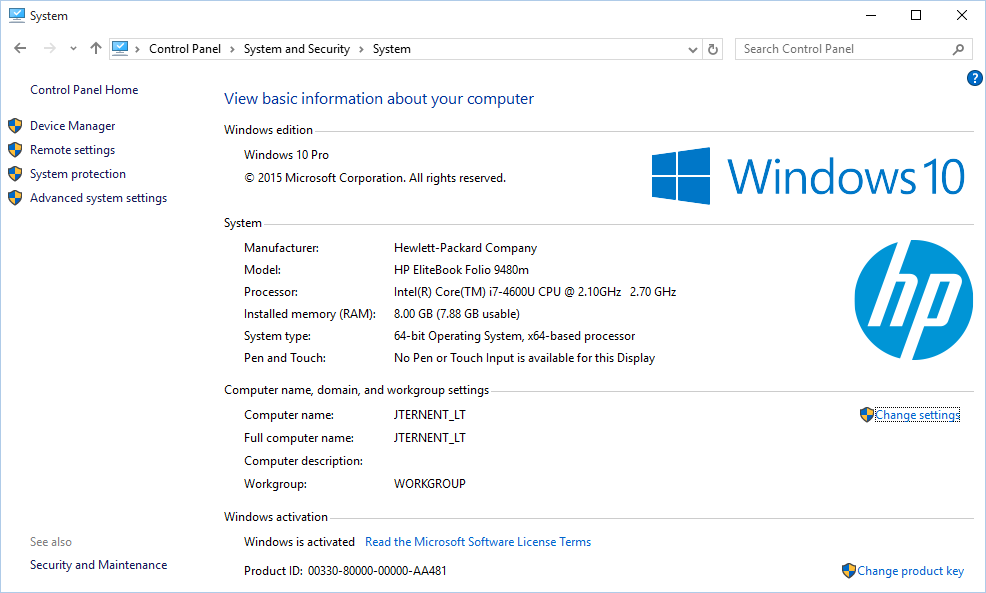
☞The only additional library used is a new python csv parser for Spark that can create DataFrames automatically and deals more gracefully with header rows in csv files than default pySpark behavior. PySpark-csv is available from github at : https://github.com/seahboonsiew/pyspark-csv. To use it, just upload the Python file to the Vagrant home directory, then import the file in the notebook as shown in step 1 below.

**Expediting Reproducibility**

In addition to the ipynb and data files, the project is also being stored as a raw Python file, which may make it easier to reproduce results on a vanilla Hortonworks Sandbox with Spark. The Python file is attached to this project as well.

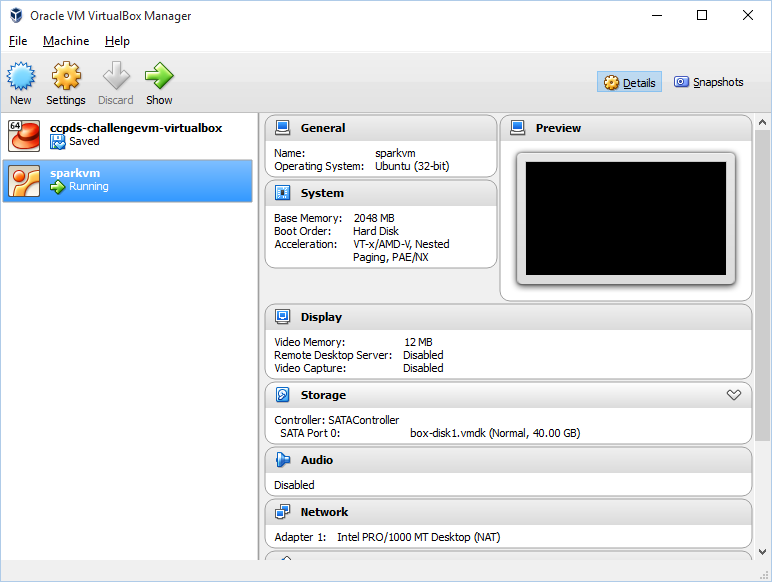
**Hardware Environment**

For the purposes of this exercise, Spark MLLib can be run on a single machine. These exercises were run on an HP Elitebook laptop computer. The relevant specifications are below, but it’s a relatively modest machine with 8GB of RAM and an Intel Core i7 processor.



The virtual machine environment itself is running on Oracle VirtualBox v. 5.0.8. The virtual machine instance was downloaded using the provided VagrantFile configuration file using Vagrant 1.7.4. By default, the user ID is vagrant and a private key is provided in the installation to authenticate to the server. The only need to connect to the server was to upload larger data files, which tend to crash the iPython notebook web user interface in the version provided.

The virtual machine specifications are shown below:



The machine itself is a 32-bit Ubuntu instance with 2GB of RAM. A 64-bit Ubuntu instance might make better use out of the underlying hardware, as might additional memory allocated to the virtual machine instance.

🖐For the purposes of this exercise, we did experience a persistent stack overflow error associated with the process that attempts to determine the best model hyperparameters for the Alternating Least Squares approach to building a recommender system. Online investigation highlighted the ability to use disk-based checkpointing to resolve this issue, but that did not help in this case. The parameters were hard-coded in the interest of illustrating the use of the recommender algorithm.

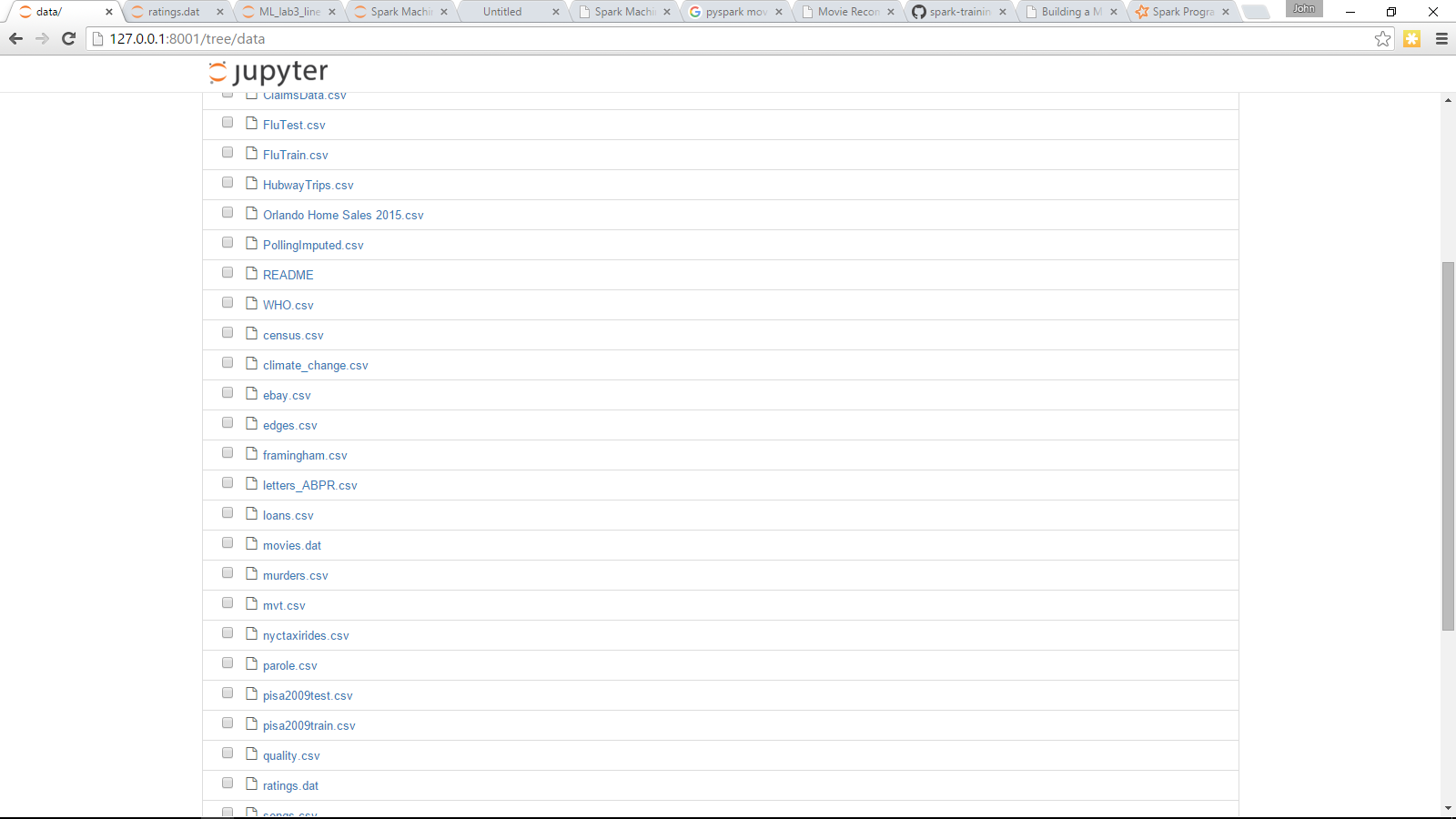
**Data Sources**

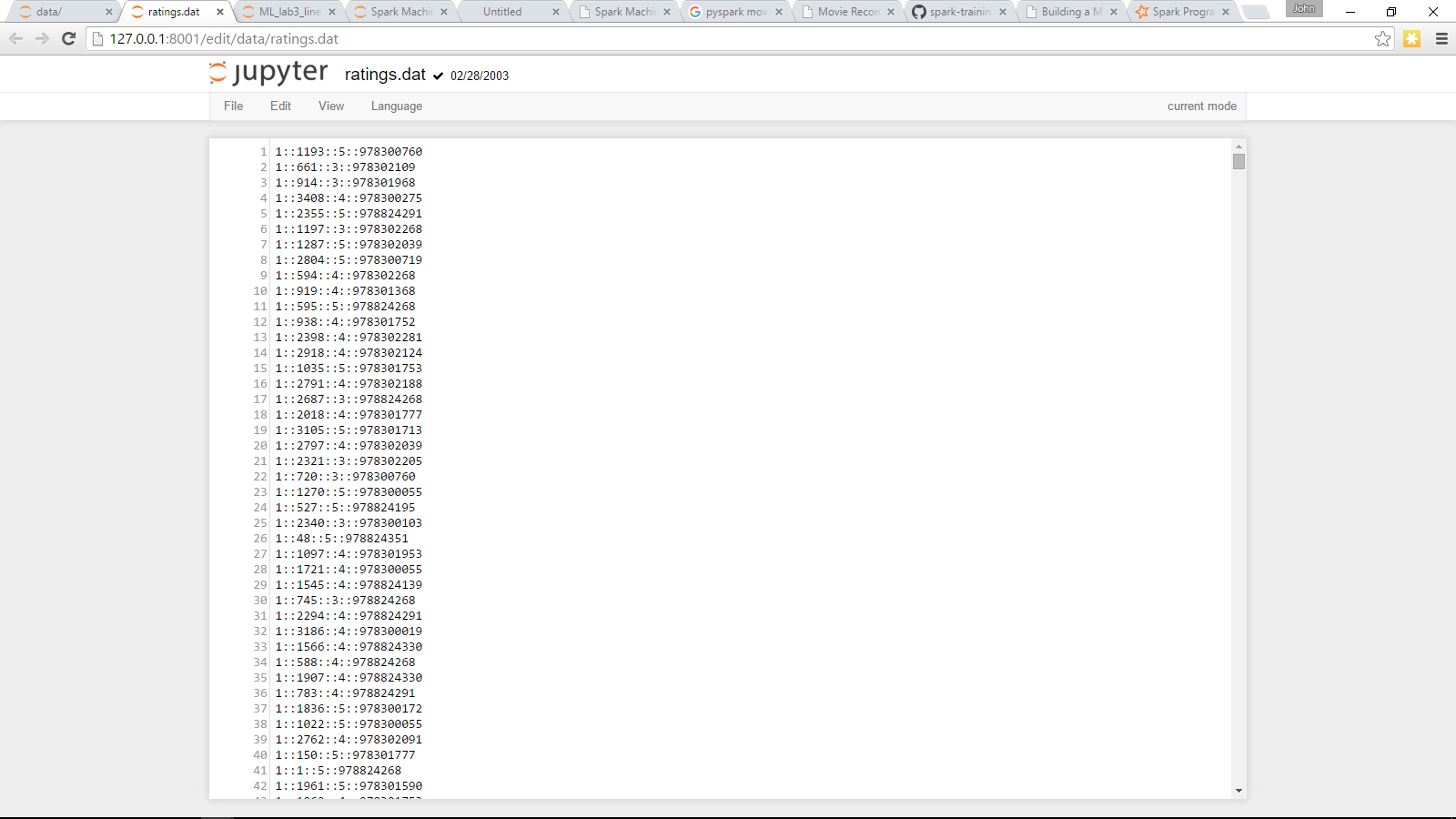
Three data sets were used for to demonstrate the machine learning algorithms:

1. Hubway use data from the Hubway challenge (we are using the edited and cleansed version that was used in the Analytics Edge MOOC, file attached and a copy online at https://github.com/gdwangh/edxTheAnalyticsEdge/blob/master/finalExam/HubwayTrips.csv) which shows bicycle rental usage statistics. This data is used to show linear regression, regression tree, and clustering examples.
2. Titanic survival data from the Kaggle Tutorial Challenge, also from <http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic3.xls>. This data is used to demonstrate classifiers and decision trees.
3. The MovieLens 1 million movies data set, which was used to build the recommender system. The data file is attached to this project and can be downloaded from <http://grouplens.org/datasets/movielens/1m/>. It contains two files – ratings.dat, the user/movie/ratings view, and movies.dat, the movie/title/genres view. Both are required for the recommender system demonstration as shown.

Data was loaded into the data/ top-level directory within the iPython notebook environment. Multiple data sets were loaded for experimentation, but only the two referred to above were actually utilized.

Some contents of the data directory, as well as a view of the ratings.dat file, are shown below:





🖐Loading the data was accomplished by using the WinSCP program using the ssh key provided with the Vagrant instance to avoid crashing the Jupyter user interface by uploading large files.

**Starting the Instance**

If the host does not have VirtualBox and Vagrant installed, please install versions 5.0.8 and 1.7.4 respectively. Newer versions should work as well; older versions may not load the iPython notebook.

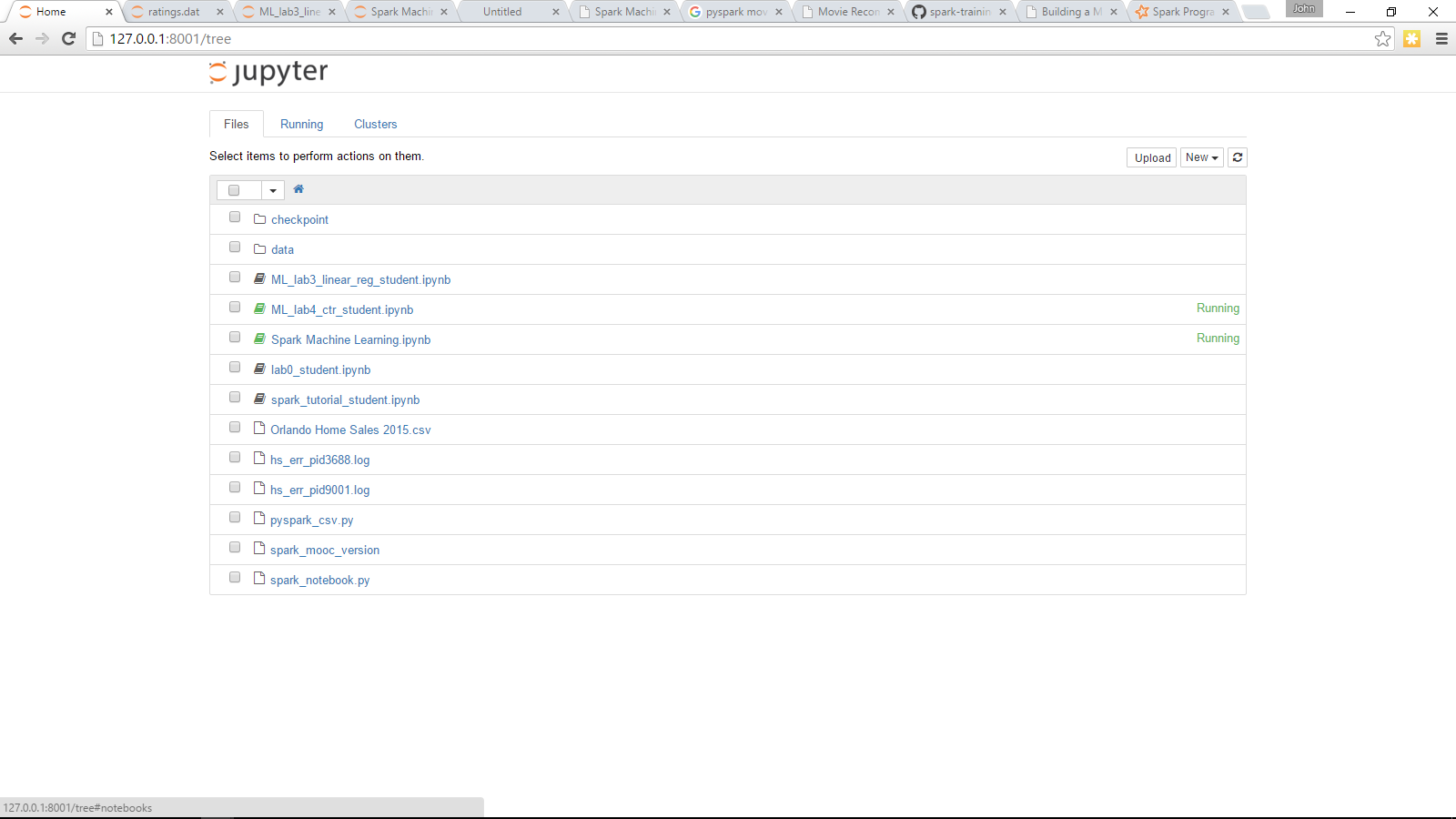
Use scp or WinSCP or FTP to upload the Hubway Trips.csv and MovieLens files. The home directory for the vagrant user (/home/vagrant) is the root of the iPython instance, so just dropping the files into /home/vagrant/data works fine.

To start the instance, go to the directory containing the VagrantFile attached to this project using a command-line terminal window and issue the command : **vagrant up**.

If all is well, vagrant will output which ports and IP addresses the instance is listening on, as well as downloading the virtual machine itself if needed.

An alternative approach is to start with an existing Hadoop instance (like Hortonworks Sandbox) and configure iPython Notebook manually as indicated in the appendix. The home directory will probably be /home/Hadoop instead of /home/vagrant in this instance, and the ports for iPython may be different.

Connect to the iPython notebook by navigating your browser to : <http://127.0.0.1:8001>



Upload the attached Spark Machine Learning.ipynb file to the home directory by clicking the Upload button, then click the Spark Machine Learning entry in the list above to start the notebook.

**High-Level Approach**

To treat this as a tutorial on Spark MLLib, the steps are sequential and increase in complexity. However, each individual machine learning step could be executed independently; for example, is someone just wanted to know how to build a decision tree, they could refer to that section in the iPython notebook without needing any of the earlier steps except for the initial data load.

Some familiarity with Spark is assumed. That is, the tutorial doesn’t deal with many Spark concepts such as resilient distributed datasets (RDDs) or MapReduce or SparkContext.

**Part 1 – Loading and Building RDDs**

**Step 1a – Loading Hubway data**

In this step, we define a custom parsing function to parse the raw csv line, then demonstrate one way to drop the header rows, which native Spark doesn’t handle nicely. We then show some basic statistics and counts of the data.

**def** parseLine**(**x**):**

**return** **[**int**(**y**)** **for** y **in** x**.**split**(**','**)]**

**import** os**.**path

baseDir **=** os**.**path**.**join**(**'data'**)**

fileName **=** os**.**path**.**join**(**baseDir**,** 'HubwayTrips.csv'**)**

rawData **=** sc**.**textFile**(**fileName**)**

#Drop header row

#Duration,Morning,Afternoon,Evening,Weekday,Male,Age

header **=** sc**.**parallelize**(**rawData**.**take**(**1**))**

rawDataNoHeaders **=** rawData**.**subtract**(**header**)**

#Parse the lines

rideRDD **=** rawDataNoHeaders**.**map**(**parseLine**).**cache**()**

numRecords **=** rideRDD**.**count**()**

minDuration **=** rideRDD**.**map**(lambda** x **:** x**[**0**]).**min**()**

maxDuration **=** rideRDD**.**map**(lambda** x **:** x**[**0**]).**max**()**

**print** "Number of records : %d " **%** numRecords

**print** "Minimum duration : %d " **%** minDuration

**print** "Maximum duration : %d " **%** maxDuration

Results

Number of records **:** 185190

Minimum duration **:** 180

Maximum duration **:** 85040

**Step 1b – Demonstrate the MLLib Statistics object**

Use the Statistics.colStats object to easily show how to derive basic statistics for all columns (similar to R’s summary() function on a data frame).

**from** pyspark**.**mllib**.**stat **import** Statistics

summary **=** Statistics**.**colStats**(**rideRDD**)**

**print** "Duration\tMorning\tAfternoon\tEvening\tWeekday\tMale\tAge\n"

**print(**"%8.2f\t%8.2f\t%8.2f\t%8.2f\t%8.2f\t%8.2f\t%8.2f\n"**)** **%** tuple**(**summary**.**mean**())**

**print(**"%8.2f\t%8.2f\t%8.2f\t%8.2f\t%8.2f\t%8.2f\t%8.2f\n"**)** **%** tuple**(**summary**.**variance**())**

**print(**"%8.2f\t%8.2f\t%8.2f\t%8.2f\t%8.2f\t%8.2f\t%8.2f\n"**)** **%** tuple**(**summary**.**numNonzeros**())**

Results

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Duration | Morning | Afternoon | Evening | Weekday | Male | Age |
| 721.55 | 0.33 | 0.4 | 0.25 | 0.83 | 0.74 | 35.37 |
| 1562024 | 0.22 | 0.24 | 0.19 | 0.14 | 0.19 | 120.89 |
| 185190 | 60399 | 74021 | 46264 | 153689 | 136505 | 185190 |

**Step 1c – Demonstrate the MLLib correlation function**

MLLib allows for some easy computations of certain statistical tests, like correlation, chi-squared, and goodness of fit. Here we demonstrate taking the correlation of two fields in an RDD.

durationRDD **=** rideRDD**.**map**(lambda** x **:** x**[**0**])** # Extract duration from the RDD

ageRDD **=** rideRDD**.**map**(lambda** x **:** x**[**6**])** # Extract Age from the RDD

**print(**Statistics**.**corr**(**durationRDD**,** ageRDD**,** method**=**"pearson"**))** # Print the Pearson correlation of Age vs. Duration

Result

0.0109175889218

So, we didn’t pick a great pair of variables to correlate – just at 1% correlation, so basically unrelated. This is going to cause some problems when doing linear regression, but it will illustrate the ability of decision trees to work with less correlated data, so worth continuing the example.

**Part 2 – Regression**

**Step 2a – Basic Plotting**

One of the powerful features of PySpark and the IPython Notebooks is the ability to do inline plotting using Matplotlib. Theoretically plot.ly and ggplot2 work as well, but those packages were not installed in my distribution. Matplotlib is probably more familiar to Python programmers anyways, so worth illustrating for exploratory data analysis

# Plot Age Vs. Duration

**%**matplotlib inline

**import** matplotlib**.**pyplot **as** plt

plt**.**scatter**(**ageRDD**.**collect**(),** durationRDD**.**collect**(),**alpha**=**0.5**)**

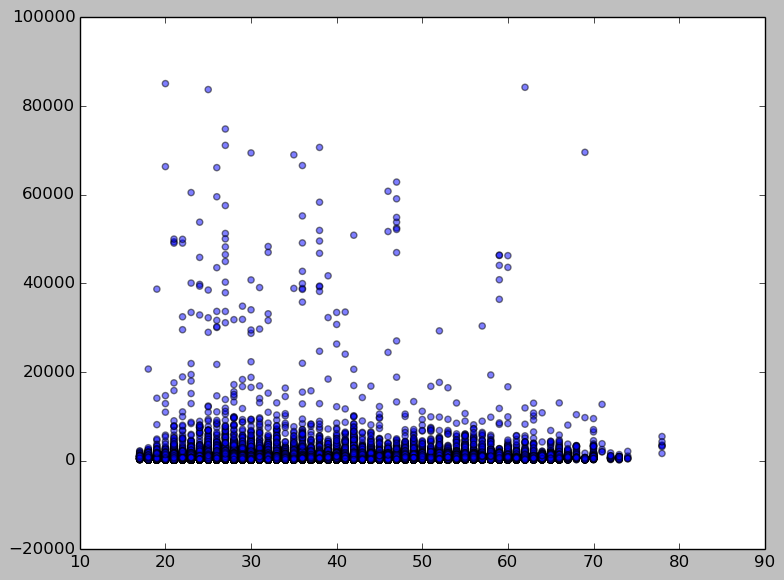
plt**.**xlabel**=**"Age"

plt**.**ylabel**=**"Duration"

plt**.**tight\_layout**()**

plt**.**show**()**

Result



Also, plot a histogram of durations to show just how skewed they are towards 0.

# Plot Duration Histogram

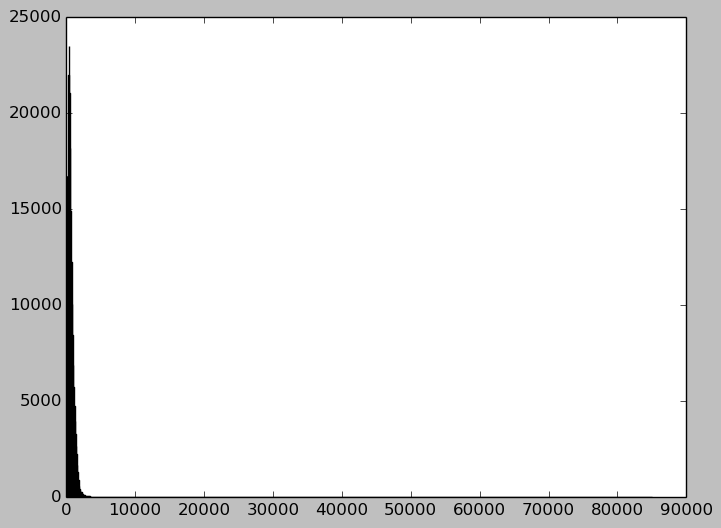
plt**.**hist**(**durationRDD**.**collect**(),**bins**=**1000**)**

plt**.**xlabel**=**"Duration"

plt**.**ylabel**=**"Count"

plt**.**show**()**

Result



As expected, highly skewed data. We had to increase the bins to 1000 in the plot to show any gradation at all in the duration data.

🖐 In spite of significant research, the axis labels don’t show up in these plots. It may be unique to being in the notebook environment and some weirdness with layout, or it may be necessary to increase the bounding box manually, but it didn’t seem worth much more than the time already invested to try to get that to work.

**Step 2b – Introducing LabeledPoints**

Key to the workings of MLLib is the LabeledPoints object. Basically, for anything with an objective value or label, like a regression problem (it would be the y value or expected outcome) or any binary variable in a logistic classification problem, the desired value is known as the “label”. Spark requires inputs to the MLLib functions to be of the format [label, [features]].

Here we show how to build a set of LabeledPoints using the Hubway data with “Duration” as the label we’re trying to predict.

**from** pyspark**.**mllib**.**regression **import** LabeledPoint**,** LinearRegressionWithSGD**,** LinearRegressionModel

**def** parsePoint**(**x**):**

**return** LabeledPoint**(**x**[**0**],**x**[**1**:])** #first field is label, all others are features

labeledRDD **=** rideRDD**.**map**(**parsePoint**).**cache**()** #use the parsepoint function to convert the RDD

**print** labeledRDD**.**take**(**5**)**

Result

**[**LabeledPoint**(**415.0**,** **[**0.0**,**1.0**,**0.0**,**1.0**,**1.0**,**30.0**]),** LabeledPoint**(**415.0**,** **[**0.0**,**1.0**,**0.0**,**1.0**,**1.0**,**30.0**]),** LabeledPoint**(**571.0**,** **[**1.0**,**0.0**,**0.0**,**0.0**,**1.0**,**21.0**]),** LabeledPoint**(**772.0**,** **[**1.0**,**0.0**,**0.0**,**1.0**,**0.0**,**28.0**]),** LabeledPoint**(**484.0**,** **[**0.0**,**1.0**,**0.0**,**1.0**,**1.0**,**32.0**])]**

**Step 3b – Linear Regression**

Now we build a simple linear regression model of all features trying to predict duration. “WithSGD” means that MLLib is using Stochastic Gradient Descent as an optimization function to find the result instead of the closed form linear algebra solutions. We then take the predicted values vs. the known values and try to compute the mean squared error (MSE) to rate the usefulness of the model.

# Build the model

model **=** LinearRegressionWithSGD**.**train**(**labeledRDD**)**

**print** model

# Evaluate the model on training data

valuesAndPreds **=** labeledRDD**.**map**(lambda** p**:** **(**p**.**label**,** model**.**predict**(**p**.**features**)))**

MSE **=** valuesAndPreds**.**map**(lambda** **(**v**,** p**):** **(**v **-** p**)\*\***2**).**reduce**(lambda** x**,** y**:** x **+** y**)** **/** valuesAndPreds**.**count**()**

**print(**"Mean Squared Error = " **+** str**(**MSE**))**

Result

**(**weights**=[-**6.32866322061e+233**,-**7.68045135233e+233**,-**4.28301224765e+233**,-**1.57278798171e+234**,-**1.39439410975e+234**,-**7.24240810206e+235**],** intercept**=**0.0**)**

Mean Squared Error **=** inf

And as expected with a nearly flat line and so many short durations, there’s no predictive value in the regression model.

**Part 3 - Clustering**

**Step 3a – K-Means Clustering**

While the linear regression was horrifyingly unhelpful, it may still be possible to identify patterns within the Hubway data. We switch our approach to clustering to see if any interesting patterns emerge. K-means clustering is the most common and most accessible clustering algorithm, so we demonstrate that. We somewhat arbitrarily select 5 as the number of clusters and 10 as the number of iterations to ensure the model terminates quickly.

**from** pyspark**.**mllib**.**clustering **import** KMeans**,** KMeansModel

# Build the model (cluster the data)

clusters **=** KMeans**.**train**(**rideRDD**,** 5**,** maxIterations**=**10**,**

runs**=**10**,** initializationMode**=**"random"**)**

**print** "Duration\tMorning\tAfternoon\tEvening\tWeekday\tMale\tAge\n"

**for** center **in** clusters**.**clusterCenters**:**

**print** **(**"%d8\t%8.2f\t%8.2f\t%8.2f\t%8.2f\t%8.2f\t%8d\n"**)** **%** tuple**(**center**)**

#print clusters.clusterCenters

Results

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Duration | Morning | Afternoon | Evening | Weekday | Male | Age |
|  |  |  |  |  |  |  |
| 6778 | 0.33 | 0.39 | 0.25 | 0.84 | 0.72 | 35 |
|  |  |  |  |  |  |  |
| 451038 | 0.2 | 0.12 | 0.56 | 0.84 | 0.74 | 34 |
|  |  |  |  |  |  |  |
| 3458 | 0.35 | 0.39 | 0.23 | 0.85 | 0.79 | 35 |
|  |  |  |  |  |  |  |
| 43948 | 0.23 | 0.49 | 0.26 | 0.62 | 0.62 | 37 |
|  |  |  |  |  |  |  |
| 12828 | 0.27 | 0.43 | 0.28 | 0.78 | 0.67 | 35 |

So, some interesting insights can be gleaned here. Some very short trips, cluster 3, almost 80% male, 85% weekday, 74% morning/afternoon. The medium trips are mostly afternoon, with more females and more weekends – couples biking together? Some very long trips are almost exclusively weekday evenings – overnight rentals?

**Part 4 – Regression Tree**

Given how badly the linear regression performed, it’s a toss-up whether a regression tree will give any insights. We’ll demonstrate how to build a regression tree to try to predict duration based on the other features in the data.

**from** pyspark**.**mllib**.**tree **import** DecisionTree**,** DecisionTreeModel

# Split the data into training and test sets (30% held out for testing)

**(**trainingData**,** testData**)** **=** labeledRDD**.**randomSplit**([**0.7**,** 0.3**])**

# Train a DecisionTree model.

# Empty categoricalFeaturesInfo indicates all features are continuous.

# Categorical features have already been converted in this data set

model **=** DecisionTree**.**trainRegressor**(**trainingData**,** categoricalFeaturesInfo**={},**

impurity**=**'variance'**,** maxDepth**=**5**,** maxBins**=**32**)**

# Evaluate model on test instances and compute test error

predictions **=** model**.**predict**(**testData**.**map**(lambda** x**:** x**.**features**))**

labelsAndPredictions **=** testData**.**map**(lambda** lp**:** lp**.**label**).**zip**(**predictions**)**

testMSE **=** labelsAndPredictions**.**map**(lambda** **(**v**,** p**):** **(**v **-** p**)** **\*** **(**v **-** p**)).**sum**()** **/** float**(**testData**.**count**())**

**print(**'Test Mean Squared Error = ' **+** str**(**testMSE**))**

**print** **(**'0:Morning,1:Afternoon,2:Evening,3:Weekday,4:Male,5:Age'**)**

**print(**'Learned regression tree model:'**)**

**print(**model**.**toDebugString**())**

Results (truncated for readability)

Test Mean Squared Error **=** 1677704.3489

0**:**Morning**,**1**:**Afternoon**,**2**:**Evening**,**3**:**Weekday**,**4**:**Male**,**5**:**Age

Learned regression tree model**:**

DecisionTreeModel regressor of depth 5 **with** 63 nodes

If **(**feature 4 **<=** 0.0**)**

If **(**feature 3 **<=** 0.0**)**

If **(**feature 5 **<=** 41.0**)**

If **(**feature 5 **<=** 24.0**)**

If **(**feature 1 **<=** 0.0**)**

Predict**:** 709.2555720653789

Else **(**feature 1 **>** 0.0**)**

Predict**:** 795.6678899082568

Else **(**feature 5 **>** 24.0**)**

If **(**feature 1 **<=** 0.0**)**

Predict**:** 847.3722037125178

Else **(**feature 1 **>** 0.0**)**

Predict**:** 929.4401490451793

Else **(**feature 5 **>** 41.0**)**

The regression tree does appear to be able to differentiate among durations. Young females rent for slightly longer in the afternoons (features 1 > 0), but if they’re over 24 and renting in the morning they rent almost 100 minutes longer, and older than 24 renting in the afternoon rent for almost 100 minutes longer than that. Definitely some insights to be gleaned from the regression tree analysis, but we must watch for overfitting.

**Part 5 – Classification Models**

**Step 5a – Logistic Regression**

We now switch to the Titanic survivors data to demonstrate how to build a classification model. The Titanic data has some missing values that will break the classifier, so in our parsing we need to flag bad records and then filter them out. We also use pycsv to read the csv file and not worry about dropping the header row. By default it creates a DataFrame, so we’ll convert that to an RDD before cleansing the data.

**from** pyspark**.**sql **import** SQLContext**,** Row

**from** pyspark**.**mllib**.**linalg **import** Vectors

**from** pyspark**.**mllib**.**classification **import** LogisticRegressionWithSGD

sqlCtx **=** SQLContext**(**sc**)**

fileName **=** os**.**path**.**join**(**baseDir**,** 'titanic3.csv'**)**

plaintext\_rdd **=** sc**.**textFile**(**fileName**)**

titanicRawRDD **=** pycsv**.**csvToDataFrame**(**sqlCtx**,** plaintext\_rdd**).**rdd

#remove blank rows

titanicRDD **=** titanicRawRDD**.**filter**(lambda** r **:** **(**r**[**2**]** **!=** **None)** **)**

#pclass,survived,name,sex,age,sibsp,parch,ticket,fare,cabin,embarked,boat,body,home\_dest

**def** parseRow**(**r**):**

pclass **=** r**[**0**]**

sex **=** 0 **if** r**[**3**]** **==** 'female' **else** 1

age **=** r**[**4**]** **if** r**[**4**]** **!=** **None** **else** **-**1 #flag invalid ages for filtering

sibsp **=** r**[**5**]**

parch **=** r**[**6**]**

fare **=** r**[**8**]** **if** r**[**8**]** **!=** **None** **else** **-**1 #flag missing fares for filtering

**try:**

lp **=**LabeledPoint**(**r**[**1**],** **[**pclass**,**sex**,**age**,**sibsp**,**parch**,**fare**])**

**except** ValueError**:**

lp **=** **None**

**return** lp

#filter out records flagged by parseRow as bad

parsedTitanicRDD **=** titanicRDD**.**map**(**parseRow**).**filter**(lambda** lp **:** **(**lp**.**features**[**2**]** **!=** **-**1**)** **and** **(**lp**.**features**[**5**]** **!=** **-**1**))**

#build distinct lists of passenger classes and features – the model will

#automatically convert those to categorical variables when we ask it to.

pclasses **=** parsedTitanicRDD**.**map**(lambda** lp **:** lp**.**features**[**0**]).**distinct**().**collect**()**

sexes **=** parsedTitanicRDD**.**map**(lambda** lp **:** lp**.**features**[**1**]).**distinct**().**collect**()**

**print** pclasses

**print** sexes

**(**trainingData**,** testData**)** **=** parsedTitanicRDD**.**randomSplit**([**0.7**,** 0.3**])**

# Train model

model **=** LogisticRegressionWithSGD**.**train**(**trainingData**)**

# evaluate the model on test data

results **=** testData**.**map**(lambda** p**:** **(**p**.**label**,** model**.**predict**(**p**.**features**)))**

# calculate the error

err **=** results**.**filter**(lambda** **(**v**,** p**):** v **!=** p**).**count**()** **/** float**(**testData**.**count**())**

# Print results

**print(**"Model Error = " **+** str**(**err**))**

Results

**[**1.0**,** 2.0**,** 3.0**]**

**[**0.0**,** 1.0**]**

Model Error **=** 0.434504792332

So, not a great model, only about 57% predictive. Adding more features or using an ensemble method will probably increase the predictive value.

**Step 5b – Classification Tree**

Another technique for classification is the classification tree, a close counterpart to the regression tree shown above. We tell the tree that pclass and sex are categorical variables so it can automatically handle that for us.

🖐Due to memory constraints on a laptop, we don’t show how to determine the hyperparameters (minInfoGain, maxDepth, maxBins) which require multiple iterations to compute.

# Train a DecisionTree model.

# Empty categoricalFeaturesInfo indicates all features are continuous.

# Categorical features have already been converted in this data set

model **=** DecisionTree**.**trainClassifier**(**trainingData**,** numClasses**=**2**,** categoricalFeaturesInfo**={**0**:**len**(**pclasses**)+**1**,**1**:**len**(**sexes**)},**

impurity**=**'gini'**,** maxDepth**=**5**,** maxBins**=**32**,** minInfoGain**=**0.001**)**

# Evaluate model on test instances and compute test error

predictions **=** model**.**predict**(**testData**.**map**(lambda** x**:** x**.**features**))**

labelsAndPredictions **=** testData**.**map**(lambda** lp**:** lp**.**label**).**zip**(**predictions**)**

testMSE **=** labelsAndPredictions**.**map**(lambda** **(**v**,** p**):** **(**v **-** p**)** **\*** **(**v **-** p**)).**sum**()** **/** float**(**testData**.**count**())**

**print(**'Test Mean Squared Error = ' **+** str**(**testMSE**))**

**print(**'Learned classification tree model:'**)**

**print** **(**'Class:Sex:Age:SibSp:ParCh:Fare'**)**

**print(**model**.**toDebugString**())**

Results

Test Mean Squared Error **=** 0.191693290735

Learned classification tree model**:**

Class**:**Sex**:**Age**:**SibSp**:**ParCh**:**Age

DecisionTreeModel classifier of depth 5 **with** 51 nodes

If **(**feature 1 **in** **{**1.0**})**

If **(**feature 2 **<=** 8.0**)**

If **(**feature 3 **<=** 2.0**)**

If **(**feature 5 **<=** 15.85**)**

If **(**feature 5 **<=** 12.875**)**

Predict**:** 1.0

Else **(**feature 5 **>** 12.875**)**

Predict**:** 0.0

Else **(**feature 5 **>** 15.85**)**

Predict**:** 1.0

Else **(**feature 3 **>** 2.0**)**

If **(**feature 4 **<=** 1.0**)**

As with our regression tree, the classifier tree has the ability to more finely discriminate between cases. It selects gender (feature 1) as the most predictive feature, which is consistent with other Titanic modeling scenarios, followed by age (women and children first theory).

**Part 6 – Recommenders**

**Step 6a – Loading data**

We now switch to our final data set, the MovieLens 1 million ratings data set. As with the earlier data sets, first we write a custom parser to split it and store it in an RDD – one for ratings, and one for movies.

**def** parseRatings**(**row**):**

**(**userID**,**MovieID**,**Rating**,**Timestamp**)** **=** row**.**split**(**"::"**)**

**return** **(**int**(**userID**),**int**(**MovieID**),**float**(**Rating**))**

**def** parseMovies**(**row**):**

**(**MovieID**,**Title**,**Genres**)** **=** row**.**split**(**"::"**)**

**return** **(**int**(**MovieID**),**Title**,**Genres**)**

ratingsRDD **=** sc**.**textFile**(** os**.**path**.**join**(**baseDir**,** 'ratings.dat'**)).**map**(**parseRatings**).**cache**()**

**print** ratingsRDD**.**take**(**3**)**

moviesRDD **=** sc**.**textFile**(** os**.**path**.**join**(**baseDir**,** 'movies.dat'**)).**map**(**parseMovies**).**cache**()**

**print** moviesRDD**.**take**(**3**)**

Results

**[(**1**,** 1193**,** 5.0**),** **(**1**,** 661**,** 3.0**),** **(**1**,** 914**,** 3.0**)]**

**[(**1**,** u'Toy Story (1995)'**,** u"Animation|Children's|Comedy"**),** **(**2**,** u'Jumanji (1995)'**,** u"Adventure|Children's|Fantasy"**),** **(**3**,** u'Grumpier Old Men (1995)'**,** u'Comedy|Romance'**)]**

This just validates that the data is loaded correctly. User 1 rated movie 1193 a 5, Movie 1 is “Toy Story”, and so on.

**Step 6b – Summary statistics and a more complex example**

Here we show how many ratings, users, and movies are in the rating data set using the lambda column extraction function and the distinct() function on the resulting RDD. We also compute a more complex example – count ratings by movie using a wordcount-style MapReduce function, join the result to the main movies RDD to get the titles, flatten the results out, and take the 25 movies with the most ratings to display. The key parameter on .takeOrdered says “sort by the 3rd feature (x[2] – total ratings) descending (negative sign).

# Some summary stats on the movie data

numUsersRating **=** ratingsRDD**.**map**(lambda** r **:** r**[**0**]).**distinct**().**count**()**

numMoviesRated **=** ratingsRDD**.**map**(lambda** r **:** r**[**1**]).**distinct**().**count**()**

totalRatings **=** ratingsRDD**.**count**()**

distinctRatings **=** ratingsRDD**.**map**(lambda** r **:** r**[**2**]).**distinct**().**collect**()**

**print** "Total ratings %d, Total users %d, Total movies %d" **%** **(**totalRatings**,** numUsersRating**,** numMoviesRated**)**

**print** "Distinct Ratings : %s" **%** distinctRatings

mostRatedMovies **=** **(**ratingsRDD

**.**map**(lambda** r **:** **(**r**[**1**],**1**))**

**.**reduceByKey**(lambda** a**,**b **:** a**+**b**)**

**.**join**(**moviesRDD**.**map**(lambda** r **:** **(**r**[**0**],**r**[**1**])))**

**.**map**(lambda** **(**id**,(**numRatings**,** movieTitle**))** **:** **(**id**,**movieTitle**,**numRatings**))**

**.**takeOrdered**(**25**,**key**=lambda** x **:** **-**x**[**2**])**

**)**

**print** mostRatedMovies

Results

Total ratings 1000209**,** Total users 6040**,** Total movies 3706

Distinct Ratings **:** **[**1.0**,** 2.0**,** 3.0**,** 4.0**,** 5.0**]**

**[(**2858**,** u'American Beauty (1999)'**,** 3428**),** **(**260**,** u'Star Wars: Episode IV - A New Hope (1977)'**,** 2991**),** **(**1196**,** u'Star Wars: Episode V - The Empire Strikes Back (1980)'**,** 2990**),** **(**1210**,** u'Star Wars: Episode VI - Return of the Jedi (1983)'**,** 2883**),** **(**480**,** u'Jurassic Park (1993)'**,** 2672**),** **(**2028**,** u'Saving Private Ryan (1998)'**,** 2653**),** **(**589**,** u'Terminator 2: Judgment Day (1991)'**,** 2649**),** **(**2571**,** u'Matrix, The (1999)'**,** 2590**),** **(**1270**,** u'Back to the Future (1985)'**,** 2583**),** **(**593**,** u'Silence of the Lambs, The (1991)'**,** 2578**),** **(**1580**,** u'Men in Black (1997)'**,** 2538**),** **(**1198**,** u'Raiders of the Lost Ark (1981)'**,** 2514**),** **(**608**,** u'Fargo (1996)'**,** 2513**),** **(**2762**,** u'Sixth Sense, The (1999)'**,** 2459**),** **(**110**,** u'Braveheart (1995)'**,** 2443**),** **(**2396**,** u'Shakespeare in Love (1998)'**,** 2369**),** **(**1197**,** u'Princess Bride, The (1987)'**,** 2318**),** **(**527**,** u"Schindler's List (1993)"**,** 2304**),** **(**1617**,** u'L.A. Confidential (1997)'**,** 2288**),** **(**1265**,** u'Groundhog Day (1993)'**,** 2278**),** **(**1097**,** u'E.T. the Extra-Terrestrial (1982)'**,** 2269**),** **(**2628**,** u'Star Wars: Episode I - The Phantom Menace (1999)'**,** 2250**),** **(**2997**,** u'Being John Malkovich (1999)'**,** 2241**),** **(**318**,** u'Shawshank Redemption, The (1994)'**,** 2227**),** **(**858**,** u'Godfather, The (1972)'**,** 2223**)]**

In this data set, *American Beauty* is the most rated movie, with 3428 ratings, followed by *Star Wars*.

**Step 6c – Build the Recommender**

We now split the ratings data into training and testing sets, build a recommender, and evaluate its performance against the test set.

**import** itertools

**import** math

**from** pyspark**.**mllib**.**recommendation **import** ALS

sc**.**setCheckpointDir**(**'checkpoint/'**)**

ALS**.**checkpointInterval **=** 2

#Create training and test sets (could also create a validation set if required)

**(**trainingData**,** testData**)** **=** ratingsRDD**.**randomSplit**([**0.7**,** 0.3**])**

model **=** ALS**.**train**(**trainingData**,** 8**,** 5**,** 0.1**)**

test\_for\_predict\_RDD **=** testData**.**map**(lambda** x**:** **(**x**[**0**],** x**[**1**]))**

predictions **=** model**.**predictAll**(**test\_for\_predict\_RDD**).**map**(lambda** r**:** **((**r**[**0**],** r**[**1**]),** r**[**2**]))**

rates\_and\_preds **=** testData**.**map**(lambda** r**:** **((**int**(**r**[**0**]),** int**(**r**[**1**])),** float**(**r**[**2**]))).**join**(**predictions**)**

error **=** math**.**sqrt**(**rates\_and\_preds**.**map**(lambda** r**:** **(**r**[**1**][**0**]** **-** r**[**1**][**1**])\*\***2**).**mean**())**

**print** 'For testing data the RMSE is %s' **%** **(**error**)**

**print** rates\_and\_preds**.**take**(**10**)**

“test\_for\_predict\_RDD” basically takes the test data and removes the rating so we just pass user,movie to the recommender model. We then evaluate the known rating to the predicted in “rates\_and\_preds”.

Results

For testing data the RMSE **is** 0.896336806338

**[((**1861**,** 3911**),** **(**3.0**,** 3.3456925527076757**)),** **((**5283**,** 2081**),** **(**3.0**,** 3.415509846366383**)),** **((**1150**,** 322**),** **(**5.0**,** 2.4951223727567147**)),** **((**3378**,** 2124**),** **(**4.0**,** 3.2772129053995087**)),** **((**1051**,** 593**),** **(**5.0**,** 4.076649866427813**)),** **((**77**,** 2723**),** **(**1.0**,** 3.0302801910753807**)),** **((**696**,** 1196**),** **(**4.0**,** 4.048061945614126**)),** **((**3200**,** 1208**),** **(**5.0**,** 4.067235318955074**)),** **((**4604**,** 1610**),** **(**2.0**,** 3.11472507852392**)),** **((**874**,** 2366**),** **(**3.0**,** 3.704751356330039**))]**

The Root Mean Squared Error is 0.89, which means that on average we’re less than one star away from the actual rating, which is not awful, but not great.

**Step 6d – Add personalized recommendations**

For the final step, I rate a few movies based on my own tastes, retrain the model, and generate predictions for movies I have not rated (by excluding the movies I’ve seen from the candidates RDD).

#Rate some movies to get personalized recommendations. Since userId 0 is not used, use that for my ratings.

myRatingsRDD **=** sc**.**parallelize **([**

**[**0**,**2858**,**3.0**],**

**[**0**,**260**,**5.0**],**

**[**0**,**1196**,**4.0**],**

**[**0**,**480**,**4.0**],**

**[**0**,**589**,**5.0**],**

**[**0**,**1270**,**5.0**],**

**[**0**,**1198**,**5.0**],**

**[**0**,**1097**,**4.0**],**

**[**0**,**858**,**2.0**]**

**])**

#Retrain the model with my preferences

#Exclude movies I've rated from the prediction set

newTrainingData **=** ratingsRDD**.**union**(**myRatingsRDD**)**

model **=** ALS**.**train**(**newTrainingData**,** 8**,** 5**,** 0.1**)**

#Generate (0,movieid) pairs for movies I haven't rated -- candidates for scoring. I’m userid 0, since it’s not used in the data.

moviesIRated **=** myRatingsRDD**.**map**(lambda** row **:** row**[**1**]).**distinct**().**collect**()**

myUnratedMoviesRDD **=** **(**moviesRDD**.**filter**(lambda** x**:** x**[**0**]** **not** **in** moviesIRated**).**map**(lambda** x**:** **(**0**,** x**[**0**])))**

predictions **=** model**.**predictAll**(**myUnratedMoviesRDD**).**collect**()**

recommendations **=** sorted**(**predictions**,** key**=lambda** x**:** x**[**2**],** reverse**=True)[:**20**]**

movies **=** moviesRDD**.**collect**()** #bring locally to print

**print** "Movies recommended for you:"

**for** i **in** xrange**(**len**(**recommendations**)):**

**print** **(**"%2d: %s" **%** **(**i **+** 1**,** movies**[**recommendations**[**i**][**1**]])).**encode**(**'ascii'**,** 'ignore'**)**

Results

Movies recommended **for** you**:**

1**:** **(**3451**,** u"Guess Who's Coming to Dinner (1967)"**,** u'Comedy|Drama'**)**

2**:** **(**2631**,** u'Frogs for Snakes (1998)'**,** u'Comedy|Film-Noir|Thriller'**)**

3**:** **(**38**,** u'It Takes Two (1995)'**,** u'Comedy'**)**

4**:** **(**576**,** u'Fausto (1993)'**,** u'Comedy'**)**

5**:** **(**2196**,** u'Knock Off (1998)'**,** u'Action'**)**

6**:** **(**561**,** u'Killer (Bulletproof Heart) (1994)'**,** u'Thriller'**)**

7**:** **(**1920**,** u'Small Soldiers (1998)'**,** u"Animation|Children's|Fantasy|War"**)**

8**:** **(**2266**,** u"Butcher's Wife, The (1991)"**,** u'Comedy|Romance'**)**

9**:** **(**970**,** u'Beat the Devil (1954)'**,** u'Comedy|Drama'**)**

10**:** **(**1804**,** u'Newton Boys, The (1998)'**,** u'Crime|Drama'**)**

11**:** **(**1228**,** u'Raging Bull (1980)'**,** u'Drama'**)**

12**:** **(**1311**,** u'Santa with Muscles (1996)'**,** u'Comedy'**)**

13**:** **(**3672**,** u'Benji (1974)'**,** u"Adventure|Children's"**)**

14**:** **(**1707**,** u'Home Alone 3 (1997)'**,** u"Children's|Comedy"**)**

15**:** **(**2640**,** u'Superman (1978)'**,** u'Action|Adventure|Sci-Fi'**)**

16**:** **(**3383**,** u'Big Fella (1937)'**,** u'Drama|Musical'**)**

17**:** **(**577**,** u'Andre (1994)'**,** u"Adventure|Children's"**)**

18**:** **(**765**,** u'Jack (1996)'**,** u'Comedy|Drama'**)**

19**:** **(**3647**,** u'Running Free (2000)'**,** u'Drama'**)**

20**:** **(**112**,** u'Rumble in the Bronx (1995)'**,** u'Action|Adventure|Crime'**)**

Generally, not great results. However, with 1 million recommendations and only a few ratings, it’s very difficult for the algorithm to do better. As with the Netflix recommender, the more ratings are added the better the algorithm performs.

🖐 An opportunity for improvement would be to create multiple sets of input ratings and give a “Usefulness Score” to each of the recommended movie sets.

**Conclusion**

Spark MLLib has done a lot to make machine learning at scale accessible to the masses. There are many algorithms beyond the ones we explored, including ensemble methods like RandomForest and dimensionality reduction algorithms like PCA, as well as other classifiers like NaïveBayes and Support Vector Machines.

Our objective was to take a survey of the capabilities of Spark MLLib and demonstrate how the more common machine learning algorithms are now implemented on a platform where they can be run against arbitrarily large data sets on a Hadoop cluster by leveraging Spark.