Lecture 11

Machine Learning with Spark MLlib

Zoran B. Djordjević

Reference

- This lecture follows to a great measure the text:

  Machine Learning with Spark by Nick Pentreah, PAKT Publishing, 2015

- Another very good book on advanced Spark processing:

  Advanced Analytics with Spark, by Sandy Ryza et al., O'Reilly, 2015

Need for Machine Learning System

1. The scale of data involved means that full human involvement quickly becomes infeasible as any business grows.

2. If we could write software that would do everything the humans do we could get rid of humans. They are so expensive.

3. Model-driven approaches such as machine learning and statistics can often benefit from uncovering patterns that cannot be seen by humans (due to the size and complexity of the datasets)

4. Model-driven approaches can avoid human and emotional biases (as long as the correct processes are carefully applied)

Spark MLLib

- MLlib is Spark's machine learning (ML) library. Its goal is to make practical machine learning scalable and easy.

- Mllib consists of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as lower-level optimization primitives and higher-level pipeline APIs.

- Spark's Machine Learning API divides into two packages:
  - `spark.mllib` contains the original API built on top of RDDs.
  - `spark.ml` provides higher-level API built on top of DataFrames for constructing ML pipelines.

- Using spark.ml is recommended because with DataFrames the API is more versatile and flexible.

- Spark will keep supporting spark.mllib along with the development of spark.ml.

- Users should be comfortable using spark.mllib features and expect more features coming.

- Developers should contribute new algorithms to spark.ml if they fit the ML pipeline concept well, e.g., feature extractors and transformers.;

MLlib Supported Features and Algorithms

- Basic statistics
  - summary statistics and correlations
  - stratified sampling and hypothesis testing
  - streaming significance testing
  - random data generation
- Classification and regression
  - linear models (SVMs, logistic regression, linear regression)
  - naive Bayes
  - decision trees
  - ensembles of trees (Random Forests and Gradient-Boosted Trees)
  - isotonic regression
- Collaborative filtering
  - alternating least squares (ALS)
- Clustering
  - k-means
  - Gaussian mixture
  - power iteration clustering (PIC)
  - latent Dirichlet allocation (LDA)
  - bisecting k-means and streaming k-means
- Dimensionality reduction
  - singular value decomposition (SVD)
  - principal component analysis (PCA)
- Feature extraction and transformation

Spark MLlib is about High End Scalability

- For large problems MLLib scales better than competing ML packages, Vowpal Wabbit or Matlab.

- Image below indicates that Mllib when applied to a regression analysis does not scale anywhere close to the ideal linear behavior, but still scales better than its close competitor Vowpal Wabbit.
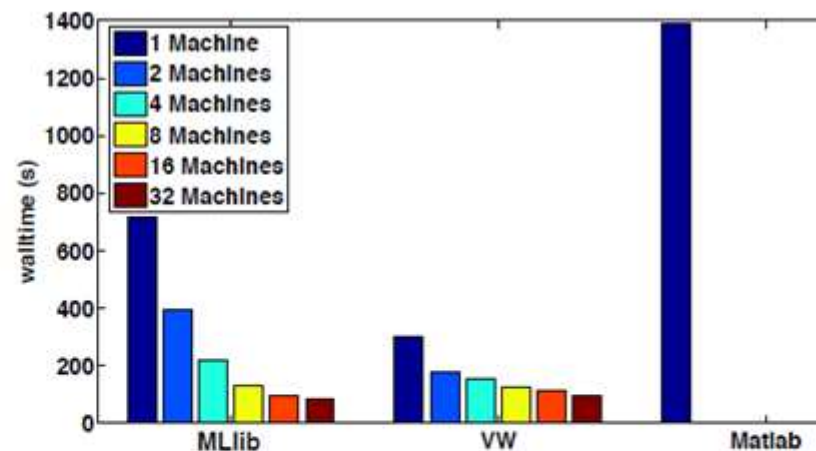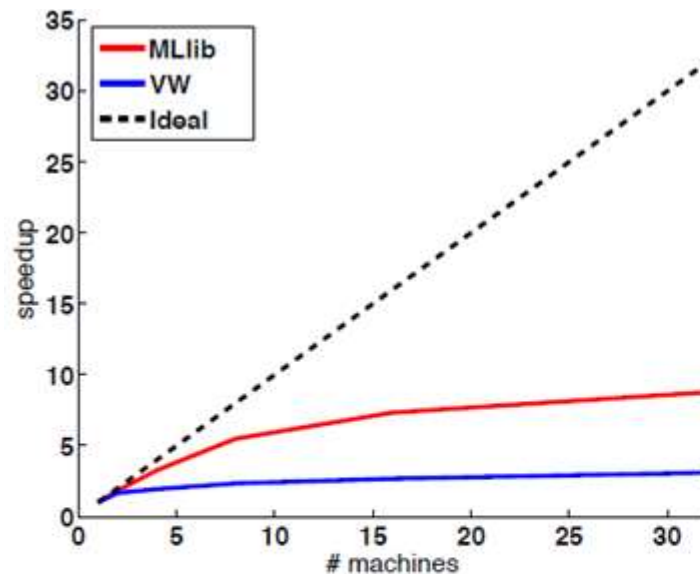
- Fixed Dataset: 50K images, 160K dense features.
- MLlib exhibits better scaling properties.
- MLlib is faster than VW with 16 and 32 machines.

For small data sets use other packages

- If your datasets are small or moderate in size, you are better of with one of many Machine Learning toolkits:

- Scikit-learn is very comprehensive Python Machine Learning toolkit.
- Matlab can do anything that smacks of Math and Statistics. Large number of implemented ML algorithms
- R – your favorite. Does not scale as well as MLLib but has a much larger number of implemented algorithms
- WEKA is Java
- Rapid Miner is Java
- Apache Mahout is Java uses Spark in the background. Used to be very popular.
- Microsoft Azure ML is an excellent platform that scales well.
- Accord.MachineLearning is a .Net package
- Vopal Wabbit is C++
- MultiBoost and Shogun are C++

Some Public Datasets

- University of California Irvine: http://archive.ics.uci.edu/ml/

  This is a collection of almost 300 datasets of various types and sizes for tasks including classification, regression, clustering, and recommender systems.

- Amazon AWS public datasets: http://aws.amazon.com/public-data-sets/

  Some of available datasets are:

  - Landsat on AWS: An ongoing collection of moderate-resolution satellite imagery of all land on Earth produced by the Landsat 8 satellite
  - NASA NEX: A collection of Earth science data sets maintained by NASA, including climate change projections and satellite images of the Earth's surface
  - Common Crawl Corpus: A corpus of web crawl data composed of over 5 billion web pages
  - 1000 Genomes Project: A detailed map of human genetic variation
  - Google Books Ngrams: A data set containing Google Books n-gram corpuses
  - US Census Data: US demographic data from 1980, 1990, and 2000 US Censuses

Some Public Datasets

- *Kaggle,* a collection of datasets used in machine learning competitions. Areas include classification, regression, ranking, recommender systems, and image analysis. These datasets can be found under the *Competitions* section at: http://www.kaggle.com/competitions

- *Europeana Linked Open Data* contains open metadata on 20 million texts, images, videos and sounds gathered by Europeana - the trusted and comprehensive resource for European cultural heritage content: http://labs.europeana.eu/api/linked-open-data/introduction/

- *MIT Cancer Genomics gene expression datasets and publications* from MIT Whitehead Center for Genome Research:

- *KDnuggets*: http://www.kdnuggets.com/datasets/index.html contains a large number of links to very large and interesting datasets and code sources.

Python, Ipython, if not on Clouder VM

- My CentOS 6.5 came with Python 2.6.6., what is good enough for Spark.

- Ipython is an advanced interactive shell for Python. Also, to do Machine Learning, you have to engage in some Math (even do fractions ☹) and you need a bunch of Python's Math packages.

- If you want Python that works for ML and Math rather than fish around for various Python Math libraries, you are advised to install a prebuilt scientific Python, such as Anaconda, available at http://continuum.io/downloads.

- Once you download `Anconda-2.2.0-Linux-x86_64.sh` (or whatever is the latest 64 bit version), you run the command

`$ bash Anconda-2.2.0-Linux-x86_64.sh`

- You type `bash` whether you run bash shell or some other Linux shell. Also, read the license and answer `yes`, when asked, otherwise the installation would not proceed. You will be asked for the installation directory. You can choose `/usr/local` or whichever directory you prefer.

- Once the installation is done, in your `.bash_profile` file, create `ANACONDA` environmental variable pointing to the above installation directory and place `$ANACONDA/bin` in your `PATH`.

- Do not do this on Cloudera VM with CDH 5.5. It uses Python 2.6 which does not support Ipython.

Install `numpy, scipy`

- For Machine Learning you must have linear algebra and other Math libraries.

- NumPy is and an excellent library. Another is SciPy. You should install both.

```
$ sudo yum -y  install python2.7   # Don't do this on Cloudera VM

$ sudo yum install numpy scipy
```

`IPython, PySpark` console, not on Cloudera VM

- If you examine `../anaconda/bin` you will see a bunch executables. One of them is Python installation utility `pip` and the other is `ipython`, the interface we want to use. If you try running

`$pip install ipython`

- `pip` will probably tell you: `ipython` is there and there is nothing to do.

- PySpark allow us to choose which option to use when running Python shell. You do that by adding these to your `.bash_profile` file:

```
IPYTHON=1
export IPYTHON
IPYTHON_OPTS="—-pylab"
export IPYTHON_OPTS
```

- If you care, `pylab` is one of Python libraries that makes IPython resemble MATLAB interface and patterns. MATLAB is a commercial software tool for people who are not afraid of fractions and Java.

- Once you source your `.bash_profile` file, and invoke `pyspark,` your console will look like the one on the next slide.

IPython notebook

- If we change `IPYTHON_OPTS` to `IPYTHON_OPTS=notebook` and then run `pyspark`, a web server (Tornado) will be started and your default browser will pop-up open at port 8888.

- The web application in the browser is the `Notebook`.

- To become a Python geek, you are obliged to use IPython notebook.

Regression Models with Spark

- Regression is concerned with predicting target variables that can take any real value. In statistics, **linear regression** is an approach for modeling the relationship between a scalar target (dependent) variable(s) y and one or more explanatory variables (independent variables, predictors or features) denoted X.

- The case of one explanatory variable is called *simple linear regression*. For more than one explanatory variable, the process is called *multiple linear regression*.
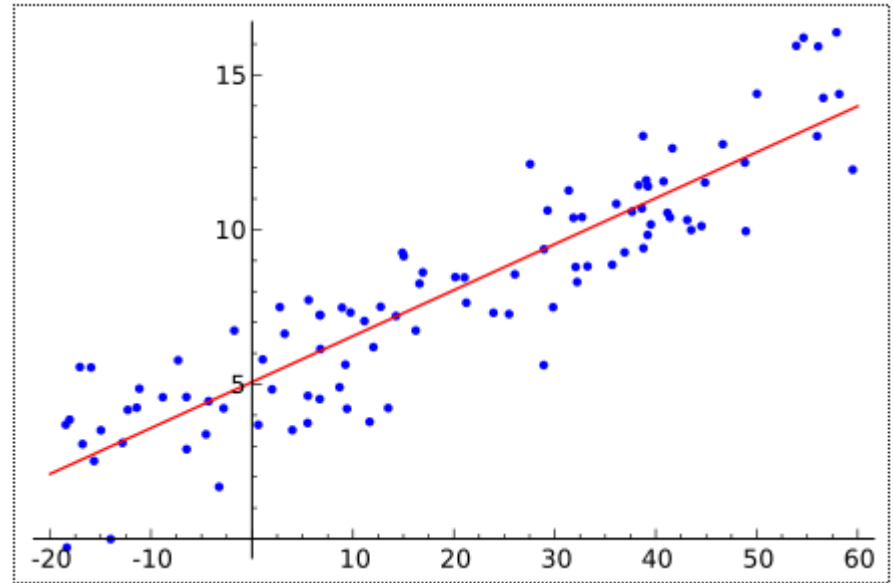
One is frequently asked how is this technique related to learning and predicting. As we will see, regression presumes that there exists a relationship (a model) between features and the target variables. Learning process is the process of determining that relationship. Once we have the relationship we could use it to estimate (predict) new target values, give new predictors.

Experimental Data with many (n) independent variables (features)

Collect prices of house. There are multiple features (variables).

Size (feet²)	Number of bedrooms	Number of floors	Age of home (years)	Price ($1000)
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
…	…	…	…	…

Notation:

n = number of features

$x^{(i)}$ = input (features) of ith training example.

$x_j^{(i)}$ = value of feature j in i^{th} training example.

Hypothesis and the Cost Function

- First of all we make a hypothesis that the best fit to our cloud of experimental points is a linear function of independent variables, i.e. features.
- If we had one feature the hypothesis would be a simple line:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

- When we have many feature it is a multidimensional "plane"

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- For convenience of notation, define $x_0 = 1$ and write the hypothesis as:

$$h_\theta(x) = \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

- We are after n unknown parameters $\theta_0, \theta_1, \ldots, \theta_n$

- We determine those by minimizing a cost function, usually defined as:

$$J(\theta_0, \theta_1, \ldots, \theta_n) = \frac{1}{2\,n} \sum_{i=1}^{n} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Iterative Solution: Gradient Descent

- For a small number of features and measurements the above "optimization" problem could be solved exactly. In most instances, when we have large number of features and a very large number of measurements, the optimal value of the set of parameters $\{\theta_i, i = 0, \dots, n\}$ has to be determined numerically.
- To that end we usually use an iterative method called Gradient Descent

Gradient descent: Repeat $\left\{ \ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \dots, \theta_n) \right\}$

(simultaneously update for every $j = 0, \dots, n$)
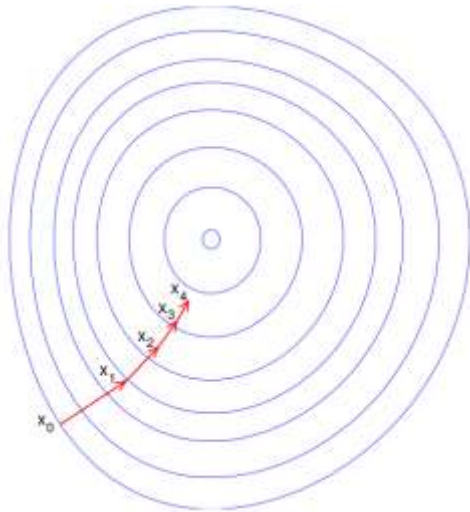
- For simple cost functions this method converges rapidly and accurately. For complex cost functions it might not.
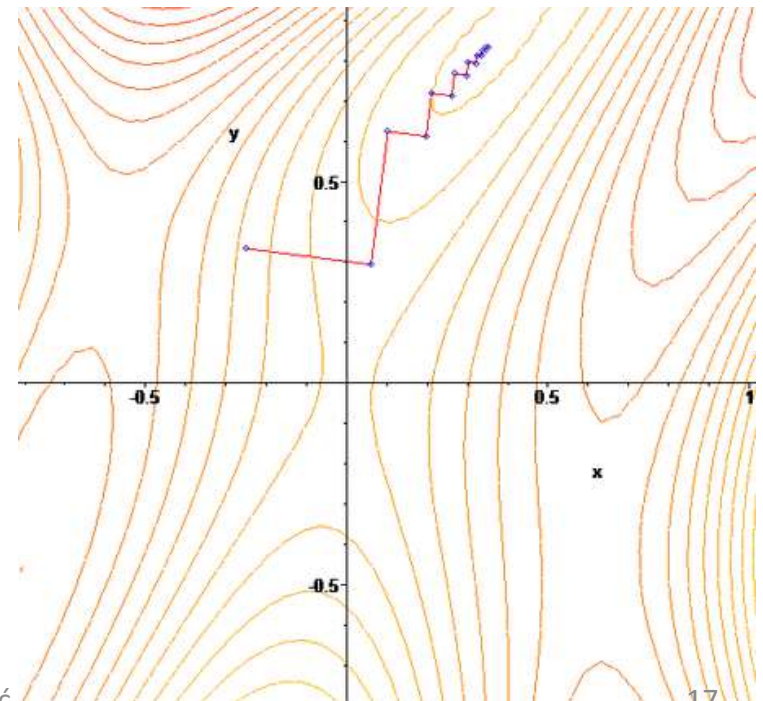
Gradient Descent

Previously (n=1):

Repeat $\{$

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$$

$$\underbrace{\qquad\qquad\qquad}_{\frac{\partial}{\partial \theta_0} J(\theta)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})x^{(i)}$$

(simultaneously update θ_0, θ_1)

$\}$

New algorithm $(n \geq 1)$:

Repeat $\{$

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

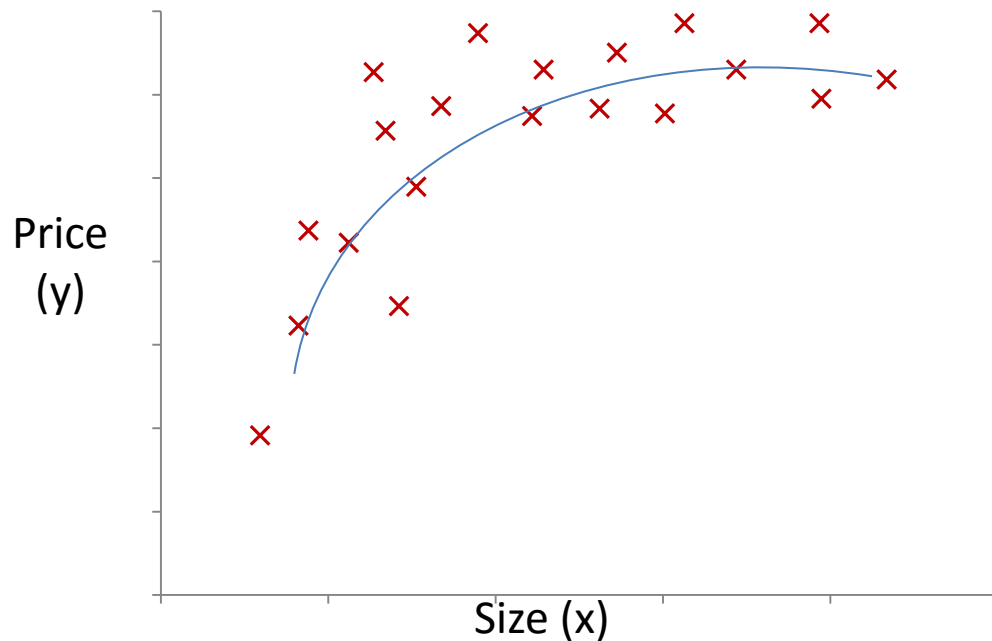(simultaneously update θ_j for

$$j = 0, \ldots, n)$$

$\}$

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})x_0^{(i)}$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})x_1^{(i)}$$

$$\theta_2 := \theta_2 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})x_2^{(i)}$$

Polynomial Regression

- Quite often it is simply obvious from the experimental data that a "liner" model is a poor fit.
- A simple way to extend the regression analysis is to add "new (derived) features". Those could be higher powers of the independent variables.

$$\theta_0 + \theta_1 x + \theta_2 x^2$$

$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3$$

- The problem is still a linear optimization or search for the best parameters $\{\theta_i, i = 0, \dots, n\}$ which minimize the cost function $J(\theta_i)$.

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$$
$$= \theta_0 + \theta_1 (size) + \theta_2 (size)^2 + \theta_3 (size)^3$$

$$x_1 = (size)$$
$$x_2 = (size)^2$$
$$x_3 = (size)^3$$

Avoid Overfitting

- For a small number of experimental points `n` one might be tempted to say: If I have `n` points, I do not have to look for an approximate solution, I could find a polynomial of order `n-1` which passes exactly through those points.
- Curiously, this would result in a very poorly "fitting" function, which is unsuitable for predicting future results. Such polynomial would "over-fit" the solution.
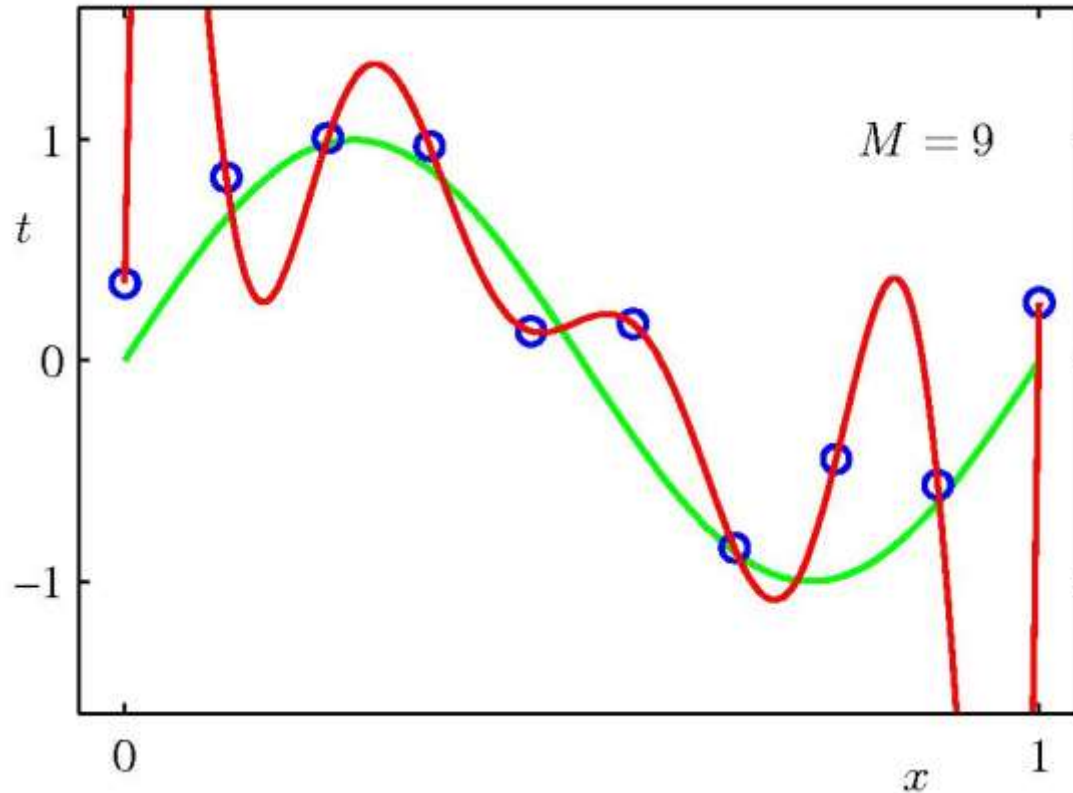
Regularization

- One "automatic" way of avoiding overfitting is to add "regularization" term to our cost function.
- We add a nonlinear term to the cost function which prohibits large values of parameters and then choose vector θ to minimize the overall cost function:
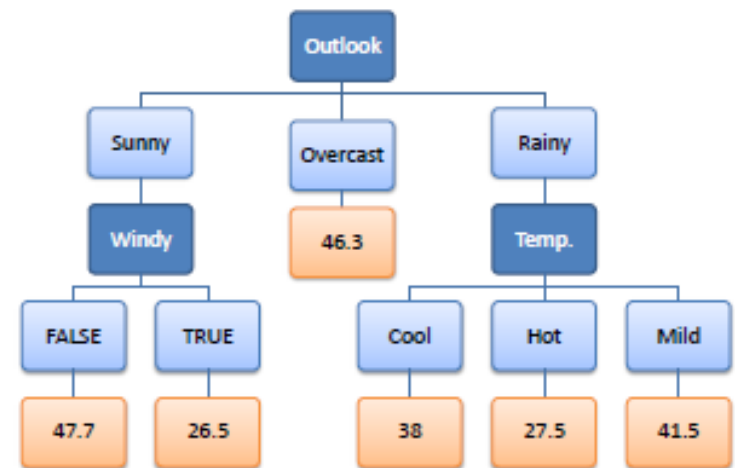
$$J(\theta) = \frac{1}{2m} \left[\sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \lambda \sum_{j=1}^{n} \theta_j^2 \right]$$

- $\lambda \sum_{j=1}^{n} \theta_j^2$ is the regularization term. We vary λ as well.
  This quadratic regularization is referred to as L2 regularization

- Another popular regularization term uses absolute values: $\lambda \sum_{j=1}^{n} |\theta_j|$ of unknown parameters. This term is referred to as L1 regularization. You choose one or the other or some different regularization which works the best with your concrete problem.

Decision Tree - Regression

- Decision tree builds regression or classification models in the form of a tree structure. It brakes down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed.

- The final result is a tree with **decision nodes** and **leaf nodes**. A decision node (e.g., Outlook) has two or more branches (e.g., Sunny, Overcast and Rainy), each representing values for the attribute tested. Leaf node (e.g., Hours Played) represents a decision on the numerical target.

- The topmost decision node in a tree which corresponds to the best predictor called **root node**. Decision trees can handle both categorical and numerical data.

Decision Tree Algorithm

- The core algorithm for building decision trees (*J. R. Quinlan: http://hunch.net/~coms-4771/quinlan.pdf*) employs a top-down, greedy search through the space of possible branches with no backtracking. The ID3 algorithm construct a decision tree thru *Standard Deviation Reduction*.

Hours
25
30
46
45
52
23
43
35
38
46
48
52
44
30

$$S = \sqrt{\frac{\sum (x - \mu)^2}{n}}$$

Standard Deviation

$$S = 9.32$$

- Standard deviation is a measure of the spread of data values.
- Small value of S (standard deviation) means that data values are fairly uniform.
- Value of S=0 means that all values are identical.
- Large S means that data are spread over a large range and are not uniform.
- The algorithm seeks islands (groups of data points) of high uniformity.

Standard Deviation for Target Values Grouped by Features

- Consider data from the previous slide broken by the value of feature (Weather) Outlook.

- We see that by grouping target values (Hours) by the feature (Outlook) we created several (3) groups some of which have a lower standard deviation S, and therefore a higher uniformity.

- Standard deviation calculated as the mean of variations for all groups is apparently reduced when compared to the overall standard variation S = 9.32

$$S(T,X) = \sum_{c \in X} P(c)S(c)$$

		Hours (StDev)	Count
Outlook	Overcast	3.49	4
	Rainy	7.78	5
	Sunny	10.87	5
			14

S(Hours, Outlook) = P(Sunny)*S(Sunny) + P(Overcast)*S(Overcast) + P(Rainy)*S(Rainy)

= (4/14)*3.49 + (5/14)*7.78 + (5/14)*10.87

= 7.66

Standard Deviation Reduction, Process

- The standard deviation reduction is based on the decrease in standard deviation after a dataset is split on an feature. Constructing a decision tree is all about finding features that returns the highest standard deviation reduction (i.e., the most homogeneous branches).

Step 1: The standard deviation of the whole target is calculated.

Step 2: The dataset is then split on different features. The standard deviation for each branch is calculated. The resulting standard deviation for each branch is subtracted from the standard deviation before the split. The result is the standard deviation reduction for the branch (feature).

Outlook		Hours (StDev)
	Overcast	3.49
	Rainy	7.78
	Sunny	10.87
	SDR=1.66	

Temp.		Hours (StDev)
	Cool	10.51
	Hot	8.95
	Mild	7.65
	SDR=0.17	

Humidity		Hours (StDev)
	High	9.36
	Normal	8.37
	SDR=0.28	

Windy		Hours (StDev)
	False	7.87
	True	10.59
	SDR=0.29	

$$SDR(T, X) = S(T) - S(T, X)$$

SDR(Hours , Outlook) = **S**(Hours) − **S**(Hours, Outlook)

$$= 9.32 - 7.66 = 1.66$$

Standard Deviation Reduction, Process

Step 3: The attribute with the largest standard deviation reduction is chosen for the decision node: Outlook in our case.

Step 4a: Dataset is divided based on the values of the selected attribute.

		Hours (StDev)
Outlook	Overcast	3.49
	Rainy	7.78
	Sunny	10.87
SDR=1.66		
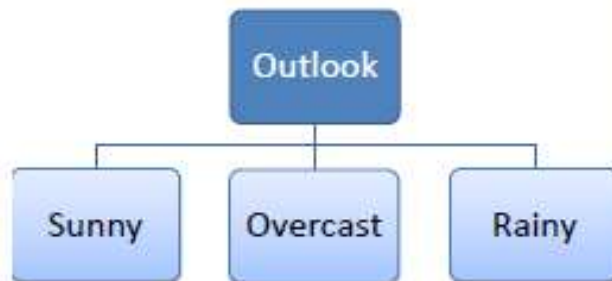
| Outlook	Temp	Humidity	Windy	Hours
Sunny	Mild	High	FALSE	45
Sunny	Cool	Normal	FALSE	52
Sunny	Cool	Normal	TRUE	23
Sunny	Mild	Normal	FALSE	46
Sunny	Mild	High	TRUE	30
Rainy	Hot	High	FALSE	25
Rainy	Hot	High	TRUE	30
Rainy	Mild	High	FALSE	35
Rainy	Cool	Normal	FALSE	38
Rainy	Mild	Normal	TRUE	48
Overcast	Hot	High	FALSE	46
Overcast	Cool	Normal	TRUE	43
Overcast	Mild	High	TRUE	52
Overcast	Hot	Normal	FALSE	44

Standard Deviation Reduction, Process

- **Step 4b:** A branch set with standard deviation more than 0 needs further splitting.
- In practice, we need some termination criteria. For example, when standard deviation for the branch becomes smaller than a certain fraction (e.g., 5%) of standard deviation for the full dataset OR when too few instances remain in the branch (e.g., 3).
- **Step 5:** The process is run recursively on the non-leaf branches, until all data is processed.
- When the number of instances is more than one at a leaf node, we calculate the average as the final value for the target.

Temp	Humidity	Windy	Hours
Mild	High	FALSE	45
Cool	Normal	FALSE	52
Mild	Normal	FALSE	46
Cool	Normal	TRUE	23
Mild	High	TRUE	30

	★	Hours (StDev)
Windy	False	3.09
	True	3.50
SDR= 7.62		

$SDR = 10.87 - ((3/5)*3.09 + (2/5)*3.5)$
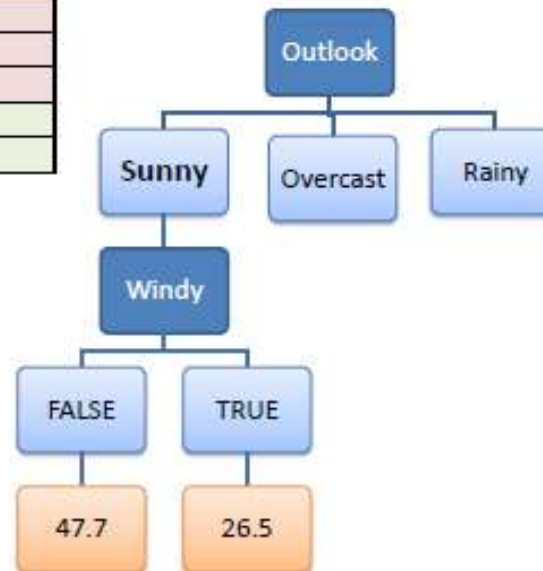
Regression vs. Decision Tree

- Every method/algorithm has its advantages.
- Regression produces smooth boundaries/predictions.
- Decision Trees could give very non-linear predictions.
- The following image is just a sketch to illustrate the difference:
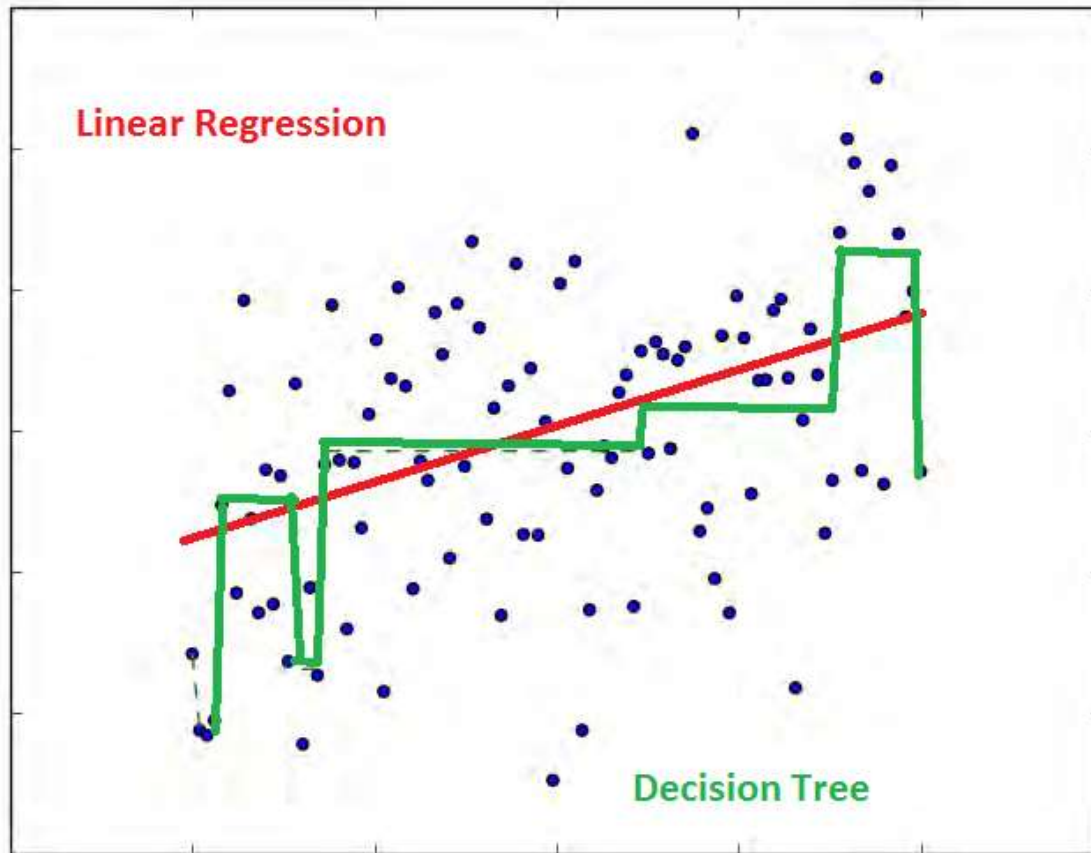
Regression Model with Spark

- Classification models deal with outcomes that represent discrete classes.

- Regression models are concerned with target variables that can take any real value. The underlying principle is very similar—we wish to find a model that maps input features to target variables. That mapping is the prediction. Like classification, regression is also a form of supervised learning.

- Regression models can be used to predict just about any variable of interest.

    - Gambling returns on stocks and other economic variables

    - Loss amounts for loan defaults (this can be combined with a classification model that predicts the probability of default, while the regression model predicts the amount in the case of a default)

    - Recommendations

    - Predicting **customer lifetime value** (**CLTV**) in a retail, mobile, or other business, based on user behavior and spending patterns

Types of Regression Models

- Spark's MLlib library offers two broad classes of regression models: linear regression models and decision tree regression models.

- MLlib provides a standard least squares regression model (although other types of generalized linear models for regression are planned).

- The loss function used for least squares is the squared loss, which is defined as: $\frac{1}{2} (\theta^T x - y)^2$

- Here, y is the target variable (this time, real valued), θ is the weight factor (vector)or set of parameters, and x is the feature vector.

- Decision trees for regression allows a more complex, nonlinear model to be fitted to the data.

Dataset

- We will be using the bike sharing dataset which contains hourly records of the number of bicycle rentals in Washington, D.C., US capital.

- The dataset contains variables related to date and time, weather, and seasonal and holiday information.

- The dataset is available at University of California Irvine site:

http://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset

- Click on the **Data Folder** link and then download the

  `Bike-Sharing-Dataset.zip` file.

- Dataset is described and analyzed in the following paper:

Fanaee-T, Hadi and Gama Joao, Event labeling combining ensemble detectors and background knowledge, *Progress in Artificial Intelligence*, pp. 1-15, Springer Berlin Heidelberg, 2013.

- The paper is available at

http://link.springer.com/article/10.1007%2Fs13748-013-0040-3

Extracting Data

- File `Bike-Sharing-Dataset.zip` file, contains the `day.csv, hour. csv,` and the `Readme.txt` files. The `Readme.txt` file contains the variable names and descriptions. We will use:
- `instant`: The record ID
- `dteday`: The raw date
- `season`: Different seasons such as spring, summer, winter, and fall
- `yr`: year (2011 or 2012)
- `mnth`: Month of the year
- `hr`: Hour of the day
- `holiday`: Whether the day was a holiday or not
- `weekday`: Day of the week
- `workingday`: Indicates whether the day was a working day (1) or not (0)
- `weathersit`: This is a categorical variable that describes the weather at a particular time
- `temp`: Normalized temperature (in ℃?, 0-41)
- `atemp`: Normalized apparent temperature (in ℃?, 0-41)
- `hum`: Normalized humidity
- `windspeed`: Normalized wind speed
- **`cnt`**: This is the target variable, that is, the count of bike rentals for that hour
- We will work with the hourly data contained in `hour.csv`. The first line contains the column names as a header. We can remove that line by the following `sed` command (or by any other means)

```
$ sed 1d hour.csv > hour_noheader.csv
```

Examining Data

- We prefer to use IPython Notebook, which we start with the command:

```
$ pyspark   % this works if IPYTHON_OPTS=notebook is set
```

- We'll start as usual by loading the dataset and inspecting it:

```
path = "file:///home/cloudera/lecture13/hour_noheader.csv"
raw_data = sc.textFile(path)
num_data = raw_data.count()
records = raw_data.map(lambda x: x.split(","))
first = records.first()
print first
print num_data
```

- We should see the following output:

```
[u'1', u'2011-01-01', u'1', u'0', u'1', u'0', u'0', u'6',
u'0', u'1',
u'0.24', u'0.2879', u'0.81', u'0', u'3', u'13', u'16']
17379
```

- We have 17,379 hourly records. We will ignore the `record ID` and `raw date` columns. We will also ignore the `casual` and `registered` count target variables and focus on the overall count variable, `cnt` (which is the sum of the other two counts).

- We are left with 12 variables. The first eight are categorical, while the last 4 are normalized real-valued variables.

Treatment of Categorical Variables

- To deal with the eight categorical variables, we will use the binary encoding approach. The four real-valued variables will be left as is.

- In some ML algorithms, categorical features are not used as input in their raw form. Those variables are not numbers; instead, they are members of a set of possible values that the variable can take. One typical categorical variable is the name of the day of the week.

- Categorical variables are also known as **nominal** variables where there is no concept of order between the values of the variable (job occupations, colors, etc.).

- When there is a concept of order between variables such as the ratings or month, day of the week, etc., (e.g. a rating of 5 is conceptually higher or better than a rating of 1), we refer to **ordinal** variables.

- To transform categorical variables into a numerical representation, we can use a common approach known as **1-of-k** encoding. An approach such as 1-of-k encoding is required to represent nominal variables in a way that makes machine learning tasks more efficient.

- Ordinal variables might be used in their raw form but are often encoded in the same way as nominal variables.

Binary Encoding of Categorical Features

- Assume that there are K(7) possible values that the variable can take. If we assign each possible value an index from the set of 1 to K, then we can represent a given state of the variable using a binary vector of length K. All entries will be zero, except the entry at the index that corresponds to the given state of the variable. That entry is set to one.

- For example, we can collect all the possible states of the day of the week variable:

```
all_days = sc.parallelize(['Monday','Tuesday',…,'Sunday'])
```

- We can then assign index values to each possible day in turn (note that we start from zero, since Python, Scala, and Java arrays all use zero-based indices):

```
idx = 0
all_days_dict= {}
for o in all_days.collect():        #  transform RDD into a list
    all_days_dict[o] = idx
    idx +=1
```

- Next we try a few examples to see what "1-of-k" encoding is assigned

```
print "Encoding of 'Monday': %d" % all_days_dict['Monday']
print "Encoding of 'Wednesday': %d" % all_days_dict['Wednesday']
```

- We will see the following output:

```
Encoding of 'Monday': 0
Encoding of 'Wednesday': 2
```

Binary Encoding of Categorical Features

- Next, we can binary encode days of the week. We start by creating a `numpy` (Math library) array of a length that is equal to the number of possible days (`K` in this case) and filling that array with zeros. We use `numpy` function `zeros()` create this array.

- We then extract the index of the word "`Wednesday`" and assign a value of 1 to the value of the array at this index:

```
import numpy as np
K = len(all_days_dict)
binary_x = np.zeros(K)
k_wednesday = all_days_dict['Wednesday']
binary_x[k_wednesday] = 1
print "Binary feature vector: %s" % binary_x
print "Length of binary vector: %d" % K
```

- Thus the resulting binary feature vector for days of the week has length 7:

```
Binary feature vector: [0. 0. 1. 0. 0. 0. 0.]
Length of binary vector: 7
```

Binary Encoding of Categorical Variables in Bike Dataset

- In order to extract/transform each categorical feature into a binary vector form, we need to know the feature mapping of each feature value to the index of the nonzero value in our binary vector.

- We define a function that extracts this mapping from the dataset for a given column.

- We will first `cache()` the dataset, since we will read from it many times:

```
records.cache()
def get_mapping(rdd, idx):
    return rdd.map(lambda fields: fields[idx]).distinct().
    zipWithIndex().collectAsMap()
```

- Function `get_mapping` first maps the field to its unique values and then uses the `zipWithIndex` transformation to associate the value with a unique index such that a key-value RDD is formed, where the key is the variable and the value is the index.

- This index will be the index of the nonzero entry in the binary vector representation of the feature. We will finally collect this RDD back to the driver as a Python dictionary.

- We can test our function on the third variable column (index 2):

```
print "Mapping of first categorical feature column: %s" % get_
mapping(records, 2)
```

- The preceding line of code will give us the following output:

```
Mapping of first categorical feature column: {u'1': 0, u'3': 2,
u'2': 1, u'4': 3}
```

Binary Encoding of Categorical Columns 2 to 9

- Now, we can apply this function to each categorical column (that is, for variable indices 2 to 9):

```
mappings = [get_mapping(records, i) for i in range(2,9)]
cat_len = sum(map(len, mappings))
num_len = len(records.first()[10:13])
total_len = num_len + cat_len
```

- We now have the mappings for each variable, and we can see how many values in total we need for our binary vector representation:

```
print "Feature vector length for categorical features: %d" % cat_len
print "Feature vector length for numerical features: %d" % num_len
print "Total feature vector length: %d" % total_len
```

- The output of the preceding code is as follows:

```
Feature vector length for categorical features: 57
Feature vector length for numerical features: 4
Total feature vector length: 61
```

Creating Feature Vectors for Linear Model

- The next step is to use our extracted mappings to convert the categorical features to binary-encoded features.
- It will be helpful to create a function that we can apply to each record in our dataset.
- We will also create a function to extract the target variable from each record.
- Import `numpy` for linear algebra utilities and MLlib's `LabeledPoint` class to wrap our feature vectors and target variables:

```python
from pyspark.mllib.regression import LabeledPoint
import numpy as np
def extract_features(record):
    cat_vec = np.zeros(cat_len)
    i = 0
    step = 0
    for field in record[2:9]:
        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]])
    return np.concatenate((cat_vec, num_vec))

def extract_label(record):
    return float(record[-1])
```

Creating Feature Vectors for Linear Model

- In `extract_features` function, we ran through each column in the row of data. We extracted the binary encoding for each variable in turn from the mappings we created previously. The step variable ensures that the nonzero featureindex in the full feature vector is correct (and is somewhat more efficient than, say, creating many smaller binary vectors and concatenating them). The numeric vector is created directly by first converting the data to floating point numbers and wrapping these in a numpy array. The resulting two vectors are then concatenated.

- The `extract_label` function simply converts the last column variable (the count) into a float.

- With the utility functions defined, we can proceed with extracting feature vectors and labels from our data records:

```
data = records.map(lambda r:
LabeledPoint(extract_label(r),extract_features(r)))
```

Inspecting Features RDD

- Let's inspect the first record in the extracted feature RDD:

```
first_point = data.first()
print "Raw data: " + str(first[2:])
print "Label: " + str(first_point.label)
print "Linear Model feature vector:\n" + str(first_point.features)
print "Linear Model feature vector length: " + str(len(first_point.
features))
```

- We should see output similar to the following:

```
Raw data: [u'1', u'0', u'1', u'0', u'0', u'6', u'0', u'1', u'0.24',
u'0.2879', u'0.81', u'0', u'3', u'13', u'16']
Label: 16.0
Linear Model feature vector: [1.0,0.0,0.0,0.0,0.0,1.0,0.0,1.0,0.0,0.0,0.0
,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,
0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0
.0,0.0,0.0,1.0,0.0,1.0,0.0,0.0,0.0,0.0,0.24,0.2879,0.81,0.0]
Linear Model feature vector length: 56
```

- We converted the raw data into a feature vector made up of binary categorical and real numeric features, and ended up with vector of total length of 56.

Creating feature vectors for the decision tree

- Decision tree models typically work on raw features (that is, it is not required to convert categorical features into a binary vector encoding;

- In decision trees categorical features can, instead, be used directly).

- We create a separate function to extract the decision tree feature vector, it simply converts all the values to floats and wraps them in a `numpy` array:

```
def extract_features_dt(record):
    return np.array(map(float, record[2:13]))


data_dt = records.map(lambda r: LabeledPoint(extract_label(r),
        extract_features_dt(r)))
first_point_dt = data_dt.first()
print "Decision Tree feature vector: " + str(first_point_dt.features)
print "Decision Tree feature vector length: " +
str(len(first_point_dt.features))
```

- The following output shows the extracted feature vector, and we can see that we have a vector length of 12, which matches the number of raw variables we are using:

```
Decision Tree feature vector:
[1.0,0.0,1.0,0.0,0.0,6.0,0.0,1.0,0.24,0.287
9,0.81,0.0]
Decision Tree feature vector length: 11
```

MLlib Data Types, Python

- MLlib supports local vectors and matrices stored on a single machine, as well as distributed matrices backed by one or more RDDs.

- Local vectors and local matrices are simple data models that serve as public interfaces.

- A training example used in supervised learning is called a "labeled point"

- A local vector has integer-typed and 0-based indices and double-typed values, stored on a single machine.

- MLlib supports two types of local vectors: dense and sparse.

- A dense vector is backed by an array of doubles representing its entry values.

- A sparse vector is backed by two parallel arrays: indices and values.

- For example, a vector (1.0, 0.0, 3.0) can be represented in
  - dense format as [1.0, 0.0, 3.0] or in
  - sparse format as (3, [0, 2], [1.0, 3]), where 3 is the size of the vector.

- Dense vectors are NumPy's `array` objects or Python's lists [1,2,3]

- Sparse vectors are Mllib's `SparceVectors` or SciPy's `csc_matrix` objects

MLLib Types, Scala, Java

- In Java, the base class of local vectors is
  `org.apache.spark.mllib.linalg.Vector and Vectors`

- It has 2 implementations: `DenseVector` **and** `SparseVector`.

- Factory methods implemented in Vectors create local vectors.

```
Vector dv = Vectors.dense(1.0, 0.0, 3.0);     # dense vector
```

- // Sparse vector (1.0, 0.0, 3.0) with indices and values

```
Vector sv = Vectors.sparse(3, new int[] {0, 2}, new double[]
{1.0, 3.0});
```

- In Scala, the base class of local vectors is `Vector`, **with two**
  implementations: `DenseVector  and  SparseVector`.

- The factory methods implemented in Vectors create local vectors.

```
val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
```

- Create a sparse vector (1.0, 0.0, 3.0) by specifying its indices and values
  corresponding to nonzero entries.

```
val sv1: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0))
```

- // Create a sparse vector (1.0, 0.0, 3.0) by specifying its nonzero entries.

```
val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
```

LabeledPoint

- A labeled point is a local vector, either dense or sparse, associated with a label/response.

- `LabeledPoint`-s are used in supervised learning algorithms. Labels are stored as variables of type `double`, so we can use labeled points in both regression and classification.

- For binary classification, a label should be either 0 (negative) or 1 (positive). For multiclass classification, labels should be class indices starting from zero: 0, 1, 2, ….

```
from pyspark.mllib.linalg import SparseVector
from pyspark.mllib.regression import LabeledPoint

# Create a labeled point with a positive label and a dense feature vector.
pos = LabeledPoint(1.0, [1.0, 0.0, 3.0])

# Create a labeled point with a negative label and a sparse feature vector.
neg = LabeledPoint(0.0, SparseVector(3, [0, 2], [1.0, 3.0]))
```

LabeledPoint, Scala, Java

- ### Scala

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint

// Create a labeled point with a positive label and a dense feature vector.
val pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))

// Create a labeled point with a negative label and a sparse feature
vector.
val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0, 2), Array(1.0,
3.0)))
```

- ### Java

```
import org.apache.spark.mllib.linalg.Vectors;
import org.apache.spark.mllib.regression.LabeledPoint;

// Create a labeled point with a positive label and a dense feature vector.
LabeledPoint pos = new LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0));

// Create a labeled point with a negative label and a sparse feature
vector.
LabeledPoint neg = new LabeledPoint(0.0, Vectors.sparse(3, new int[] {0,
2}, new double[] {1.0, 3.0}))
```

Training and using regression models

- To train regression models using decision trees or linear models we simply pass the training data contained in a `[LabeledPoint] RDD` to the relevant train method.

- In Scala and Java, if we wanted to customize the various model parameters (such as regularization and step size for the SGD optimizer), we are required to instantiate a new model instance and use the optimizer field to access these available parameter setters.

- In Python, we are provided with a convenience method that gives us access to all the available model arguments, so we only have to use this one entry point for training.

- We can see the details of these convenience functions by importing the relevant modules and then calling the help function on the train methods:

```
from pyspark.mllib.regression import LinearRegressionWithSGD
from pyspark.mllib.tree import DecisionTree
help(LinearRegressionWithSGD.train)
```

help(LinearRegressionWithSGD.train)

```
Help on method train in module pyspark.mllib.regression:

train(cls, data, iterations=100, step=1.0, miniBatchFraction=1.0, initialWeights=None,
regParam=0.0, regType=None, intercept=False, validateData=True, convergenceTol=0.001)
method of __builtin__.type instance
    Train a linear regression model using Stochastic Gradient
    Descent (SGD).
    This solves the least squares regression formulation

        f(weights) = 1/(2n) ||A weights - y||^2,

    which is the mean squared error.
    Here the data matrix has n rows, and the input RDD holds the
    set of rows of A, each with its corresponding right hand side
    label y. See also the documentation for the precise formulation.

    :param data:              The training data, an RDD of
                              LabeledPoint.
    :param iterations:        The number of iterations
                              (default: 100).
    :param step:              The step parameter used in SGD
                              (default: 1.0).
    :param miniBatchFraction: Fraction of data to be used for each
                              SGD iteration (default: 1.0).
    :param initialWeights:    The initial weights (default: None).
    :param regParam:          The regularizer parameter
                              (default: 0.0).
```

help(LinearRegressionWithSGD.train), contuned

```
:param regType:          The type of regularizer used for
                             training our model.
                         :Allowed values:
                         - "l1" for using L1 regularization (lasso),
                         - "l2" for using L2 regularization (ridge),
                         - None for no regularization  (default: None)
:param intercept:         Boolean parameter which indicates the
                             use or not of the augmented representation
                             for training data (i.e. whether bias
                             features are activated or not,
                             default: False).
:param validateData:     Boolean parameter which indicates if
                             the algorithm should validate data
                             before training. (default: True)
:param convergenceTol:   A condition which decides iteration
                         termination.
                             (default: 0.001)


    .. versionadded:: 0.9.0
```

help(DecisionTree.trainRegressor)

```
Help on method trainRegressor in module pyspark.mllib.tree:

trainRegressor(cls, data, categoricalFeaturesInfo, impurity='variance',
maxDepth=5, maxBins=32, minInstancesPerNode=1, minInfoGain=0.0) method of
__builtin__.type instance
    Train a DecisionTreeModel for regression.


    :param data: Training data: RDD of LabeledPoint. Labels are real
numbers.
    :param categoricalFeaturesInfo: Map from categorical feature index to
number                                      of categories.
            Any feature not in this map is treated as continuous.
    :param impurity: Supported values: "variance"
    :param maxDepth: Max depth of tree.
            E.g., depth 0 means 1 leaf node.
            Depth 1 means 1 internal node + 2 leaf nodes.
    :param maxBins: Number of bins used for finding splits at each node.
    :param minInstancesPerNode: Min number of instances required at
            child nodes to create the parent split
    :param minInfoGain: Min info gain required to create a split
    :return: DecisionTreeModel
```

help(DecisionTree.trainRegressor), continued

```
Example usage:

    >>> from pyspark.mllib.regression import LabeledPoint
    >>> from pyspark.mllib.tree import DecisionTree
    >>> from pyspark.mllib.linalg import SparseVector
    >>>
    >>> sparse_data = [
    ...     LabeledPoint(0.0, SparseVector(2, {0: 0.0})),
    ...     LabeledPoint(1.0, SparseVector(2, {1: 1.0})),
    ...     LabeledPoint(0.0, SparseVector(2, {0: 0.0})),
    ...     LabeledPoint(1.0, SparseVector(2, {1: 2.0}))
    ... ]
    >>>
    >>> model = DecisionTree.trainRegressor(sc.parallelize(sparse_data), {})
    >>> model.predict(SparseVector(2, {1: 1.0}))
    1.0
    >>> model.predict(SparseVector(2, {1: 0.0}))
    0.0
    >>> rdd = sc.parallelize([[0.0, 1.0], [0.0, 0.0]])
    >>> model.predict(rdd).collect()
    [1.0, 0.0]

    .. versionadded:: 1.1.0
```

Training a Linear Regression Model on Bike Sharing Dataset

- First, we will train the linear regression model and take a look at the first few predictions that the model makes on the data:

```
linear_model = LinearRegressionWithSGD.train(data, iterations=100,
step=0.01, intercept=False)
true_vs_predicted = data.map(lambda p: (p.label, linear_model.
predict(p.features)))
print "Linear Model predictions: " + str(true_vs_predicted.take(5))
```

- We have not used the default settings for iterations and step here. We've changed the number of iterations so that the model does not take too long to train. As for the step size, we will see why this has been changed from the default a little later. We will see the following output:

```
Linear Model predictions: [(16.0, 69.763529728385762), (40.0,
68.931901688418534), (32.0, 68.852836015878225), (13.0,
68.593298041266806), (1.0, 68.553402914573269)]
```
From a quick glance at these predictions, it appears that the linear model is quite a way off in its predictions.

- Let us see whether the decision tree model might work better.

Save and Load Model

- Save and load model to directory `/home/cloudera/linear_model`

```
from pyspark.mllib.regression import LinearRegressionModel
linear_model.save(sc, "file:///home/cloudera/linear_model")
sameModel = LinearRegressionModel.load(sc,
"file:///home/cloudera/linear_model")
```

Training a Decision Tree Regression Model on Bike Dataset

- We will train the decision tree model simply using the default arguments to the `trainRegressor` method (which equates to using a tree depth of 5). Note that we need to pass in the other form of the dataset, `data_dt`, that we created from the raw feature values (as opposed to the binary encoded features that we used for the linear model).

- We also need to pass in an argument for `categoricalFeaturesInfo`. This is a dictionary that maps the categorical feature index to the number of categories for the feature. If a feature is not in this mapping, it will be treated as continuous. For our purposes, we will leave this as is, passing in an empty mapping:

```
dt_model = DecisionTree.trainRegressor(data_dt,{})
preds = dt_model.predict(data_dt.map(lambda p: p.features))
actual = data.map(lambda p: p.label)
true_vs_predicted_dt = actual.zip(preds)
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())
```

- This should output these predictions:

```
Decision Tree predictions: [(16.0, 54.913223140495866),
(40.0, 54.913223140495866), (32.0, 53.171052631578945), (13.0,
14.28402366863905 3), (1.0, 14.284023668639053)]
Decision Tree depth: 5
Decision Tree number of nodes: 63
```

Evaluating the Performance of Regression Models

- When dealing with regression models, it is very unlikely that our model will precisely predict the target variable, because the target variable can take on any real value. However, we would naturally like to understand how far away our predicted values are from the true values, so will we utilize a metric that takes into account the overall deviation.

- Some of the standard evaluation metrics used to measure the performance of regression models include the **Mean Squared Error** (**MSE**) and **Root Mean Squared Error** (**RMSE**), the **Mean Absolute Error** (**MAE**), the R-squared coefficient, and many others.

Mean Squared Error and Root Mean Squared Error

- MSE is the average of the squared error that is used as the loss function for least squares regression

$$\sum_{i=1}^{n} \frac{(\theta^T x(i) - y(i))^2}{n}$$

- It is the sum, over all the data points, of the square of the difference between the predicted and actual target variables, divided by the number of data points.

- RMSE is the square root of MSE. MSE is measured in units that are the square of the target variable, while RMSE is measured in the same units as the target variable.

- Due to its formulation, MSE, just like the squared loss function that it derives from,

- effectively penalizes larger errors more severely.

- In order to evaluate our predictions based on the mean of an error metric, we will first make predictions for each input feature vector in an RDD of `LabeledPoint` instances by computing the error for each record using a function that takes the prediction and true target value as inputs. This will return a [Double] RDD that contains the error values. We can then find the average using the mean method of RDDs that contain Double values.

- Let's define our squared error function as follows:

```
def squared_error(actual, pred):
return (pred - actual)**2
```

Mean Absolute Error, Root Mean Squared Log Error

- MAE is the average of the absolute differences between the predicted and actual targets:

$$\sum_{i=1}^{n} \frac{|x(i) - y(i)|}{n}$$

- MAE is similar in principle to MSE, but it does not punish large deviations as much. Our function to compute MAE is as follows:

```
def abs_error(actual, pred):
    return np.abs(pred - actual)
```

Root Mean Squared Log Error

- RMSLE not as widely used as MSE and MAE, but it is used as the metric for the Kaggle competition that uses the bike sharing dataset. It is effectively the RMSE of the log-transformed predicted and target values. This measurement is useful when there is a wide range in the target variable, and you do not necessarily want to penalize large errors when the predicted and target values are themselves high. It is also effective when you care about percentage errors rather than the absolute value of errors. We calculate RMSLE as

```
def squared_log_error(pred, actual):
    return (np.log(pred + 1) - np.log(actual + 1))**2
```

Comparing Performance Metrics

- For linear regression model, we will apply the relevant error function to each record in the RDD we computed earlier, `true_vs_predicted`

```
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
```

- This outputs the following metrics:

```
Linear Model - Mean Squared Error: 28166.3824
Linear Model - Mean Absolute Error: 129.4506
Linear Model - Root Mean Squared Log Error: 1.4974
```

Comparing Performance Metrics

- We will use the same approach for the decision tree model, using the `true_vs_predicted_dt` RDD:

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: %2.4f" % mse_dt

print "Decision Tree - Mean Absolute Error: %2.4f" % mae_dt

print "Decision Tree - Root Mean Squared Log Error: %2.4f" % rmsle_dt

- We should see output similar to this:

```
Decision Tree - Mean Squared Error: 11560.7978
Decision Tree - Mean Absolute Error: 71.0969
Decision Tree - Root Mean Squared Log Error: 0.6259
```

- Looking at the results, we can see that our initial guess about the decision tree model being the better performer is indeed true.

Improving Performance and Tuning Parameters

- Many machine learning models, including linear models, work better with certain distributions of the input data as well as target variables.

- In particular, linear regression works best with data of normal distribution.

- In many real-world cases, the distribution preference of linear regression does not hold. In our case, for example, we know that the number of bike rentals can never be negative. This alone should indicate that the assumption of normality might be problematic.

- To get a better idea of the target distribution, it is often a good idea to plot a histogram of the target values.

- To genrate plots, if you are using IPython Notebook, enter the magic function,

`%pylab inline`

- to import pylab (that is, the `numpy` and `matplotlib` plotting functions) into the workspace. This will also create any figures and plots inline within the Notebook cell.

- If you are using the standard IPython console, you can use

`%pylab`

- to import the necessary functionality (your plots will appear in a separate window).

Target Variable Distribution

- To create a plot of the raw target variable distribution we will use the following piece of code:

```
targets = records.map(lambda r: float(r[-1])).collect()
hist(targets, bins=40, color='lightblue', normed=True)
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(16, 10)
```

- Looking at the histogram plot, we can see that the distribution is highly skewed and certainly does not follow a normal distribution
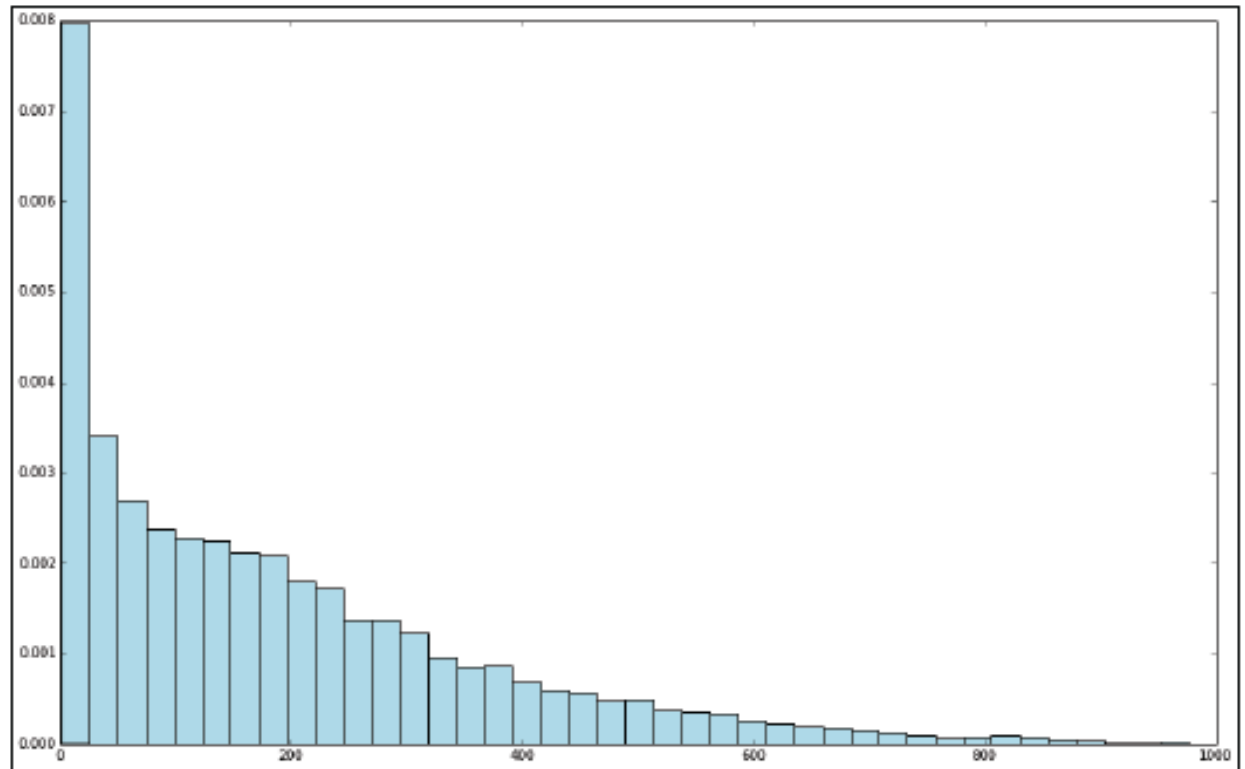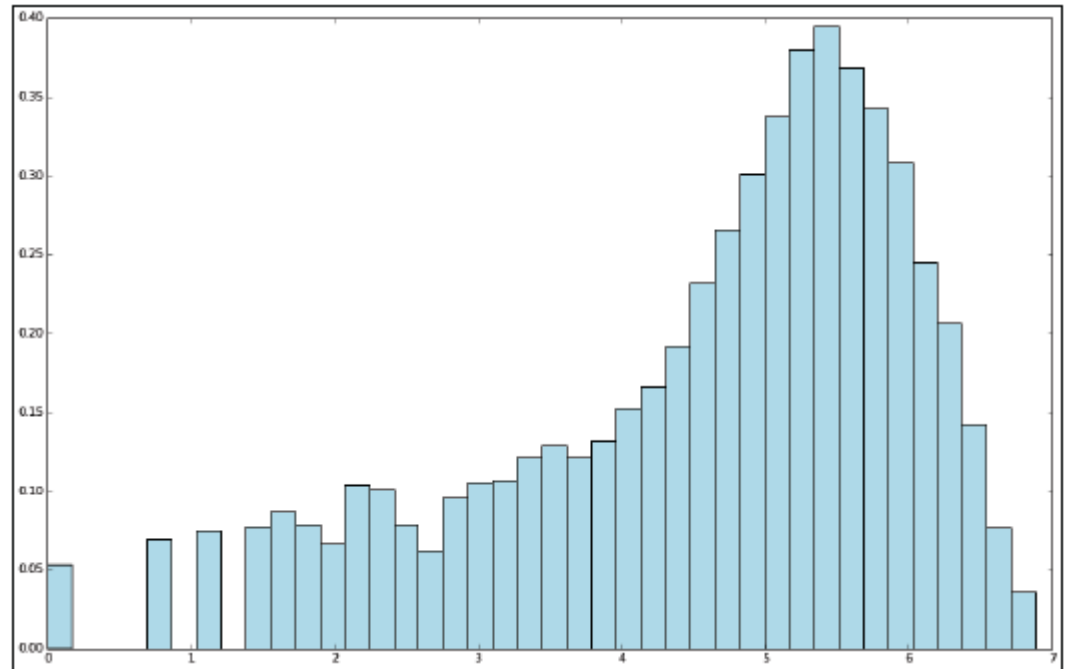
Log-Transform Target Variables

- We could apply a transformation to the target variable, such that we take the logarithm of the target value instead of the raw value. This is often referred to as log-transforming the target variable (this transformation can also be applied to feature values).

```
histogram of the log-transformed values:
log_targets = records.map(lambda r: np.log(float(r[-1]))).collect()
hist(log_targets, bins=40, color='lightblue', normed=True)
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(16, 10)
```

- As we can see, the resulting, transformed, values have a much better almost "bell –like" distribution.

Square Root Transformed Target Variable

- A second type of transformation that is useful in the case of target values that do not take on negative values and, in addition, might take on a very wide range of values, is to take the square root of the variable.

```
sqrt_targets = records.map(lambda r: np.sqrt(float(r[-1]))).collect()
hist(sqrt_targets, bins=40, color='lightblue', normed=True)
fig = matplotlib.pyplot.gcf()
fig.set_size_inches(16, 10)
```

- From the plots of the log and square root transformations, we can see that both
- result in a more even distribution relative to the raw values.
- While they are still not normally distributed, they are a lot closer to a normal distribution when compared to the original target variable.
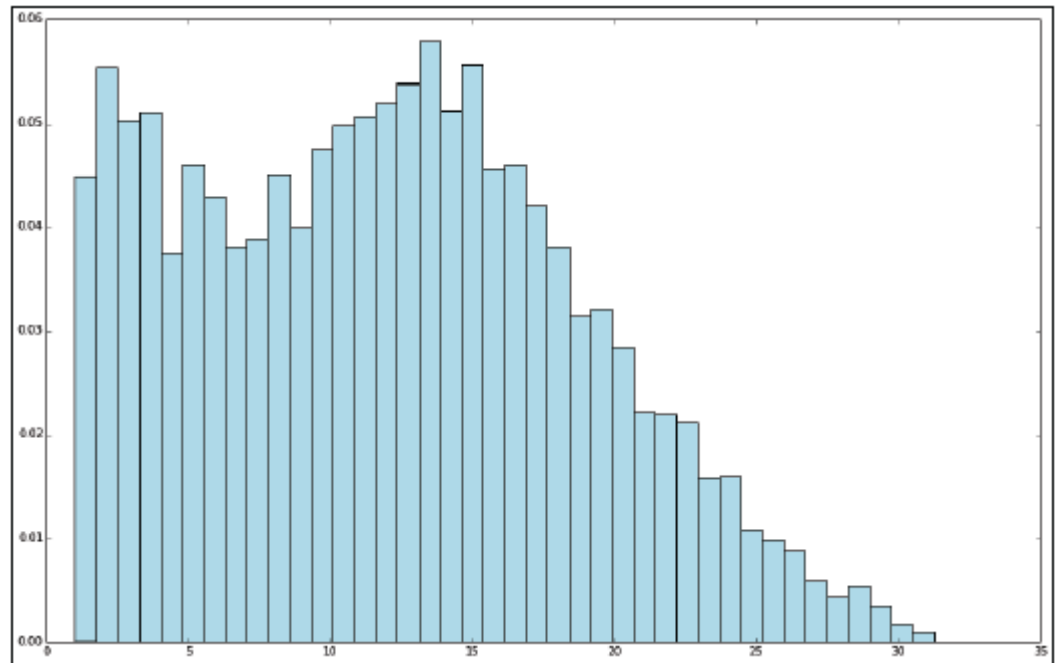
Impact of Log-transformed Targets on Training

- We need to assess whether applying these transformations have any impact on model performance?

- Let's evaluate the various metrics we used previously on log-transformed data as an example. We will do this first for the linear model by applying the numpy log function to the label field of each LabeledPoint RDD. Here, we will only transform the target variable, and we will not apply any transformations to the features.

```
data_log = data.map(lambda lp: LabeledPoint(np.log(lp.label),lp.features))
```

- We will then train a model on this transformed data and form the RDD of predicted versus true values:

```
model_log = LinearRegressionWithSGD.train(data_log, iterations=10,step=0.1)
```

- Note that now that we have transformed the target variable, the predictions of the model will be on the log scale, as will the target values of the transformed dataset. Therefore, in order to use our model and evaluate its performance, we must first transform the log data back into the original scale by taking the exponent of both the predicted and true values using the `numpy exp` function.

```
true_vs_predicted_log = data_log.map(lambda p: (np.exp(p.label),
np.exp(model_log.predict(p.features))))
```

MSE, MAE, and RMSLE Metrics for Linear Model

- We will compute the MSE, MAE, and RMSLE metrics for the model:

```
mse_log = true_vs_predicted_log.map(lambda (t, p): squared_error(t,p)).mean()
mae_log = true_vs_predicted_log.map(lambda (t, p): abs_error(t, p)).mean()
rmsle_log = np.sqrt(true_vs_predicted_log.map(lambda (t, p):
 squared_log_error(t, p)).mean())
print "Mean Squared Error: %2.4f" % mse_log
print "Mean Absolue Error: %2.4f" % mae_log
print "Root Mean Squared Log Error: %2.4f" % rmsle_log
print "Non log-transformed predictions:\n" + str(true_vs_predicted.take(3))
print "Log-transformed predictions:\n" + str(true_vs_predicted_log.take(3))
```

- We should see output similar to the following:

```
Mean Squared Error: 38606.0875
Mean Absolue Error: 135.2726
Root Mean Squared Log Error: 1.3516
Non log-transformed predictions:
[(16.0, 119.30920003093594), (40.0, 116.95463511937378), (32.0,
116.57294610647752)]
Log-transformed predictions:
[(15.999999999999998, 45.860944832110015), (40.0,
43.255903592233274), (32.0, 42.311306147884252)]
```

- If we compare these results to the results on the raw target variable, we see that while we did not improve the MSE or MAE, we improved the RMSLE.

MSE, MAE, and RMSLE Metrics for Decision Tree Model

- We will perform the same analysis for the decision tree model:

```
data_dt_log = data_dt.map(lambda lp:
LabeledPoint(np.log(lp.label), lp.features))
dt_model_log = DecisionTree.trainRegressor(data_dt_log,{})
preds_log = dt_model_log.predict(data_dt_log.map(lambda p:p.features))
actual_log = data_dt_log.map(lambda p: p.label)
true_vs_predicted_dt_log = actual_log.zip(preds_log).map(lambda (t,p):
(np.exp(t), np.exp(p)))
mse_log_dt = true_vs_predicted_dt_log.map(lambda (t, p):
squared_error(t, p)).mean()
mae_log_dt = true_vs_predicted_dt_log.map(lambda(t,p):abs_error(t,p))
.mean()
rmsle_log_dt = np.sqrt(true_vs_predicted_dt_log.map(lambda (t, p):
squared_log_error(t, p)).mean())
print "Mean Squared Error: %2.4f" % mse_log_dt
print "Mean Absolue Error: %2.4f" % mae_log_dt
print "Root Mean Squared Log Error: %2.4f" % rmsle_log_dt
print "Non log-transformed predictions:\n" +
str(true_vs_predicted_dt.take(3))
print "Log-transformed predictions:\n" +
str(true_vs_predicted_dt_log.take(3))
```

MSE, MAE, and RMSLE Metrics for Decision Tree Model

- From the results bellow, we can see that we actually made our metrics slightly worse for the decision tree:

```
Mean Squared Error: 14781.5760
Mean Absolue Error: 76.4131
Root Mean Squared Log Error: 0.6406
Non log-transformed predictions:
[(16.0, 54.913223140495866), (40.0, 54.913223140495866), (32.0,
53.171052631578945)]
Log-transformed predictions:
[(15.999999999999998, 37.530779787154508), (40.0,
37.530779787154508), (32.0, 7.2797070993907287)
```

- It is probably not surprising that the log transformation results in a better RMSLE performance for the linear model. As we are minimizing the squared error, once we have transformed the target variable to log values, we are effectively minimizing a loss function that is very similar to the RMSLE.

Creating Training and Test Data

- Spark's Python API does not yet provide the `randomSplit` convenience method that is available in Scala. We need to create a training and test dataset manually.

- One relatively easy way to do this is by first taking a random sample of, say, 20 % of our data as our test set. We will then define our training set as the elements of the original RDD that are not in the test set RDD.

- We can achieve this using the sample method to take a random sample for our test set, followed by using the `subtractByKey` method, which takes care of returning the elements in one RDD where the keys do not overlap with the other RDD.

- Method `subtractByKey,` as the name suggests, works on the keys of the RDD elements that consist of key-value pairs. Therefore, here we will use `zipWithIndex` on our RDD of extracted training examples. This creates an RDD of (`LabeledPoint, index`) pairs.

Creating Training and Test Data

- We will then reverse the keys and values so that we can operate on the index keys:

```
data_with_idx = data.zipWithIndex().map(lambda (k, v): (v,
k))
test = data_with_idx.sample(False, 0.2, 42)
train = data_with_idx.subtractByKey(test)
```

- Once we have the two RDDs, we will recover just the `LabeledPoint` instances we need for training and test data, using map to extract the value from the key-value pairs:

```
train_data = train.map(lambda (idx, p): p)
test_data = test.map(lambda (idx, p) : p)
train_size = train_data.count()
test_size = test_data.count()
print "Training data size: %d" % train_size
print "Test data size: %d" % test_size
print "Total data size: %d " % num_data
print "Train + Test size : %d" % (train_size + test_size
```

Confirm Existence of two Datasets

- We can confirm that we now have two distinct datasets that add up to the original dataset in total:

```
Training data size: 13934
Test data size: 3445
Total data size: 17379
Train + Test size : 17379
```

- The final step is to apply the same approach to the features extracted for the decision tree model:

```
data_with_idx_dt = data_dt.zipWithIndex().map(lambda (k, v): (v, k))
test_dt = data_with_idx_dt.sample(False, 0.2, 42)
train_dt = data_with_idx_dt.subtractByKey(test_dt)
train_data_dt = train_dt.map(lambda (idx, p): p)
test_data_dt = test_dt.map(lambda (idx, p) : p)
```

Impact of Parameter Settings on Linear Models

- Now that we have prepared our training and test sets, we are ready to investigate the impact of different parameter settings on model performance. We will create a convenience function to evaluate the relevant performance metric by training the model on the training set and evaluating it on the test set for different parameter settings.

- We will use the RMSLE evaluation metric. The evaluation function is defined as:

```
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
```

Iterations

- As we saw when evaluating our classification models, we generally expect that a model trained with SGD will achieve better performance as the number of iterations increases, although the increase in performance will slow down as the number of iterations goes above some minimum number. Note that here, we will set the step size to 0.01 to better illustrate the impact at higher iteration numbers:

```
params = [1, 5, 10, 20, 50, 100]
metrics = [evaluate(train_data, test_data, param, 0.01,
0.0, l2',
False) for param in params]
print params
print metrics
```

- The output shows that the error metric indeed decreases as the number of iterations increases. It also does so at a decreasing rate, again as expected. What is interesting is that eventually, the SGD optimization tends to overshoot the optimal solution, and the RMSLE eventually starts to increase slightly:

```
[1, 5, 10, 20, 50, 100]
[2.3532904530306888, 1.6438528499254723, 1.4869656275309227,
1.4149741941240344, 1.415964126731959, 1.4539667094611679]
```

Metric for Varying number of Iterations

- Here, we will use the `matplotlib` library to plot a graph of the RMSLE metric against the number of iterations. We will use a log scale for the *x* axis to make the output easier to visualize:

```
plot(params, metrics)
fig = matplotlib.pyplot.gcf()
pyplot.xscale('log')
```
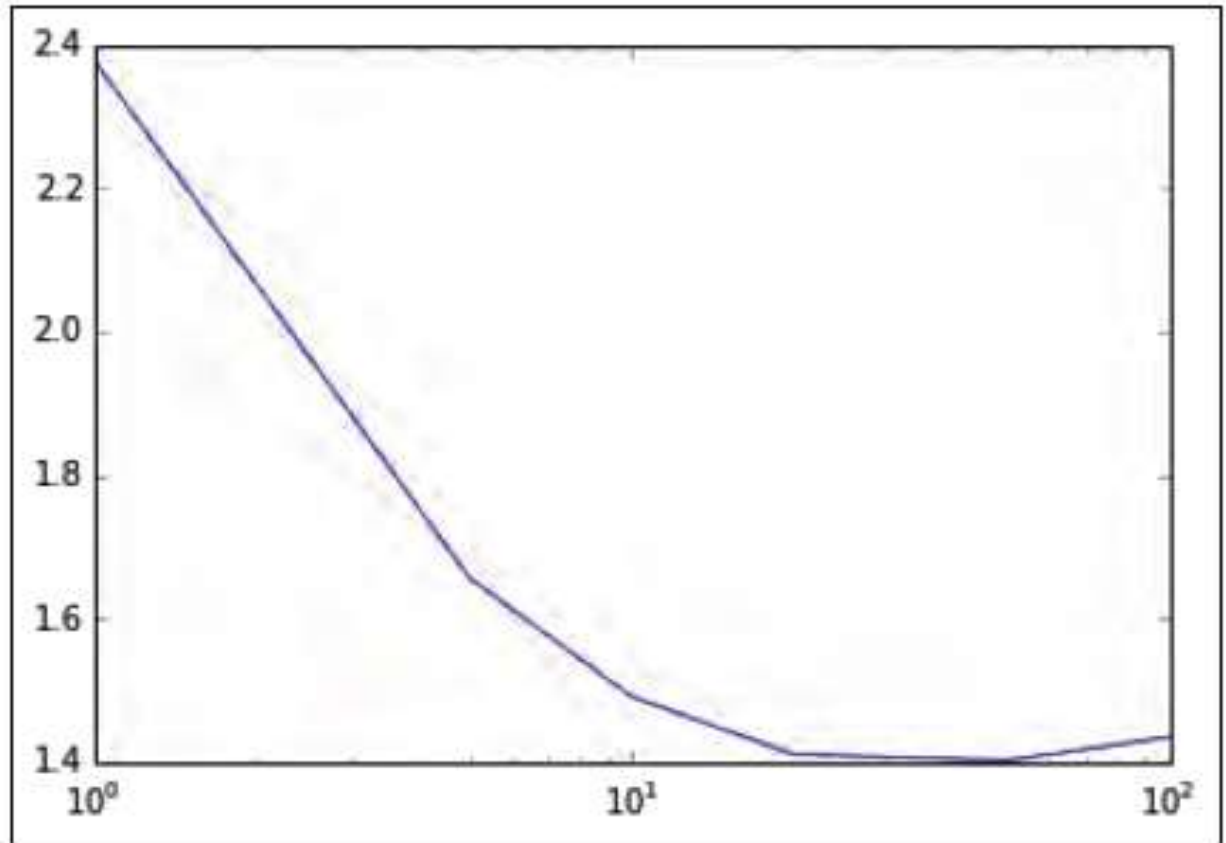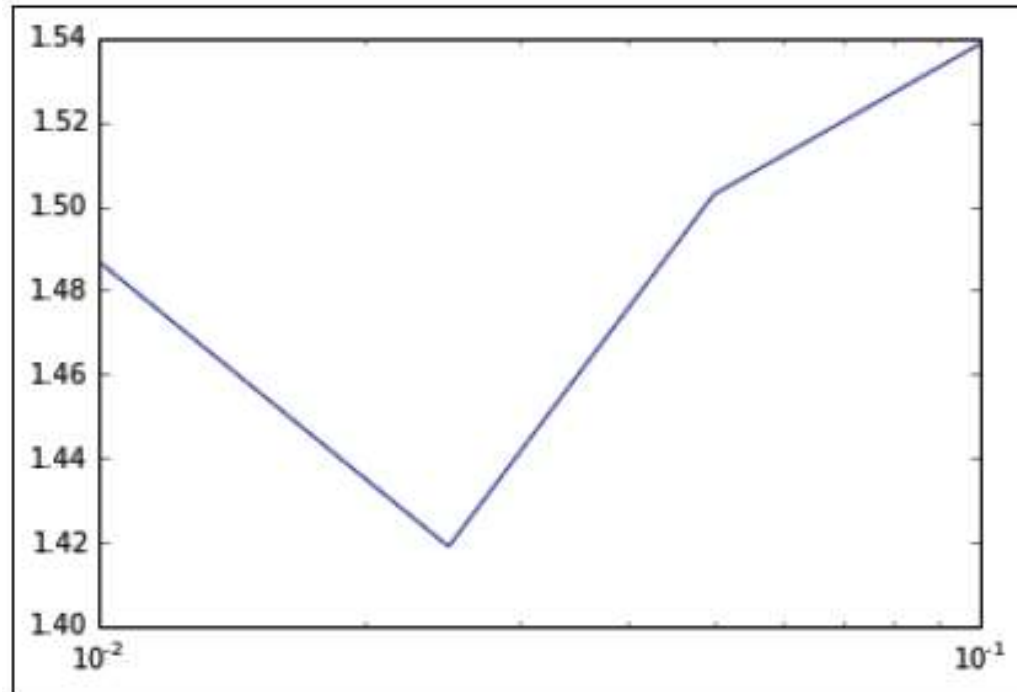
Step Size

- To examine how the step size influences results of linear regression analysis we will perform an analysis for the step size :

```
params = [0.01, 0.025, 0.05, 0.1, 1.0]
metrics = [evaluate(train_data, test_data, 10, param, 0.0, 'l2',
False) for param in params]
print params
print metrics
```

- The output of the preceding code:

```
[0.01, 0.025, 0.05, 0.1, 0.5]
[1.4869656275309227,
1.4189071944747715,
1.5027293911925559,
1.5384660954019973, nan]
```

Effect of Varying Step Size

- Now, we can see why we avoided using the default step size when training the linear model originally. The default is set to *1.0*, which, in this case, results in a nan (not-a-number) output for the RMSLE metric. This typically means that the SGD model has converged to a very poor local minimum in the error function that it is optimizing. This can happen when the step size is relatively large, as it is easier for the optimization algorithm to overshoot good solutions.

- We can also see that for low step sizes and a relatively low number of iterations (we used 10 here), the model performance is slightly poorer. However, in the preceding *Iterations* section, we saw that for the lower step-size setting, a higher number of iterations will generally converge to a better solution.

- Generally speaking, setting step size and number of iterations involves a trade-off. A lower step size means that convergence is slower but slightly more assured. However, it requires a higher number of iterations, which is more costly in terms of computation and time, in particular at a very large scale. Selecting the best parameter settings can be an intensive process that involves training a model on many combinations of parameter settings and selecting the best outcome.

Impact of Parameter Settings on Decision Tree Models

- Decision trees provide two main parameters: maximum tree depth and the maximum number of bins. We will now perform the same evaluation of the effect of parameter settings for the decision tree model.

- Our starting point is to create an evaluation function for the model, similar to the one used for the linear regression earlier. The function is defined as:

```
def evaluate_dt(train, test, maxDepth, maxBins):
    model = DecisionTree.trainRegressor(train, {},
        impurity='variance', maxDepth=maxDepth, maxBins=maxBins)
    preds = model.predict(test.map(lambda p: p.features))
    actual = test.map(lambda p: p.label)
    tp = actual.zip(preds)
    rmsle = np.sqrt(tp.map(lambda (t, p): squared_log_error(t,
        p)).mean())
    return rmsle
```

Tree Depth

- We would generally expect performance to increase with more complex trees (that is, trees of greater depth). Having a lower tree depth acts as a form of regularization, and it might be that there is a tree depth that is optimal with respect to the test set performance.

- Here, we will try to increase the depths of trees to see what impact they have on test set RMSLE, keeping the number of bins at the default level of 32:

```
params = [1, 2, 3, 4, 5, 10, 20]
metrics = [evaluate_dt(train_data_dt, test_data_dt, param, 32)for
param in params]
print params
print metrics
plot(params, metrics)
fig = matplotlib.pyplot.gcf()
```
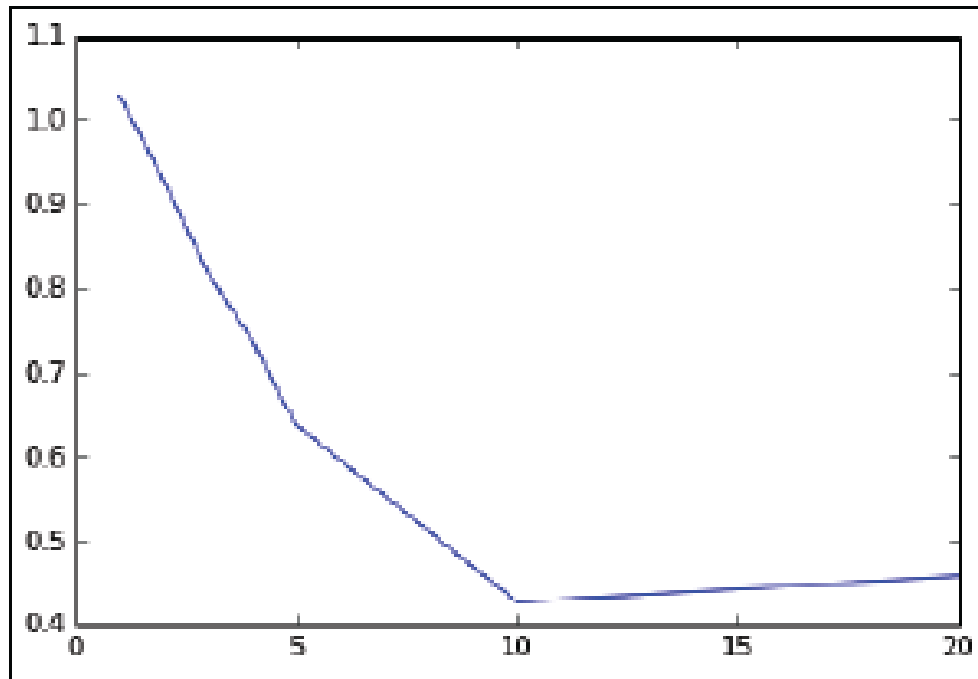
Tree Depth

- The output of the tree depth is as follows:

```
[1, 2, 3, 4, 5, 10, 20]
[1.0280339660196287, 0.92686672078778276, 0.81807794023407532,
0.74060228537329209, 0.63583503599563096, 0.4285136041869247,
0.4550008049779139]
```

- In this case, it appears that the decision tree starts over-fitting at deeper tree levels. An optimal tree depth appears to be around 10 on this dataset.

Number of bins

- Finally, we will perform our evaluation on the impact of setting the number of bins for the decision tree. As with the tree depth, a larger number of bins should allow the model to become more complex and might help performance with larger feature dimensions. After a certain point, it is unlikely that it will help any more and might, in fact, hinder performance on the test set due to over-fitting:

```
params = [2, 4, 8, 16, 32, 64, 100]
metrics = [evaluate_dt(train_data_dt, test_data_dt, 5, param) for
param in params]
print params
print metrics
plot(params, metrics)
fig = matplotlib.pyplot.gcf()
```

- The output is:

```
[2, 4, 8, 16, 32, 64, 100]
[1.3069788763726049, 0.81923394899750324,
0.75745322513058744,
0.62328384445223795, 0.63583503599563096,
0.63583503599563096,
0.63583503599563096]
```

Maximum Number of bins

- Below we show the output as we vary the number of bins (while keeping the tree depth at the default level of 5). In this case, using a small number of bins hurts performance, while there is no impact when we use around 32 bins (the default setting) or more.

- There seems to be an optimal setting for test set performance at around 16-20 bins.
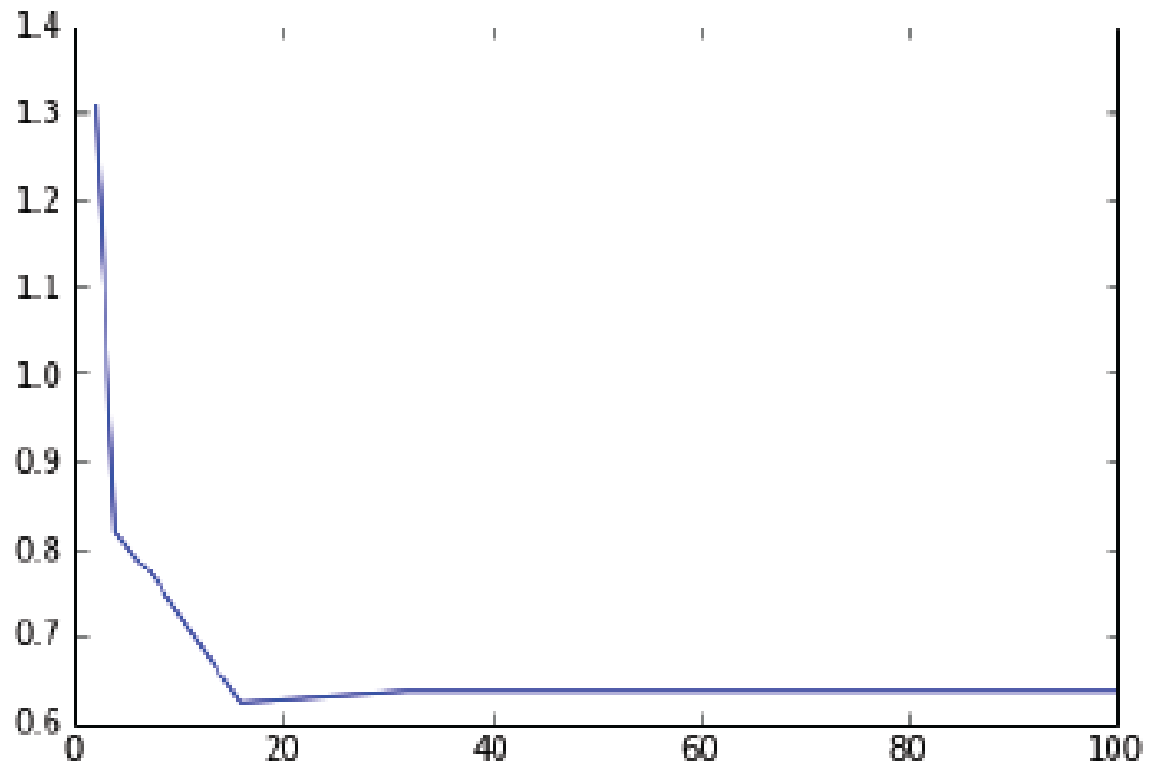
Instead of Conclusion

- Spark provides a sophisticated if not necessarily intuitive API for standard Machine Learning tasks.

- Spark's for Machine Learning MLlib is evolving and gaining both performance and (hopefully) convenience.

- MLlib is not the only game in town. At the moment appears very promising.