Sibyl: a system for large scale machine learning

Tushar Chandra, Eugene Ie, Kenneth Goldman,
Tomas Lloret Llinares, Jim McFadden, Fernando Pereira,
Joshua Redstone, Tal Shaked, Yoram Singer

Machine Learning Background

Use the past to predict the future

Core technology for internet-based prediction tasks

Examples of problems that can be solved with machine learning:

- Classify email as spam or not
- Estimate relevance of an impression in context:
  - Search, advertising, videos, etc.
  - Rank candidate impressions

The internet adds a scaling challenge:

- 100s of millions of users interacting every day
- Good solutions require a mix of theory and systems

Overview of Results

Built a large scale machine learning system:

- Used recently developed machine learning algorithm
- Algorithms have provable convergence & quality guarantees
- Solves internet scale problems with reasonable resources
- Flexible: various loss functions and regularizations

Used numerous well known systems techniques

- MapReduce for scalability
- Multiple cores and threads per computer for efficiency
- GFS to store lots of data
- Compressed column-oriented data format for performance

Inference and Learning

- Objective: draw reliable inferences from all the evidence in our data
  - Is this email SPAM?
  - Is this webpage porn?
  - Will this user click on that ad?
- Learning: create concise representations of the data to support good inferences

Many, Sparse Features

- Many elementary features: words, etc.
- Most elementary features are infrequent
- Complex features:
  - combination of elementary features
  - discretization of real-valued features
- Most complex features don't occur at all
- We want algorithms that scale well with number of features that are actually present, **not** with the number of possible features

Supervised Learning

- Given feature-based representation
- Feedback through a label:
  - Good or Bad
  - Spam or Not-spam
  - Relevant or Not-relevant
- Supervised learning task:
  - Given training examples, find an accurate model that predicts their labels

Machine learning overview

Training data

Label	Feature 1,	...	Feature n
Label	Feature 1',	...	Feature n'
Label	Feature 1'',	...	Feature n''

Machine learning overview

Training data

Label	Feature 1,	...	Feature n
Label	Feature 1',	...	Feature n'
Label	Feature 1'',	...	Feature n''

Model

Feature 1 = 0.2, ... Feature n = -0.5

Machine learning overview

Training data

Label	Feature 1,	...	Feature n
Label	Feature 1’,	...	Feature n’
Label	Feature 1”,	...	Feature n”

Model

Feature 1 = 0.2, ... Feature n = -0.5

+

Feature 1’”, ... Feature n’”

Machine learning overview

Training data

Label Feature 1, ... Feature n

Label Feature 1’, ... Feature n’

Label Feature 1”, ... Feature n”

Model

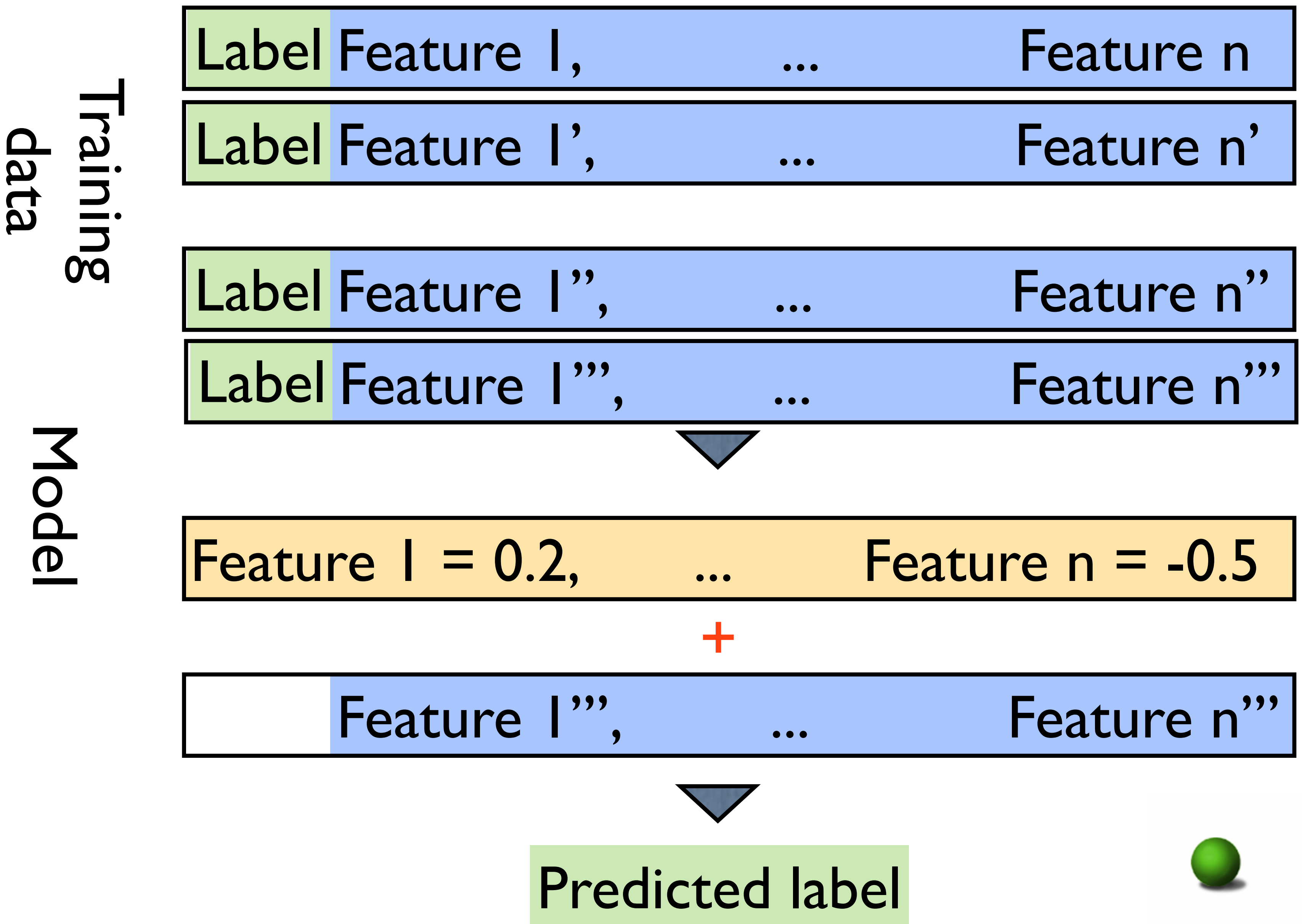
Feature 1 = 0.2, ... Feature n = -0.5

+

Feature 1’”, ... Feature n’”

Predicted label

Machine learning overview



Example: Spam Prediction

- Feedback on emails:
  "Move to Spam" , "Move to Inbox"
- Lots of features:
  - `Viagra ∈ Document`
  - `IP Address of sender is bad`
  - `Sender's domain @google.com`
  - `...`
- Feedback returned daily and grows with time
- New features appear every day

From Emails to Vectors

- User receives an email from an unknown sender
- Email is tokenized:

```
...
Viagra  ∈ Document
Sudafed ∈ Document
Find a young wife ∈ Document
...
```

- Compressed instance:

$$\mathbf{x} \in \{0,1\}^n \quad (0,0,1,0,1,0,\ldots,0,0,1,0)$$

From Emails to Vectors

- User receives an email from an unknown sender
- Email is tokenized:

```
...
Viagra  ∈ Document
Sudafed ∈ Document
Find a young wife ∈ Document
...
```

- Compressed instance:

$$\mathbf{x} \in \{0,1\}^n \quad (0,0,1,0,1,0,\ldots,0,0,1,0)$$

Prediction Models

Captures importance of features

```
Viagra  ∈ Document => score +2.0
Sudafed ∈ Document => score +0.5
Sender's domain @google.com => score -1.0
```

Represented as a vector of weights

w = (0, 0, 2.0, -0.1, 0.5, ..., -1.0, ...)

Scoring the email

w.x = 2.0 + 0.5 - 1.0

Logistic regression (used for probability predictions)

$$\text{Probability} = \frac{1}{1 + e^{-w.x}}$$

Prediction Models

Captures importance of features

```
Viagra  ∈ Document => score +2.0
Sudafed ∈ Document => score +0.5
Sender's domain @google.com => score -1.0
```

Represented as a vector of weights

w = (0, 0, 2.0, -0.1, 0.5, ..., -1.0, ...)

Scoring the email

w.x = 2.0 + 0.5 - 1.0

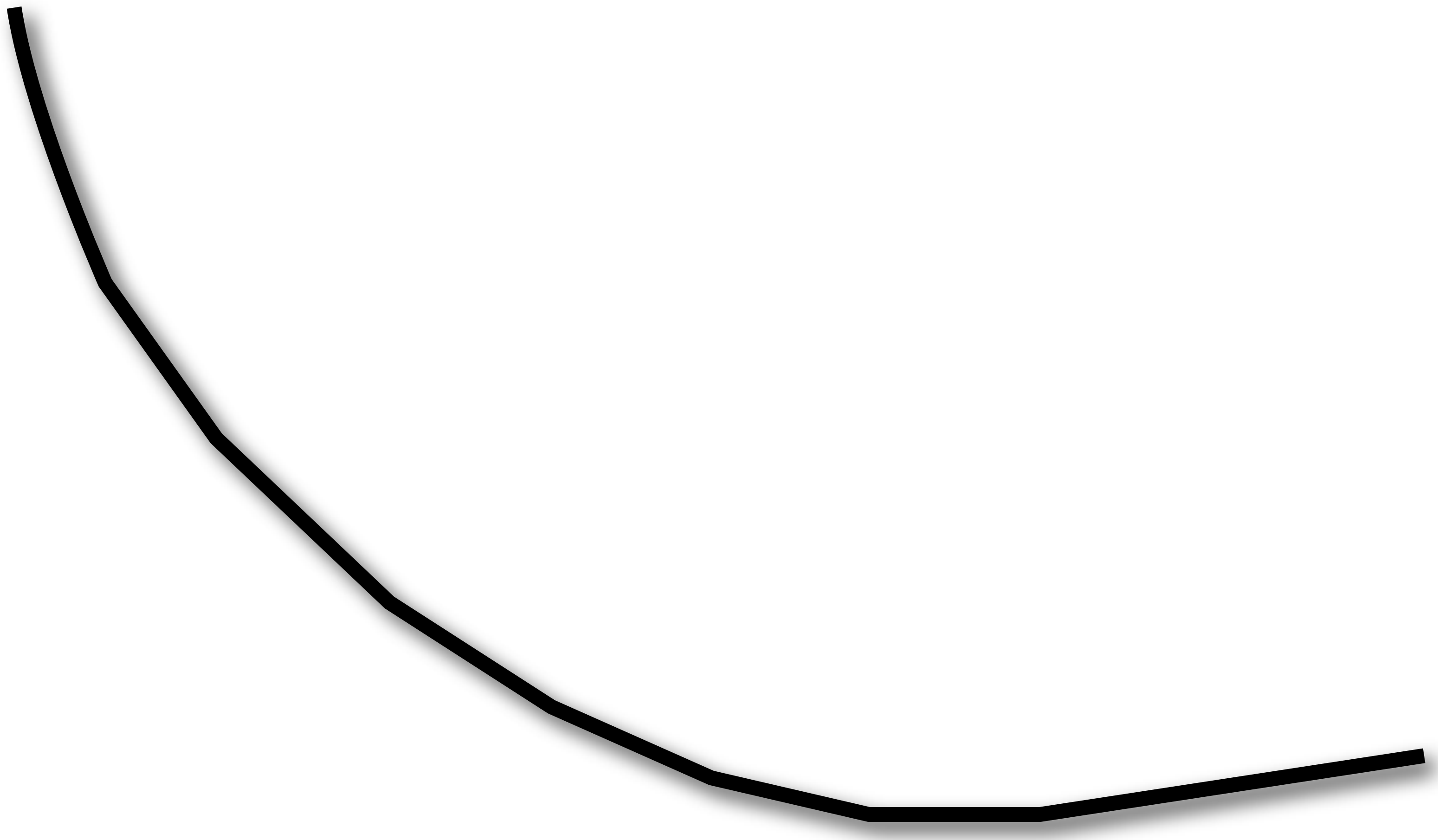
Logistic regression (used for probability predictions)

$$\text{Probability} = \frac{1}{1 + e^{-w.x}}$$

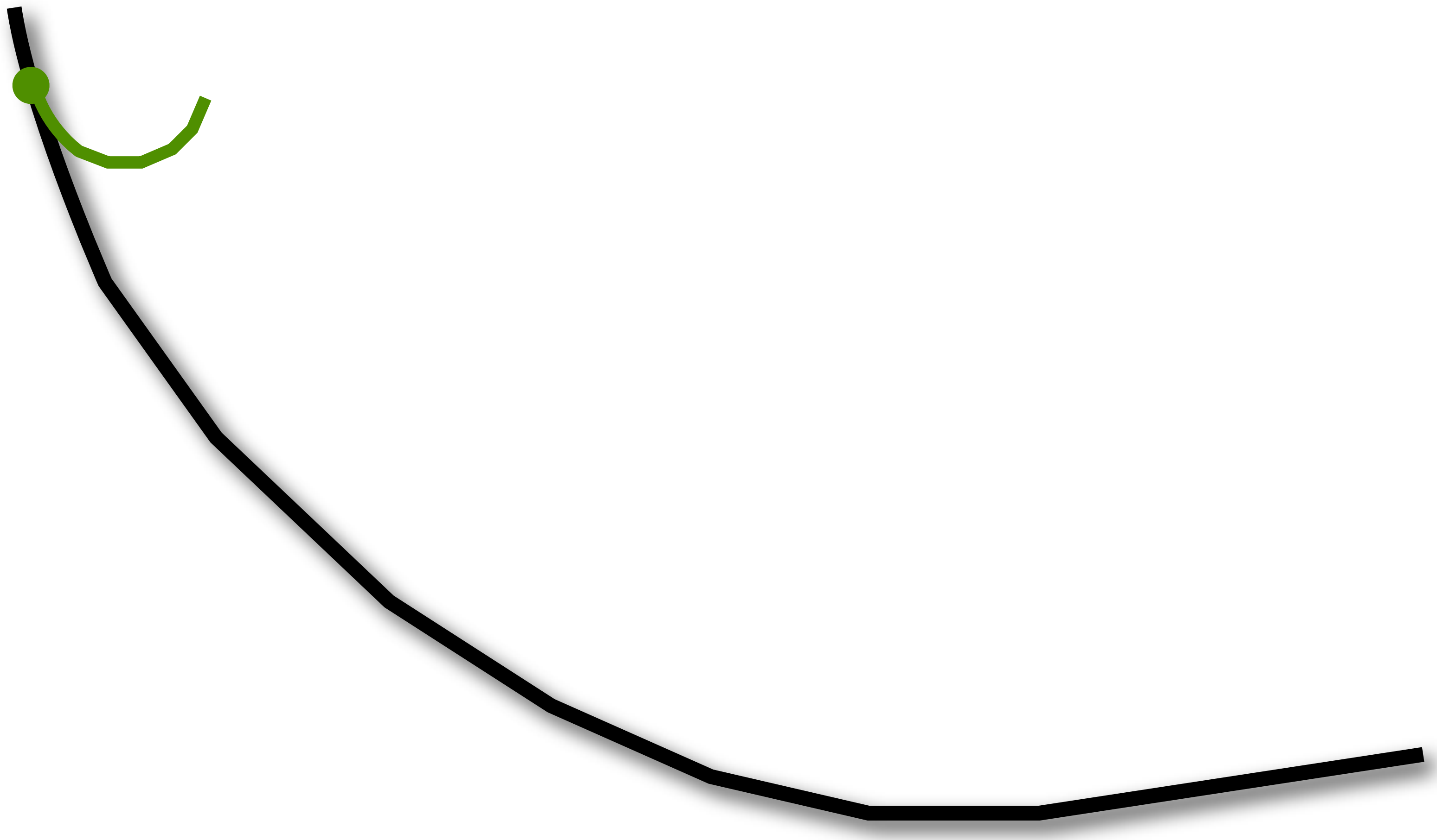
Parallel Boosting (Collins, Schapire, Singer 2001)

- Iterative algorithm, each iteration improves model
- Number of iterations to get within ϵ of the optimum:
  $$\log(m)/\epsilon^2$$
- Updates correlated with gradients,
  but not a gradient algorithm
- Self-tuned step size,
  large when instances are sparse

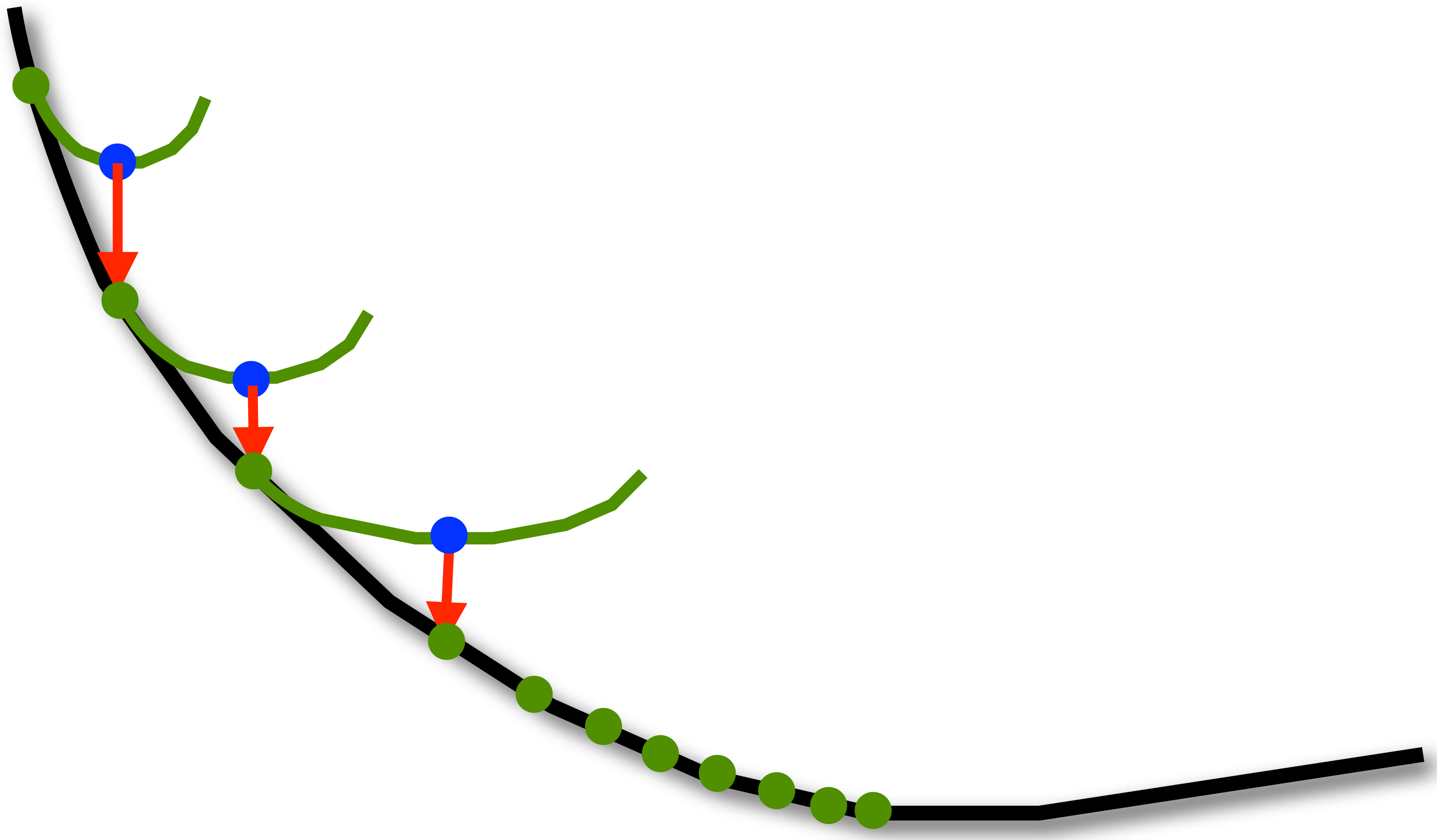
Boosting: Illustration



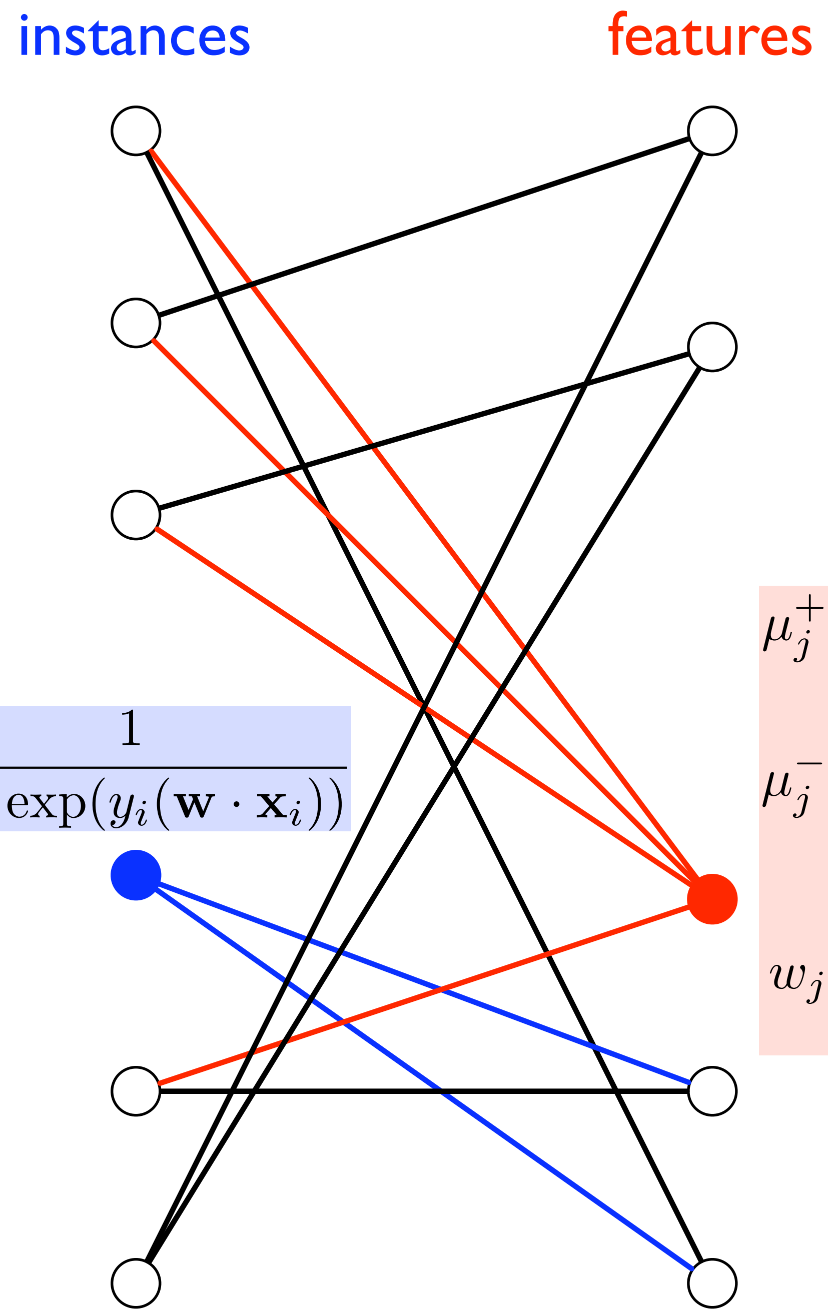
Boosting: Illustration



Boosting: Illustration



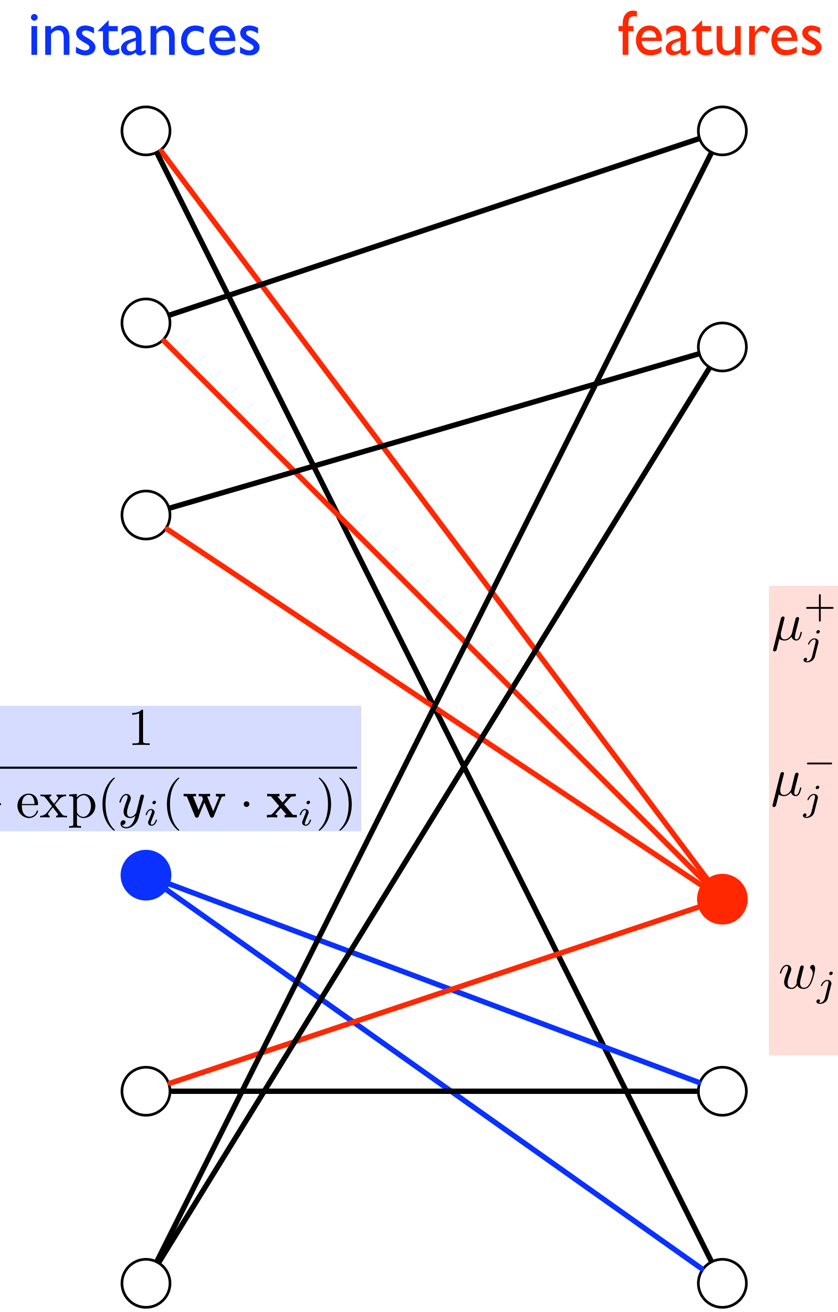
Parallel Boosting Algorithm



$$q(i) = \frac{1}{1 + \exp(y_i(\mathbf{w} \cdot \mathbf{x}_i))}$$

$$\begin{aligned}
\mu_j^+ &= \sum_{i: y_i = 1 \wedge x_{ij} = 1} q(i) \\
\mu_j^- &= \sum_{i: y_i = -1 \wedge x_{ij} = 1} q(i) \\
w_j &\mathrel{+}= \eta \log\left(\frac{\mu_j^+}{\mu_j^-}\right)
\end{aligned}$$

Parallel Boosting Algorithm



$$q(i) = \frac{1}{1 + \exp(y_i(\mathbf{w} \cdot \mathbf{x}_i))}$$

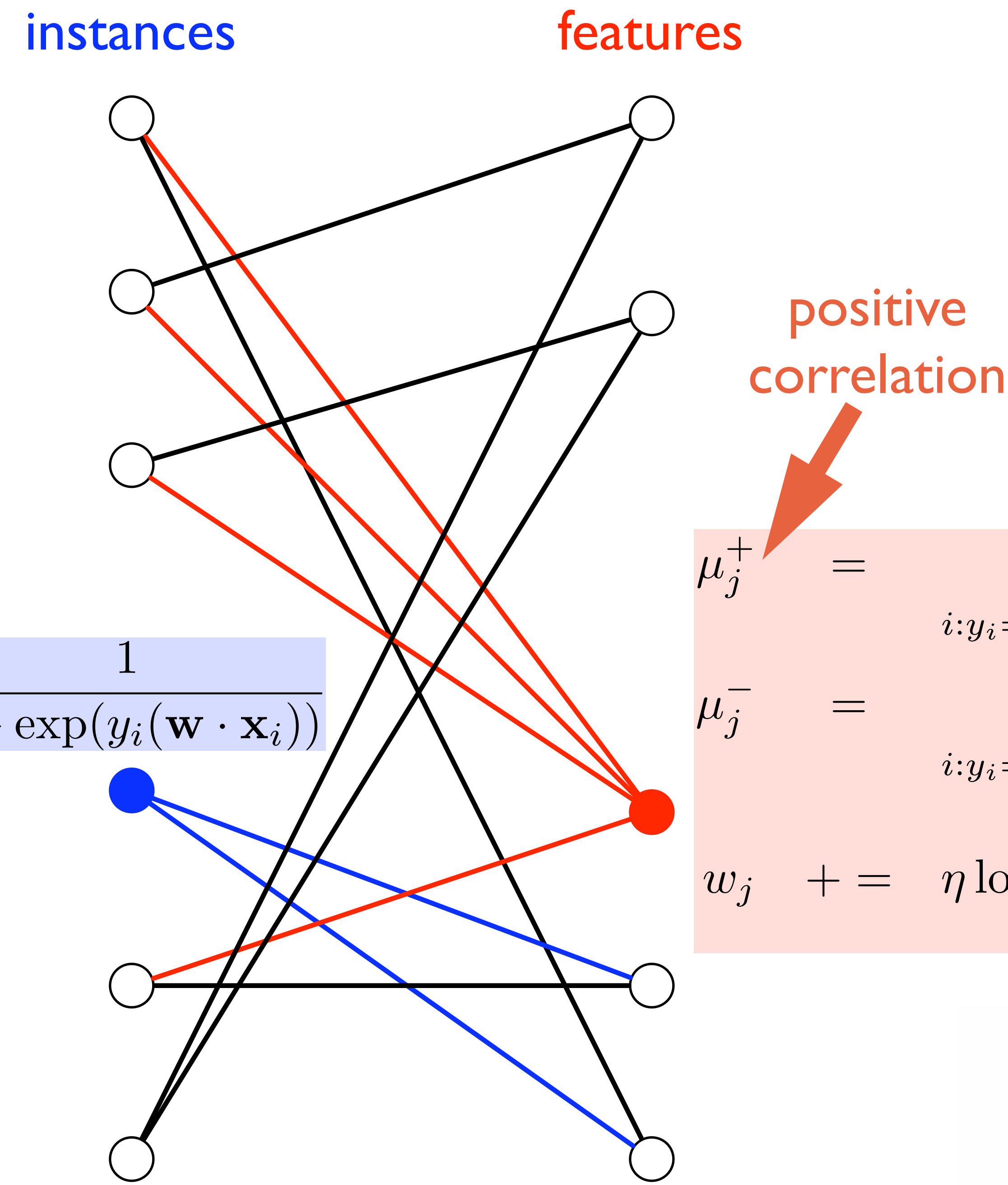
mistake probability

$$\mu_j^+ = \sum_{i: y_i = 1 \wedge x_{ij} = 1} q(i)$$

$$\mu_j^- = \sum_{i: y_i = -1 \wedge x_{ij} = 1} q(i)$$

$$w_j \mathrel{+}= \eta \log \left(\frac{\mu_j^+}{\mu_j^-} \right)$$

Parallel Boosting Algorithm



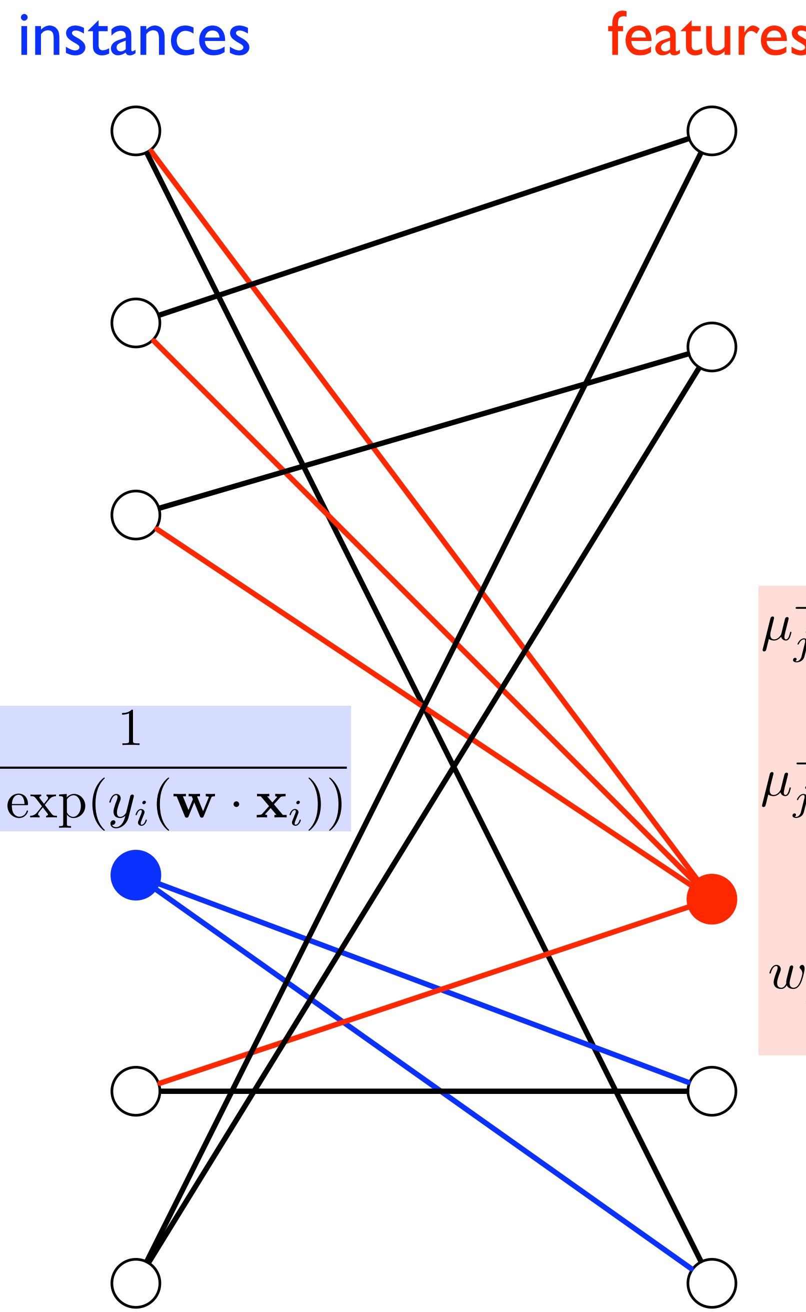
$$q(i) = \frac{1}{1 + \exp(y_i(\mathbf{w} \cdot \mathbf{x}_i))}$$

$$\mu_j^+ = \sum_{i: y_i = 1 \wedge x_{ij} = 1} q(i)$$

$$\mu_j^- = \sum_{i: y_i = -1 \wedge x_{ij} = 1} q(i)$$

$$w_j \mathrel{+}= \eta \log \left(\frac{\mu_j^+}{\mu_j^-} \right)$$

Parallel Boosting Algorithm



$$q(i) = \frac{1}{1 + \exp(y_i(\mathbf{w} \cdot \mathbf{x}_i))}$$

mistake probability

positive correlation

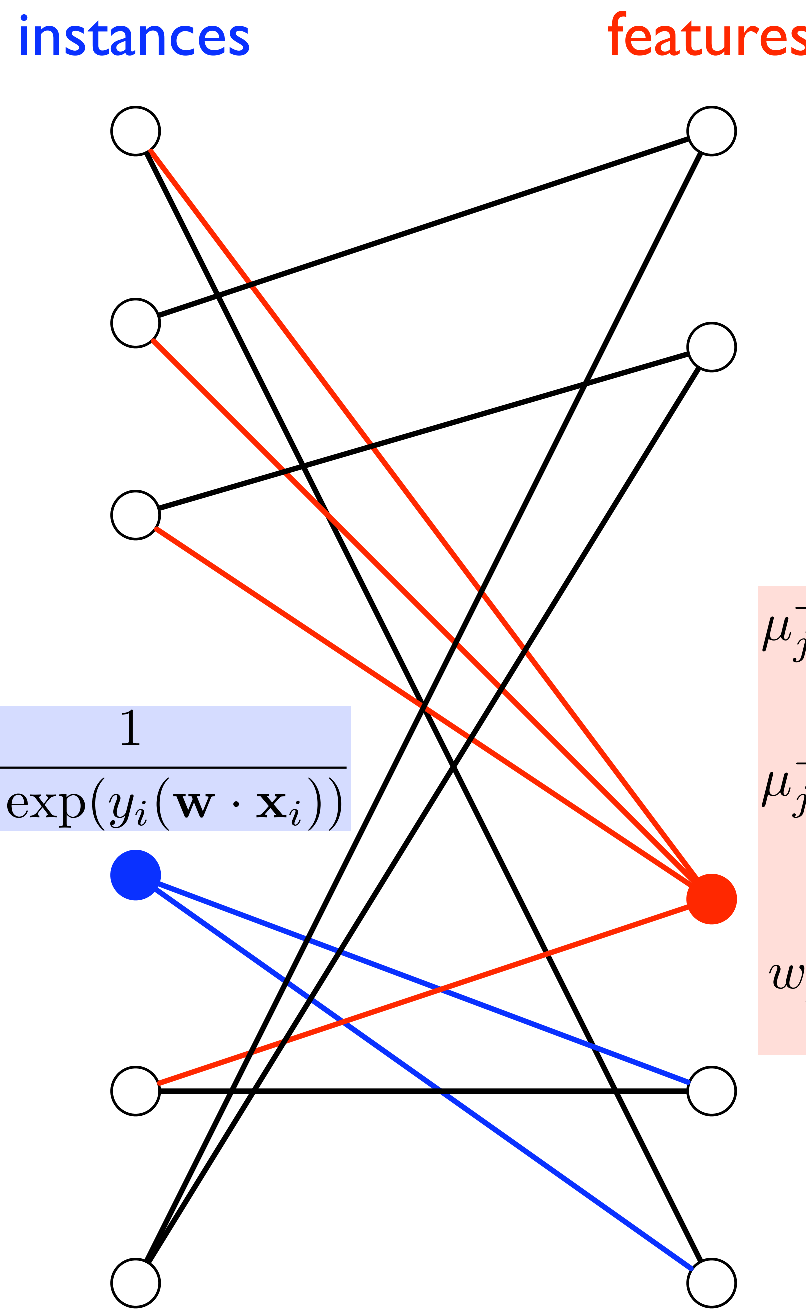
negative correlation

$$\mu_j^+ = \sum_{i: y_i = 1 \wedge x_{ij} = 1} q(i)$$

$$\mu_j^- = \sum_{i: y_i = -1 \wedge x_{ij} = 1} q(i)$$

$$w_j \mathrel{+}= \eta \log \left(\frac{\mu_j^+}{\mu_j^-} \right)$$

Parallel Boosting Algorithm



$$q(i) = \frac{1}{1 + \exp(y_i(\mathbf{w} \cdot \mathbf{x}_i))}$$

mistake probability

positive correlation

negative correlation

$$\mu_j^+ = \sum_{i: y_i = 1 \wedge x_{ij} = 1} q(i)$$

$$\mu_j^- = \sum_{i: y_i = -1 \wedge x_{ij} = 1} q(i)$$

$$w_j \mathrel{+}= \eta \log \left(\frac{\mu_j^+}{\mu_j^-} \right)$$

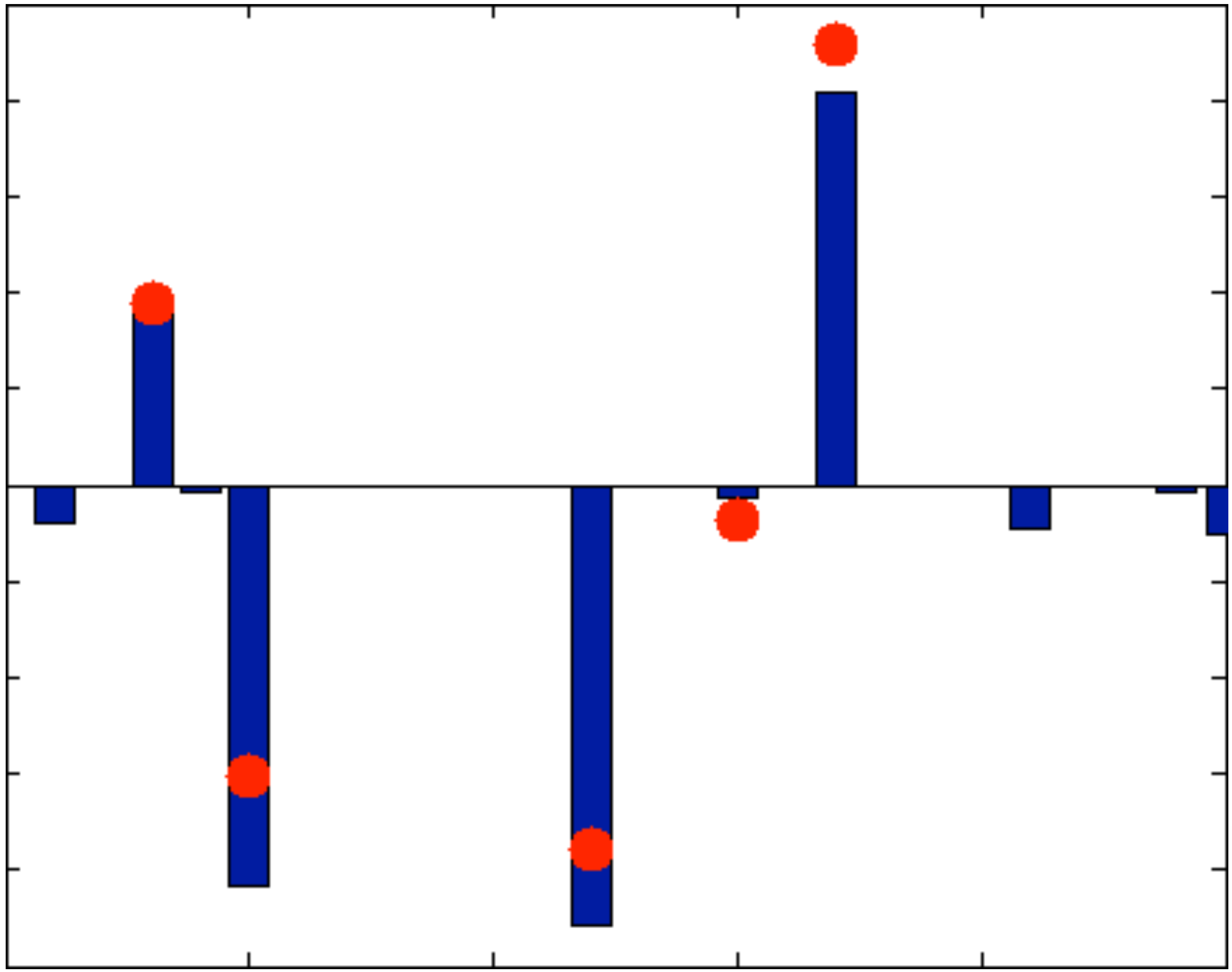
step size

Properties of parallel boosting

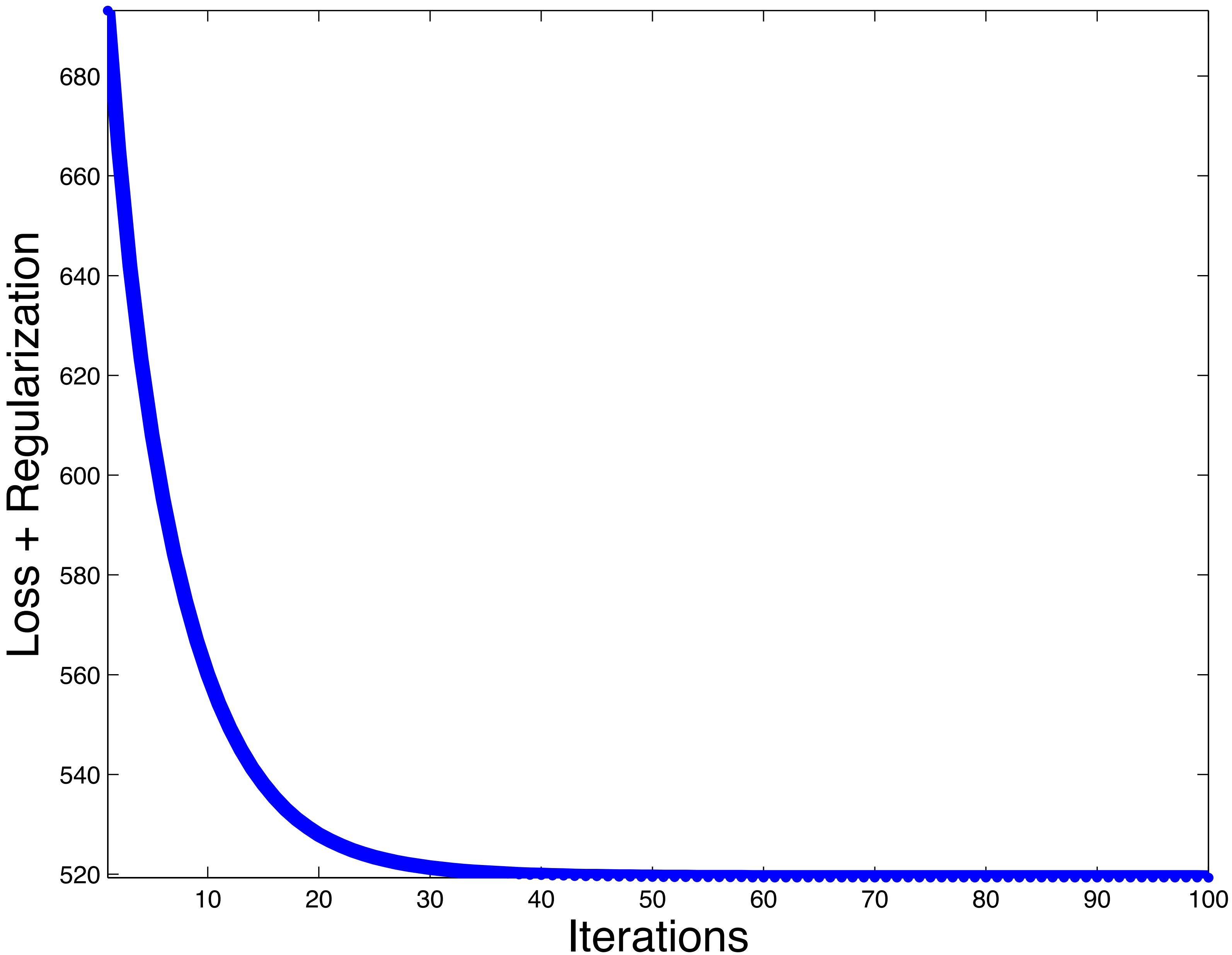
- Embarrassingly parallel:
  1. Computes feature correlations for each example in parallel
  2. Feature are updated in parallel
  - We need to "shuffle" the outputs of Step 1 for Step 2
- Step size inversely proportional to number of active features per example
  - **Not** total number of features
  - Good for sparse training data
- Needs some form of regularization

Learning w/ L_1 Regularization

Learning w/ L_1 Regularization

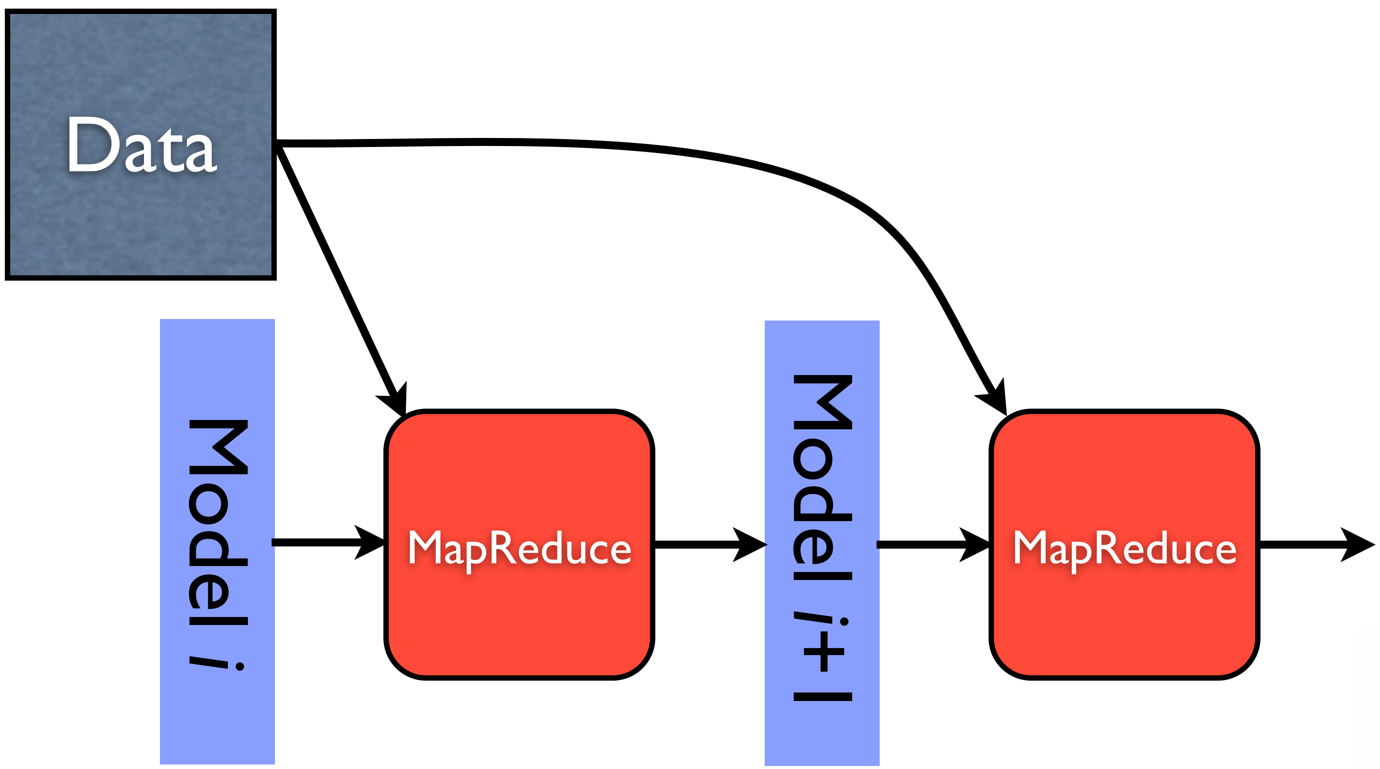


Learning w/ L_1 Regularization



Implementing Parallel Boosting

- + Embarrassingly parallel
- + Stateless, so robust to transient data errors
- + Each model is consistent, sequence of models for debugging
- \- 10-50 iterations to converge



Some observations

- We typically train multiple models
  - To explore different types of features
    - Don't read unnecessary features
  - To explore different levels of regularization
    - Amortize fixed costs across similar models
- Computers have lots of RAM
  - Store the model and training stats in RAM at each worker
- Computers have lots of cores
  - Design for multi-core
- Training data is highly compressible

Design principle: use column-oriented data store

- Column for each field
- Each learner only reads relevant columns
- Benefits
  - Learners read much less data
  - Efficient to transform fields
  - Data compresses better

Design principle: use model sets

- Train multiple similar models together
- Benefit: amortize fixed costs across models
  - Cost of reading training data
  - Cost of transforming data
- Downsides
  - Need more RAM
  - Shuffle more data

Design principle: "Integerize" features

- Each column has its own dense integer space
- Encode features in decreasing order of frequency
- Variable-length encoding of integers
- Benefits:
  - Training data compression
  - Store in-memory model and statistics as arrays rather than hash tables
    - Compact, faster, less data to shuffle

Design principle: store model and stats in RAM

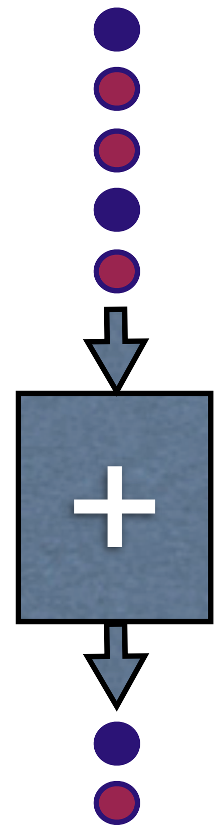
- Each worker keeps in RAM
  - A copy of the previous model
  - Learning statistics for its training data
- Boosting requires O(10 bytes) per feature
- Possible to handle billions of features

Design principle: optimize for multi-core

- Share model across cores
- MapReduce optimizations
  - Multi-shard combiners
    - Share training statistics across cores

Design principle: use combiners to limit communication

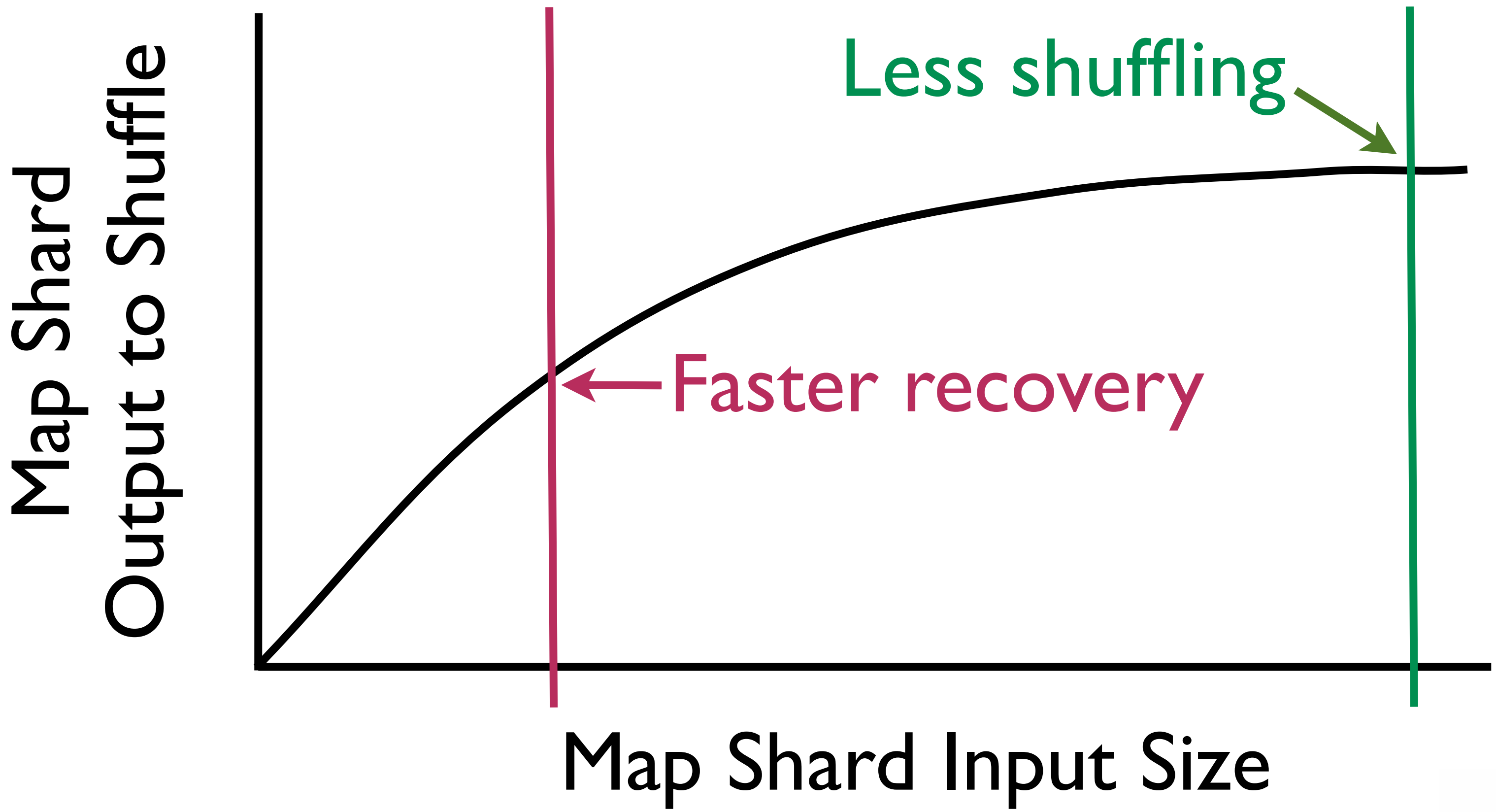
Standard
Mapper



Mapper
with
Combiner

Design principle: use combiners to limit communication

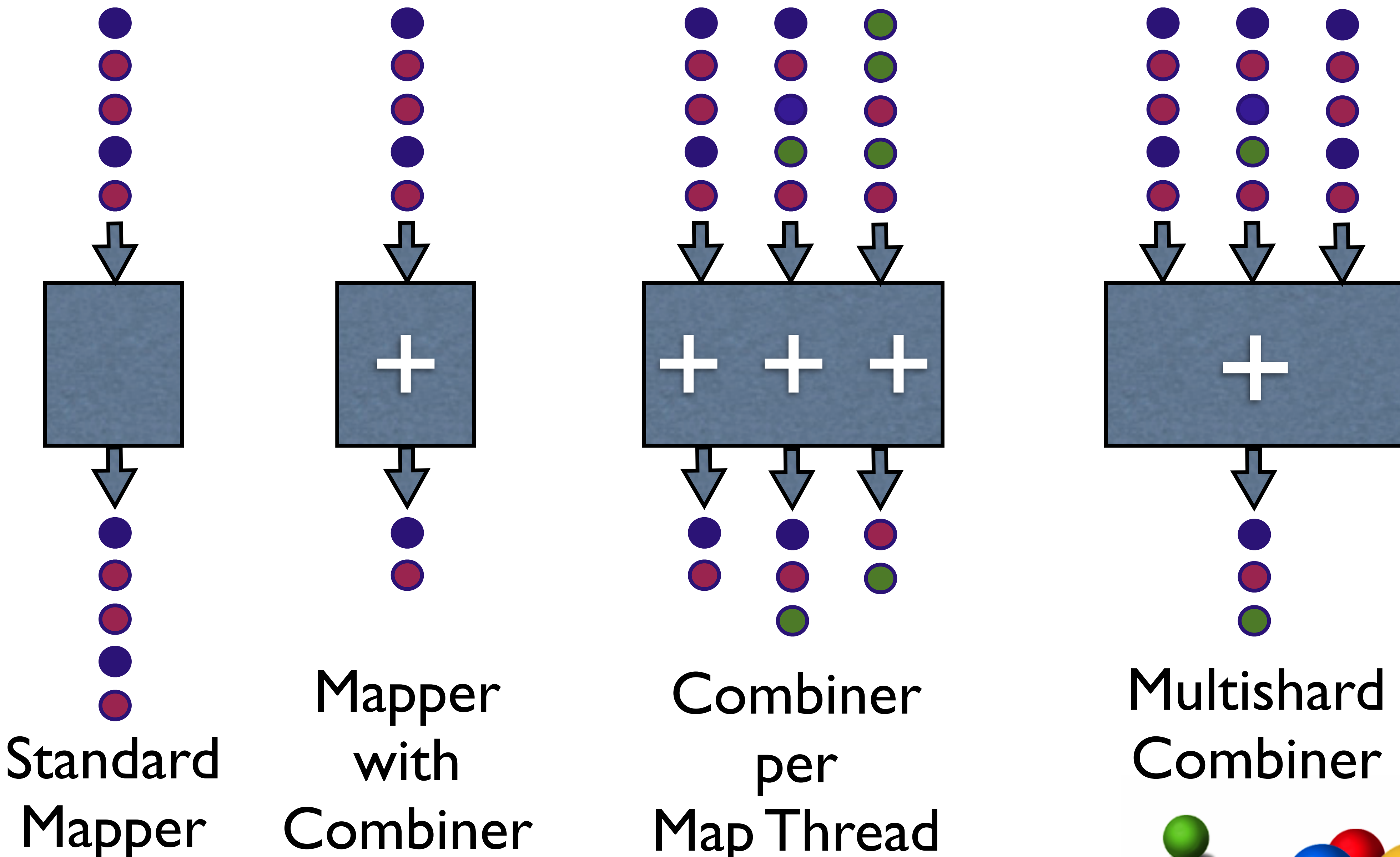
- Fewer large shards mean less shuffling, but possible stragglers when shards fail



Design principle: use combiners to limit communication

- Solution: Multishard Combining
  - Multiple threads per worker
  - Many small map shards per thread
  - One accumulator shared across threads
  - One supershard per worker... less shuffling
  - Spread shards from failed workers across the remaining workers ... fewer stragglers

Design principle: use combiners to limit communication



Compression results

- Data Set 1
  - 3.2x compression (source is unsorted and has medium compression)
  - 2.6x compression (source is sorted and has medium compression)
  - 1.7x compression (source is sorted and has max compression)
  - string -> int map overhead < 0.5%
- Data Set 2
  - 1.8x compression (default compression options)
  - string -> int map overhead < 0.5%

Performance results

Number of models in model set

Cores	1	2	3	4	5
80	1.8M	4.0M	4.4M	5.4M	4.5M
160	1.3M	2.4M	3.0M	4.4M	3.5M
240	1.4M	2.2M	3.0M	3.9M	3.5M
320	1.2M	2.0M	2.4M	2.9M	3.3M
400	1.1M	1.7M	2.4M	2.1M	2.7M

Measurements in features/second per core

Infrastructure challenges

Sibyl is an HPC workload running on infrastructure designed for the web

- Rapidly opens lots of files
    - GFS master overload
- Concurrently reads 100s of files per machine
    - Cluster cross-sectional bandwidth overload
    - Denial of service for co-resident processes
- Random accesses into large vectors
    - Prefetch performance
    - Page-table performance
- MapReduce challenges
    - Multi-shard combiners, column-oriented format
- Column oriented data format creates lots of small files
    - Outside the GFS sweet spot