Linear regression:

- using a linear function to model the relationship between two variables by fitting a linear equation to observed data.

Notation:
n = number of features
x^(i) = inpute features of ith training example
(x^(i))j = value of features j in ith training example
m = number of examples

- choose the parameters we want to estimate so that the function/model we learn is close to y on all the training examples
- cost function (or loss function)
  - quantify the difference between the estimate value of model and true value

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

easy for derivatives averaged square of differences

Goal: choose θ_0 and θ_1 to minimize the cost function

RDUE

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

easy for derivatives averaged square of differences

Goal: choose θ_0 and θ_1 to minimize the cost function

RDUE

Outline:

- start with (random parameters)
- keep changing parameter to reduce cost function until we end up at a minimum

Gradient descent Algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \quad (\text{for } j = 0 \text{ and } j = 1)$$

}

learning rate — partial derivative of θ_j → move to the steepest direction

Linear regression with one variable

Gradient descent algorithm

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

$$(\text{for } j = 1 \text{ and } j = 0)$$

}

Linear Regression Model

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

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2$$

Hypothesis: $h_\theta(x) = \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$ ($x_0 = 1$)

Parameters: $\theta_0, \theta_1, \ldots, \theta_n$

Cost function:

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

Gradient descent:

Repeat {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \ldots, \theta_n)$$

}

(simultaneously update for every $j = 0, \ldots, n$)

PURDUE

Large Scale Machine Learning

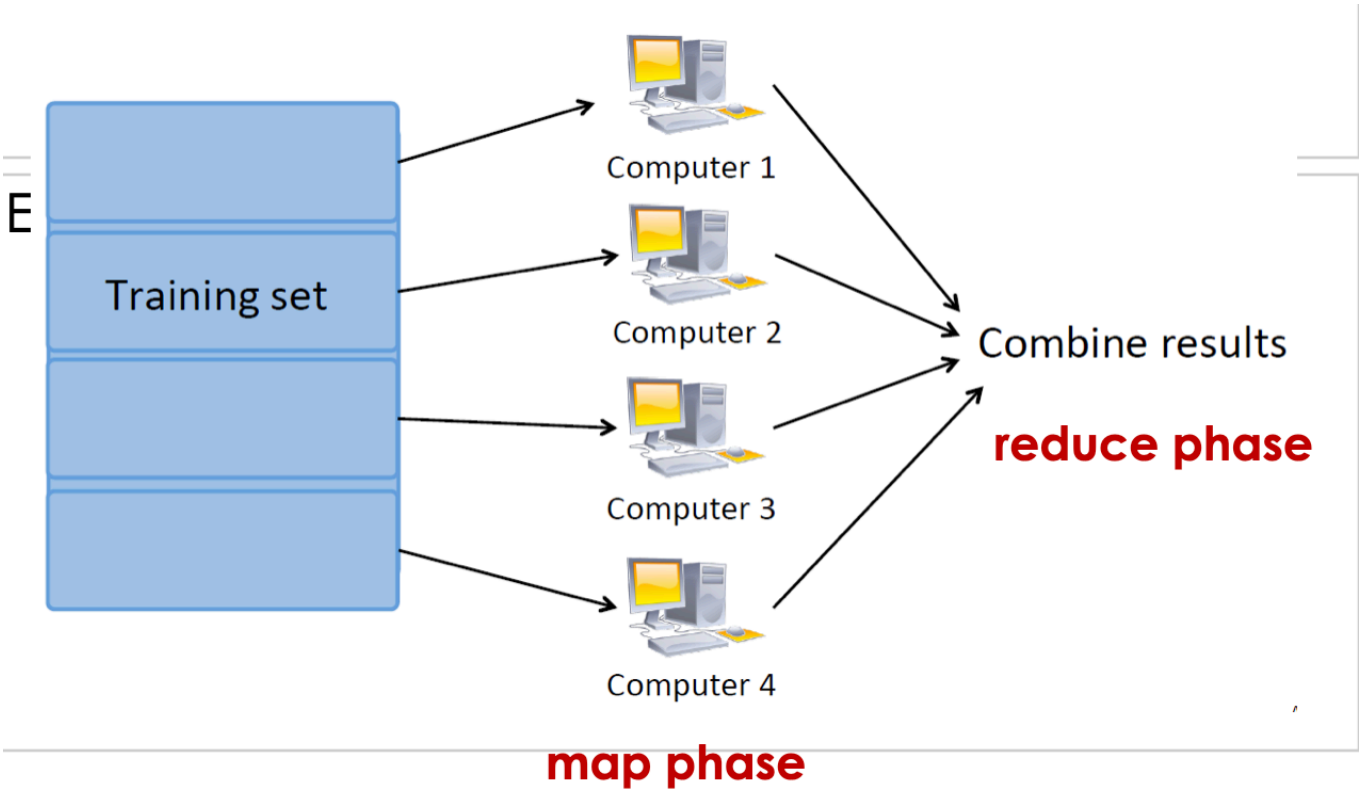
- If data is huge with billions or trillions of training examples, each iteration is very slow

Distributed ML

- Use many machines
- Each machine processes a partition of data
- Each machine computes the gradient on that machine
- Finally, combine all the gradients

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

parallelize this part



Distributed ML with Map-Reduce

- Map
  - workers do computation locally
  - map the training examples to temp
- Reduce
  - compute the sum of all temp

- Server side
  - compute the derivatives
  - Broadcast parameters to all workers
  - continue the next iteration

ML in Spark

- all the advantages of Spark extend to machine learning
  - spark's distributed nature: leverage Spark's RDDs to scale to large-scale data
  - Spark's unifying nature: offer a platform for performing most tasks in man applications
    - can collect, prepare, and analyze the data (various types)
      MLlib (spark, mllib)
- based on Mlbase project in Berkeley
- more mature
- based on RDDs

ML (spark.ml)

- new ml package, still developing
- end-to-end pipeline
- based on dataframe

Data Types in MLlib

- Two building blocks
  - vectors, matrices

For each building block there are:

- local version vs distributed version:
  - local: stored on a single node
  - distributed: based on RDD, stored on multiple nodes
- Dense version vs sparse version
  - sparse: contains a lot of 0's
  - dense: not that many 0's
- Underlying linear algebra operations are provided by Breeze and jblas

Dense vs Sparse Vectors

- Dense vector
  - it can take all elements as inline arguments or
  - it can take an array of elements

- Sparse vector
  - Need to specify a vector size, an array with indices (of non-zero values), and an array with values

Matrix

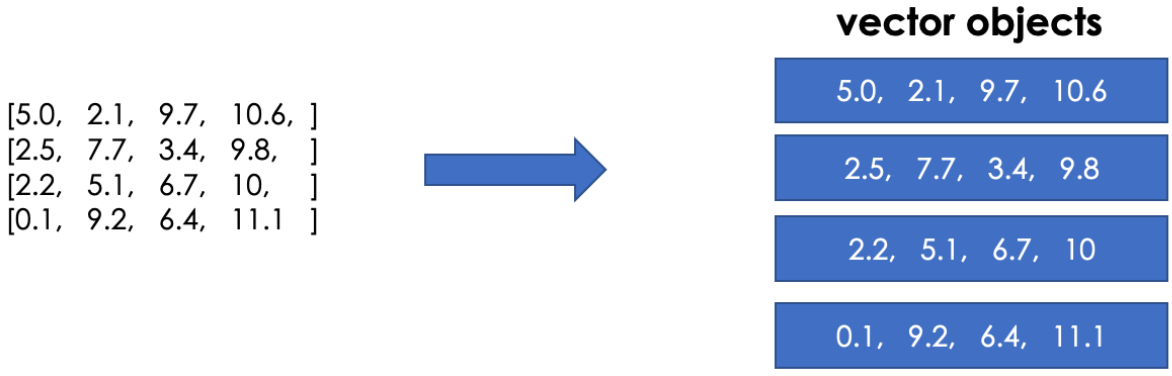
- a local matrix has integer-typed row and column indices and double-typed values, stored on a single machine.
  - dense matrix: entry values are stored in a single double array in column major
  - sparse matrix: compressed sparse column format

Distributed Matrix

- Distributed matrices are necessary when you're using ml algorithms on huge datasets
- they're stored across many machines, and they can have a large number of rows and columns
- Different forms
  - RowMatrix: used widely and we'll focus on it
  - IndexedRowMatrix
  - CoordinateMatrix
  - BlockMatrix

Row Matrix:

- Stores each row as a vector object



LabeledPoint

- LabeledPoint is another important data type
- it's a specialized vector that includes label and a feature vector

Training and Validation Data

- split the data into training and validation sets

- training set is used to train the model and a validation set is used to see how well the model performs on data that wasn't used to train it
- the usual split ration is 80% for the training set and 20% for the validation set

Predication

- you can now use the trained model to predict target values of vectors in the validation set by running predict on every element
- you can see how well the model is doing on the validation set by examining the contents of validPredicts:

Model Evaluataion

- Some predictions are close to original labels, and some are further off. To quantify the success of your model, calculate the root mean squared error (root of the cost function defined previously)