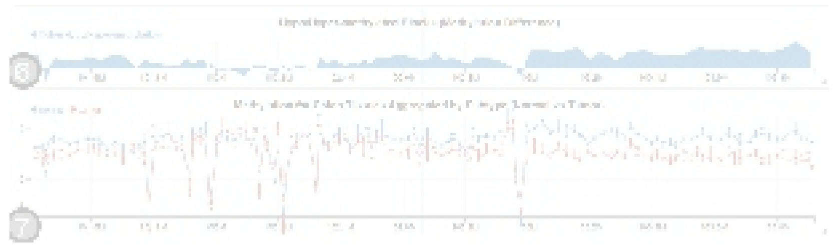


Large Scale Learning

Héctor Corrada Bravo

University of Maryland, College Park, USA

CMSC 643: 2017-11-28

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js

Large-scale Learning

Analyses we have done in class are for in-memory data:

- Datasets can be loaded onto memory of a single computing node.

Database systems can execute SQL queries, which can be used for efficient learning of some models (e.g. decision trees) over data on disk

- Operations are usually performed by a single computing node.

Large-scale Learning

In the 90s database systems that operate over multiple computing nodes became available

Basis of the first generation of large data warehousing.

In the last decade, systems that manipulate data over multiple nodes have become standard.

Large-scale Learning

Basic observation

for very large datasets, many of the operations for aggregation and summarization, which also form the basis of many learning methods, can be parallelized.

Large-scale Learning

For example:

- partition observations and perform transformation on each partition as a parallel process
- partition variables and perform transformation on each variable as a parallel process
- for summarization (`group_by` and `summarize`), partition observations based on `group_by` expression, perform `summarize` on each partition.

Large-scale Learning

Efficiency of implementation of this type of parallelism depends on underlying architecture:

Shared memory vs. Shared storage vs. Shared nothing

For massive datasets, shared nothing is usually preferred since fault tolerance is perhaps the most important consideration.

Map-reduce

Map-Reduce is an implementation idea for a shared nothing architecture.

It is based on:

- distributed storage
- data proximity (perform operaations on data that is physically close)
- fault tolerance.

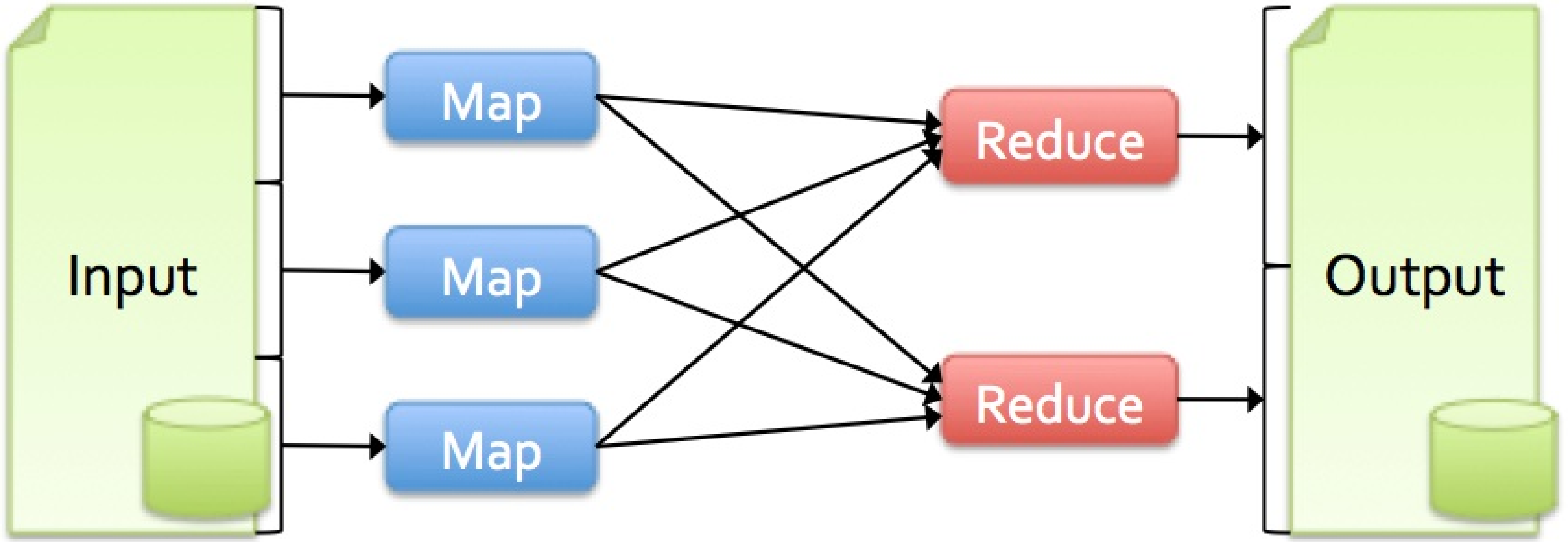
Map-reduce

Basic computation paradigm is based on two operations:

- reduce: perform operation on subset of observations in parallel
- map: decide which parallel process (node) should operate on each observation

Map-reduce

The fundamental operations that we have learned very well in this class are nicely represented in this framework: `group_by` clause corresponds to `map`, and `summarize` function corresponds to `reduce`.



Map-reduce

Map-reduce is most efficient when computations are organized in an acyclic graph.

Data is moved from stable storage to computing process and the result moved to stable storage without much concern for operation ordering.

This architecture provides runtime benefits due to flexible resource allocation and strong failure recovery.

Existing implementations of Map-reduce systems do not support interactive use, or workflows that are hard to represent as acyclic graphs.

Spark

Recent system based on the general map-reduce framework

Designed for ultra-fast data analysis.

Provides efficient support for interactive analysis (the kind we do in Jupyter)

Designed to support iterative workflows needed by many Machine Learning algorithms.

Spark

The basic data abstraction in Spark is the resilient distributed dataset (RDD).

Applications keep working sets of data in memory and support iterative algorithms and interactive workflows.

Spark

RDDs are

(1) inmutable and partitioned collections of objects,
(2) created by parallel transformations on data in stable storage (e.g., map, filter, group_by, join, ...)
(3) cached for efficient reuse
(4) operated upon by actions defeind on RDDs (count, reduce, collect, save, ...)

Spark

Fault Tolerance

RDDs maintain lineage, so partitions can be reconstructed upon failure.

Spark

The components of a SPARK workflow

Transformations: Define new RDDs

https://spark.apache.org/docs/latest/programming-guide.html#transformations

Actions: Return results to driver program

https://spark.apache.org/docs/latest/programming-guide.html#actions

Spark

Spark was designed first for Java with an interactive shell based on Scala. It has strong support in Python and increasing support in R SparkR.

- Spark programming guide: https://spark.apache.org/docs/latest/programming-guide.html
- More info on python API: https://spark.apache.org/docs/0.9.1/python-programming-guide.html

Stochastic Gradient Descent

Other learning methods we have seen, like regression and SVMs (or even PCA), are **optimization problems**

We can design gradient-descent based optimization algorithms that process data efficiently.

We will use linear regression as a case study of how this insight would work.

Case Study

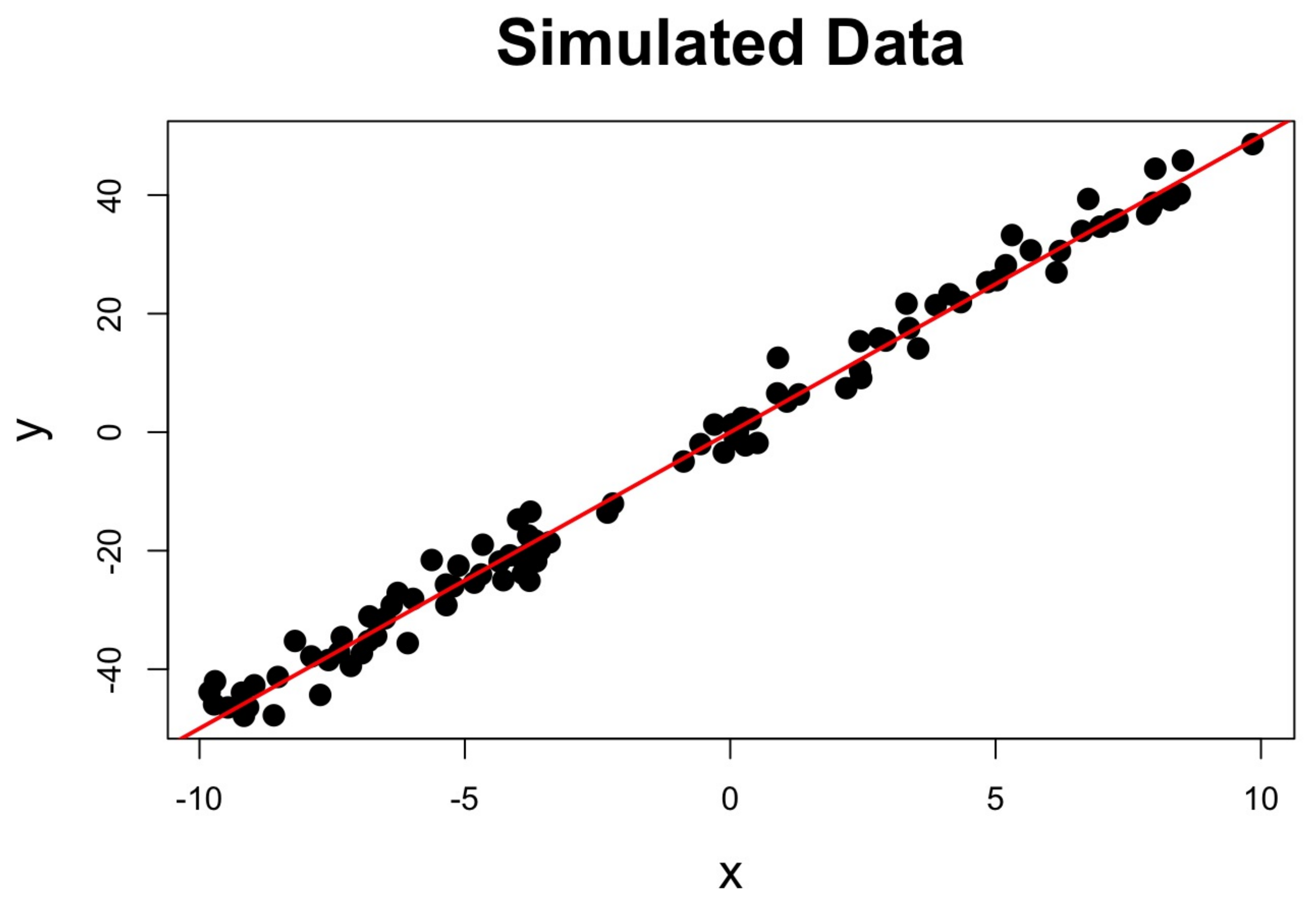
Let's use linear regression with one predictor, no intercept as a case study.

Given: Training set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, with continuous response y_i and single predictor x_i for the i-th observation.

Do: Estimate parameter β_1 in model $y = \beta_1 x$ to solve

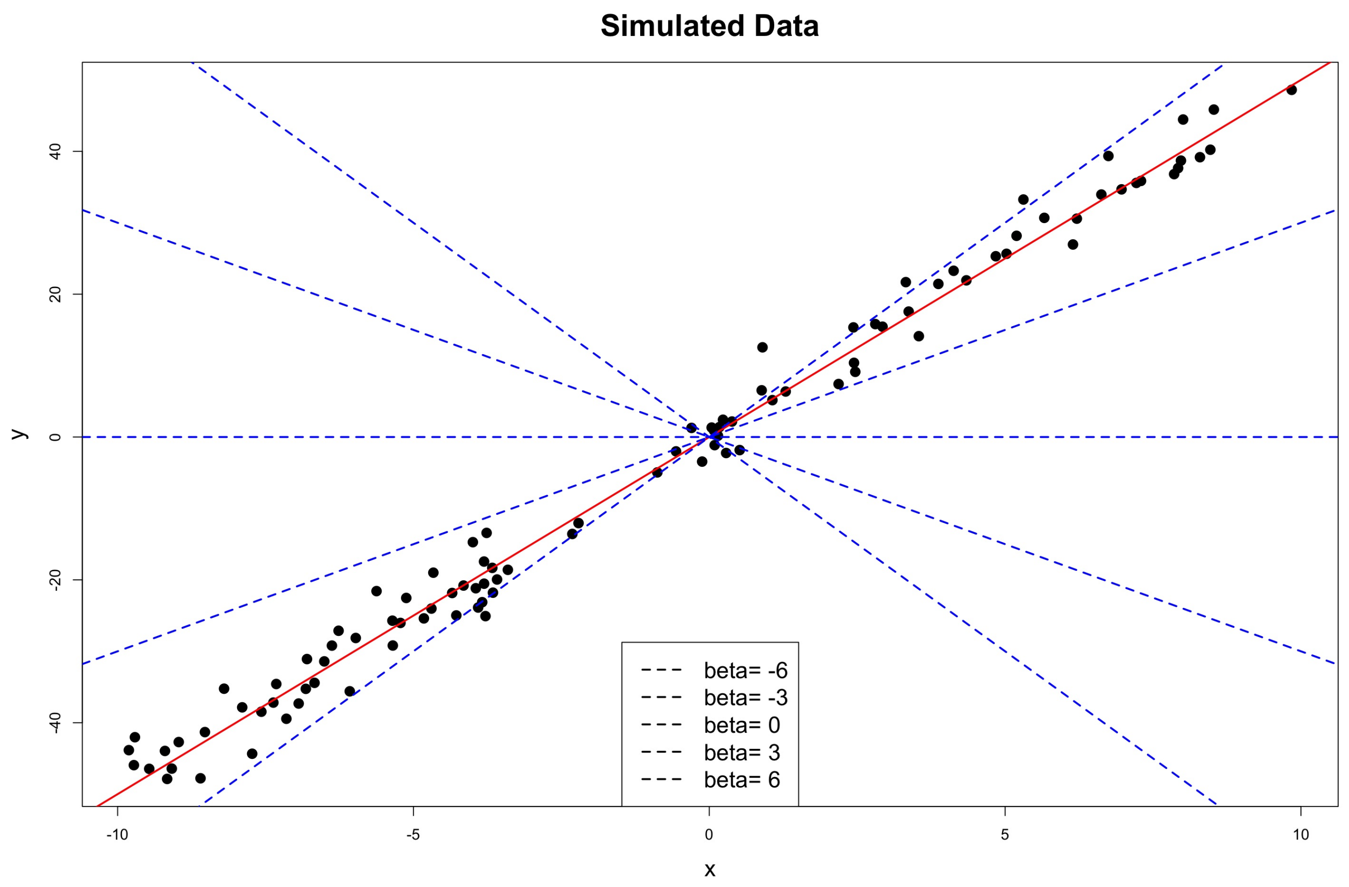
$$\min_{\beta_1} L(\beta_1) = \frac{1}{2} \sum_{i=1}^{n} (y_i - \beta_1 x_i)^2$$

Case Study



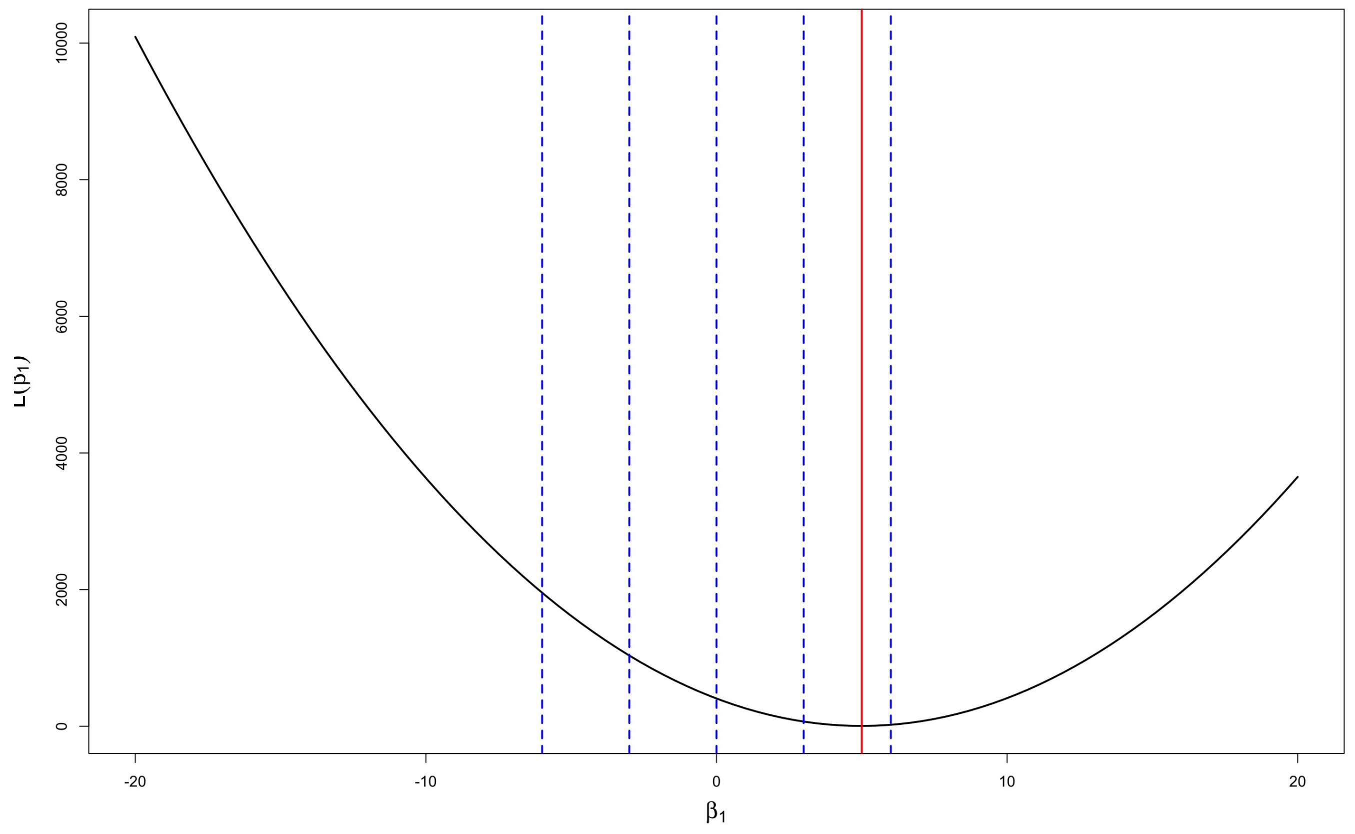
Suppose we want to fit this model to the following (simulated) data:

Case Study



Our goal is then to find the value of `(\beta_1)` that minimizes mean squared error. This corresponds to finding one of these many possible lines:

Case Study



Each of which has a specific error for this dataset:

Case Study

1) As we saw before in class, loss is minimized when the derivative of the loss function is 0

2) and, the derivative of the loss (with respect to β_1) at a given estimate β_1 suggests new values of β_1 with smaller loss!

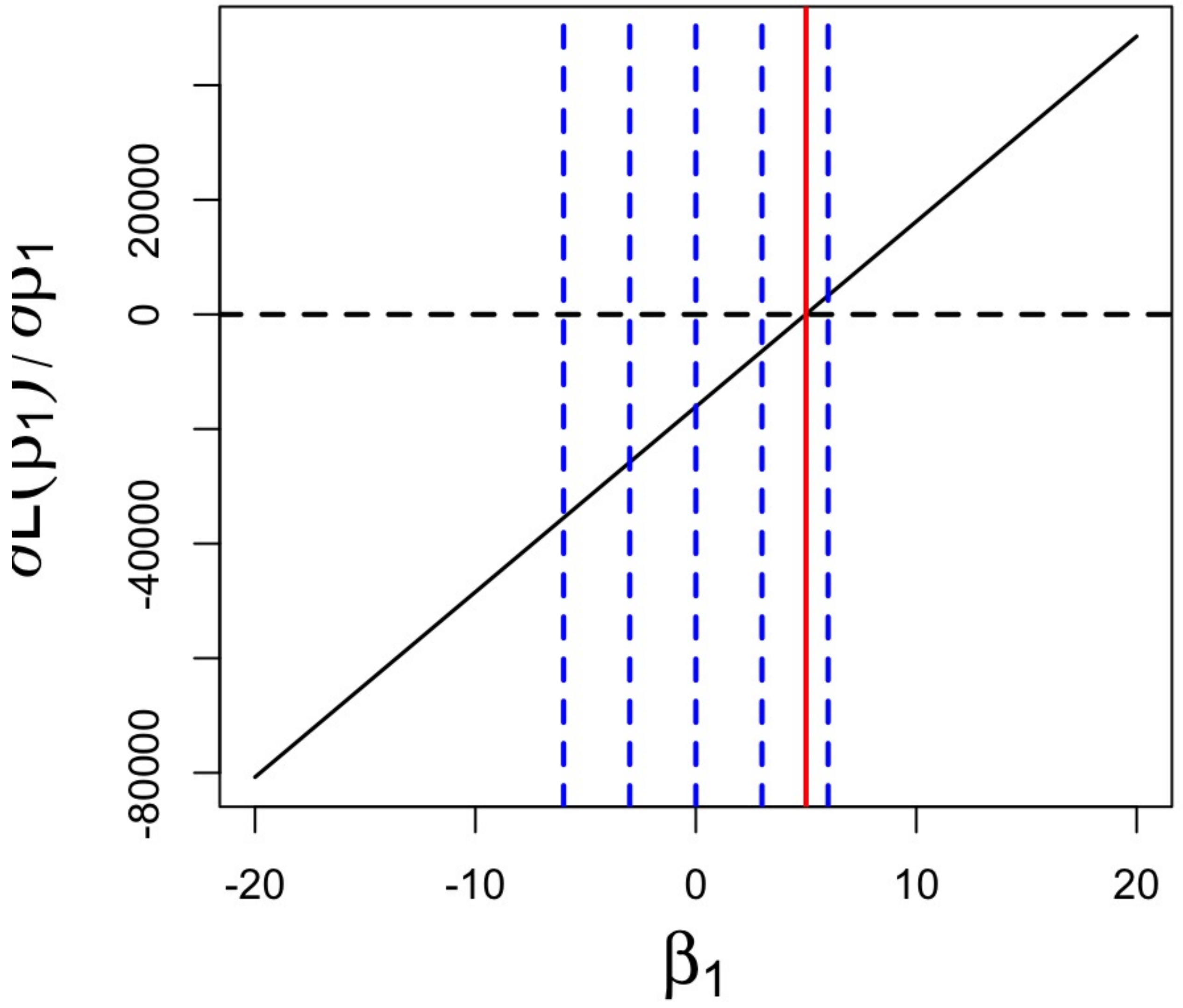
Case Study

Let's take a look at the derivative:

$$
\frac{\partial}{\partial \beta_1} L(\beta_1) =
$$

$$
\frac{\partial}{\partial \beta_1} \frac{1}{2} \sum_{i=1}^{n} (y_i - \beta_1 x_i)^2
$$

$$
= \sum_{i=1}^{n} (y_i - \beta_1 x_i)(-x_i)
$$



Gradient Descent

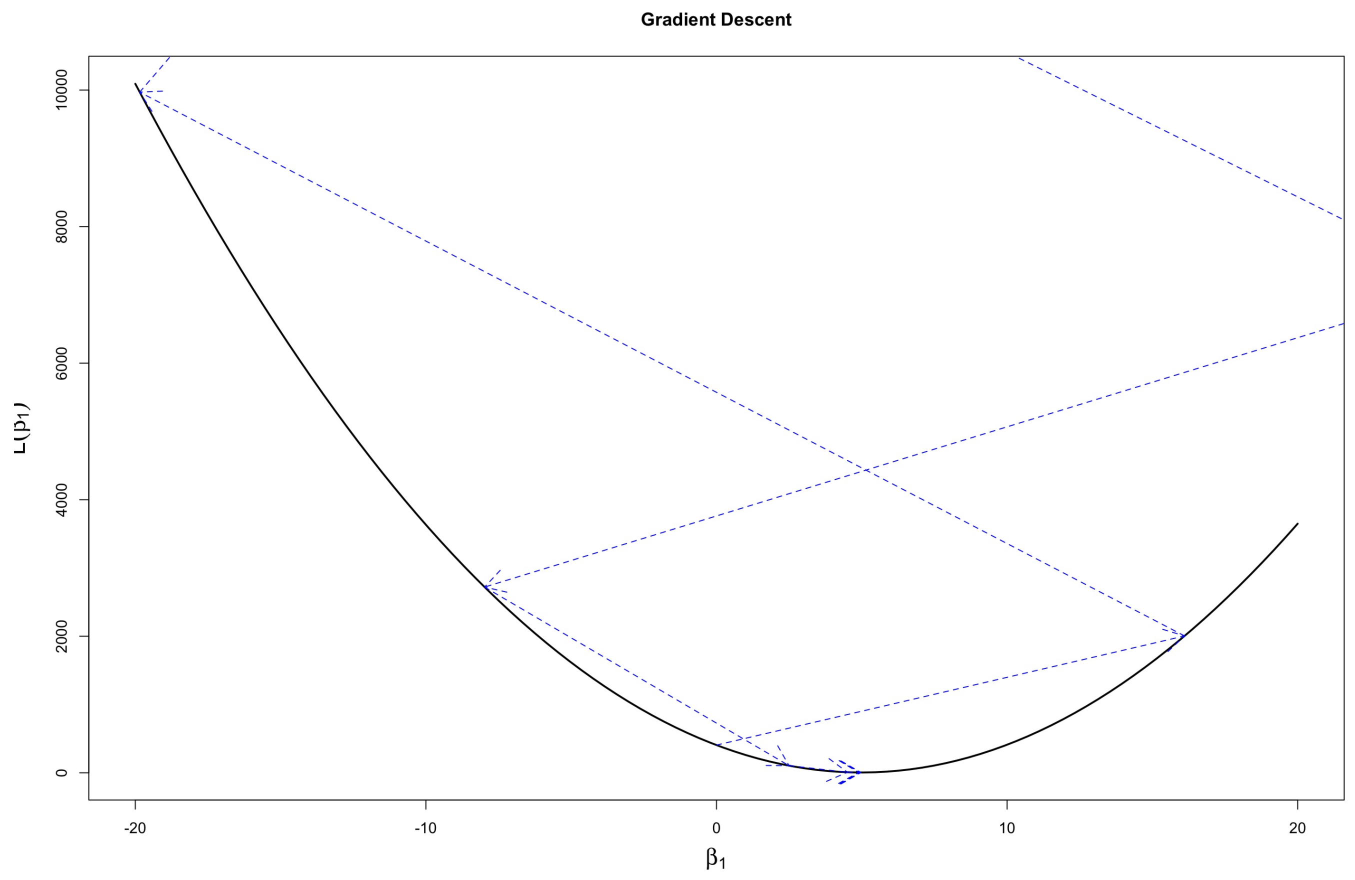
This is what motivates the Gradient Descent algorithm

1. Initialize $\beta_1 = 0$
2. Repeat until convergence
   - Set $\beta_1 = \beta_1 + \alpha \sum_{i=1}^{n} (y_i - f(x_i))x_i$

Gradient Descent

The basic idea is to move the current estimate of β_1 in the direction that minimizes loss the fastest. Another way of calling this algorithm is **Steepest Descent**.

Gradient Descent



Let's run GD and track what it does:

Gradient Descent

"Batch" gradient descent: take a step (update β_1) by calculating derivative with respect to all n observations in our dataset.

$$\beta_1 = \beta_1 + \alpha \sum_{i=1}^{n} (y_i - f(x_i, \beta_1))x_i$$

where $f(x_i) = \beta_1 x_i$.

Gradient Descent

For multiple predictors (e.g., adding an intercept), this generalizes to the gradient

$$\beta = \beta + \alpha \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i, \beta))\mathbf{x}_i$$

where $f(\mathbf{x}_i, \beta) = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip}$

Gradient Descent

Gradiest descent falls within a family of optimization methods called first-order methods (first-order means they use derivatives only). These methods have properties amenable to use with very large datasets:

1. Inexpensive updates
2. "Stochastic" version can converge with few sweeps of the data
3. "Stochastic" version easily extended to streams
4. Easily parallelizable

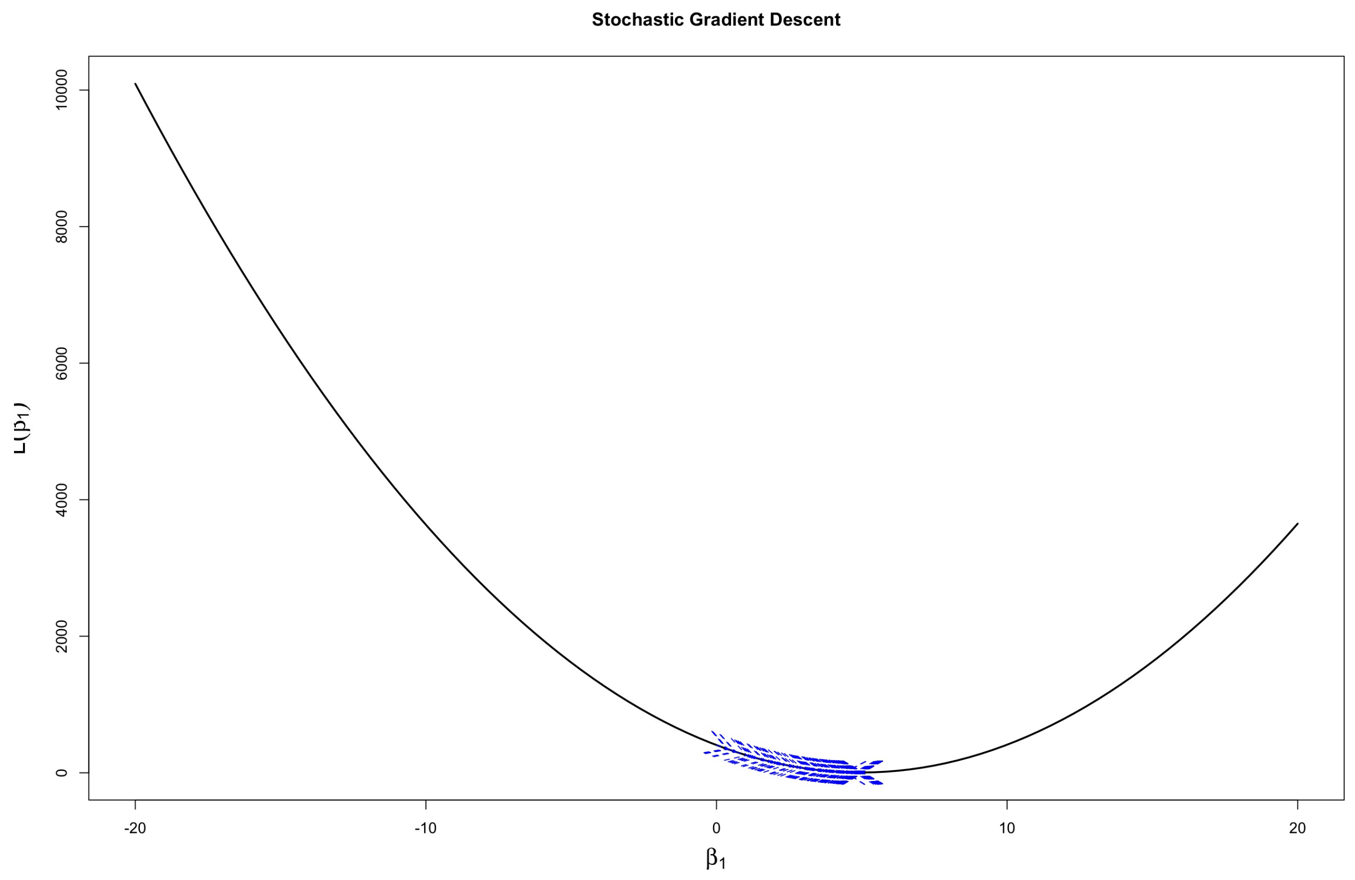
Drawback: Can take many steps before converging

Stochastic Gradient Descent

Key Idea: Update parameters using update equation one observation at a time:

1. Initialize $\beta = \mathbf{0},\ i = 1$
2. Repeat until convergence
   - For $i = 1$ to n
   - Set $\beta = \beta + \alpha(y_i - f(\mathbf{x}_i, \beta))\mathbf{x}_i$

Stochastic Gradient Descent



Let's run this and see what it does:

Stochastic Gradient Descent

Why does SGD make sense?

For many problems we are minimizing a cost function of the type

$$\arg\min_f \frac{1}{n} \sum_i L(y_i, f_i) + \lambda R(f)$$

Which in general has gradient

$$\frac{1}{n} \sum_i \nabla_f L(y_i, f_i) + \lambda \nabla_f R(f)$$

Stochastic Gradient Descent

$$\frac{1}{n}\sum_i \nabla_f L(y_i, f_i) + \lambda \nabla_f R(f)$$

The first term looks like an empirical estimate (average) of the gradient at

f_i

SGD then uses updates provided by a different estimate of the gradient based on a single point.

- Cheaper
- Potentially unstable

Stochastic Gradient Descent

In practice

- Mini-batches: use 10 or so examples at a time to estimate gradients
- Shuffle data order every pass

Stochastic Gradient Descent

SGD easily adapts to data streams where we receive observations one at a time and assume they are not stored.

This setting falls in the general category of online learning.

Online learning is extremely useful in settings with massive datasets

Stochastic Gradient Descent

Parallelizing gradient descent

Gradient descent algorithms are easily parallelizable:

- Split observations across computing units
- For each step, compute partial sum for each partition (map), compute final update (reduce)

$$\beta = \beta + \alpha * \sum_{\text{partition } p} \sum_{i \in p} (y_i - f(\mathbf{x_i}, \beta))\mathbf{x}_i$$

Stochastic Gradient Descent

This observation has resulted in their implementation if systems for large-scale learning:

1. Vowpal Wabbit
    - Implements general framework of (sparse) stochastic gradient descent for many optimization problems

Stochastic Gradient Descent

1. Spark MLlib
   - Implements many learning algorithms using Spark framework we saw previously

Sparse Regularization

For many ML algorithms prediction time-efficiency is determined by the number of predictors used in the model.

Reducing the number of predictors can yield huge gains in efficiency in deployment.

The amount of memory used to make predictions is also typically governed by the number of features. (Note: this is not true of kernel methods like support vector machines, in which the dominant cost is the number of support vectors.)

Sparse Regularization

The idea behind sparse models, and in particular, sparse regularizers.

A disadvantage of optimizing problems of the form

$$\sum_i L(y_i, w) + \lambda |w|^2$$

That they tend to never produce weights that are exactly zero.

Sparse Regularization

Instead, minimize problems of the form

$$\sum_i L(y_i, w) + \lambda |w|_1$$

where $\|w\|_1 = \sum_j |w_j|$

Sparse Regularization

This is a convex optimization problem.

Can use standard subgradient methods.

See CIML for further details.

Feature Hashing

For data sets with a large number of features/attributes an idea based on hashing can also help.

Suppose we are building a model over P features (P very large). We use hashing to reduce the number of features to smaller number p.

For each observation $x \in \mathbb{R}^P$ we transform it into observation $\tilde{x} \in \mathbb{R}^p$.

We will use hash function $h : P \rightarrow p$

Feature Hashing

Initialize $\tilde{x} = \langle 0, 0, \dots, 0 \rangle$

For each $j \in 1, \dots, P$

- Hash j to position $k = h(j)$
- Update $\tilde{x}_k \leftarrow \tilde{x}_k + x_j$

Return $\tilde{x}$

We can think of this as a feature mapping

$$\phi(x)_k = \sum_{j | j \in h^{-1}(k)} x_j$$

Feature Hashing

To see how what this does, we can see what the inner product between observations in the smaller feature space is:

$$\begin{aligned}
\phi(x)'\phi(z) &= \sum_k \left[\sum_{j|j\in h^{-1}(k)} x_j\right]\left[\sum_{j'|j'\in h^{-1}(k)} z_{j'}\right] \\
&= \sum_k \sum_{j,j'|j,j'\in h^{-1}(k)} x_j z_{j'} \\
&= \sum_j \sum_{j'|j'\in h^{-1}(h(j))} x_j z_{j'} \\
&= \sum_j x_j z_j + \sum_{j'\neq j|j'\in h^{-1}(h(j))} x_j z_{j'} = x'z + \cdots
\end{aligned}$$

Feature Hashing

So, we get the inner product in the original large dimensions plus an extra quadratic term

$$\phi(x)'\phi(z) = x'z + \sum_{j' \neq j | j' \in h^{-1}(h(j))} x_j z_{j'}$$

- We might get lucky and get a useful interaction between two features
- Nonetheless, the size of this sum is very small due to property of hash functions: expected value of product is ≈ 0

Large-Scale Learning

- Database operations for out-of-memory datasets
- Parallelization on shared-nothing architectures (map reduce)
- Spark as MR framework also supporting iterative procedures (optimization)
- Stochastic gradient descent (many low cost steps, easy to parallelize)
- Sparse models (build models with few features)
- Feature Hashing (build models over smaller number of features, retain inner product)