**Owen Galvin**

**May 11, 2016**

## Topic: Machine Learning with H2O.ai

## Concentration: Simple and Multiple Linear Regression

**Problem/Purpose:** Illustrate use of both simple and multiple linear regression in H2O.ai using movie data set. For simple, the goal will be to predict professional movie critics rating of a film by using an explanatory variable feature-engineered to encompass “star power” of the actors in the films, along with the general audience rating of given movie. Multiple Linear regression will attempt to predict same value but using a larger number of primarily categorical values. A preliminary assessment of the technology will also be performed using the Small Car dataset used in class.

**Big Data Set:**

**Data files:** movies.dat, movie\_countries.dat, movie\_directors.dat, movie\_genres.dat (hetrec2011-movielens-2k-v2.zip)

**Description of data:** Links the movies of MovieLens dataset with their corresponding web pages at Internet Movie Database (IMDb) and Rotten Tomatoes movie review systems, includes data on actors, country, genre tags, ratings from both critics and audience members.

**URL To get data:** original .zip @ on <http://grouplens.org/datasets/hetrec-2011/>. Data joined through common movie\_id values via SQL Server + feature engineering to create new values. Full massaged dataset available at [FP\_movies.psv](http://oweng.net/csci-e63/FP_movies.psv).

**Size:** Orig zip: 18MB, used .dat files: ~12MB uncompressed, final input data file: 1MB uncompressed.

**Format of data file:** csv orig, psv final export file for ingestion by H2O

**Hardware:** Windows 7 running VMware with Linux CentOS 6.

**Software:** (grayed lines were used only in data prep**)**

|  |  |
| --- | --- |
| **Technology/tools** | **Description** |
| VMWare Workstation | For hosting Linux CentOS system |
| Excel, UltraEdit | Data munging, editing |
| Microsoft SQL Server | Joining, querying, calculating and hosting data |
| Python via Anaconda2 | Programming, usage of H2O library |
| IPython/Jupyter notebook | Running and output of python/H2O |
| H2O & Flow | Python library + web notebook like UI interface (provided by H2O install) |

**Summary:**

Fast (so they say, I believe them) and well documented open source tool for performing machine learning and related Big Data analysis tasks. R, Python, Scala interfaces; my impression is that R is most fully supported/documented, followed closely by Python. Didn’t find too much Scala documentation, though I wasn’t looking for any.

**Overview of steps:**

1) cover installation and setup in a Python 2.7 environment

2) using Jupyter notebook, run through a tutorial using small car data set (100 rows) from Assignment 11

- loading of data, splitting into test/train

- creating simple linear regression model and training it on displacement in order to predict horsepower

- creating multiple regression model, using same features as from Assignment 11 in order to predict horsepower

- accuracy measures, csv output, viz of simple model results via D3, compare vs. Spark/MLlib predictions

3) similar simple and multiple regression examples for the movie dataset

- for simple, predict critic’s rating with the help of feature engineering (movie star power + audience rating)

- visualize results of above in D3

- carry out similar predictions via model using multiple feature variables

**Pro/Con:**

Silicon Valley startup but supported by some big names in the field (Rob Tibshirani & Trevor Hastie from Stanford among others), so likely to continue growing. Codebase is fast moving, which means many continual improvements but also conflicting documentation and videos on the web, i.e. old vs. new. Internal help function very comprehensive. Syntactically friendly, within Python both pandas and numpy supported/expected. Similarly named syntax between R and Python, though overall more oriented toward R in nomenclature. One “pro” is lots of knobs to play with, which is also a con. One other con is that I keep typing H20 when I mean to type H2O.

**YouTube URLs here: long:** [**https://youtu.be/OxjCc99Jer0**](https://youtu.be/OxjCc99Jer0) **short:** [**https://youtu.be/uAxmUhqxOFY**](https://youtu.be/uAxmUhqxOFY)

Code file manifest

|  |  |
| --- | --- |
| FP\_cars.ipynb  small\_car.csv  car\_MULTI.csv  car\_SIMPLE.csv | Jupyter notebook for car example   1. import small\_car.csv 2. .csv of predictions for both 3. (simple is used as source data in viz) |
| FP\_movies\_simple.ipynb  FP\_movies\_A\_SHORT\_SAMPLE\_OF\_INPUT.psv  movie\_SIMPLE\_out\_SHORT.csv | Jupyter of simple linear regression on movie dataset   1. 50 or so rows of input .psv file   (full 7k input file on my website @ [FP\_movies.psv](http://oweng.net/csci-e63/FP_movies.psv))   1. output file of predictions used in data viz |
| FP\_movies\_multi.ipynb  movie\_MULTI\_out\_SHORT\_SAMPLE\_OF\_OUTPUT.csv | Jupyter of multiple linear regression, movie data   1. 20 or so rows of output predictions |
| e63Q1.sql | SQL query to join data and create new features |
| (Viz subfolder) FP\_movies.html  movie\_SIMPLE\_out\_SHORT.csv  spark\_car.csv  car\_SIMPLE.csv  FP\_cars.css  FP\_cars.html  FP\_movies.css | both D3 visualizations |
| (PythonVersionOfJupyterNotebook subfolder) FP\_movies\_simple.py  FP\_cars.py  FP\_movies\_multi.py | .py export from Jupyter notebooks (just in case…) |

**References:**

* Documentation seems to be very fast changing, best bet is to go to main H2O ai website at [www.h2o.ai](http://www.h2o.ai)
* One extensive page = <http://h2o-release.s3.amazonaws.com/h2o/master/3092/docs-website/h2o-docs/index.html>
* Specific to Python: <http://h2o-release.s3.amazonaws.com/h2o/rel-turchin/3/docs-website/h2o-py/docs/index.html>
* Their Generalized Linear Modeling with H2O pdf ([GLM\_Vignette.pdf](http://h2o-release.s3.amazonaws.com/h2o/master/3461/docs-website/h2o-docs/booklets/GLM_Vignette.pdf)) was very helpful. I hyperlinked the file name for latest, which is April 2016 as of this writing but also recommended to google for latest version.

**Installation and setup**

Working on the VM we created, logged in as user joe. I had already installed Anaconda2 onto this CentOS machine for Assignment 11, so that should generally be considered a prerequisite for reproducing the examples. I will simply paste in my related notes from Assignment 11 below. I recall having to install pip for a Linux python environment but believe that was on the QuickStart VM, as the Anaconda documentation indicates pip is one of the included packages. I also believe 99% of the report contents could be created on a simply 2.7 w/pip… a single use of numpy is the only non-standard library that immediately comes to mind.

|  |  |  |
| --- | --- | --- |
| **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.** |

**Download of H2O.ai for Python**

URL: <http://www.h2o.ai/download/h2o/choose>

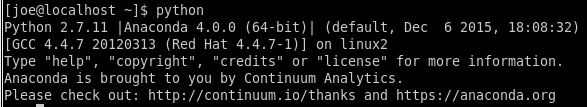
And click on the python pic



Which leads to a series of steps that can be run in python itself in order to get H2O installed.

Prerequisite: Python 2.7 or 3.5

Check my python, which is part of the Anaconda 2.x family.



(Certain other documentation mentioned an installation of 64-bit Java, 1.6 or newer, this VM has 1.8.)

The instructions then indicate a series of steps requiring pip already be installed for Python, so that may be an additional prerequisite. In my case, I had already installed pip as part of completing Assignment 10, Spark MLib, so I ran those directly. (My joe user has sufficient permissions and I don’t need to sudo each command as the instructions indicate might be necessary). Since I was running under Anaconda there was a decent change most if not all packages would already by present but there also shouldn’t be any harm in upgrading to latest

Install dependencies (prepending with `sudo` if needed):

*pip install -U requests*

*pip install -U tabulate*

*pip install -U future*

*pip install -U six*

Run each of those in sequence, apparently only the last was fully up to date on my machine. Easier to include full screen details vs. slightly smaller snippets.



Next command calls for uninstalling any H2O, followed by the main installation. I’d be very surprised if there were any existing H2O components but perhaps Continuum Analytics as included something, no harm in doing the uninstall before the primary installation.

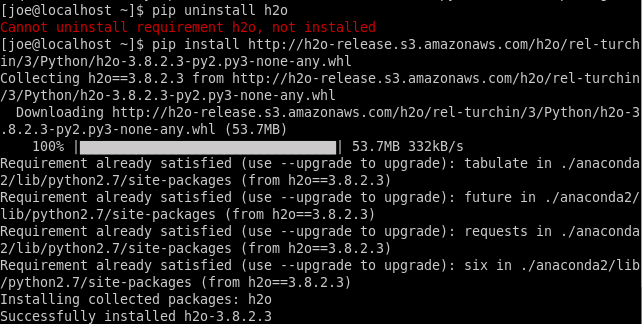
*# The following command removes the H2O module for Python.*

*pip uninstall h2o*

*# Next, use pip to install this version of the H2O Python module.*

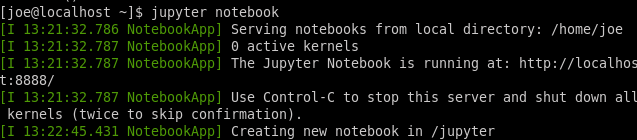
*pip install http://h2o-release.s3.amazonaws.com/h2o/rel-turchin/3/Python/h2o-3.8.2.3-py2.py3-none-any.whl*

The H2O version in the install path above points to, 3.8.2.3, is presumably dynamically generated as appropriate. This version appears to have a release date of 4/25/16, as a point of reference 3.6 came out in December, 2015 and first 3.8.0 in February of 2016.



If you care to run the provided Jupyter notebook, the installation of Anaconda2 should have installed the necessary components. Simply start it like below

*Jupyter notebook*

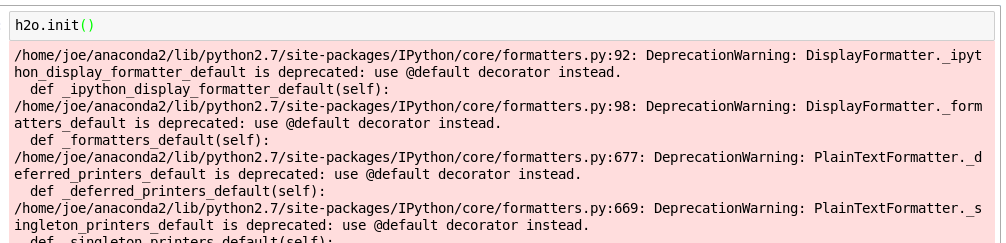


And in internet browser enter the url = <http://localhost:8888>

Tried a sample iPython/Jupyter notebook and first attempt at initializing the environment failed, where it was apparently unable to find the h2o .jar. Then I used a recommended debugging command, jar\_paths(), to find where it was trying to load the .jar from (hint, it is in place right where it is looking). For whatever reason it decided to work on this run, .init() prints out a high level summary of cluster status.

|  |
| --- |
|  |

I’ll also point out that on first use, usually got a series of warnings, apparently a combination of my Anaconda/IPython install & the code run by h2o init(). Running init() again cleared warnings from Jupyter output and set h2o up correctly.



**Tutorial with Small Car data**

Since a primary goal of this report is to serve as a tutorial, I thought it made sense to spend a decent amount of time on re-implementing one of the linear regression examples from class before moving on to my own more “Big Data” run through. In that vein, going to show how to do both simple and multiple linear regression in H2O using the small car dataset. For simple, try to predict Horsepower using the single feature variable of Displacement, followed later on by a visual display via D3. For multiple example the goal will be to predict Horsepower again, but using multiple features.

Within a new notebook named FP\_cars.ipynb, begin with importing the h2o module and initializing it. By default it will run on local machine but here is also where you could pass in the IP address/port of a remote H2O cluster. The remove\_all() (and both comments for that matter) are from one of the provided tutorials. May not be necessary on a local machine, or perhaps even most helpful on a local machine, but either way unlikely to hurt anything when you and your environment are in learning mode. As noted in output, default port number is 54321 and this is also where the H2O Flow tool will listen. The latter is a web based user interface into H2O, resembling Jupyter/IPython notebook in many aspects.

|  |
| --- |
|  |

The linear regression capabilities are in the H2OGeneralizedLinearEstimator module, import that as well as numpy. I think I use the latter only for some very simple math that could just as easily have been done via python builtins but it is notable how the Python interface of H2O has been designed to work with both numpy and pandas libraries. In xome cases there are corresponding H2O objects that behave similarly to those modules, e.g. H2OFrame is similar to a pandas dataframe . In other situations it is simply notable how H2O works with their internal objects, both in terms of execution and the documentation provide by H2O – the software was built with understanding that Python end users are likely to have used either or both numpy and pandas.

Create a H2OFrame object by importing the sample file from class, in this case the file resides in the same Linux directory as the Jupyter notebook. It could have just as easily been a full path or a Hadoop path, e.g. “hdfs://path/to/data.csv”. Create train and test dataframes via split\_frame – I found in general the syntax to be more R-oriented than anything, as can be seen here. In class we used a 90/10 split but 80/20 makes a little more sense for reasons I’ll discuss later. Interactively, you would see the progress bar run up to 100%, which of course takes almost no time with so few records. Other operations, such as generating a linear regression model, display similar progress “bars”.

|  |
| --- |
|  |

I’ll run the same w/uncommented help(h2o.import\_file) to illustrate internal documentation. In this case I did need to run it when working on my pipe-delimited primary dataset, to find syntax for supplying a non-default delimiter. Some of the other objects I ran help() on produced even more voluminous output. On the one hand you are probably only looking for one tiny part of the output, on the other the detail level is great. Especially in a fast-moving codebase like this, where much of the documentation and examples found online are likely to be outdated, even if only by months. One assumes the information here matches the version of H2O installed.

|  |
| --- |
|  |

For simple regression, set target/dependent variable we are trying to predict to name of column in the loaded dataframe, Horsepower, and the explanatory variable to the feature used to make that prediction. The multiple version is similar but the explanatory variable is set to a Python list of the columns that will be used in that model.

|  |
| --- |
|  |

Next we create the actual models, one for simple and another for multiple. Each is created with a model\_id value that will then appear in the Flow tool (otherwise it is a somewhat arbitrary value that will be hard to track down\_. For linear regression, the family to use is ‘gaussian’ , other values are appropriate to other uses. At this point I’ll mention the “Generalized Linear Modeling with H2O” GLM\_Vignette.pdf available from H2O at <http://www.h2o.ai/resources/>. I’ve linked to the general page as the pdf itself appears to be continually updated, with the latest version currently from April 2016. The pdf has many details, many of which were beyond my current understanding of machine learning. Usually when I had a implementation question I’d search in the pdf for a keyword and try to understand the context. The family parameter is one value discussed within, as well as the solver. Of the latter there are only two, with IRLSM being the default; other documentation states “*IRLSM is fast on on problems with small number of predictors and for lambda-search with L1 penalty, while L\_BFGS scales better for datasets with many columns*.” (Notable that Spark’s MLlib also has an L-BFGS “solver” but we used a stochastic gradient descent library in class.) The available lambda parameter is a regularization parameter and I won’t discuss in detail because the details are beyond me but suffice to say if we set lambda\_search = True, the H2O engine will do most of the heavy lifting.

|  |
| --- |
|  |

Below shows the training of the two models, first parameter is the predictor feature list of columns (or single column name for the simple below), followed by the target variable, and finally the dataframe on which to train each model. At least one straightforward way of improving the training would have been to, in an earlier step above, create a third “validation” data frame in addition to test & train and pass this in as the validation\_frame. The resulting cross-validation would likely improve the model’s accuracy. Note the progress bars, which would helpful when building more complicated models.

|  |
| --- |
|  |

Simply executing the model in Jupyter will display a nicely formatted summary of model details. Below is the top portion for the simple model. Output includes the Mean Squared Error aka MSE for the model, which was discussed in class.

|  |
| --- |
|  |

Here is where we do the actual work of making predictions. The first thing to note is that I’ve thrown away the original test dataframe in favor of using the exact same 10 records that were used to test Spark MLlib model in Lecture 11. Best scenario would of course have been to train the model here on the same 90 from that model but I didn’t have time to get the split exactly the same. By training the H2O model on only 80% of the full dataset I’ve at least slight reduced likelihood of overlap between train set of H2O model and imported test set. But I need to use Lec 11 set in order to create an appropriate visualization that compares the predictions between the MLlib and H2O models.

The MSE value is directly available as an attribute on the model, get the square root in order to display a Root Mean Squared Error. Call .predict on the model, passing in the test\_simple data to get a single vector dataframe of predictions. Then to create data for use in the visualization, need values for Displacement and then the actual and predicted values for Horsepower. To create that, call the cbind (aka column bind, as in R) to combine the test dataset with the predictions dataframe. The py\_df\_simple variable was created only as an example to show how easy it is to convert the H2OFrame object into a natively Python list of lists + to show how a pandas dataframe could have been created instead. Finally call .export\_file to create the external .csv. The syntactic coherence of H2O is something I appreciated here as I was googling for the method to export a file. If I remembered that I earlier used import\_file, it would have been easy to guess at this one. Instead I googled for an answer and was only able to find R’s exportFile. From there I could guess the Python equivalent would be in lower case and separated by an underscore.

|  |
| --- |
|  |

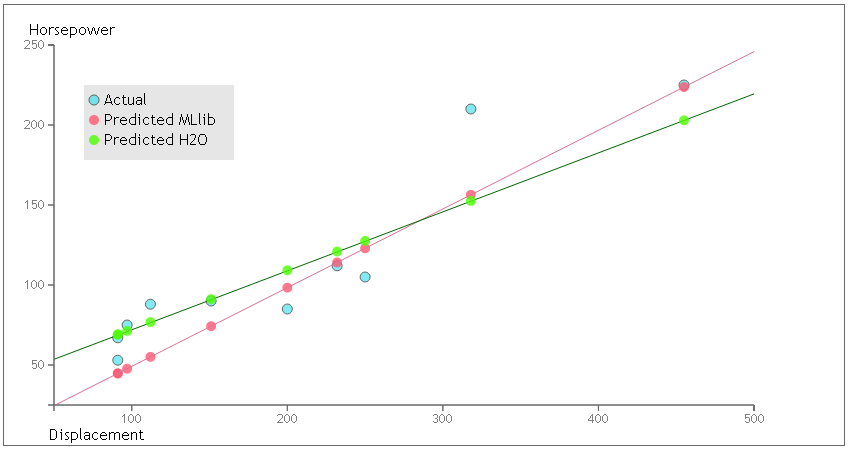
Here is the output of above. We see that H2O came up with a notably more accurate model, going by measurement of RMSE. In and of itself that shows nothing, between the tiny data size and the default model creations used for both H2O and MLlib – an experienced data scientist would no doubt have used additional parameters and perhaps even different “solvers”. And/or she may have judged that there were better approaches than linear regression. The .head on the data to be output shows reasonable guess and the python list shows the more precise predict values held underneath.

|  |
| --- |
|  |

Similar code and output for the multiple linear regression model.

|  |
| --- |
|  |

Now to move on to the data visualization aspect of the small car dataset, beginning with a display of the output. Here we begin with actual 10 Displacement/Horsepower sets from Assignment 11 in blue. Then predictions from the Spark MLlib data from same assignment as red dots, along the related model line in red. Added to that are the values predicted by the H2O model, green dots for specific predictions and green line for the overall model. The corresponding RMSE measurements indicate it is likely the green dots are more accurate predictions than the red but with only ten observations the opposite is possible for this specific example. Either way, pretty to look at but not too much information gained by only ten items.



As for the corresponding code, it is based on the HTML from Assignment 11, with a few alterations to support the dual datasets. I discussed in detail for that Assignment and will attempt a briefer overview below.

Create some global variables that can be set during the initial load and then read by the accessor functions for each D3 data load. Simple linear scales set for each, with data ranges that make sense for the min/max values in the ten record csv.

|  |
| --- |
| var xAxis, yAxis;  var data\_spark, data\_h2o;  var loaded = 0;  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;  var svg = d3.select('body').append('svg')  .attr('width', width + margin.right + margin.left)  .attr('height', height + margin.top + margin.bottom)  .attr('class', 'frame');  //horizontal/Displacement 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]) |

Two d3.csv functions that will load in appropriate data sets. For the H2O data, I’m using the orig csv exported by python and its header values don’t match that used for Assignment 11, so I alias them by creating a new row object for each row of data and return that. Each accessor function calls drawStuff(), where the actual visualization happens.

|  |
| --- |
| d3.csv('spark\_car.csv', function (d) {  return d;  }, function (data) {  data\_spark = data;  drawStuff();  });  d3.csv('car\_SIMPLE.csv', function (d) {  //normalize data row column names to match those used in Lecture 11  var row = {  displacement: d.Displacement,  actual: d.Horsepower,  predicted: d.predict  }  return row;  }, function (data) {  data\_h2o = data;  drawStuff();  }); |

First I’ll look at the setup – what happens as soon as the first dataset is loaded. Create the x/y axes, hard-coding in the desired tick values. Then create a simple legend, playing around to get the x,y coordinates that look best. Some of the css classes, e.g. for dot color, will be shared between legend and the primary svg.

|  |
| --- |
| function drawStuff() {  loaded += 1;  console.log(loaded);  if (loaded < 2) {  xAxis = d3.svg.axis().scale(xScale).orient('bottom')  .tickValues([100, 200, 300, 400, 500]);  yAxis = d3.svg.axis().scale(yScale).orient('left')  .tickValues([50, 100, 150, 200, 250]);  svg.append("text").attr("x", 25).attr("y", 30)  .text("Horsepower").attr('class', 'axis-label');  svg.append("text").attr("x", 45).attr("y", height + 35)  .text("Displacement").attr('class', 'axis-label');  //legend  svg.append("rect").attr("x", 80).attr("y", 80)  .attr("height", 75).attr("width", 150).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 MLlib").attr('class', 'axis-label');  svg.append("text").attr("x", 100).attr("y", 140)  .text("Predicted H2O").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');  svg.append("circle").attr("cx", 90).attr("cy", 135)  .attr('r', 5).attr('class', 'predicted2');  } |

The else block is executed once both datasets have been loaded and first section within plots the actual displacement/horsepower values from the original dataset, in this case the copy that is the spark\_car.csv file.

|  |
| --- |
| else {  console.log('loaded 2x');  var dot\_actual = svg.selectAll('circle-actual')  .data(data\_spark).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);  }); |

Plot the x/y displacement/horsepower values predicted from the Spark MLlib package.

|  |
| --- |
| var dot\_pred\_spark = svg.selectAll('circle-predicted')  .data(data\_spark).enter().append('circle')  .attr('class', function (d) {  return 'predicted';  });  dot\_pred\_spark.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 'MLlib: ' + Math.round(d.displacement) + ' Displacement, Predicted hp: ' + Math.round(d.predicted);  }); |

Same for the H2O dataset.

|  |
| --- |
| var dot\_pred\_h2o = svg.selectAll('circle-predicted')  .data(data\_h2o).enter().append('circle')  .attr('class', function (d) {  return 'predicted2';  });  dot\_pred\_h2o.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 'H2O: ' + Math.round(d.displacement) + ' displacement, predicted hp: ' + Math.round(d.predicted);  }); |

Code for the two model lines, the spark version is from Assignment 11, where the weights value was embedded as a property in the model object. The corresponding slope and intercept for the H2O data is available in the Flow web tool.

|  |
| --- |
| //some quick & dirty algebra gave me a slope = 0.49  //linear\_model.weights = [0.491718981427]  var spark\_model\_weights = 0.491718981427;  svg.append("line").attr("x1", xScale(xMin)).attr("y1", yScale(xMin \* spark\_model\_weights))  .attr("x2", xScale(xMax)).attr("y2", yScale(xMax \* spark\_model\_weights))  .attr('class', 'model\_spark');  var h2o\_disp\_coeff = 0.3686; //0.3686 = Displacement coefficient for model, reported by Flow in H20  var h2o\_intercept = 35.1316; //reported as Intercept in Flow  svg.append("line").attr("x1", xScale(xMin)).attr("y1", yScale(xMin \* h2o\_disp\_coeff + h2o\_intercept))  .attr("x2", xScale(xMax)).attr("y2", yScale(xMax \* h2o\_disp\_coeff + h2o\_intercept))  .attr('class', 'model\_h2o');  } |

And finally the actual x and y axes are drawn below, based on the xAxis and yAxis objects created earlier.

|  |
| --- |
| //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);  }; |

FP\_cars.css with inline comments.

|  |
| --- |
| /\*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\*/  opacity: 0.7;  }  /\*predicted value dots\*/  circle.predicted2 {  fill: #4cff00; /\*green\*/  opacity: 0.7;  }  /\*actual/true value dots\*/  circle.actual {  fill: #30DCEC; /\*blue\*/  opacity: 1;  stroke: black;  }  /\*square around legend\*/  rect.legend {  opacity: 0.10;  }  /\*linear model straight line\*/  line.model\_spark {  stroke: palevioletred;  stroke-width: 1;  }  line.model\_h2o {  stroke: darkgreen;  stroke-width: 1;  } |

**Example via Movie dataset**

The dataset I chose for the report is available as a file named hetrec2011-movielens-2k.zip, which itself contains a series of tab-separated .dat files, see <http://grouplens.org/datasets/hetrec-2011/>. A brief description from the data’s readme:

|  |
| --- |
| Version 1.0 (May 2011)  -----------  Description  -----------  This dataset is an extension of MovieLens10M dataset, published by GroupLeans  research group.  http://www.grouplens.org    It links the movies of MovieLens dataset with their corresponding web pages at  Internet Movie Database (IMDb) and Rotten Tomatoes movie review systems.  http://www.imdb.com  http://www.rottentomatoes.com  From the original dataset, only those users with both rating and tagging information  have been mantained.    The dataset is released in the framework of the 2nd International Workshop on  Information Heterogeneity and Fusion in Recommender Systems (HetRec 2011)  http://ir.ii.uam.es/hetrec2011  at the 5th ACM Conference on Recommender Systems (RecSys 2011)  http://recsys.acm.org/2011 |

I wound up using only a subset of the available files:

* movies.dat, movie\_countries.dat, movie\_directors.dat, movie\_genres.dat

Below is some info from that same readme that relates to the these files. After cleanup the primary dataset winds up being around seven thousand rows.

|  |
| --- |
| 2113 users  10197 movies    20 movie genres  20809 movie genre assignmentss  avg. 2.040 genres per movie  4060 directors  95321 actors  avg. 22.778 actors per movie  72 countries  10197 country assignments  avg. 1.000 countries per movie |

To get the data into my local SQL Server, used the relatively simple Import Data function, uses below wizard.

|  |
| --- |
|  |

The only real tricky part came when adjusting mappings since the wizard defaults to importing everything as 50 character string data types, which often truncated data and/or imported numbers as strings. The below .dat happens to only have two columns, I changed the movieID into an INT that can be joined on other tables.

|  |
| --- |
|  |

The wizard has several screens, at the end is a summary.

|  |
| --- |
|  |

There were a number of other cleanup tasks I needed to, like removing duplicate rows from the primary movie dataset. In fact I imported the primary movie.dat into table named raw\_movies, meaning to create a cleaned table named ‘movies’ but I ran ahead with queries instead and didn’t get around to it. I’ll provide a link to the full dataset as imported into H2O, hosted on my web site - here it is: [FP\_movies.psv](http://oweng.net/csci-e63/FP_movies.psv).

The final project is targeted toward learning the technology, both for me and then for colleagues. Given that all the data munging etc. is presumably a secondary concern at best. So the next huge block of code is not particularly helpful in that regard. On the other hand it involves feature engineering, which can be a key aspect of machine learning. Of just as much import -- it took a good bit of effort so I felt the need to include it in the report.

Overall idea for the eventual linear regression model was to predict one of the observations available in the data, “rtAllCriticsRating” using other values in same data, whether in their raw form or in some kind of value-add feature engineered value. The Rotten Tomatoes website acts in part as an aggregator of ratings given both by professional movie critics and regular moviegoers. Within the data, the rtAllCriticsRating column holds the average rating given by movie critics who appear on the site, with a scale from 1 to 10.

Here is the query, comments are inline, contents are in e63Q1.sql should it matter.

|  |
| --- |
| use Movies  go  --Where there is no AudienceRating, substitute the mean value over all movies  DECLARE @AvgAudRating DECIMAL(6,1);  SET @AvgAudRating = (SELECT AVG(CAST(m.rtAudienceRating AS float)) FROM raw\_movies m  WHERE CAST(rtAudienceRating AS float) > 0 ) --max of 5, min of 1.5 in this dataset  --SELECT @AvgAudRating -- average = 3.3900  --For StarPower, try to weigh by the number of films by that country in the dataset  -- Values range from 1.0 for countries w/few entries, e.g. Greece only has 3, to 0.33 for USA with over 6k  ;WITH cteCountry  AS (  select movieID, country,  CAST(1 - COUNT(\*) OVER (PARTITION BY country)/(COUNT(\*) OVER() \* 1.0) AS DECIMAL(18,2)) as CountryFactor  from movie\_countries  )  --movie\_actors has a Ranking = order in which they appear on IMDB for a given movie  -- Grab a subset containing only actors who were given billing w/in top 3 movies  -- Tom Cruise is included in Minory Report because he had top billing, ranking = 1  -- but not included for Tropic Thunder where his ranking = 16 since he only had a cameo  ,cteMostPop  AS (  SELECT actorID, actorName, COUNT(\*) As MostPopCount  FROM movie\_actors  WHERE ranking < 4  GROUP BY actorID, actorName  )  --Similar to above, but include rankings from 4 to 7 inclusive  -- These will receive less weighting when calculating StarPower, vs. above  -- 'not\_applicaple' was a manual cleanup step, where I updated certain rows in movie\_actors  -- to have that value if there was no valid actor name in the data - only affected 2 records below  ,cteQuitePop  AS (  SELECT actorID, actorName, COUNT(\*) As QuitePopCount  FROM movie\_actors  WHERE ranking BETWEEN 4 AND 7 AND actorID <> 'not\_applicable'  GROUP BY actorID, actorName  )  --For TotalPopCount, any appearance in the top 3 ranking is given twice the weight of appearance in 4-7  ,cteStarPower AS  (  SELECT mp.actorName, (MostPopCount \* 2) + ISNULL(QuitePopCount, 0) AS TotalPopCount  FROM cteMostPop mp  LEFT JOIN cteQuitePop qp ON qp.actorID = mp.actorID  )  -- Now getting into even more arbitrary and ad-hoc calculations, with a strong emphasis on "arbitrary"  -- Playing around to get a gut feel for what might make sense given a business domain = movie ratings  -- MovieStarPowerWCntry is the numeric value most of the previous calculations were aiming for  ,cte1 AS  (  SELECT m.id, m.title, m.year, c.country, d.directorName, a.actorName, a.ranking, m.rtAllCriticsRating  ,SUM(sp.TotalPopCount) OVER (PARTITION BY m.id) As MovieStarPower  ,SUM(sp.TotalPopCount) OVER (PARTITION BY m.id) \* c.CountryFactor As MovieStarPowerWCntry  ,CASE WHEN CAST(m.rtAudienceRating AS DECIMAL(6,1)) = 0  THEN @AvgAudRating ELSE CAST(m.rtAudienceRating AS DECIMAL(6,1))  END AS AudienceRating  FROM raw\_movies m  INNER JOIN movie\_actors a ON a.movieID = m.id AND a.ranking < 4 --have to have appeared in top 3 billing at least once  INNER JOIN cteStarPower sp ON sp.actorName = a.actorName --but rank 4 to 7 still counts in popularity  INNER JOIN cteCountry c ON c.movieID = m.id  INNER JOIN movie\_directors d ON d.movieID = m.id  WHERE 1=1  -- AND m.id < 100  AND ISNUMERIC(m.rtAllCriticsRating) = 1 AND CAST(m.rtAllCriticsRating AS float) > 0  AND ISNULL(m.note, '') <> 'bad\_actor\_ranking' --account for bad data, or at least bad data in terms of my model  )  -- Include the top 3 actor names for each movie in the dataset, don't really now if these will be useful in model generation  -- Some movies don't even list 3 people, use placeholder values in that scenario.  ,cte2 AS  (  SELECT id, title, year, country, directorName  ,[1] AS Actor1  ,ISNULL([2], CASE WHEN id % 2 = 0 THEN 'George Spelvin' ELSE 'Georgette Spelvin' END) AS Actor2  ,ISNULL([3], CASE WHEN id % 2 = 0 THEN 'Georgette Spelvin' ELSE 'George Spelvin' END) AS Actor3  ,rtAllCriticsRating, MovieStarPower, MovieStarPowerWCntry, AudienceRating  --KEY LINE HERE = after my early models showed MovieStarPowerWCntry having almost no predictive ability I felt  -- it necessary to add in AudienceRating... and kept on reducing impact of MovieStarPowerWCntry until it had  -- very little weighting in the final value. Oh well.  ,(MovieStarPowerWCntry/100) + AudienceRating AS StarAud  FROM cte1  PIVOT (  MAX(actorName)  FOR ranking IN ([1],[2],[3])  )  AS pvt  )  --movie\_genres has one-to-many between movies and 20 possible genres, e.g. Crime, Musical, Film-Noir  -- Each movie has at least one, highest has 8 (for 'Gwoemul', Korean movie better known as 'The Host' in US)  -- Goal is to wind up with three movie genres that movie is tagged with, where the top 3 "rarest" genres are included  -- Rarity is calc based on number of movies for that genre in the dataset, idea is that if a movie is tagged with both  -- 'Drama' (5076 movies) and 'Western' (261), it is more important that the 'Western' tag be included since that  -- would likely have a greater affect on a critic's evaluation vs. it being a Drama  -- Next 4 queries all deal with this genre calculation.  ,cteGenreRnk AS (  SELECT A.genre, ROW\_NUMBER() OVER (ORDER BY cnt) AS rnk  FROM (  SELECT genre, COUNT(\*) as cnt  FROM movie\_genres  GROUP BY genre  ) A  )  ,cteGenres AS  (  SELECT g.movieID, g.genre, rnk.rnk, ROW\_NUMBER() OVER (PARTITION BY g.movieID ORDER BY rnk.rnk) AS FinalRank  FROM movie\_genres g  INNER JOIN cteGenreRnk rnk ON rnk.genre = g.genre  )  ,cteTopGenreRnk AS  (  SELECT rm.id, rm.title, rm.Year, g.genre, g.FinalRank--, g.rnk  FROM raw\_movies rm  LEFT JOIN cteGenres g ON g.movieID = rm.id  WHERE 1=1  AND g.FinalRank <= 3  )  ,cteGenrePivot AS  (  SELECT id, title, year  ,[1] AS Genre1  ,COALESCE([2], [1]) AS Genre2 --if only one genre, put that here  ,COALESCE([3], [1]) AS Genre3 --if only two genre tags, put most-rare here  FROM cteTopGenreRnk  PIVOT (  MAX(genre)  FOR FinalRank IN ([1],[2],[3])  )  AS pvt  )  SELECT c2.\*, p.Genre1, p.Genre2, p.Genre3  --bonus feature = presuming female given names more likely to end with letter 'a',  -- throw into the mix the likelihood that top billed is an actress  ,CASE WHEN SUBSTRING(c2.actor1, CHARINDEX(' ', c2.actor1, 0)-1, 1) = 'a' Then 'T'  ELSE 'F' End AS Actor1NameEndsInA  FROM cte2 c2  INNER JOIN cteGenrePivot p ON p.id = c2.id |

One of the main points for including the query was the one highlighted line from above, which appears again below. Originally I wanted to predict Box Office earnings by using features that included the actors and their billing, i.e. did they have a lead role, but had trouble finding an open dataset. I carried over the initial idea into predicting the (average) rating a movie received from film critics. The movie\_actors table had a many to one relation to the raw\_movies table – it might list 10 actors for a movie, each one with a rating as to how high they appeared on the correspoinding IMDB page for the movie. Higher ranking value presumably meant higher billing, in advertisements etc. One of the primary calculations in the above SQL is a numeric MovieStarPowerWCntry that attempted to measure the star power of a movie (if top actors in given movie received high ranking in other movies, star power increases) but also tried to account for movies from smaller countries that don’t have much representation in the dataset (top movie in Greece would have low star power since actors didn’t appear in many/any other movies in data). Unfortunately that value, in an early H2O simple linear regression model, had very little predictive power, with almost all predictions within a point of the overall rtAllCriticsRating for all movies. Resulting D3 graph showed a very steep line centered around 6.0.

From there I added in the AudienceRating, a numeric value ranging from 1 to 5, reducing the influence of MovieStarPowerWCntry until I got a more reasonable model (and graphical output). Using AudienceRating alone would probably result in a lower MSE but I don’t really want to know that and using that alone does not result in another objective of the final project, that of having fun. A typical calculation for the below might be something like ‘Crimson Tide’, which had a calculated MovieStarPowerWCntry = 62.37 and an Audience Rating = 3.5, resuling in a StarAud value ( the explanatory variable for simple regression example) = 4.1237.

|  |
| --- |
| ,(MovieStarPowerWCntry/100) + AudienceRating AS StarAud |

Below shows how a few rows from the dataset and it reveals one immediate problem – any movie with JCVD should automatically get a StarPower rating of 1000 ☺.

(bigger pic = <http://oweng.net/csci-e63/data_snapshot.png> )

|  |
| --- |
|  |

The SQL query produces 7070 rows and that becomes the final dataset to be used in H2O. Export as pipe-delimited file, text version of above snapshot is below.

|  |
| --- |
| id|title|year|country|directorName|Actor1|Actor2|Actor3|rtAllCriticsRating|MovieStarPower|MovieStarPowerWCntry|AudienceRating|StarAud|Genre1|Genre2|Genre3|Actor1NameEndsInA  2|Jumanji|1995|USA|Joe Johnston|Robin Williams|Bonnie Hunt|Kirsten Dunst|5.6|159|52.47|3.2|3.724700|Children|Fantasy|Adventure|F  4|Waiting to Exhale|1995|USA|Forest Whitaker|Whitney Houston|Angela Bassett|Loretta Devine|5.6|52|17.16|3.3|3.471600|Romance|Comedy|Drama|F  9|Sudden Death|1995|USA|Peter Hyams|Jean-Claude Van Damme|Powers Boothe|Raymond J. Barry|5.2|75|24.75|2.6|2.847500|Action|Action|Action|F  11|The American President|1995|USA|Rob Reiner|Michael Douglas|Annette Bening|Martin Sheen|7|138|45.54|3.2|3.655400|Romance|Comedy|Drama|F  18|Four Rooms|1995|USA|Alexandre Rockwell|Tim Roth|Valeria Golino|Jennifer Beals|3.5|82|27.06|3.5|3.770600|Thriller|Comedy|Drama|F |

Now back to H2O and a new Jupyter notebook named FP\_movies\_simple – as implied by the name, this notebook will only handle simple regression, using my calculated StarAud value to predict rtAllCriticsRating number. To re-iterate, StarAud represents the average audience rating of a given movie + some (modest) arbitrary weighting that attempts to measure how many “big actors” also had major roles in said movie.

This review can go more quickly since most details were covered in the earlier small car tutorial. Import h2o module and intitialize, show partial output.

|  |
| --- |
|  |

Import the main GLM module from H2O and numpy to help out later. Import file from local directory and print out the shape property of resulting H2OFrame, which shows 7,071 rows including header and 17 columns/observations.

|  |
| --- |
|  |

There is enough data here to warrant a histogram, by first turning the H2OFrame into a python list of lists and transforming that into a simple list of rtAllCriticsRating values. Looks like a reasonably normal distribution of data with a slight left skew. (Last minute problem with variable names forced me to summarize only 3000 records.)

|  |
| --- |
|  |

Peak at train data.

|  |
| --- |
|  |

And at test.

|  |
| --- |
|  |

Create the model and train on the training data. Goal is to predict rtAllCriticsRating and we can see I progressively tried various columns by themselves, first the MovieStarPower and then MovieStarPowerWCntry that tried to normalize the data by taking into account country origin distribution within the dataset. Then finally the composed StarAud column that is primarily the AudienceRating with a bit of MovieStarPowerWCntry.

|  |
| --- |
|  |

Next step is to make the predictions and a csv consisting of the movie title, predictor variable (StarAud), the actual rating (rtAllCriticsRating), and the predicted rating. Note the RMSE = 1.25, which would be based on the 1 to 10 scale used for the rating. As such, 1.25 indicates a decent amount of predictive power but can imagine it could be improved upon.

|  |
| --- |
|  |

Finally, create a shorter version of the predictions by using split\_frame to take a small sampling. These 143 are what will go into the D3 display.

|  |
| --- |
|  |

I’m not going to discuss the Flow tool in this report, in part because I’m not able to adequately interpret most of the very detailed data presented. But I will illustrate the one thing I use it for by opening a web page on the H2O machine and pointing to <http://localhost:54321/flow/index.html>. Within there I can click on the getModels hyperlink below

|  |
| --- |
|  |

Which will display a list of actively generated models, using the tag passed in as model\_id when the model was created, assuming that was done.

|  |
| --- |
|  |

Click on the model\_id hyperlink, open the OUTPUT- COEFFICIENTS section and note the Intercept and independent variable coefficient values – those will be used in D3.

|  |
| --- |
|  |

Moving on to the D3 visualization, things are very similar to that of the cars example, enough so that there really isn’t much to explain, beyond it being simpler because there is only the one .csv to process. Of course the range for the scale is different, adjusted for movie data. I’ll discuss the model line after the code.

|  |
| --- |
| var xAxis, yAxis;  var data;  var remaining = 2;  var loaded = 0;  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;  var svg = d3.select('body').append('svg')  .attr('width', width + margin.right + margin.left)  .attr('height', height + margin.top + margin.bottom)  .attr('class', 'frame');  //horizontal  var xMin = 1, xMax = 6;  var xScale = d3.scale.linear()  .domain([xMin, xMax])  .range([50, width]);  var yScale = d3.scale.linear()  .domain([1, 10])  .range([height, 40])  d3.csv('movie\_SIMPLE\_out\_SHORT.csv', function (d) {  var row = {  title: d.title,  starAud: d.StarAud,  actual: d.rtAllCriticsRating,  predicted: d.predict  }  return row;  }, function (d) {  data = d;  drawStuff();  });  function drawStuff() {  xAxis = d3.svg.axis().scale(xScale).orient('bottom');  yAxis = d3.svg.axis().scale(yScale).orient('left');  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);  svg.append("text").attr("x", 20).attr("y", 20)  .text("Star Power + Audience Rating").attr('class', 'axis-label');  svg.append("text").attr("x", 50).attr("y", height + 35)  .text("RT Critics Rating").attr('class', 'axis-label');  //legend  svg.append("rect").attr("x", 70).attr("y", 80)  .attr("height", 50).attr("width", 200).attr('class', 'legend');  svg.append("text").attr("x", 100).attr("y", 100)  .text("Actual RT Critic Rating").attr('class', 'axis-label');  svg.append("text").attr("x", 100).attr("y", 120)  .text("Predicted H20").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');    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.starAud);  })  .attr('cy', function (d) {  return yScale(d.actual);  })  .attr('r', function (d) {  return 3;  })  .append('title')  .text(function (d) {  return '"' + d.title + '"\r\n'  + Math.round(d.starAud \* 100) / 100  + ' StarAud, actual RT Rating: '  + Math.round(d.actual \* 100) / 100;  });  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.starAud);  })  .attr('cy', function (d) {  return yScale(d.predicted);  })  .attr('r', function (d) {  return 3;  })  .append('title')  .text(function (d) {  return '"' + d.title + '"\r\n'  + Math.round(d.starAud \* 100) / 100  + ' StarAud, actual RT Rating: '  + Math.round(d.predicted \* 100) / 100;  });  //H20 Flow, info from glm model  var coeff = 1.7855;  var intercept = -0.5835;  svg.append("line").attr("x1", xScale(xMin)).attr("y1", yScale(xMin \* coeff + intercept))  .attr("x2", xScale(xMax)).attr("y2", yScale(xMax \* coeff + intercept))  .attr('class', 'model');  }; |

I wanted to discuss the last few lines highlighted above. Here is where I was able to just plug in the intercept and StarAud coefficient values in order to have D3 draw a full line that covers the full axes range.

Not much new with .css either here it is.

|  |
| --- |
| /\*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 on axes\*/  .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. 'RT Critics Rating' \*/  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, will be on the line so reduce opacity\*/  circle.predicted {  fill: #FF5872; /\*red\*/  opacity: 0.4;  }  /\*actual/true value dots\*/  circle.actual {  fill: #30DCEC; /\*blue\*/  }  /\*square around legend\*/  rect.legend {  opacity: 0.10;  }  /\*linear model straight line\*/  line.model {  stroke: red;  stroke-width: 1;  } |

And now the data viz. We can see some limitations of the feature approach – “Orgy of the Dead” had a predicted value up around 5 but an actual average critics rating of 1.2, can probably be considered an outlier. One can assume that a general audience member for a film with that title will have certain expectations and their ratings will be correlated to those presumably positive expectations (otherwise why pay to see the movie, in both money and time?). A critic will likely have negative preconceptions and will be trying to judge against a wider range of movie standards.

If movie sounds interesting to the reader, IMDB summarizes the plot as “A captive couple watch as half-naked ghouls dance for their gleeful emperor.” Director = Ed Wood.

|  |
| --- |
|  |

Next we will review a multiple linear regression example using the same movies dataset.

First few pieces of python code are exactly the same as with simple linear regression and will be skipped over. Same seed is used for the test/train split so as I understand we should have same data pieces as in previous example.

Of course the model creation will be different, though with the same target variable. The features aka explanatory variables can be submitted as a simple Python list. I chose the below list of features for the example but a more serious attempt would involve testing different sets of columns, also no doubt there is information available in the model details that give an idea as to how much weight each of the features influence the final prediction.

|  |
| --- |
|  |

Also important to note regarding the features is this note from the H2O documentation, which indicates the binary vector expansion we went over in class for categorical variables in MLlib is simply unnecessary with H2O. There are probably caveats but everything seems to be working as-is.

|  |
| --- |
|  |

Either way, some model details are output. Note that if the explanatory variable list is reduced to only year and country the number\_of\_active\_predictors jumps up to 58, so this is not directly related to the number of features. No doubt there is documentation detailing everything. Beyond that, we can see the MSE = 1.1055 is lower than the 1.556 registered for the simple regression model, an almost 30% improvement in raw comparison.

|  |
| --- |
|  |

I’ll go ahead and create an output file (and display RMSE), though there isn’t going to be an attempt at trying to visualize this particular model.

|  |
| --- |
|  |

Output for above, 1.0514 RMSE vs. 1.2473 with simple model

|  |
| --- |
|  |

That concludes my report, thank you for reading.

**YouTube URLs here: long:** [**https://youtu.be/OxjCc99Jer0**](https://youtu.be/OxjCc99Jer0) **short:** [**https://youtu.be/uAxmUhqxOFY**](https://youtu.be/uAxmUhqxOFY)