Overview

Review machine-learning approaches in the context of MapReduce and distributed learning.

Types of Learning

- **Unsupervised learning** models a set of inputs. like clustering.
- **Supervised learning** generates a function that maps inputs to desired outputs. For example, in a **classification** problem, the learner approximates a function mapping a vector into classes by looking at input-output examples of the function.
- **Semisupervised learning** combines both labeled and unlabeled examples to generate an appropriate function or classifier.
- **Reinforcement learning**:
    - One-shot decision making versus lifetime-value modeling.
    - Learns how to act given an observation of the world. Every action has some impact in the environment, and the environment provides feedback in the form of rewards that guide the learning algorithm.
      Transduction tries to predict new outputs based on training inputs, training outputs, and test inputs.

Machine Learning Background

Machine learning (ML): "a computer program that improves its performance at some task through experience" (Mitchell, 1997).

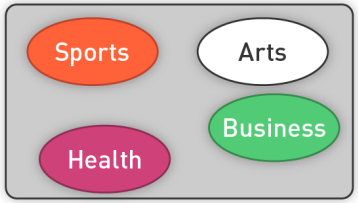
Given: Input data are a table of attribute values and associated class values (in the case of supervised learning).

Goal: Approximate $f(x_1,...,x_n) \rightarrow y$.

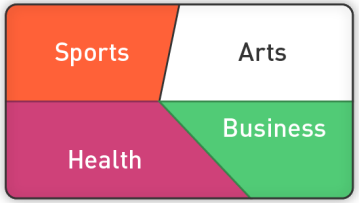
Instance\Attr	x_1	x_2	...	x_n	*y*
1	3	0	...	7	-1
2					+1
...	...	...	...	...	...
L (aka m)	0	4	...	8	-1

Families of Supervised Learning

- Generative Classifier (Bottom-up learning)
  - Build model of each class.
  - Assume the underlying form of the classes, and estimate their parameters (e.g., a Gaussian).



- Discriminative Classifier (Top down)
  - Build model of boundary between classes.
  - Assume the underlying form of the discriminant, and estimate its parameters (e.g., a hyperplane).



Generative vs. Discriminative

- Generative learning (e.g., Bayesian networks, HMM, naïve Bayes) typically more flexible
  - More complex problems
  - More flexible predictions

- Discriminative learning (e.g., linear regression, ANN, SVM) typically more accurate
  - Better with small datasets
  - Faster to train

Parametric vs. Nonparametric ML Algorithms

- Parametric ML algorithms (e.g., OLS, decision trees; SVMs, NNs)
  - Model-based methods, such as neural networks and the mixture of Gaussians, use the data to build a parameterized model. After training, the model is used for predictions, and the data are generally discarded.
- Nonparametric (lowess(); knn; some flavors SVMs)
  - In contrast, "memory-based'' methods are nonparametric approaches that explicitly retain the training data and use it each time a prediction needs to be made.
  - The term "nonparametric" (roughly) refers to the fact that the amount of stuff we need to keep in order to represent the hypothesis/model grows linearly with the size of the training set.
  - Very little learning; hyperparameter tuning.
    - Hyperparameter tuning can be readily done in MapReduce.
    - Prediction is challenging.

Matrix Operations

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & \vdots \\ \vdots & \ddots & \ddots & A_{m-1,n} \\ A_{m,1} & \cdots & A_{m,n-1} & A_{m,n} \end{bmatrix}$$

Ax	$Ax = b$	$\min \|Ax - b\|$	$Ax = \lambda x$
Operations	Linear systems	Least squares	Eigenvalues

Life Is a Matrix, and Then You Multiply!

Matrix computations are not just linear algebra (e.g., SQL query).

- Linear regression in closed form.
- Principle components analysis.
- PageRank for graph problems.
- Graphics.
- Navigation: shortest path.
- Robotics and automation: Matrices are the base elements for the robot movements.
- And many others.

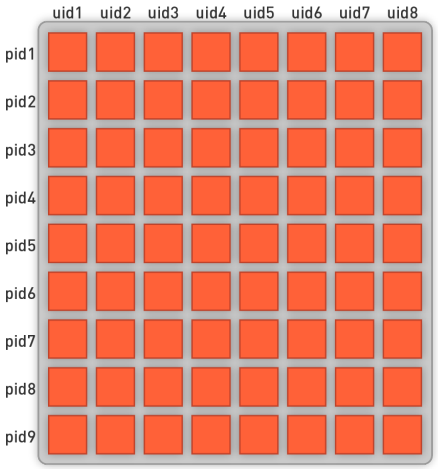
Example: Nonlinear Operation With Matrices

```
SELECT
    …
    AVG(pr.rating)
…
GROUP BY p.product_id
```

product_ratings

pid8	uid2	4
pid9	uid9	1
pid2	uid9	5
pid6	uid8	4
pid1	uid2	4
pid3	uid4	4
pid5	uid9	2
pid9	uid8	4
pid9	uid9	1

Is a matrix!

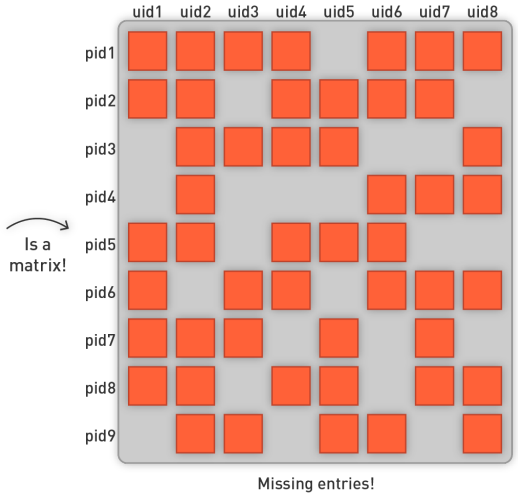


Source: David Gleich, Purdue. bit.ly/10SIe1A

But It's a Weird Matrix

product_ratings

pid8 uid2 4
pid9 uid9 1
pid2 uid9 5
pid6 uid8 4
pid1 uid2 4
pid3 uid4 4
pid5 uid9 2
pid9 uid8 4
pid9 uid9 1

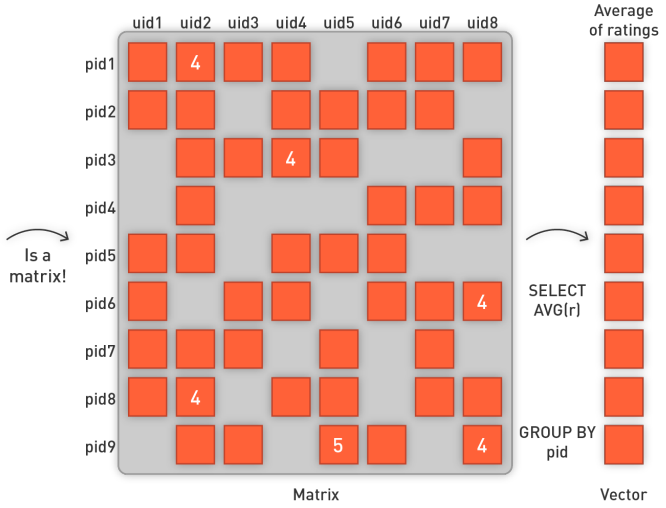


Source: David Gleich, Purdue. [bit.ly/10SIe1A](bit.ly/10SIe1A)

Group Up to Get Average

product_ratings

pid8	uid2	4
pid9	uid9	1
pid2	uid9	5
pid6	uid8	4
pid1	uid2	4
pid3	uid4	4
pid5	uid9	2
pid9	uid8	4
pid9	uid9	1



Source: David Gleich, Purdue. [bit.ly/10SIe1A](bit.ly/10SIe1A)

Not a Linear Operation

product_ratings

pid8 uid2 4
pid9 uid9 1
pid2 uid9 5
pid6 uid8 4
pid1 uid2 4
pid3 uid4 4
pid5 uid9 2
pid9 uid8 4
pid9 uid9 1

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & A_{2,2} & \cdots & \vdots \\ \vdots & \ddots & \ddots & A_{m-1,n} \\ A_{m,1} & \cdots & A_{m,n-1} & A_{m,n} \end{bmatrix}$$

$$\text{avg}(A) = \begin{bmatrix} \sum_j A_{1,j} \Big/ \sum_j \text{"}A_{1,j} \neq 0\text{"} \\ \sum_j A_{2,j} \Big/ \sum_j \text{"}A_{2,j} \neq 0\text{"} \\ \vdots \\ \sum_j A_{m,j} \Big/ \sum_j \text{"}A_{m,j} \neq 0\text{"} \end{bmatrix}$$

Source: David Gleich, Purdue. bit.ly/10SIe1A

More Matrix Computations

Wordcount is a matrix computation, too.

	term 1	**term 2**		**term *n***
doc 1	$A_{1,1}$	$A_{1,2}$	$\cdots$	$A_{1,n}$
doc 2	$A_{2,1}$	$A_{2,2}$	$\cdots$	$\vdots$
	$\vdots$	$\ddots$	$\ddots$	$A_{m-1,n}$
doc *m*	$A_{m,1}$	$\cdots$	$A_{m,n-1}$	$A_{m,n}$

$= A$

wordcount = colsum(A) = A^Te
e is the vector of all 1s.

Transposed matrix is a matrix computation, too.

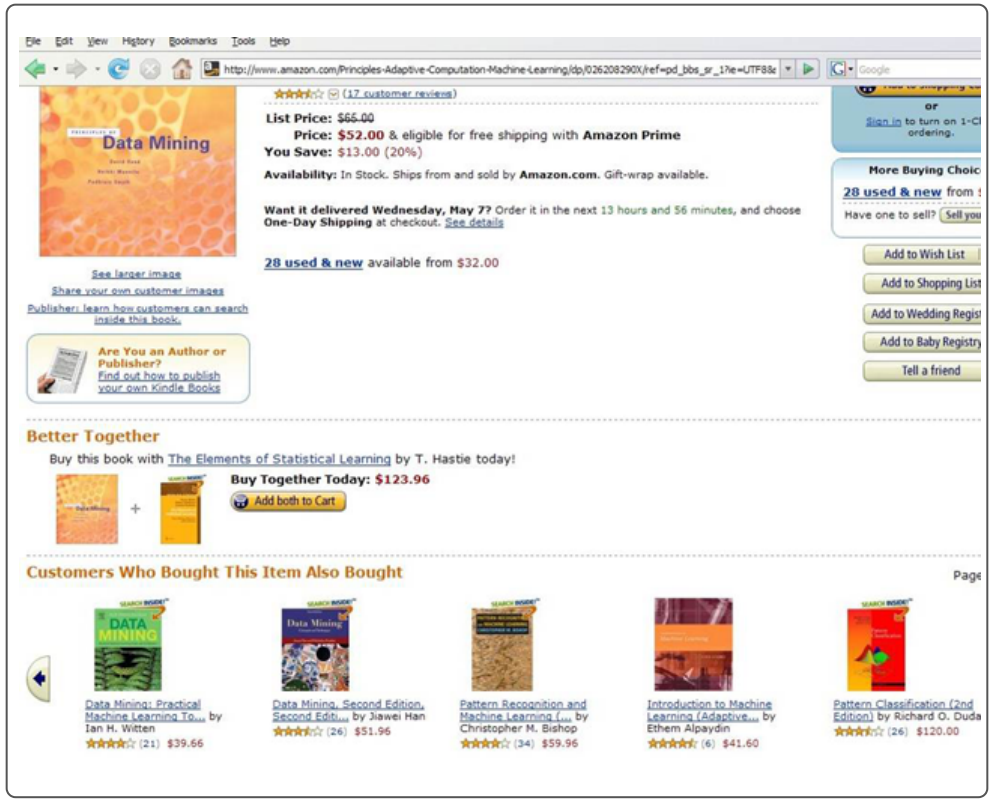
	doc 1	**doc 2**		**doc *n***
term 1	$A_{1,1}$	$A_{1,2}$	$\cdots$	$A_{1,n}$
term 2	$A_{2,1}$	$A_{2,2}$	$\cdots$	$\vdots$
	$\vdots$	$\ddots$	$\ddots$	$A_{m-1,n}$
term *m*	$A_{m,1}$	$\cdots$	$A_{m,n-1}$	$A_{m,n}$

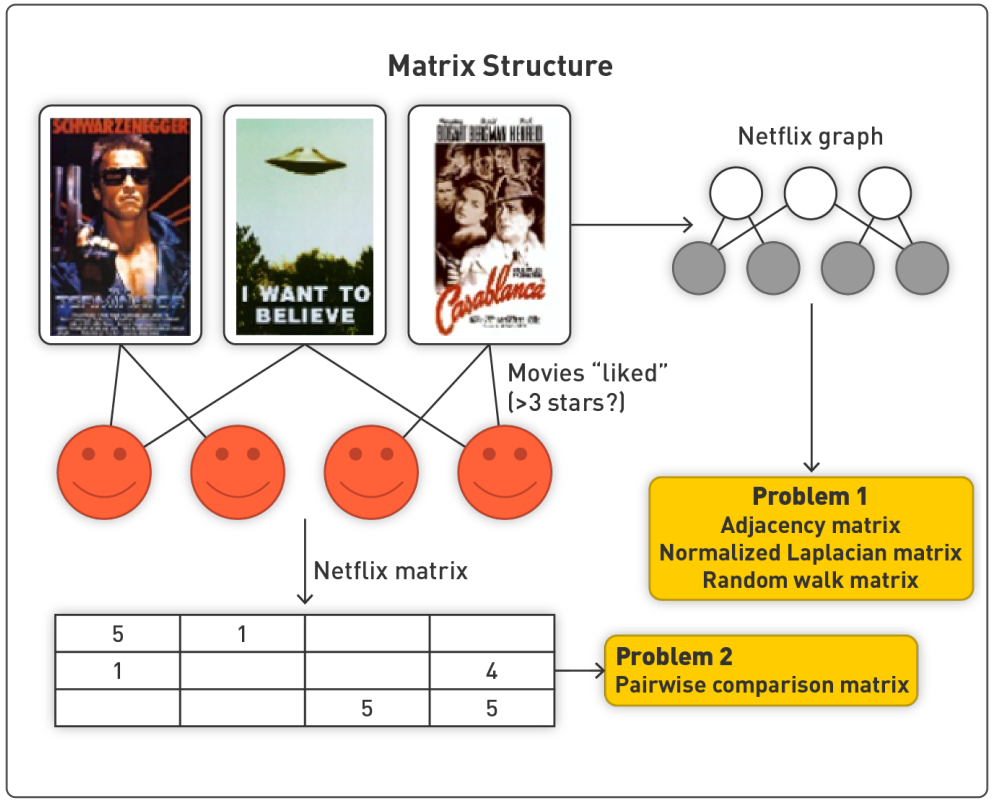
$= A^T$

Recommender Systems: User-Item Matrix

What is Bob's Rating of *The Matrix*?

	The Matrix	Alien	Inception
			
Alice	5	1	4
Bob	?	2	5
Peter	4	3	2


File Edit View History Bookmarks Tools Help
http://www.amazon.com/Principles-Adaptive-Computation-Machine-Learning/dp/026208290X/ref=pd_bbs_sr_1?ie=UTF8&
Google
(17 customer reviews)
Data Mining
List Price: $65.00
Price: $52.00 & eligible for free shipping with Amazon Prime
You Save: $13.00 (20%)
Availability: In Stock. Ships from and sold by Amazon.com. Gift-wrap available.
Want it delivered Wednesday, May 7? Order it in the next 13 hours and 56 minutes, and choose One-Day Shipping at checkout. See details
28 used & new available from $32.00
See larger image
Share your own customer images
Publisher: learn how customers can search inside this book.
Are You an Author or Publisher?
Find out how to publish your own Kindle Books
Sign in to turn on 1-Cl ordering.
More Buying Choic
28 used & new from
Have one to sell? Sell you
Add to Wish List
Add to Shopping List
Add to Wedding Regis
Add to Baby Registry
Tell a friend
Better Together
Buy this book with The Elements of Statistical Learning by T. Hastie today!
Buy Together Today: $123.96
Add both to Cart
Customers Who Bought This Item Also Bought
Page
Data Mining: Practical Machine Learning To... by Ian H. Witten
(21) $39.66
Data Mining, Second Edition, Second Editi... by Jiawei Han
(26) $51.96
Pattern Recognition and Machine Learning (... by Christopher M. Bishop
(34) $59.96
Introduction to Machine Learning (Adaptive... by Ethem Alpaydin
(6) $41.60
Pattern Classification (2nd Edition) by Richard O. Duda
(26) $120.00


Matrix Structure
SCHWARZENEGGER
I WANT TO BELIEVE
Casablanca
Netflix graph
Movies "liked" (>3 stars?)
Problem 1
Adjacency matrix
Normalized Laplacian matrix
Random walk matrix
Netflix matrix
5 1
1 4
5 5
Problem 2
Pairwise comparison matrix


News Feed Ranking:
1 Billion Personalized Feeds
David Vickrey

The Web as a Graph

Connectivity: Is it possible to reach most pages from most pages?

The web is a bow tie!

The web graph is also scale-free, fractal: Many slices and subgraphs exhibit similar properties.

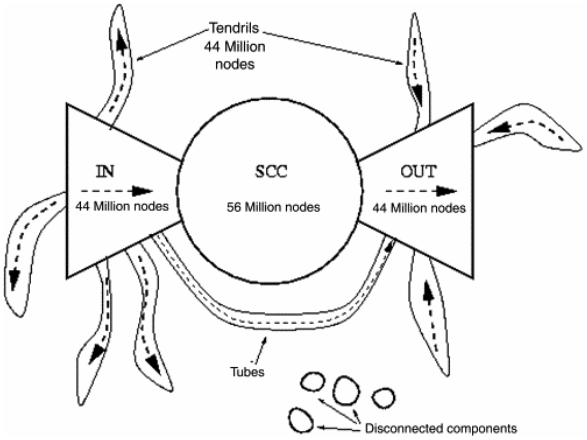
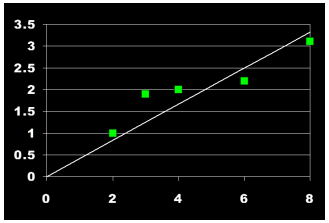


Image source: Graph Structure in the Web, Broder et al. (2000)

Closed Form of Linear Regression



$$E = \sum_i (y_i - wx_i)^2$$
$$= \sum_i y_i^2 - (2\sum_i x_i y_i)w + (\sum_i x_i^2)w^2$$

Minimize quadratic function of w.
E minimized with:

$$w^* = (\sum_i x_i y_i) / (\sum_i x_i^2)$$

so ML model is $Out(x) = w^* x$.

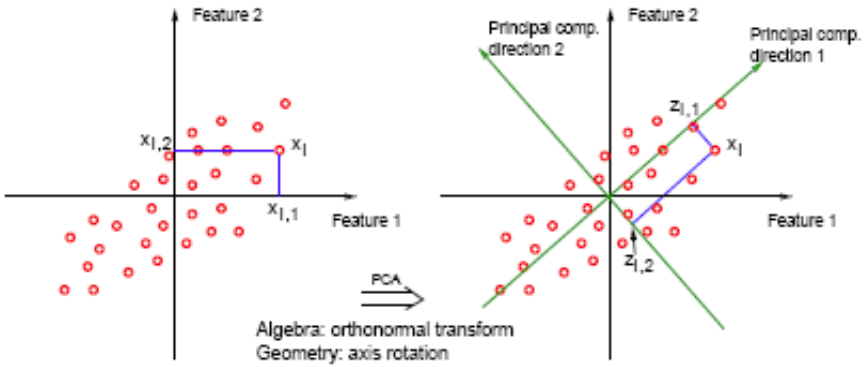
What if inputs are vectors?
n data points, D components

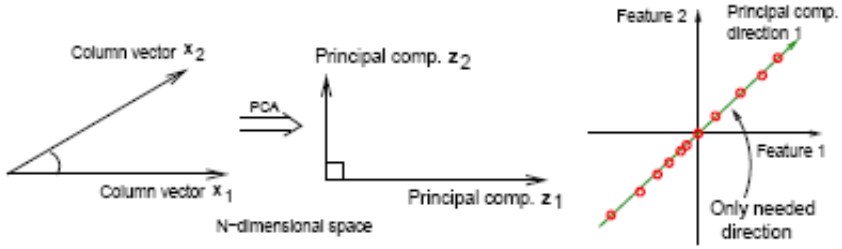
$$X = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \quad Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

(X: $n \times D$)

ML solution:
$w = (X^T X)^{-1}(X^T Y)$

PCA





Source: https://onlinecourses.science.psu.edu/stat557

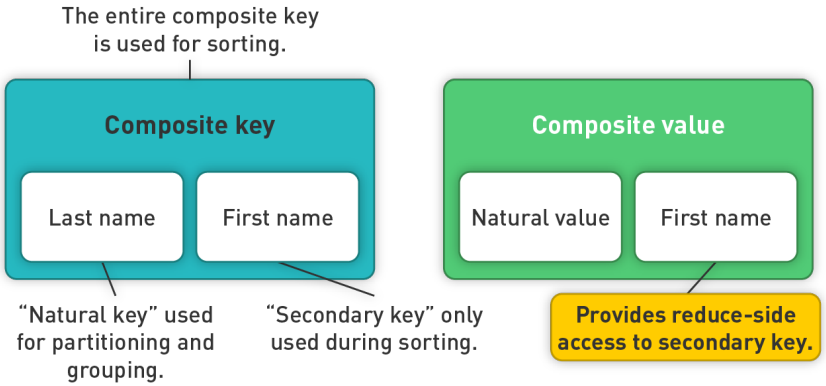
Conclusion

- Matrices are wonderful things.
- Wouldn't it be great if we could scale them?
- Can we distribute them in a MapReduce framework?

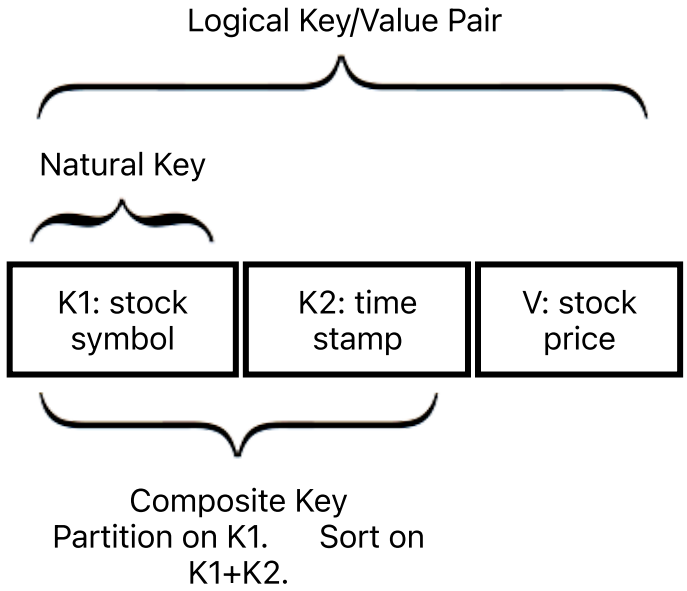
Matrix multiplication is the backbone of many great algorithms, including linear regression and page rank.

Composite Key: Natural Key + Secondary Key

Very useful: gets the MapReduce framework to do all the heavy lifting!



Composite Key: Partition and Sort



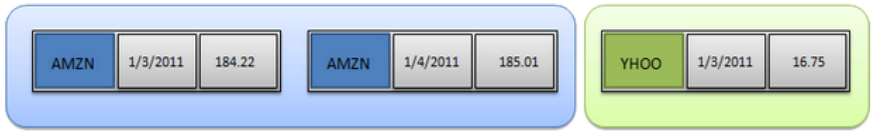
The Composite Key gives MapReduce framework the needed information during the shuffle to perform a sort not only on the "stock symbol" but on the time stamp as well.

Secondary Sort: Composite Key: "Stock Symbol" + Time Stamp

0	0		0	1		0	2		0	3		1	0		1	1	
K1	K2	V															

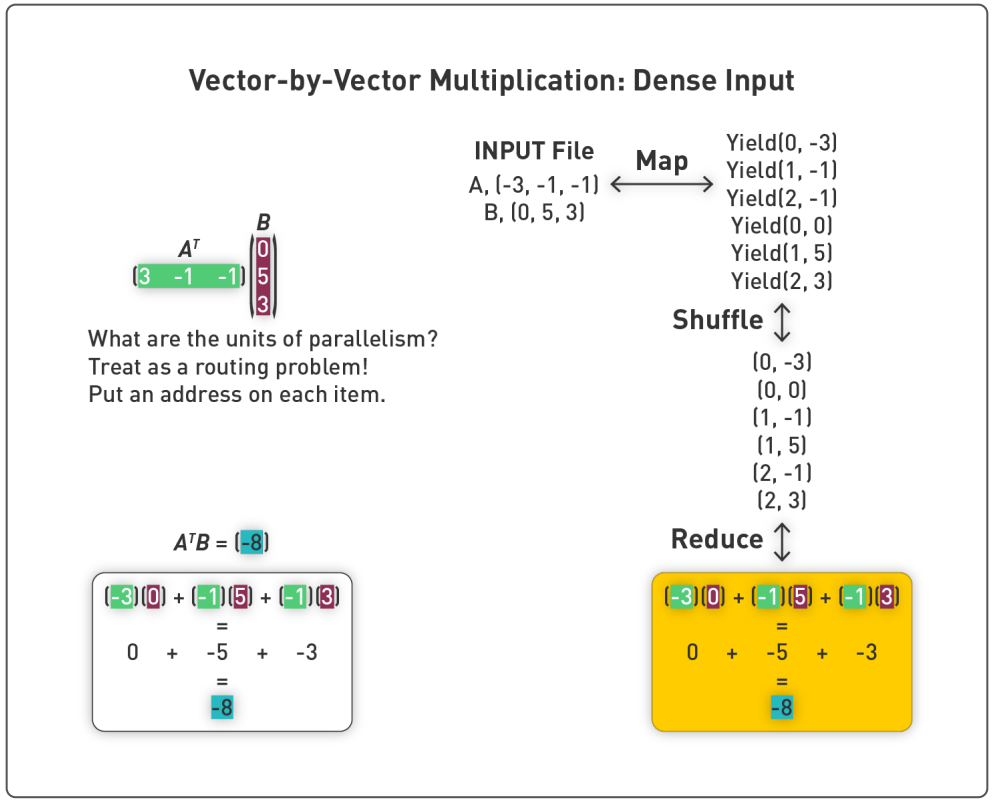
Sorting by the Composite Key, using integers for both keys for simplicity

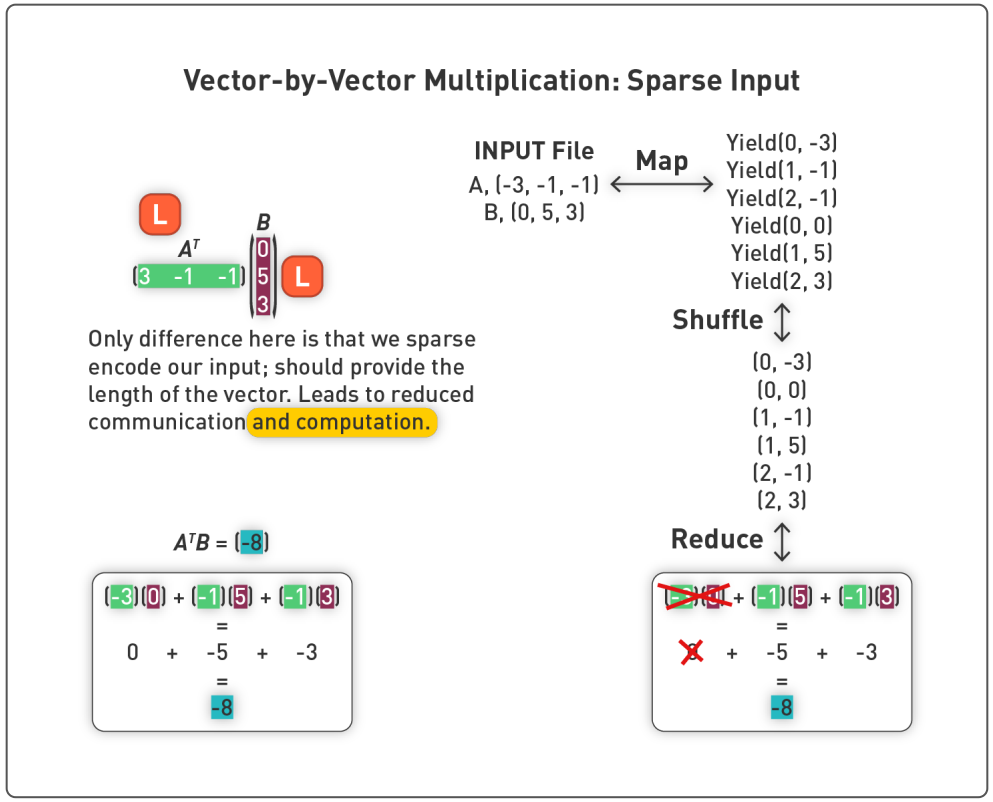
The Composite Key gives MapReduce framework the needed information during the shuffle to perform a sort not only on the "stock symbol" but on the time stamp as well.

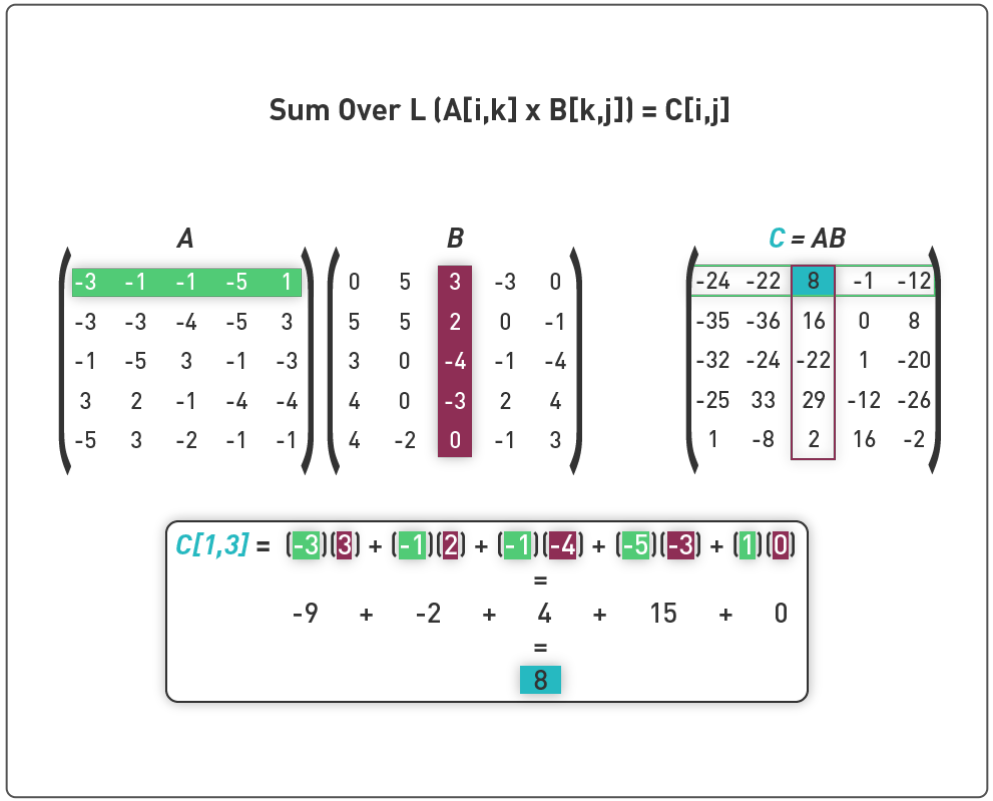


"AMZN" partitioning by the natural key.

"YHOO" partitioning by the natural key.

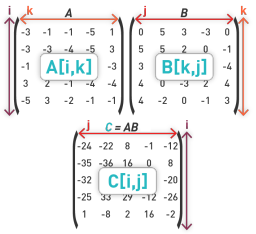

Vector-by-Vector Multiplication: Dense Input
INPUT File
A, (-3, -1, -1)
B, (0, 5, 3)
Map
Yield(0, -3)
Yield(1, -1)
Yield(2, -1)
Yield(0, 0)
Yield(1, 5)
Yield(2, 3)
Shuffle
(0, -3)
(0, 0)
(1, -1)
(1, 5)
(2, -1)
(2, 3)
Reduce
A^T
B
(3 -1 -1)
0
5
3
What are the units of parallelism?
Treat as a routing problem!
Put an address on each item.
A^TB = (-8)
(-3)(0) + (-1)(5) + (-1)(3)
=
0 + -5 + -3
=
-8
(-3)(0) + (-1)(5) + (-1)(3)
=
0 + -5 + -3
=
-8


Vector-by-Vector Multiplication: Sparse Input
INPUT File
A, (-3, -1, -1)
B, (0, 5, 3)
Map
Yield(0, -3)
Yield(1, -1)
Yield(2, -1)
Yield(0, 0)
Yield(1, 5)
Yield(2, 3)
Shuffle
(0, -3)
(0, 0)
(1, -1)
(1, 5)
(2, -1)
(2, 3)
Reduce
L
A^T
(3 -1 -1)
B
0
5
3
L
Only difference here is that we sparse encode our input; should provide the length of the vector. Leads to reduced communication and computation.
A^TB = (-8)
(-3)(0) + (-1)(5) + (-1)(3)
=
0 + -5 + -3
=
-8
(-1)(5) + (-1)(3)
=
+ -5 + -3
=
-8


Sum Over L (A[i,k] x B[k,j]) = C[i,j]
A
B
C = AB
C[1,3] = (-3)(3) + (-1)(2) + (-1)(-4) + (-5)(-3) + (1)(0)
=
-9 + -2 + 4 + 15 + 0
=
8

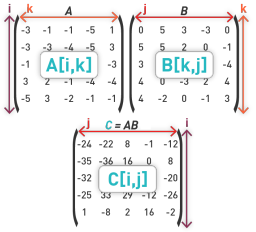
Distribute Matrix-by-Matrix Multiplication

- Once again, look at the the output matrix elements C_{ij}; all are independent of each other from the start of the operation to the end result.
- Atomic level of parallelization.



- A number of ways to tackle this problem.

One Job: Matrix-Vector Multiplication



- Need to know *k*, the number of columns in matrix A (or the number of rows in matrix B)
- Best to know I, K, and J:
  - I number of rows in A
  - K number of columns in A (number of rows in B)
  - J number of columns in B

- Emits a lot of superfluous data into the mapper-to-reducer stream, i.e., a dense emit
  - If value exists, then emit a value for each *k* (the number of columns in matrix A) and targets that could potentially involve that cell, i.e., all cells in row *k* in matrix B
  - In short B[k,*]

Input Data Format: Dense vs. Sparse

- MapReduce needs to divide and aggregate data (matrix) by key-value pairs.
- It is easier to represent data as (*rowIndex*, *columnIndex*, *value*).
- Regardless of the input data format, we can derive input value (*rowIndex*, *columnIndex*, *value*) from matrix.
    - Dense matrix in array format can be easily scanned to be transformed to (*rowIndex*, *columnIndex*, *value*).
    - Sparse format will be in this format already or can be transformed to this format easily.

Version 1 One-Step MapReduce

- To calculate C_{ij}, we need A_{ik}, and B_{kj} for *k*=1, ... *K*

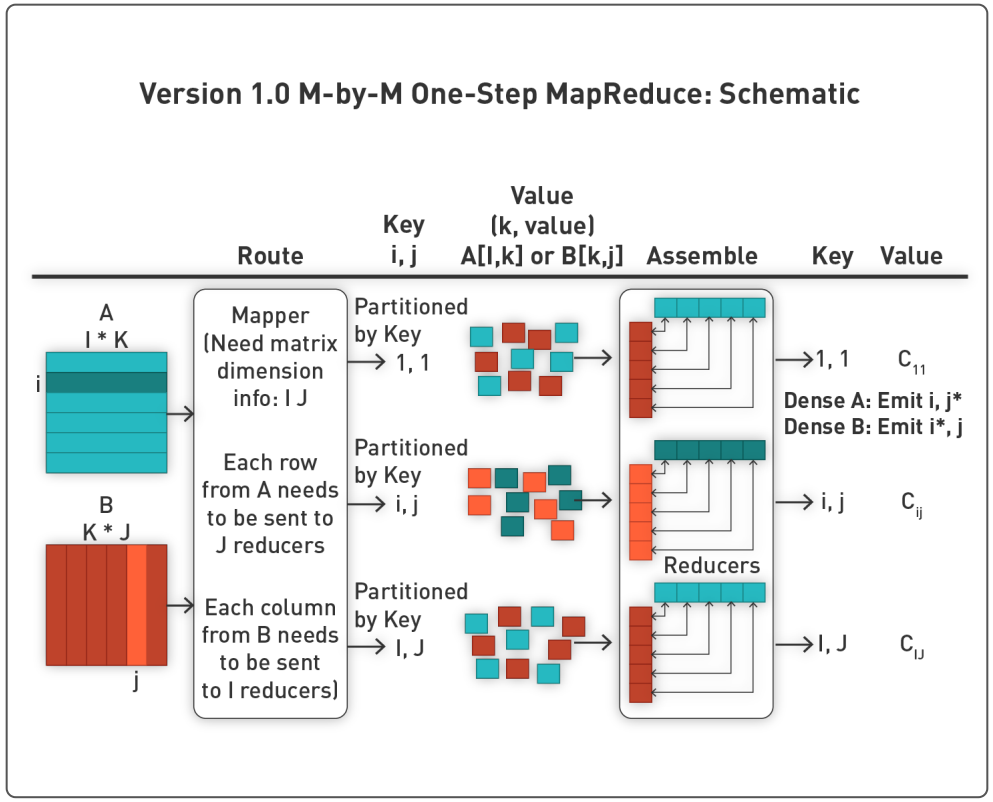
$$C_{ij} = \sum_{k=1}^{K} A_{ik} B_{kj}$$

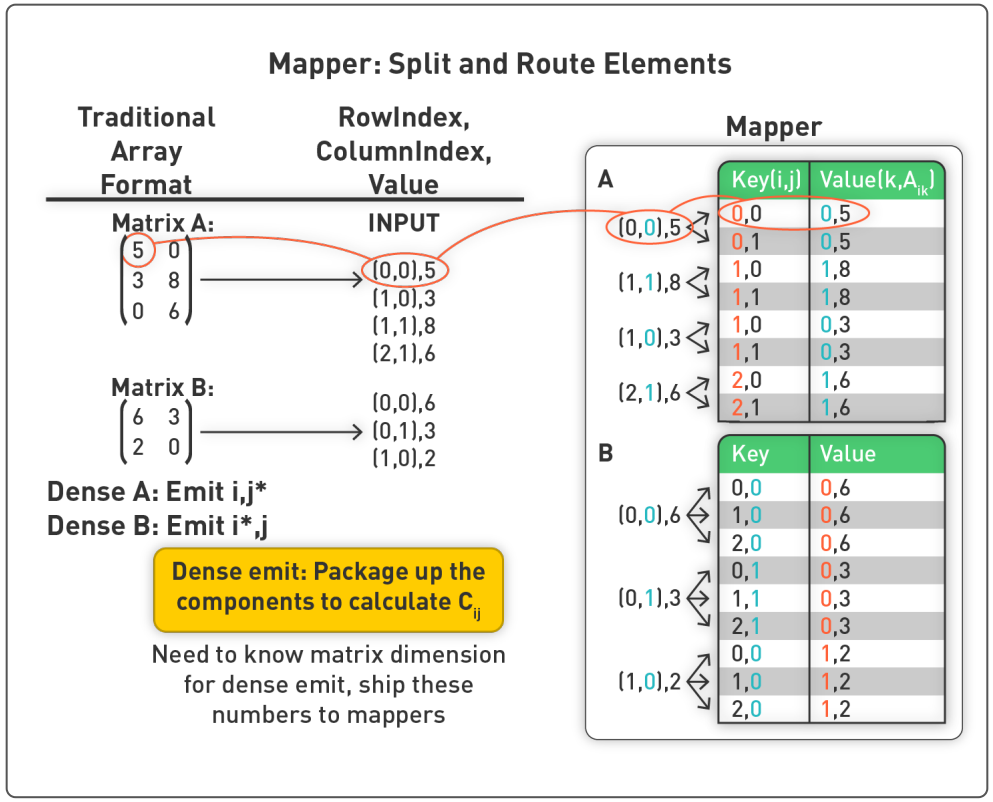
- In the map phase:
  - For each element (*i*, *k*) of *A*, emit
    - Key = (*i*, *j*), Value=(*k*, A_{ik}) for *j* in 1 ... *J*
  - For each element (*k*, *j*) of *B*, emit
    - Key = (*i*, *j*), Value = (*k*, B_{kj}) for *i* in 1 ... *I*

Dense A: Emit i,j

Dense B: Emit i*,j

- In the reduce phase, emit
  - Key = (*i*, *j*), Value = $\Sigma_k \left(A_{ij} \cdot B_{jk} \right)$


Version 1.0 M-by-M One-Step MapReduce: Schematic
Route
Key
i, j
Value
(k, value)
A[I,k] or B[k,j]
Assemble
Key
Value
A
I * K
i
B
K * J
j
Mapper
(Need matrix dimension info: I J
Each row from A needs to be sent to J reducers
Each column from B needs to be sent to I reducers)
Partitioned by Key
1, 1
Partitioned by Key
i, j
Partitioned by Key
I, J
Reducers
1, 1
C11
Dense A: Emit i, j*
Dense B: Emit i*, j
i, j
Cij
I, J
CIJ


Mapper: Split and Route Elements
Traditional Array Format
RowIndex, ColumnIndex, Value
Matrix A:
INPUT
(0,0),5
(1,0),3
(1,1),8
(2,1),6
Matrix B:
(0,0),6
(0,1),3
(1,0),2
Dense A: Emit i,j*
Dense B: Emit i*,j
Dense emit: Package up the components to calculate Cij
Need to know matrix dimension for dense emit, ship these numbers to mappers
Mapper
A
Key(i,j)
Value(k,Aik)
0,0 0,5
0,1 0,5
1,0 1,8
1,1 1,8
1,0 0,3
1,1 0,3
2,0 1,6
2,1 1,6
B
Key
Value
0,0 0,6
1,0 0,6
2,0 0,6
0,1 0,3
1,1 0,3
2,1 0,3
0,0 1,2
1,0 1,2
2,0 1,2

Reducer: Assembling Components for C_{ij}

Need in-memory hashing
Reduce-side join of all components of C_{ij}

Key (i,j)	Value (k,value)
0,0	(0,5), (0,6), (1,2)
0,1	(0,5), (0,3)
1,0	(1,8), (0,3), (0,6), (1,2)
1,1	(1,8), (0,3), (0,3)
2,0	(1,6), (0,6), (1,2)
2,1	(1,6), (0,3)

Join by *k* and sum up the product

1,[3,6]
0,[8,2]

1,18
0,16

Key (i, j)	Value C(i,j)
0,0	30
0,1	15
1,0	34
1,1	9
2,0	12
2,1	0

Write to file the final matrix result

A

$$\begin{pmatrix} 5 & 0 \\ 3 & 8 \\ 0 & 6 \end{pmatrix}$$

B

$$\begin{pmatrix} 6 & 3 \\ 2 & 0 \end{pmatrix}$$

C

$$\begin{pmatrix} 30 & 14 \\ 34 & 9 \\ 12 & 0 \end{pmatrix}$$

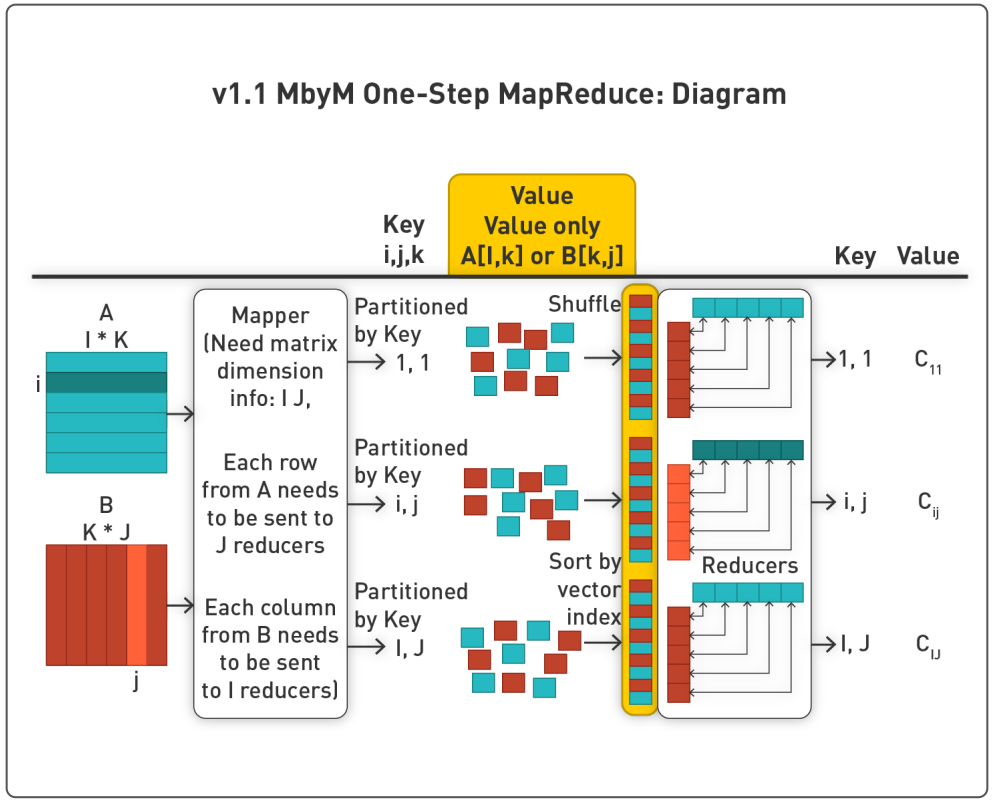
One Step MapReduce Pros and Cons

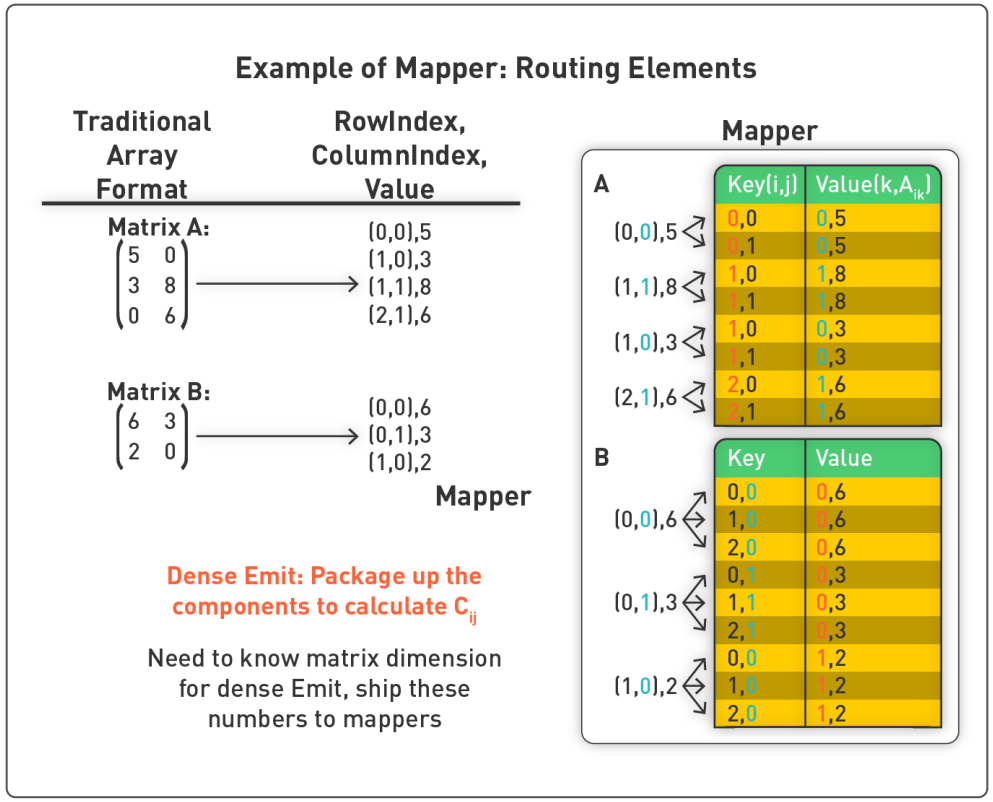
- Pros:
  - Straightforward
    - The same as manual calculation
    - Only need MapReduce step
- Cons:
  - Reducer-side join of C_{ij} in memory
    - Component when *K* is too big, in reducer method 1 cannot load all data into memory to join and sum up the product (the hash table to hold the component forC_{ij})
  - If matrices dimensions are unknown, method does not work
  - Network communication is heavy: Dense emits of individual elements of matrixA_{ik} and B_{kj}

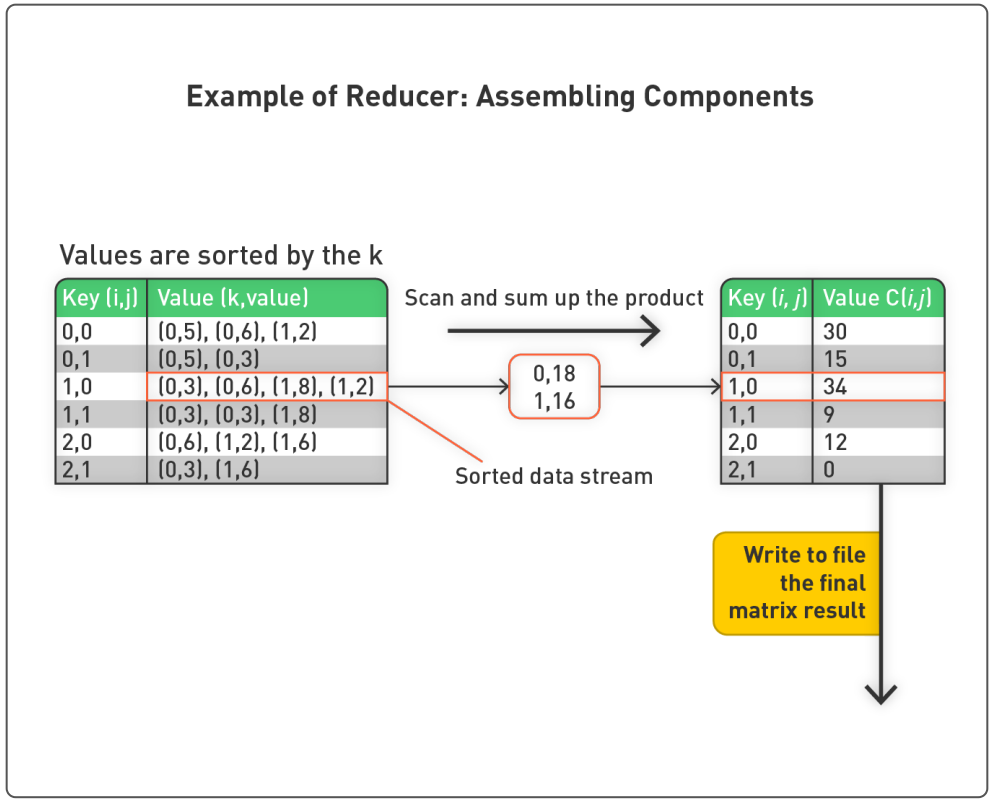
How do we make matrix multiplication more efficient and scalable?

v1.1 One-Step MapReduce (Improved) Matrix-by-Matrix Multiplication

- Drawbacks of v1.0 of the one-step MapReduce
  - When M is too big, in reducer, method1.0 cannot load all data into memory to join and sum up the product.
  - If the dimensions of two matrices are unknown, the method does not work.
  - Network communication is heavy: dense emits of individual elements of matrixA_{ik} and B_{kj}.
- Solution:
  - Secondary sort of *k* values:
    - If we sort the output values of mapper by *k*, we don't need to join values by *k* in memory, just scan values by on pass to get the sum of product without loading all data into memory.


v1.1 MbyM One-Step MapReduce: Diagram
Key
i,j,k
Value
Value only
A[I,k] or B[k,j]
Key
Value
A
I * K
i
B
K * J
j
Mapper
(Need matrix dimension info: I J,
Each row from A needs to be sent to J reducers
Each column from B needs to be sent to I reducers)
Partitioned by Key
1, 1
Partitioned by Key
i, j
Partitioned by Key
I, J
Shuffle
Sort by vector index
Reducers
1, 1
C11
i, j
Cij
I, J
CIJ


Example of Mapper: Routing Elements
Traditional Array Format
RowIndex, ColumnIndex, Value
Matrix A:
5 0
3 8
0 6
(0,0),5
(1,0),3
(1,1),8
(2,1),6
Matrix B:
6 3
2 0
(0,0),6
(0,1),3
(1,0),2
Mapper
Dense Emit: Package up the components to calculate C_{ij}
Need to know matrix dimension for dense Emit, ship these numbers to mappers
Mapper
A
Key(i,j)
Value(k,A_{ik})
0,0 0,5
0,1 0,5
1,0 1,8
1,1 1,8
1,0 0,3
1,1 0,3
2,0 1,6
2,1 1,6
(0,0),5
(1,1),8
(1,0),3
(2,1),6
B
Key
Value
0,0 0,6
1,0 0,6
2,0 0,6
0,1 0,3
1,1 0,3
2,1 0,3
0,0 1,2
1,0 1,2
2,0 1,2
(0,0),6
(0,1),3
(1,0),2


Example of Reducer: Assembling Components
Values are sorted by the k
Key (i,j)
Value (k,value)
0,0
(0,5), (0,6), (1,2)
0,1
(0,5), (0,3)
1,0
(0,3), (0,6), (1,8), (1,2)
1,1
(0,3), (0,3), (1,8)
2,0
(0,6), (1,2), (1,6)
2,1
(0,3), (1,6)
Scan and sum up the product
0,18
1,16
Sorted data stream
Key (i, j)
Value C(i,j)
0,0
30
0,1
15
1,0
34
1,1
9
2,0
12
2,1
0
Write to file the final matrix result

1.1 One-Step MapReduce: Limitations

Solution:

- Secondary sort of *k* values:
  - If we sort the output values of mapper by *k*, we don't need to join values by *k* in memory, just scan values by on pass to get the sum of product without loading all data into memory.
- Only limitations:
  - Need to provide matrix dimension.
  - Network communication is heavy: dense emits of individual elements of matrixA_{ik} and B_{kj}.

Solution

- Two-step MapReduce:
  - Intermediate products are calculated in first reducers.
  - Sum these products up at second reducers.
- Click here for a detailed description of the Two-Step MapReduce Approach to Matrix-by-Matrix-Multiplication

Next: In summary.

Matrix multiplication is complex on paper and also in a parallel framework such as MapReduce.

Which Method Is Better?

- One-step with dense emit
- Two-step with a heavy in-memory sync around *k*-based components

Method Comparisons

Distributed matrix multiplication is challenging!

	Method 1.0	Method 1.1	Method 2
Number of jobs	1	1	2
Need big memory	Yes in Reducer	No	Yes in Reducer1
Need to know dimensions of matrix	Yes	Yes	No
Dense emit	Yes	Yes	No

List of Matrix-by-Matrix Multiplications in Hadoop (Method 1)

- **Method 1 One Job: Matrix-Matrix Multiplication**
  Dense emit. Works with sparse input?
  - Mapper output key is *i,j* ($C_{i,j}$).
  - Need to know *k*, the number of columns in matrix A (or the number of rows in matrix B).
  - Risks running out of memory in the first reducer (do a sort/join in memory).
  - Emits a lot superfluous data into the mapper-to-reducer stream, i.e., a dense emit.
    - If value exists, then emit a value for each *k* (the number of columns in matrix A) and similarly for cells in matrix B.

List of Matrix-by-Matrix Multiplications in Hadoop (Method 1.1)

- **Method 1.1 One Job: Matrix-Matrix Multiplication**
  Dense emit
  - Mapper output key is *i,j*, and do a secondary sort of on *I,j,k*.
  - Need to know *k*, the number of columns in matrix A (or the number of rows in matrix B).
  - Emits a lot superfluous data into the mapper-to-reducer stream, i.e., a dense emit.
    - If value exists, then emit a value for each *k* (the number of columns in matrix A) and similarly for cells in matrix B.

List of Matrix-by-Matrix Multiplications in Hadoop (Method 2)

- **Method 2 Two Jobs: Matrix-Matrix Multiplication**
  - Mapper output key is *k* ($C_{\mathrm{i,j}}$).
  - No need to know K; more complex.
  - Risks running out of memory in the first reducer (in-memory join).

List of Matrix-by-Matrix Multiplications in Hadoop (Tile-Based)

- **Tile-Based Matrix Multiplication**
  - "MadLINQ: Large-Scale Distributed Matrix Computation for the Cloud," Qian et al, EuroSys 2012.

Overview

- In this section again we will focus on basics, including more heavy infrastructure.
- In this case we will refresh optimization theory before zeroing in on convex optimization.

Optimization Definition

- In mathematics, computer science, economics, or management science:
- **Mathematical optimization** is the selection of a best element (with regard to some criteria) from some set of available alternatives.
- In the simplest case, an optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function.

Mathematical Optimization

Finding the minimizer of a function subject to constraints:

	$\underset{x}{\text{minimize}} \quad f_o(x)$	An objective function, a loss function, or cost function (minimization)
s.t.	$f_i(x) \leq 0,\ i = \{1, \ldots, k\}$	Inequality constraints
	$h_j(x) = 0,\ j = \{1, \ldots, l\}$	Equality constraints

- Example: Stock market. "Minimize variance of return subject to getting at least $50."
- Such a formulation is called an optimization problem or a mathematical programming problem.
- Many real-world and theoretical problems may be modeled in this general framework.

Source:"Introduction to Convex Optimization for Machine Learning," John Duchi, University of California, Berkeley.

Mathematical Optimization (cont.)

- The function f_0 is called, variously, an objective function, a loss function or cost function (minimization),[2] a utility function or fitness function (maximization), or, in certain fields, an energy function or energy functional.
    - A feasible solution that minimizes (or maximizes, if that is the goal) the objective function is called an optimal solution.

Source:"Mathematical Optimization," Wikipedia.

Why Do We Care?

Optimization is at the heart of many (most practical?) machine-learning algorithms.

- Linear regression:

$$\underset{w}{\text{minimize}}\ ||xw - y||^2$$

- Classification (logistic regression or SVM):

$$\underset{w}{\text{minimize}}\ \sum_{i=1}^{n} \log\left(1 + \exp\left(-y_i x_i^T w\right)\right)$$

or

$$||w||^2 + C\sum_{i=1}^{n} \xi_i \quad \text{s.t. } \xi_i \geq 1 - y_i x_i^T w, \xi_i \geq 0.$$

Source:"Introduction to Convex Optimization for Machine Learning," John Duchi, University of California, Berkeley.

We Still Care

- Maximum likelihood estimation:

$$\underset{\theta}{\text{maximize}} \quad \sum_{i=1}^{n} \log p_\theta\left(x_i\right)$$

- Collaborative filtering:

$$\underset{w}{\text{minimize}} \quad \sum_{i \prec j} \log\left(1 + \exp\left(w^T x_i - w^T x_j\right)\right)$$

- *k*-means:

$$\underset{\mu_1, \cdots, \mu_k}{\text{minimize}} \quad J\left(\mu\right) = \sum_{j=1}^{k} \sum_{i \in C_j} ||x_i - \mu_j||^2$$

- And more (graphical models, features selection, active learning, control)

Source:"Introduction to Convex Optimization for Machine Learning," John Duchi, University of California, Berkeley.

Maximum vs. Minimum

$$f(x) = x^3 - 12x + 1$$

First derivative:

$$f'(x) = 3x^2 - 12$$

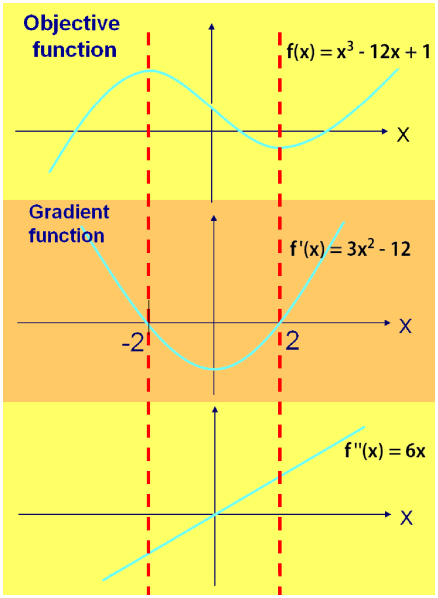
FOC:
f'(x=x*) = 0 at maximum and minimum.
These zero points are the roots!

Second derivative:

$$f''(x) = 6x$$

Gives the "rate of change of gradient"

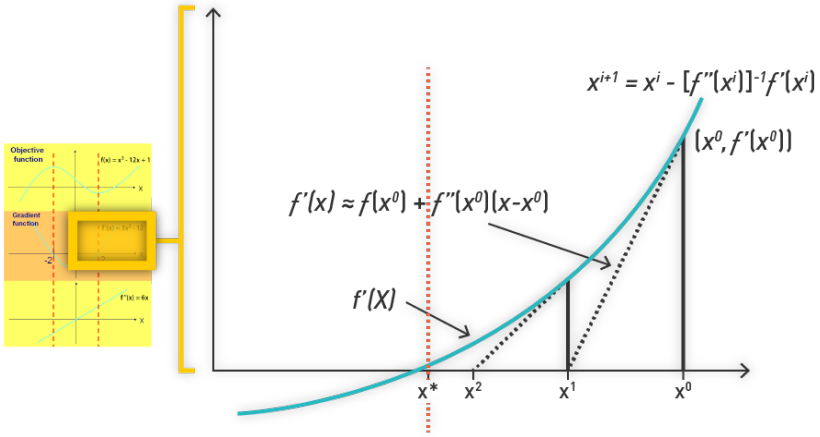
SOC:
If f''(x=x*) < 0, then maximum.
If f''(x=x*) > 0, then minimum.
If f''(x=x*) = 0, then ???



Finding the Optimum

- Finding the roots of the gradient function
  - Closed form
  - Newton-Raphson
  - Gradient descent
  - Bisection method
- Are you a root finder?

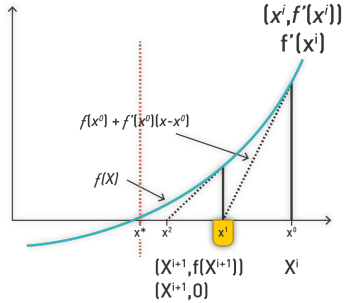
Newton-Raphson Method: Root Finding of Derivative/Gradient Function *f'(x)*



1. Initial guess: x^0 Letting $i = 0$ $x^{i+1} = x^0$.
2. Approximate *f*(*x*) by tangent at $(x^{i+1},\ f(x^{i+1}))$ # $(x^0,\ f(x^0))$ for the first iteration.
3. Find where $f_{\text{Tangent\_x0}} x = 0$, i.e.,$x^{i+1}$; better approx. of the root (*x**).
4. Repeat until convergence, i.e., *X* does not change.

Deriving Newton-Raphson Method

- Solving a nonlinear equation of the form $f(x)=0$.
- Generate a sequence of iterate x^{n-1} , x^{n} , x^{n+1} which hopefully converges to the solution x^* (the root of $f(x)$).



$$f(x) \approx f(x^0) + f'(x^0)(x - x^0) \qquad \forall \text{ surrounding } x^0$$

$$f(x^{i+1}) \approx f(x^i) + f'(x^i)(x^{i+1} - x^i) \qquad \forall \text{ surrounding } x^i$$

We desire a root (i.e., $f(x^{i+1}) = 0$)

$$0 = f(x^i) + f'(x^i)(x^{i+1} - x^i)$$

$$f'(x^i)(x^{i+1} - x^i) = -f(x^1)$$

Iteration function:

$$x^{i+1} = -\frac{f(x^i)}{f'(x^i)} + x^i$$

Sometimes written as:

$$x^{i+1} = x^i - \left[\frac{df}{dx}\left(x^i\right)\right]^{-1} f(x^i)$$

Gradient Descent (a Simpler Root Finder) in 1D

Newton-Raphson in 1D, iteration function:

$$x^{i+1} = x^i - \frac{f'(x^i)}{f''(x^i)}$$

$$x^{i+1} = x^i - \left[f''(x^i) \right]^{-1} f'(x^i)$$

- Calculating $f''(x)$, the Hessian H, and inverting it is complex, so simpler algorithms have been developed such as gradient descent.

Gradient Descent, iterate until X does not change:

$$x^{i+1} = x^i - a^i f'(x^i)$$

- How large should I step in the positive gradient direction (gradient ascent)?
  - Or in the negative gradient direction (gradient descent)?
  - Convergence criteria. E.g., decision vector X does not change that much.

Gradient: A Vector-Valued Function of Partial Derivatives

- In mathematics, the gradient is a generalization of the usual concept of derivative of a function in one dimension to a function in several dimensions.
- If $f(x_1, \ldots x_n)$ is a differentiable, scalar-valued function of standard Cartesian coordinates in Euclidean space, its gradient is the vector whose components are the *n* partial derivatives of *f*. It is thus a vector-valued function.
- Similarly to the usual derivative, the gradient represents the slope of the tangent of the graph of the function.
    - More precisely, the gradient points in the direction of the greatest rate of increase of the function, and its magnitude is the slope of the graph in that direction.
    - The components of the gradient in coordinates are the coefficients of the variables in the equation of the tangent space to the graph.
    - This characterizing property of the gradient allows it to be defined independently of a choice of coordinate system, as a vector field whose components in a coordinate system will transform when going from one coordinate system to another.

Source: http://en.wikipedia.org/wiki/Gradient.

Gradient: Nabla (∇)

- Nabla is the symbol (∇).
- ∇ = 0, FOC.
- ∇ points in the direction of greatest increase.
- The gradient at a specific point $x = x'$ is the vector whose elements are the respective partial derivatives evaluated at $X = x'$, so that

$$\nabla f(X = x') = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$$
$$\nabla (X = x') = 0; \quad \nabla f(X = x') = (0, 0, \ldots 0)$$

  at a candidate extremum (FOC).
- The significance of the gradient is that the (infinitesimal) change in x that maximizes the rate at which $f(X)$ increases is the change that is proportional to $\nabla f(X)$.

Source: http://en.wikipedia.org/wiki/Gradient.

Hessian: Matrix of Second Partial Derivatives of *f*, Jacobian vs. Hessian

In mathematics, the Hessian matrix or Hessian is a square matrix of second-order partial derivatives of a scalar-valued function, or scalar field. It describes the local curvature of a function of many variables. The Hessian matrix was developed in the 19th century by the German mathematician Ludwig Otto Hesse and later named after him. Hesse originally used the term "functional determinants."

Specifically, suppose $f : \mathbb{R}^n \to \mathbb{R}$ is a function taking as input a vector $x \in \mathbb{R}^n$ and outputting a scalar $f(x) \in \mathbb{R}$; if all second partial derivatives of f exist and are continuous over the domain of the function, then the Hessian matrix H of f is a square $n \times n$ matrix, usually defined and arranged as follows:

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

$$\nabla f(X = x') = \left(\frac{df}{dx_1}, \frac{df}{dx_2}, \ldots \frac{df}{dx_n} \right) = J(X = x')$$

Hessian: Matrix of Second Partial Derivatives of *f*, Jacobian vs. Hessian (cont.)

or, component-wise:

$$H_{i,j} = \frac{\partial^2 f}{\partial x_i \, \partial x_j}.$$

The determinant of the above matrix is also sometimes referred to as the Hessian.

The Hessian matrix can be considered related to the Jacobian matrix by $H(f)(x) = J(\nabla f)(x)$.

Hessian matrices are used in large-scale optimization problems within Newton-type methods because they are the coefficient of the quadratic term of a local Taylor expansion of a function. That is,

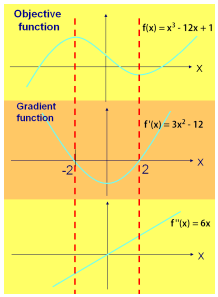
$$y = f(x + \Delta x) \approx f(x) + J(x)\Delta x + \frac{1}{2}\Delta x^T H(x)\Delta x$$

where *J* is the Jacobian matrix. The full Hessian matrix can be difficult to compute in practice; in such situations, quasi-Newton algorithms have been developed that use approximations to the Hessian. One of the most popular quasi-Newton algorithms is BFGS.

Source: https://en.wikipedia.org/wiki/Hessian_matrix.

Gradient Descent (a Simpler Root Finder) vs. Newton-Raphson

Given: Minimize $f(x)$. **Step 1**: Find the zeros of the gradient function $f'(x)$ using bisection method, Newton-Raphson, or gradient descent.



Newton-Raphson:

$$x^{i+1} = x^i - \left[f''\left(x^i\right) \right]^{-1} f'(x)$$ Univariate case

$$x^{i+1} = x^i - \left[H\left(x^i\right) \right]^{-1} J(x)$$ Multivariate case

Calculating $f''(x)$, the Hessian H in multi- variate case, and inverting it is complex so simpler algorithms have been developed such as gradient descent.

Gradient Descent: → learning rate: α

$$x^{i+1} = x^i - \alpha^i f'(x)$$ Univariate case

$$x^{i+1} = x^i - \alpha^i J(x)$$ Multivariate case

Determine the Learning Rate

- Fixed
- Decay over time
- Linesearch (one-dimensional line search)

Operational Issues: Quasi-Newton

- Calculating gradient and Hessian not very time consuming but calculating the inverse of *H* is!

Suppose that the objective *f* is a function of multiple arguments, $f(w_1, w_2, \ldots w_p)$. Let's bundle the parameters into a single vector, $\vec{w}$.

Then the Newton update is

$$\vec{w}_{n+1} = \vec{w}_n - H^{-1}(w_n)\nabla f(\vec{w}_n)$$

where ∇*f* is the gradient of *f*, its vector of partial derivatives $\left[\partial f / \partial w_1, \partial f / \partial w_2, \ldots \partial f / \partial w_p\right]$, and *H* is the Hessian of *f*, its matrix of second partial derivatives, $H_{ij} = \partial^2 f / \partial w_i \partial w_j$.

Calculating *H* and ∇*f* isn't usually very time consuming, but taking the inverse of *H* is, unless it happens to be a diagonal matrix. This leads to various quasi-Newton methods, which either approximate *H* by a diagonal matrix, or take a proper inverse of *H* only rarely (maybe just once), and then try to update an estimate of $H^{-1}(w_n)$ as w_n changes. (See section 8.3 in the textbook for more.)

Source: Hand, Mannila, Smyth, *Data Mining*, Section 8.3.

- In R, have a look at ?optim #method=BFGS :http://www.stat.cmu.edu/~cshalizi/350/2008/lectures/29/lecture-29.pdf.

Characterizing Extrema in Multi-Dims

- FOC:
  - Gradient function is zero (i.e., the objective function is at an extremum).
- Is $F(x^*)$ a minimum?
- SOC:
  - Can be established by checking if the Hessian is positive definite (second partial derivative $f''(x^*)$, i.e., $\partial^2 f \; / \; \partial x_i \, \partial j_i$ evaluated at x^*)
- A sufficient condition for an extreme point x^* (i.e., $F(x^*) = 0$) to be a minimum is to have a positive definite Hessian at x^* (i.e., has positive (nonzero) eigenvalues).

Maximum vs. Minimum

- Find the roots of the Gradient function using the Newton-Raphson algorithm or gradient descent.
- Use SOC to establish if extremum is a maximum or minimum.

$$f(x) = x^3 - 12x + 1$$

First derivative:

$$f'(x) = 3x^2 - 12$$

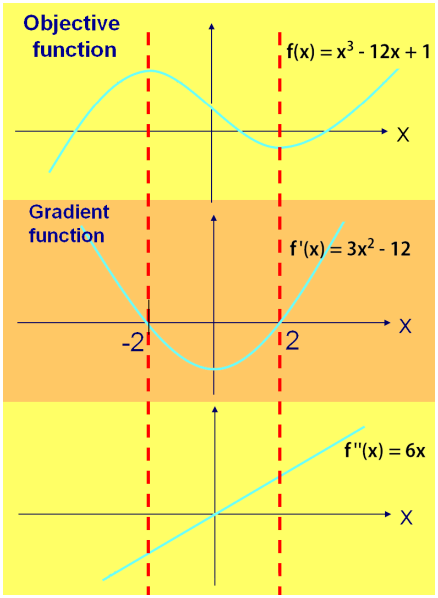
FOC:
$f'(x=x^*) = 0$ at maximum and minimum.
These zero points are the roots!

Second derivative:

$$f''(x) = 6x$$

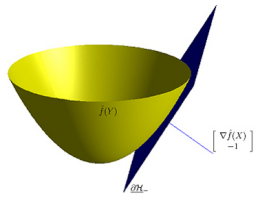
Gives the "rate of change of gradient"

SOC:
If $f''(x=x^*) < 0$, then maximum.
If $f''(x=x^*) > 0$, then minimum.
If $f''(x=x^*) = 0$, then ???

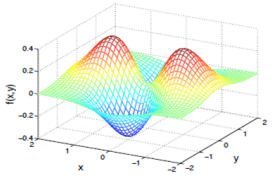


General Approach to Finding Extrema

- Well-behaved version spaces
  - Convex or concave function (±definiteness).
  - Algorithms seek a local extrema knowing that it will be global.
    - If f( ) is a concave function, then local maximum is a global maximum.
    - If f( ) is a convex function, then local minimum is a global minimum.
    - Newton-Raphson, gradient descent, conjugate gradient descent.



- Otherwise
  - We resort to local approximations.
    - Hill climbing
    - Simulated annealing
    - Commonly used in neural networks



Next, let's look at the world of convex optimization and how it relates to machine learning.

Overview

- In this section we will review convex optimization.
- Convex optimization is one of the most important pillars of math underlying machine learning.

Convex Minimization

- Convex minimization, a subfield of optimization, studies the problem of minimizing convex functions over convex sets.
- Convexity can make optimization in some sense "easier" than the general case; e.g., any local minimum must be a global minimum.
- We do *not* need to look at the SoC to determine minimum and maximum.

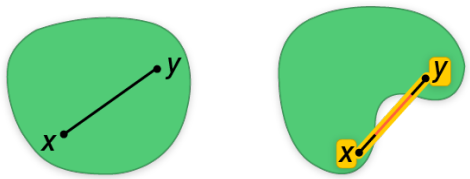
Given a real vector space $\boldsymbol{X}$ together with a convex, real-valued function

$f: \chi \to \mathbb{R}$

defined on a convex subset $\boldsymbol{\chi}$ of $\boldsymbol{X}$, the problem is to find any point x^* in $\boldsymbol{\chi}$ for which the number $f(x)$ is smallest, i.e., a point x^* such that $f(x^*)$ is smallest—i.e., a point x^* such that $f(x^*) \leq f(x)$ for all $x \in \chi$.

Convex Minimization Over Convex Sets

- In Euclidean space, an object is convex if for every pair of points within the object, every point on the straight line segment that joins them is also within the object.
- Intuitively, this means that the set is connected (so that you can pass between any two points without leaving the set) and has no dents in its perimeter.
- For example, a solid cube is convex, but anything that is hollow or has a dent in it, for example, a crescent shape, is not convex.



Convexity: Zero-Order Condition

A real-valued function is **convex** if

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta) f(y)$$

for all $x,\ y \in \mathbb{R}^n$ and all $0 \leq \theta \leq 1$.

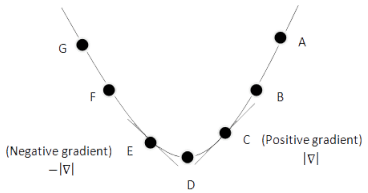
- Function is below the chord from *x* to *y*
- Show that all local minima are global minima

Convexity: First-Order Condition

A real-valued differentiable function is **convex** iff

$$f(y) \geq f(x) + \nabla f(y)^T \bullet (y - x)$$

for all $x, y \in \mathbb{R}^n$



- The function is globally above the tangent at *y*.

Source: https://www.cs.ubc.ca/~schmidtm/MLSS/convex.pdf

First-Order Condition of Convexity: Recap

3.1.3 First-order conditions

Suppose *f* is differentiable (i.e., its gradient ∇*f* exists at each point in **dom** *f*, which is open). Then *f* is convex if and only if **dom** *f* is convex and

$$f(y) \geq f(x) + \nabla f(y)^T (y - x)$$

holds for all *x*, *y* ∈ **dom** *f*.

Source: web.stanford.edu/~boyd

Convexity: First-order condition

A real-valued differentiable function is **convex** iff

$$f(y) \geq f(x) + \nabla f(y)^T \bullet (y - x)$$

for all *x*, *y* ∈ ℝ^*n*.

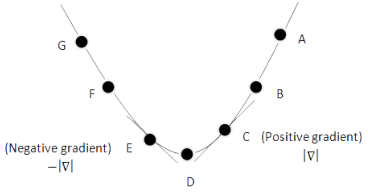
• The function is globally above the tangent at *y*.

Source: www.cs.ubc.ca/~schmidtm

Convex Optimization Problems: More Formally

Definition:
An optimization problem is *convex* if its objective is a convex function, the inequality constraints f_j are convex, and the equality constraints h_j are affine.

$$\begin{aligned} \underset{x}{\text{minimize}} & f_0(x) \quad \text{(Convex function)} \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad \text{(Convex sets)} \\ & h_j(x) = 0 \quad \text{(Affine)} \end{aligned}$$



First order condition of convexity:
Around a minimum, everywhere $f(y)=f(x)+\nabla F(x)(y-x)$ for it to be minimum if $\nabla F(x)$ is zero, otherwise we violate the condition.

Theorem:
$\nabla f(x)=0$ if and only if x is a global minimizer of $f(x)$.

Proof.
• $\nabla f(x)=0$. We have

$$f\left(y\right) \geq f\left(x\right) + \nabla f\left(x\right)^T \left(y - x\right) = f\left(x\right)$$

• $\nabla f(x)\neq 0$. There is a direction of descent.

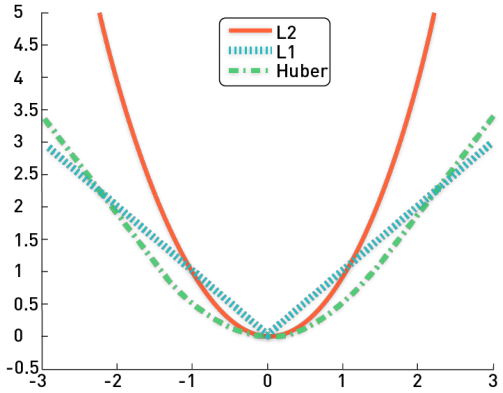
Convex Optimization in ML

- Many methods in machine learning are based on finding parameters that minimize some objective function. Very often, the objective function is a weighted sum of two terms:
  - A cost function and regularization term.
- In statistics terms, these are the (log-)likelihood and (log-)prior.
- If both of these components are convex, then their sum is also convex.
  - Loss functions are summed over examples so the sum of a convex functions is a convex function.

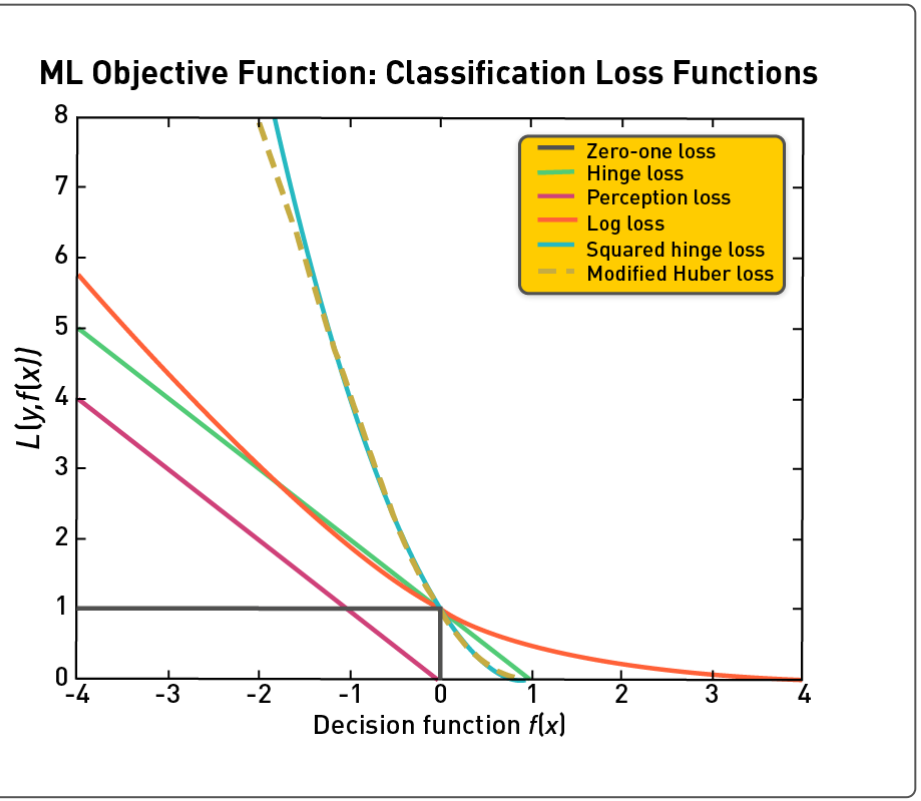
ML Objective Function: Regression Loss Function

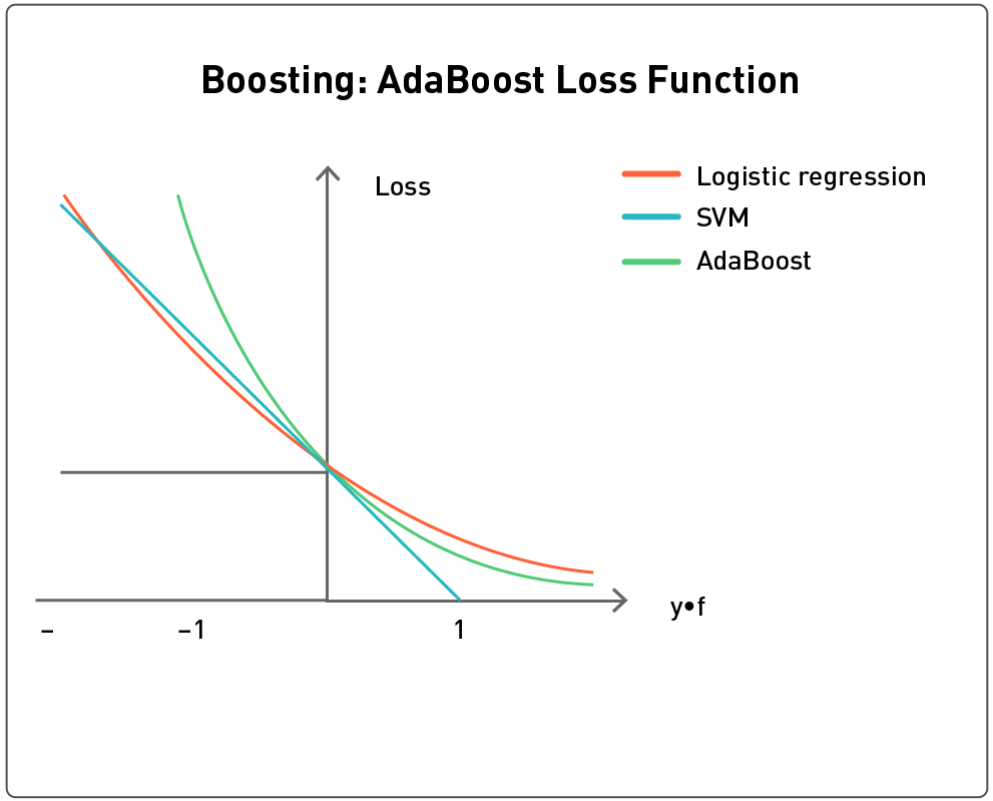
Model: $y = \theta^T x$

Goal: $\theta^* = \min_\theta \sum_{i=1}^{m} \left(\theta^T x_i - y_i\right)^2$



- Regression loss functions: All are convex.
- This is for one example in 1Dim.
- Loss: Combine over all training examples; sum of such convex functions is also convex.


ML Objective Function: Classification Loss Functions
Zero-one loss
Hinge loss
Perception loss
Log loss
Squared hinge loss
Modified Huber loss
L(y,f(x))
Decision function f(x)
8
7
6
5
4
3
2
1
0
-4
-3
-2
-1
0
1
2
3
4


Boosting: AdaBoost Loss Function
Loss
Logistic regression
SVM
AdaBoost
y•f
–
–1
1

Gradient Descent for LR Price~Size
Iteration 0

Eqn line: $y = aX + b$

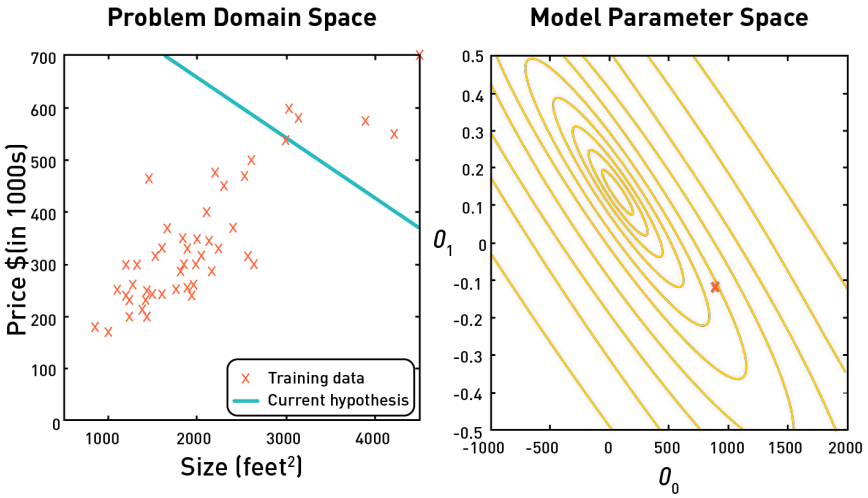
$Y = w_1 X + W_0$

(for fixed θ_0, θ_1, this is a function of x)

$J_q\left(W, X_1^t\right) = \frac{1}{m}\sum_{i=1}^{m}\left(W^T X_i - y_i\right)^2$

$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



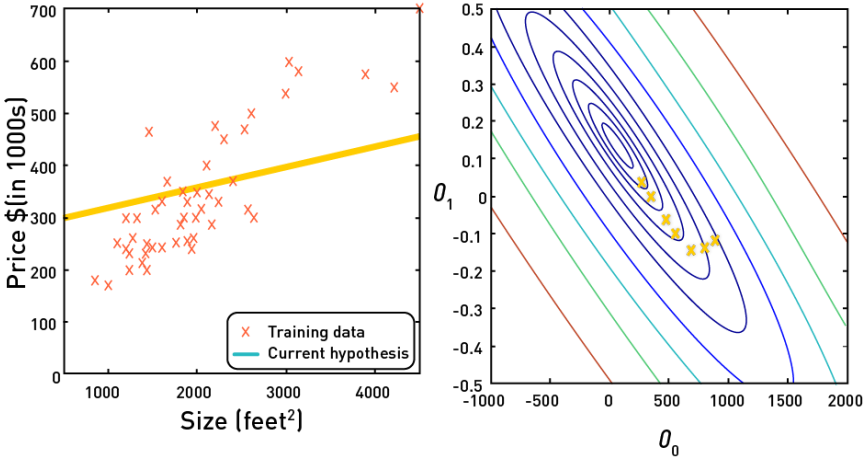
Gradient Descent for LR Price~Size Iteration 6

$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)

$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



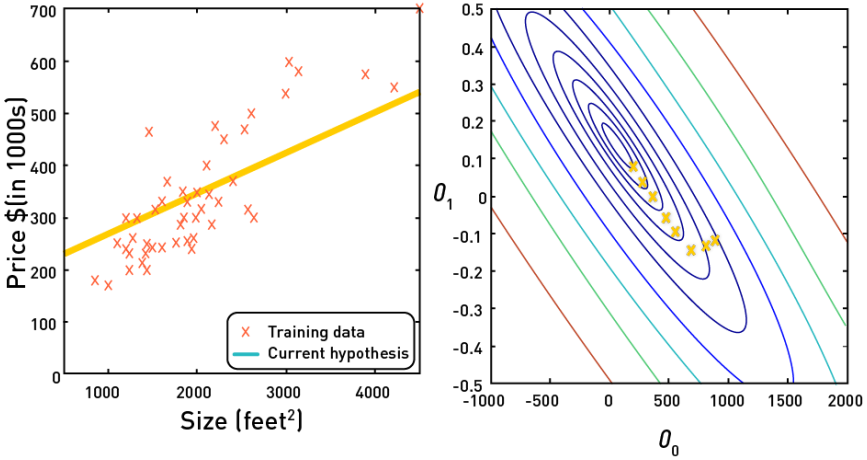
Gradient Descent for LR Price~Size Iteration 7

$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)

$J(\theta_0, \theta_1)$

(function of the parameters θ_0, θ_1)



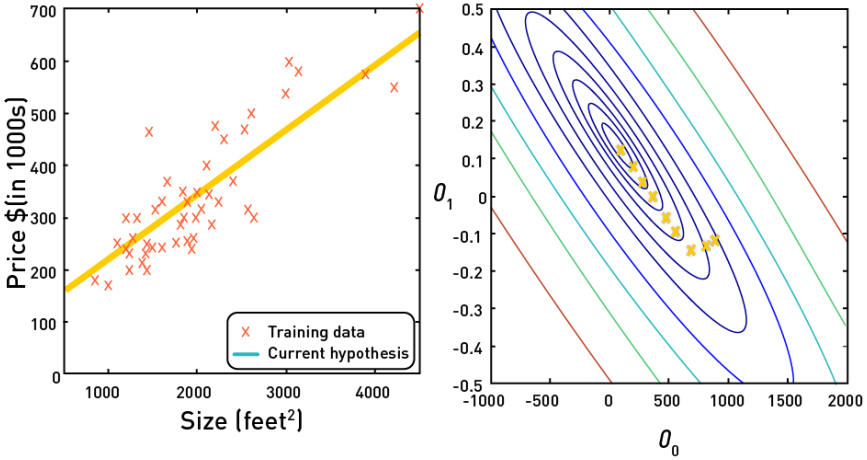
Gradient Descent for LR Price~Size Converged After Nine Steps

$h_\theta(x)$

(for fixed θ_0, θ_1, this is a function of x)

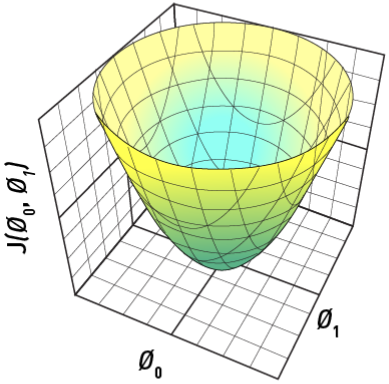
$J(\theta_0, \theta_1)$

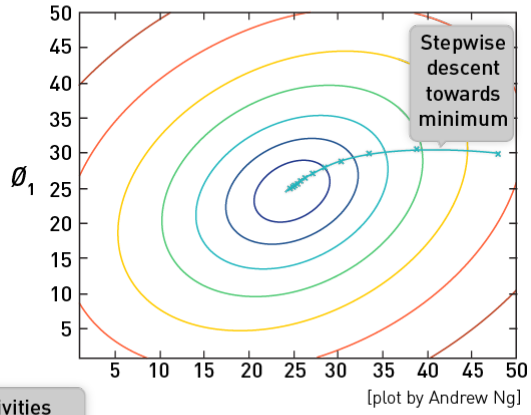
(function of the parameters θ_0, θ_1)



3D Plots and Contour Plots

$$J(\emptyset_0, \emptyset_1) = \frac{1}{2m} \sum_{i=1}^{m} ((\emptyset_0 + \emptyset_1 x^{(i)}) - y^{(i)})^2$$



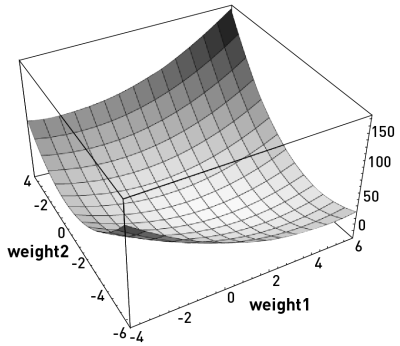


[plot by Andrew Ng]

Derivities work only for few parameters

$$\underset{\emptyset_0 \emptyset_1}{\text{minimize}}\ J(\emptyset_0, \emptyset_1) = \arg\min_{\emptyset} J(\emptyset)$$

Gradient Descent for OLS Version Space (Weights)



Error surface; each point corresponds to a different linear model (hypothesis).

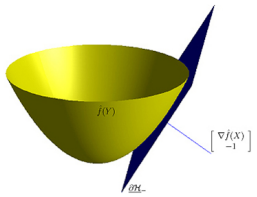
The vertical axis indicates the squared error for the training data set with regard to that weight vector.

Q: Will this surface change for different data sets?

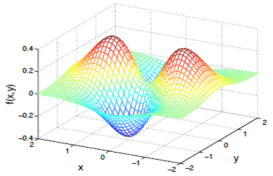
OLS with this objective has no local minima (convex as the Hessian, n x n matrix of second derivatives, if the objective function is positive definite); in this case n = 2 variables. Iterative versus closed-form solution.

General Approach to Finding Extrema

- Well-behaved version spaces
  - Convex or concave function (± definiteness)
  - Algorithms seek a local extrema knowing that it will be global
    - If f( ) is a concave function, then local maximum is a global maximum
    - If f( ) is a convex function, then local minimum is a global minimum
    - Newton-Raphson, gradient descent, conjugate gradient descent



- Otherwise …
  - We resort to local approximations
    - Hill climbing
    - Simulated annealing
    - Commonly used in neural networks



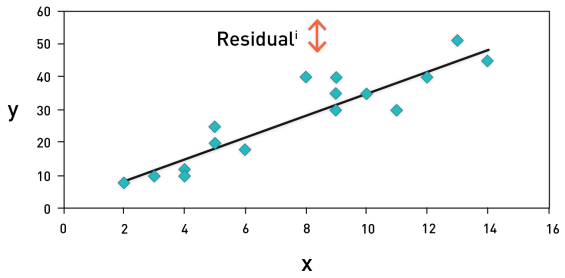
Summary

- Having a convex objective function in machine learning is very attractive, as we can leverage the FOC to determine a minimum—a minimum that will be a global minimum.
- We can use techniques such as gradient descent or Newton-Raphson to get us there.
- Let's leverage this to build some learning algorithms and run them on MapReduce frameworks.

Equation of a Line

$$\text{Residual}^i = \left(WX^i - y^i\right) \quad \Rightarrow \quad \text{Residual}^i = \left(WX^i - y^i\right)^2$$

Squared error loss gives us a twice differentiable function, and thus we can use convex optimization.



$y = mx + b$
$Y = w_1 x + w_0$

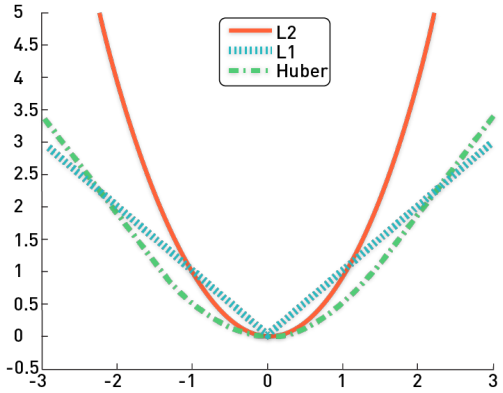
Where w_1,w_0 are model parameters

Each pair yields a different sum of squares error

ML Objective Function: Regression Loss Function

Model: $y = \theta^T x$

Goal: $\theta^* = \min_\theta \sum_{i=1}^{m} \left(\theta^T x_i - y_i\right)^2$



- Regression loss functions: All are convex.
- This is for one example in 1Dim.
- Loss: Combine over all training examples; sum of such convex functions is also convex.

Convex Operation, FOC

- Convex optimization tells us that the first-order condition $\nabla F(X) = 0$ at the global minimum.
- We are going to exploit that here to get a closed-form solution for linear regression.

Closed-Form Solution to OLS

SSE:

$$S(\theta) = \sum_i \left[y(i) - \sum_j \theta_j x_{ij} \right]^2$$

$$= \sum_i e_i^2$$

$$= \underline{e}' \underline{e}$$

$$= \left(y - X\theta\right)' (y - X\theta)$$

$$S(\theta) = \sum e^2 = e'e$$

Minimize objective function and find root vector of the gradient function:

y = N × 1 vector of target values

X = N × (p + 1) vector of input values

θ = (p + 1) × 1 vector of parameter values

where $\underline{e} = y - X\theta$

$$= \left(y - X\theta\right)' (y - X\theta)$$

$$= y'y - \theta' X' y - y' X\theta + \theta' X' X\theta$$

$$= y'y - 2\theta' X' y + \theta' X' X\theta$$

- Taking derivative of $S(\theta)$ with respect to the components of θ gives

$$dS / d\theta = -2X'y + 2X'X\theta$$

- Set this to 0 to find the extremum (minimum) of S as a function of θ.

Normal Equations and Solving for θ

Set to 0 to find the extremum (minimum) of S as a function of θ ...

$\Rightarrow \quad -2X^{'}y + 2X^{'}X\theta = 0$

$\Rightarrow \quad X^{'}X\theta = X^{'}y$ (known in statistics as the normal equations)

Letting $X^{'}X = C$, and $X^{'}Y = b$,

We have $C\theta = b$, i.e., a set of linear equations

We could solve this directly, e.g., by matrix inversion:

$\theta = C^{-1}b = \left(X^{'}X\right)^{-1}X^{'}y$ Closed-form solution to OLS

Closed-Form Solution to OLS

- To minimize *J* (aka RSS), we set its derivatives to zero and obtain the normal equations:
  - $X^T X W = X^T Y$
  - Thus the value of *W* that minimizes *J*(*W*) is given in closed form

$$
\begin{aligned}
\nabla J_{W_j}(W) = \tfrac{\partial}{\partial W_j} J(W) &= \tfrac{\partial}{\partial W_j}\left(\tfrac{1}{2}\left(f_W(x)-y\right)^2\right) \\
&= 2 * \tfrac{1}{2}\left(f_W(x)-y\right)\tfrac{\partial}{\partial W_j}\left(f_W(x)-y\right) \\
&= \left(f_W(x)-y\right)\tfrac{\partial}{\partial W_j}\left(\left(\sum_{i=0}^{n} w_i x_i\right)-y\right) \\
&\quad \left(f_W(x)-y\right)x_j \quad \text{for each j in 1:n} \\
&\quad (XW-Y)^T X \quad \text{overall and in terms of data} \\
&= X^T XW - X^T Y = 0 \\
&\quad X^T XW = X^T Y \quad \text{Normal Equations} \\
&\quad \mathrm{W} = \left(X^T X\right)^{-1} X^T Y
\end{aligned}
$$

- For a full derivation, see: http://www.stanford.edu/class/cs229/notes/cs229-notes1.pdf

OLS on a Single Computer (in R)

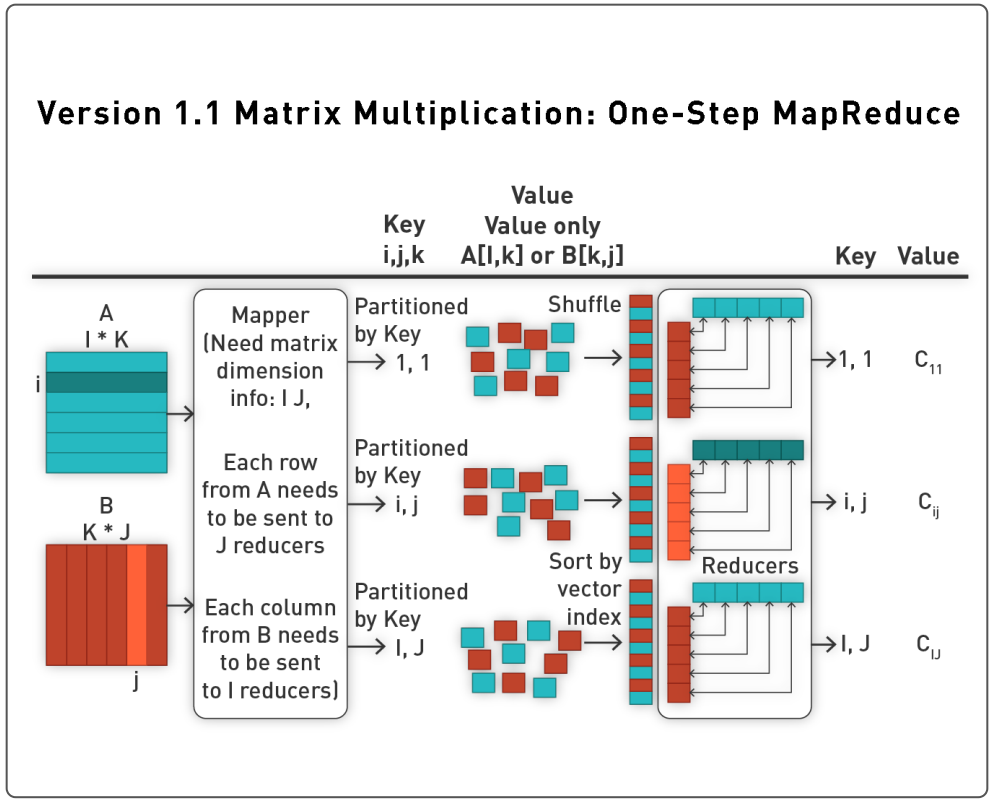
```
example.learnLSUsingClosedFormSolution = function() {
  dataEx1 = matrix(
      c(
   1,2,  # example 1
   2,3,  # example 2
   3,7
   4,8
   5,9 ),
   byrow=TRUE,
   ncol = 2)



  colnames(dataEx1)=c("time","temperature")

  designMatrix=as.matrix(dataEx1[,1]) #input variable data
  X=designMatrix=cbind(1, designMatrix) #append a constant 1 for
bias term

  y=targetValues=as.matrix(dataEx1[,2]);
  numVariables=ncol(designMatrix);
  w=rep(-2, numVariables);  #initialize weight vector
  library(MASS)  #makes ginv() the inverse of a matrix available
  w = ginv(t(X) %*% X) %*% t(X) %*% y;
  print(paste("OLS: Closed From Weight is ", w));
```

$$\theta = A^{-1}b = \left(X'X\right)^{-1}X'y$$

Closed-Form Solution

- This is the closed-form solution to linear regression and very amenable to a programmatic solution on a single-node machine.
- Can we parallelize this approach to learning a linear regression? Yes, we can can. We will look at this next.

  $\theta = A^{-1}b = \left(X'X\right)^{-1}X'y$
- Distributed closed form of LR


Version 1.1 Matrix Multiplication: One-Step MapReduce
Key
i,j,k
Value
Value only
A[I,k] or B[k,j]
Key
Value
A
I * K
i
B
K * J
j
Mapper
(Need matrix dimension info: I J,
Each row from A needs to be sent to J reducers
Each column from B needs to be sent to I reducers)
Partitioned by Key
1, 1
Partitioned by Key
i, j
Partitioned by Key
I, J
Shuffle
Sort by vector index
Reducers
1, 1
C11
i, j
Cij
I, J
CIJ

OLS on Multiple Computers: Four Map Reduce Jobs (Two for X T X,Two for X T y →) + Local Comp.

Model: y= θ T x

Goal: θ * = min θ ∑ i=1 m (θ T x i – y i) 2

Solution: Given *m* examples— (x 1 , y 1),(x 2 , y 2),...(x m , y m) —we write a matrix *X* with x 1 ,..., x m as rows and row vector Y=(y 1 , y 2 ,... y m). Then the solution is

θ*= (X T X) –1 X T y → .

- Parallel computation using matrices:

A= X T X	A= ∑ i=1 m (x i x i T)	One or two Map Reduce Jobs
b= X T y →	b= ∑ i=1 m (x i y i)	One or two Map Reduce Jobs

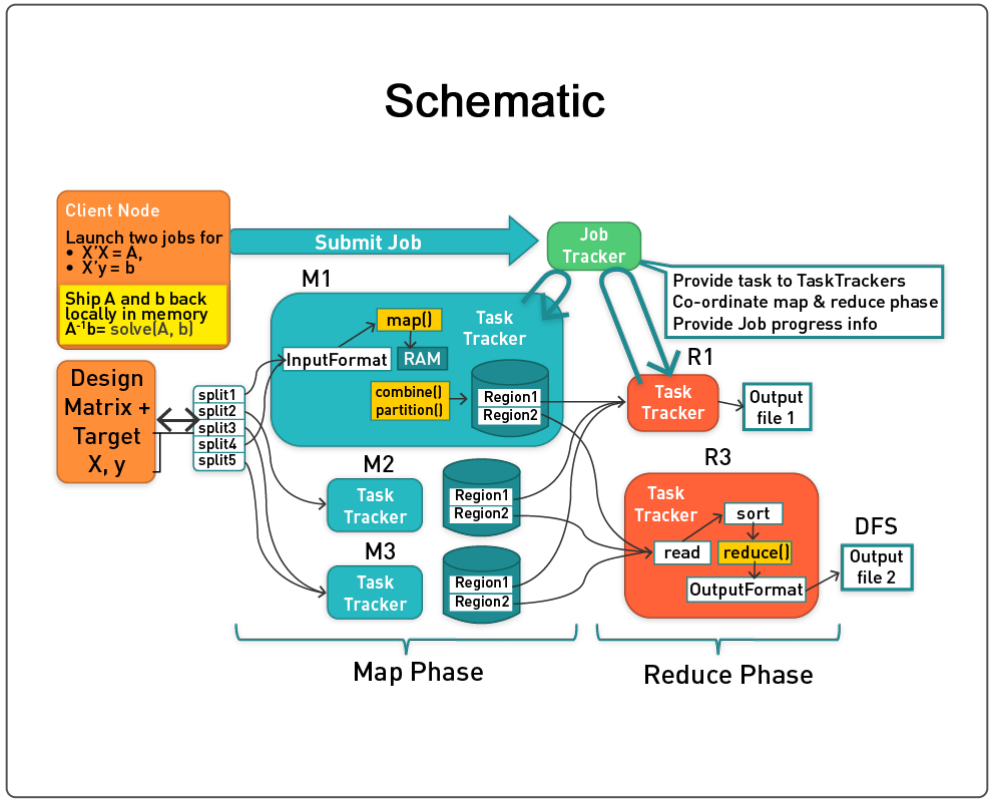
- Then ship ***A*** and ***b*** back to the master node.
- Calculate the inverse of the covariance matrix.

MapReduce Matrix Multiplication

θ*= (X T X) −1 X T y → .

A= X T X	A= ∑ i=1 m (x i x i T)	1 or 2 Map Reduce Jobs
b= X T y →	b= ∑ i=1 m (x i y i)	1 or 2 Map Reduce Jobs

- Then ship ***A*** and ***b*** back to the master node
- Calculate the Inverse of the covariance matrix
- Then multiply A −1 b
- In code it looks like this: `theta=solve(A,b)`


Schematic
Client Node
Launch two jobs for
• X'X = A,
• X'y = b
Ship A and b back locally in memory
A⁻¹b= solve(A, b)
Submit Job
Job Tracker
Provide task to TaskTrackers
Co-ordinate map & reduce phase
Provide Job progress info
Design Matrix + Target X, y
split1
split2
split3
split4
split5
M1
map()
Task Tracker
InputFormat
RAM
combine()
partition()
Region1
Region2
M2
Task Tracker
Region1
Region2
M3
Task Tracker
Region1
Region2
R1
Task Tracker
Output file 1
R3
Task Tracker
sort
read
reduce()
OutputFormat
DFS
Output file 2
Map Phase
Reduce Phase

Conclusion

- That is our first version of a linear regression.
- I hope you are excited about being able to distribute one of the workhorse algorithms of machine learning that forms the basis of many fields such as statistics, social sciences, and econometrics.
- But wait—there is more!
- This is just one way to distribute linear regression. We will address that next.

Algorithm Time Complexity Analysis

$$\theta = A^{-1}b = \left(X'X\right)^{-1}X'y$$

Single:

As an example of our running-time analysis, for single-core LWLR:

- We have to compute $A = \sum_{i=1}^{m} w_i(x_i x_i^T)$, which gives us the mn^2.
- This matrix must be inverted for n^3.
- Also, the reduce step incurs a covariance matrix communication cost of n^2.

Multi:

- Split calculation of ***A*** over worker.
- Split A^{-1} over cores.
- Communicate ***A*** back to master.

LWLR O(Calculation of A + Inversion of A):

$$O\left(mn^2 + n^3\right) \qquad O\left(\frac{mn^2}{P} + \frac{n^3}{P} + n^2\log(P)\right)$$

- *m* training examples
- *n* input variables
- *P* cores

- Locally Weighted Linear Regression (LWLR)
- Linear regression is just a degenerated LWLR where all weights are equal to 1.

Source: stanford.edu/people/ang... Complexity: wikipedia.org...

Evaluation Data Sets

Conducted an extensive series of experiments to compare the speed-up on data sets of various sizes (Table 2)

- On eight commonly used machine learning data sets from the UCI Machine Learning repository
- Two other ones from an anonymous research group (helicopter control and sensor data)
- Note that not all the experiments make sense from an output view—regression on categorical data—but our purpose was to test speed-up, so we ran every algorithm over all the data

Table 2. Data Sets: Size and Description

Data Sets	Samples (m)	Features (n)
Adult	30,162	14
Helicopter Control	44,170	21
Corel Image Features	68,040	32
IPUMS Census	88,443	61
Synthetic Time Series	100,001	10
Census Income	199,523	40
ACIP Sensor	229,564	8
KDD Cup 99	494,021	41
Forest Cover Type	581,012	55
1990 U.S. Census	2,458,285	68

Source: http://www.cs.stanford.edu/people/ang/papers/nips06-mapreducemulticore.pdf

Two Cores → 2x Speed-Up

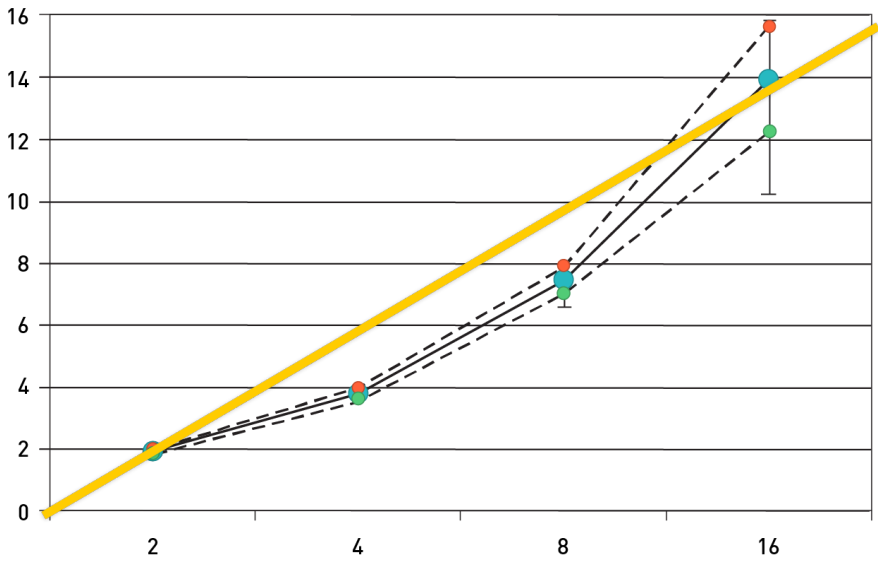
Table 3. Speed-Ups Achieved on a Dual-Core Processor, w/o Load Time.
Numbers reported are dual-core time/single-core time. Super linear speed-up sometimes occurs due to a reduction in processor idle time with multiple threads.

Data Sets	lwlr
Adult	1.922
Helicopter	1.93
Corel Image	1.96
IPUMS	1.963
Synthetic	1.909
Census Income	1.975
Sensor	1.927
KDD	1.969
Cover Type	1.961
Census	2.327
Avg.	1.985

Dual core versus single core

Scaling Out: Linear Scaling

- Linear scale (scale out by adding machines)
- Multicore simulator over the *sensor data set*
    - LR: 16 cores 14x, 32 cores 29.5x, 64 cores 53x



Conclusion

- You have just seen how one of the workhorse algorithms of machine learning, linear regression, scales out.
- We will discuss shortly one of the limitations of this approach.
- In addition, I will present an alternative approach to distributing linear regression that overcomes these limitations.

Overview

- Previously we have seen how to learn a linear regression model using distributed closed-form solution.
- This distribution solution runs into minor issues when we try to scale.

Limitations of the Closed-Form OLS: Solving for the θs

- Problem is equivalent to inverting $X'X$ matrix.
  - Inverse does not exist if matrix is not of full rank.
    - E.g., if one column is a linear combination of another (collinearity).
    - Note that $X'X$ is closely related to the covariance of the X data.
      - So we are in trouble if two or more variables are perfectly correlated.
    - Numerical problems can also occur if variables are almost collinear.
- Equivalent to solving a system of *p* linear equations.
  - Many good numerical methods for doing this, e.g., Gaussian elimination, LU decomposition, etc.
  - These are numerically more stable than direct inversion.
- Matrix inversion is not easily parallelized.

Alternative Ways of Solving OLS

- Closed-form solution
  - In this method, we will minimize RSS by explicitly taking its derivatives with respect to the β_j's (sometimes written as W, the weight vector) and setting them to zero.
  - Do this via calculus with matrices.

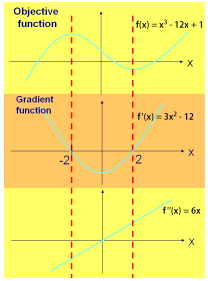
$$\theta^* = \left(X^T X\right)^{-1} X^T \vec{y}$$

- Gradient descent gives another way of minimizing RSS.
- Bayesian approach.
- Quadratic programming.
- Others.

Linear Regression via Gradient Descent

Given: Minimize *f(x)*. $J_q\left(W, X_1^m\right) =$ Minimize $\sum_{i=1}^{m}\left(W^T X_i - y_i\right)^2$

Step 1: Find the zeros of the gradient function *f'(x)* using bisection method, Newton-Raphson, or gradient descent.



RSS = Variance of ε

$$0 = \frac{\partial \sum \hat{\varepsilon}_i^2}{\partial W} = \frac{\partial\left(\sum_{j=1}^{n}\left(X_j W - y_i\right)^2\right)}{\partial W}$$

Gradient vector of partial derivatives

$$\nabla J(W) = \left(\sum_{j=1}^{n}\left(X_j W - y_i\right) X_j\right)$$

Pull model closer to examples with biggest residual.

$$W^{t+1} = W^t - \alpha^i\left(\sum_{j=1}^{n}\left(X_j W^t - y_i\right) X_j\right)$$

Gradient descent

For another derivation see: http://cs229.stanford.edu/notes/cs229-notes1.pdf.

OLS Using Gradient Descent

$J_q\left(W, X_1^m\right) =$ Minimize $\sum_{i=1}^{m}\left(W^T X_i - y_i\right)^2$ OLS objective function with decision variables W

- Initialize W = vector or zeros.
- Repeat until convergence.

$W^{t+1} = W^t - \alpha^i\left(\sum_{j=1}^{n}\left(X_j W^t - y_i\right)X_j\right)$ OLS batch update rule

- End repeat.

True gradient is approximated by the gradient of the cost function only evaluated at all examples; adjust parameters proportional to this approximate gradient.

Intuitively, drag weight vector closer to the incorrectly predicted examples.

Batch vs. Stochastic

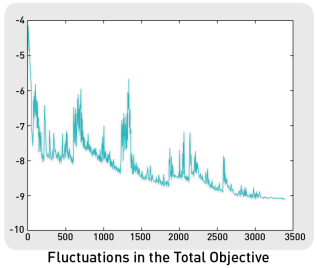
- That was batch-based descent.
- Stochastic gradient descent vs. batch.

OLS Using Gradient Descent

Stochastic gradient descent

$\nabla J_{wj}(W_t)$

Partial derivative WRT to variable w_j of error function $J(W)$ at point W_t



Fluctuations in the Total Objective

Stochastic update (after each example)

Let *W* = (0, 0,...)

Repeat

For *j* in 0...*n* #each variable

For *i* in 1...*m* #each example

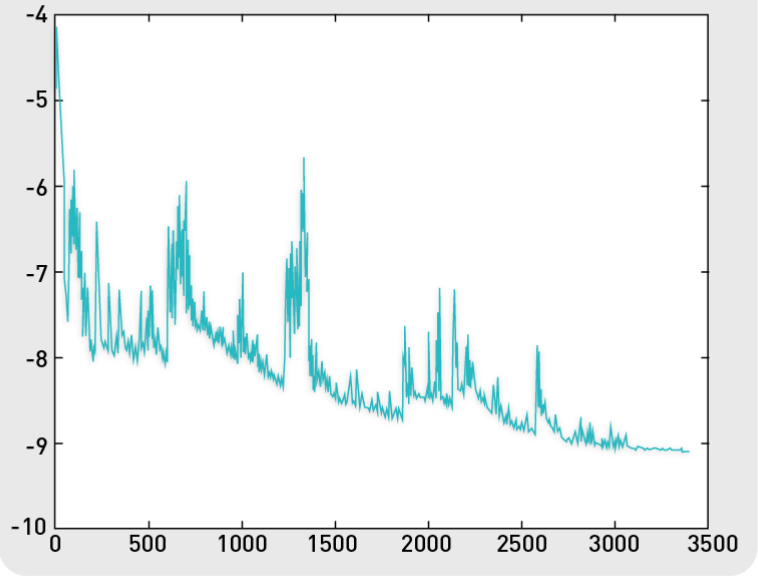
$$W^{t+1} = W^t - \alpha^t \left(\sum_{j=1}^{n} \left(X_j W^t - y_i \right) X_j \right)$$

Until convergence (i.e., no big changes in *W* or error)

Stochastic Gradient Descent vs. Batch

- Stochastic gradient descent can start making progress right away and continues to make progress with each example it looks at.
- Often, stochastic gradient descent gets W "close" to the minimum much faster than batch gradient descent.
    - Note, however, that it may never "converge" to the minimum, and the parameters W will keep oscillating around the minimum of J(W); but in practice most of the values near the minimum will be reasonably good approximations to the true minimum.
- For these reasons, particularly when the training set is large, stochastic gradient descent is often preferred over batch gradient descent.

Stochastic Updates -> Fluctuations in the Total Objective



Fluctuations in the Total Objective

Two Approaches

- Two gradient-descent approaches to linear regression
  - Batch gradient descent
  - Stochastic gradient descent based
- Are both amenable to parallelization? And if so, which do you expect to perform better?

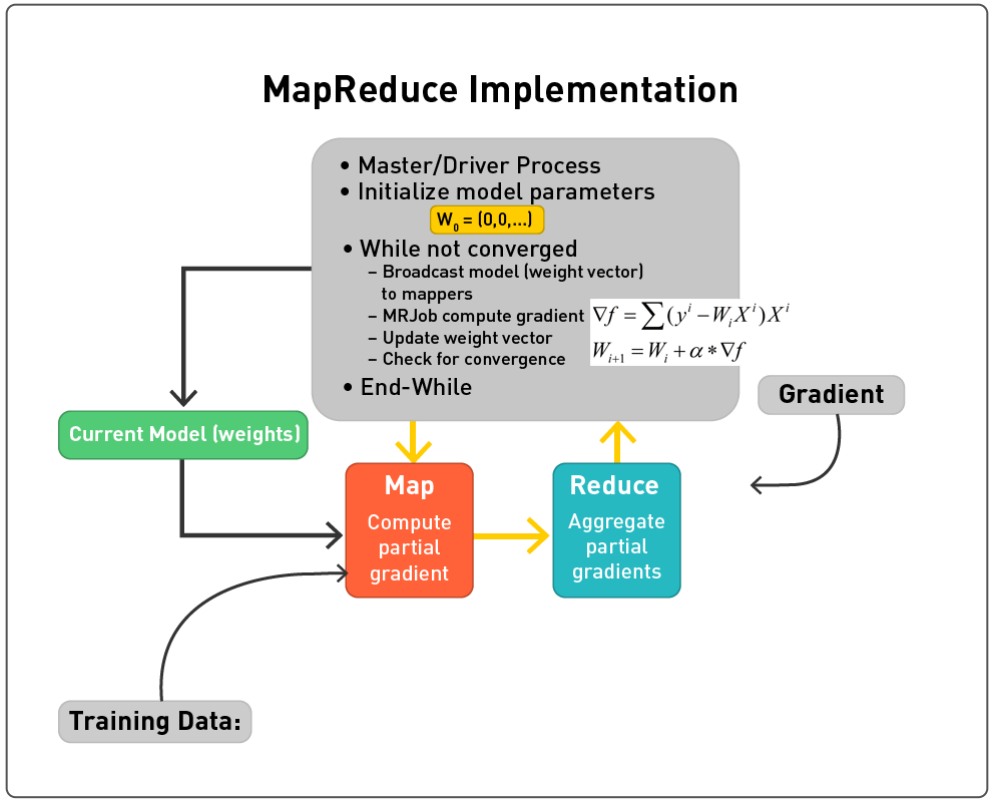
Stochastic Gradient Descent: Not Easy in MapReduce

- Stochastic gradient descent requires the weight vector to be updated after each example (or minibatch of examples).
- "Some implementations of machine-learning algorithms, such as ICA, are commonly done with stochastic gradient ascent, which poses a challenge to parallelization. The problem is that in every step of gradient ascent, the algorithm updates a common set of parameters."

Some implementations of machine-learning algorithms, such as ICA, are commonly done with stochastic gradient ascent, which poses a challenge to parallelization. The problem is that in every step of gradient ascent, the algorithm updates a common set of parameters (e.g., the unmixing *W* matrix in ICA). When one gradient ascent step (involving one training sample) is updating *W*, it has to lock down this matrix, read it, compute the gradient, update *W*, and finally release the lock. This "lock-release" block creates a bottleneck for parallelization; thus, instead of stochastic gradient ascent, our algorithms above were implemented using batch gradient ascent.

OLS via Distributed Gradient Descent

- Master/Driver process.
- Initialize model parameters $W_0 = \left(0, 0, \ \ldots \ \right)$.
- While not converged:
  - Broadcast model (weight vector) to mappers.
  - MRJob compute gradient $\nabla f = \ \Sigma \left(y^i - W_i X^i\right) X^i$.
  - Update weight vector $W_{i+1} = W_i + \alpha * \nabla f$.
  - Check for convergence.
- End-While.


MapReduce Implementation
• Master/Driver Process
• Initialize model parameters
W_0 = [0,0,...]
• While not converged
– Broadcast model (weight vector) to mappers
– MRJob compute gradient
– Update weight vector
– Check for convergence
$\nabla f = \sum (y^i - W_i X^i) X^i$
$W_{i+1} = W_i + \alpha * \nabla f$
• End-While
Gradient
Current Model (weights)
Map
Compute partial gradient
Reduce
Aggregate partial gradients
Training Data:

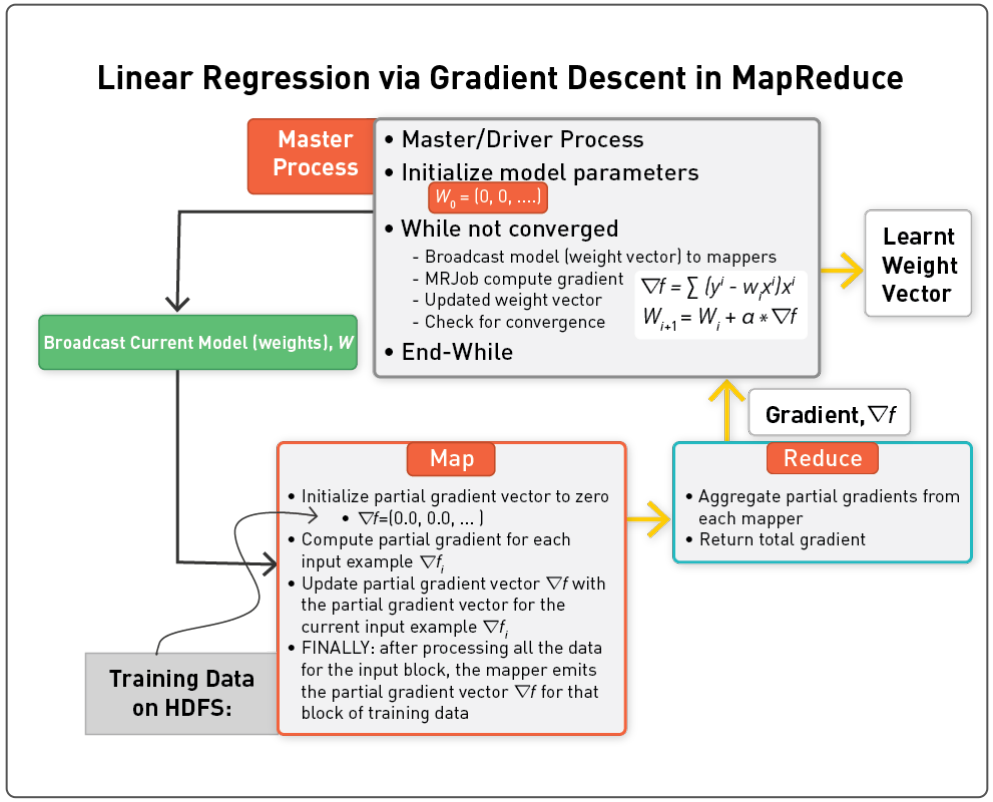
OLS via Distributed Gradient Descent: Mapper and Reducer

- Master/Driver process.
- Initialize model parameters, *W*= vector of zeros: $W_0 = \left(0, 0, \ \ldots \ \right)$.

- While not converged:
  - Broadcast model (i.e., weight vector) to the worker nodes.
  - Mapper (MANY mappers).
    - Compute partial gradient for each training example.
    - Combine in memory $\nabla f = \ \Sigma \left(y^i - W_i X^i\right) X^i$.
    - Finally yield the partial gradient.
  - Reducer (single Reducer).
    - Aggregate partial gradients.
    - Yield full gradient $\nabla f = \ \Sigma \left(y^i - W_i X^i\right) X^i$.
  - Update weight vector $W_{i+1} = W_i + \alpha * \nabla f$.
  - Check for convergence.
- End-While.

Divide and Conquer

- Can we use a combiner? Yes!
  - In-memory combiner.
  - So use mapper final.
- How many reducers?
  - We can have one (should be sufficient).
  - Or many (in the case of many, the master has to aggregate the partial reducer aggregates).


Linear Regression via Gradient Descent in MapReduce
Master Process
• Master/Driver Process
• Initialize model parameters
W_0 = (0, 0, ...)
• While not converged
- Broadcast model (weight vector) to mappers
- MRJob compute gradient
- Updated weight vector
- Check for convergence
$\nabla f = \sum (y^i - w_i x^i) x^i$
$W_{i+1} = W_i + \alpha * \nabla f$
• End-While
Learnt Weight Vector
Broadcast Current Model (weights), W
Gradient, ∇f
Map
• Initialize partial gradient vector to zero
• ∇f=(0.0, 0.0, ...)
• Compute partial gradient for each input example ∇f_i
• Update partial gradient vector ∇f with the partial gradient vector for the current input example ∇f_i
• FINALLY: after processing all the data for the input block, the mapper emits the partial gradient vector ∇f for that block of training data
Reduce
• Aggregate partial gradients from each mapper
• Return total gradient
Training Data on HDFS:

Linear Regression Summary

- Closed form (with limitations)
- Gradient descent

Distributed Closed Form vs. Gradient Descent

- Distributed closed form solution is not that scalable after all, as we need to compute the inverse of the covariance matrix $X^T X$.
- Gradient descent approach to linear regression is more scalable.

Summary: Gradient Descent Pattern

- Other machine-learning algorithms follow this pattern for distributed gradient descent.
  - Support vector machines
  - Logistic regression
- Not surprisingly, many learning algorithms do not follow this pattern.
  - E.g., stochastic gradient descent.
  - Nonconvex problems.
  - Perceptron learning is a sequential learner, but we will see how to get around this over the course of this lecture series.