RDD

Resilient Distributed Dataset

Python
Element-wise multiplication and dot product

```
u = arange(0, 5, .5)
v = arange(5, 10, .5)
u*v
dot(u,v)
```

Dense Vector in Spark

```
myDenseVector = DenseVector([3,4,5])
# Calculate the dot product between the two vectors.
denseDotProduct = numpyVector.dot(numpyVector2)
```

```
# Multiply the elements in dataset by five, keep just the even values, and sum those values
finalSum = dataset.map(lambda x: x*5).filter(lambda x: x % 2==0).reduce(lambda x, y: x+y)
```

```
pluralLengths = (pluralRDD.map(lambda x: len(x))
                  .collect())
```

Linear regression and distributed machine learning

Distributed logistic regression
w = (X^T X)^{-1} X^T y
Computation: O(nd^2+d^3) operations
Storage: O(nd+d^2) floats
Other methods including cholesky, QR, SVD have the same complexity

Distribute Computation
trainData.map(computeOuterProduct).reduce(sumAndInvert)
Basically distribute the n summation, and aggregate afterwards

Distribute Storation
Storing X and computing $X^T X$ are bottlenecks. Now, storing and operating on $X^T X$ is also a bottleneck. It can't be easily distributed!

1st Rule of Thumb
Computation and storage should be linear (in n,d)

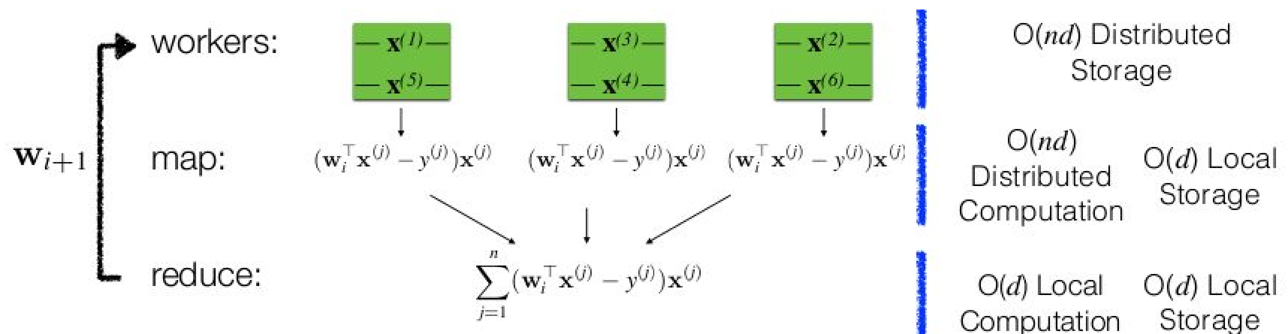
Idea 1: Exploit sparsity
- Explicit sparsity can provide orders of magnitude storage and computational gains
- Latent sparsity assumption can be used to reduce dimension, e.g. PCA, low-rank approximation.

Idea 2: Use different algorithms
Gradient descent is an iterative algorithm that requires O(nd) computation and O(d) local storage per iteration

Gradient descent
Start at a random point
Repeat
 Determine a descent direction
 Choose a step size
 Update
Until stopping criterion is satisfied

Vector Update: $$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \sum_{j=1}^{n} (\mathbf{w}_i^\top \mathbf{x}^{(j)} - y^{(j)})\mathbf{x}^{(j)}$$

Compute summands in parallel!
note: workers must all have $\mathbf{w}_i$



Least Squares, Ridge Regression and Logistic Regression are all convex!

We can move anywhere in R^d Negative gradient is direction of steepest descent!

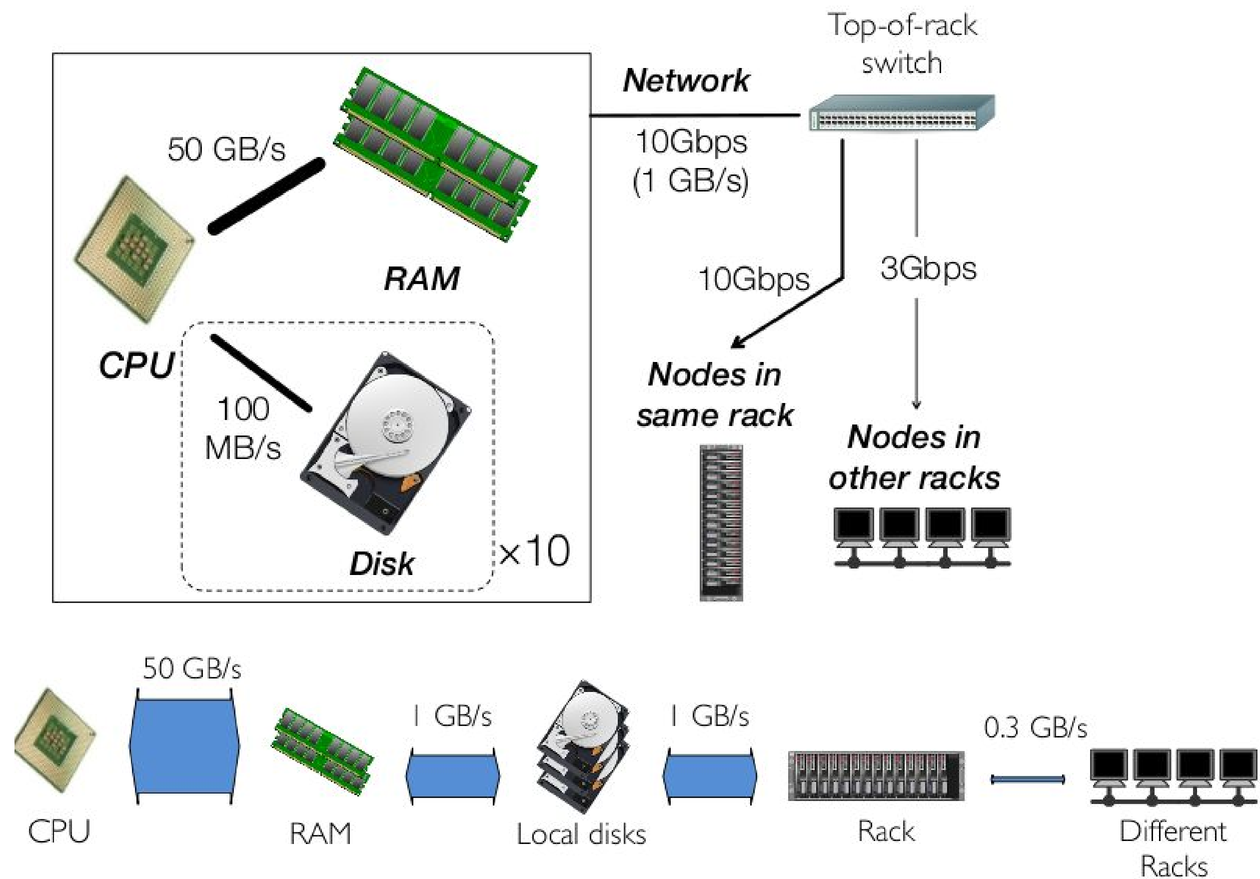
Parallel gradient descent for least squares
```
for i in range(numIters):
    alpha_i = alpha / (n * sqrt(i+1))
    gradient = train.map(lambda lp: gradientSummand(w, lp)).sum()
    w -= alpha_i * gradient
return w
```

Pros:

- Easily parallelized
- Cheap at each iteration
- Stochastic variants can make things even cheaper

Cons:
- Slow convergence especially compared with closed-form
- Requires communication across nodes

Communication hierarchy
CPU: clock speed not changing, but number of cores growing with Moore's Law
RAM: Capacity growing with Moore's law
Disk: Capacity growing exponentially, but not speed



2nd Rule of Thumb
Perform parallel and in-memory computation

Persisting in memory reduces communication
- Especially for iterative computation

Scale-up (Powerful multi-core machine)
- No network communication
- Expensive hardware, eventually hit a wall

Scale-out (distributed, e.g. cloud-based)

- Need to deal with network communication
- Commodity hardware, scales to massive problems
- train.cache() # Persist training data across iterations

3rd Rule of Thumb
Minimize network communication

First observation: We need to store and potentially communicate Data, model and intermediate objects
Solution: Keep large objects local

Example:
Linear regression, big n and big d
- Gradient descent, communicate w_i
- O(d) communication OK for fairly large d
- Compute locally on data (Data Parallel)

Hyperparameter tuning for ridge regression with small n and small d
- Data is small, so can communicate it
- Model is collection of regression models corresponding to different hyperparameters
- Train each model locally (Model Parallel)

Linear regression, big n and huge d
- Gradient descent
- O(d) communication slow with hundreds of millions parameters
- Distribute data and model (Data and Model Parallel)
- Often rely on sparsity to reduce communication

Second observation: ML methods are typically iterative
Solution: Reduce number of iterations

Distributed iterative algorithms must compute and communicate
- In Bulk Synchronous Parallel (BSP) systems, e.g. Apache Spark, we strictly alternate between the two

Distributed Computing Properties
- Parallelism makes computation fast
- Network makes communication slow

Idea: Design algorithm that compute more, communicate less
- Do more computation at each iteration
- Reduce total number of iterations

Extreme: Divide-and-conquer
- Fully process each partition locally, communicate final result
- Single iteration; minimal communication
- Approximate results

```
w = train.mapPartitions(localLinearRegression).reduce(combineLocalRegressionResults)
```

Less extreme: Mini-batch
- Do more work locally than gradient descent before communication
- Exact solution, but diminishing returns with larger batch size

```
for i in range(fewerIters):
        update =
train.mapPartitions(doSomeLocalGradientUpdates).reduce(combineLocalUpdates)
        w += update
```

Throughput: How many bytes per second can be read
Latency: Cost to send message (independent of size)

We can amortize latency
- Send larger messages
- Batch their communication
- E.g. Train multiple models together

Latency	
Memory	1e-4 ms
Hard Disk	10 ms
Network (same datacenter)	.25 ms
Network (US to Europe)	>5 ms

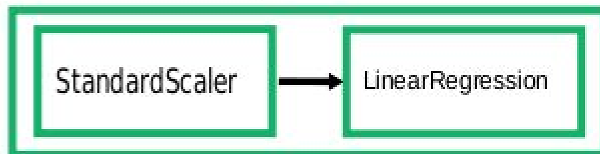
Note. Root mean squared error (RMSE) is typically used as it provides a measure that has the same units as the target variable.

MLlib and Pipelines
Common learning algorithms and utilities
- Classification, Regression, Clustering, Collaborative filtering, Dimensionality Reduction

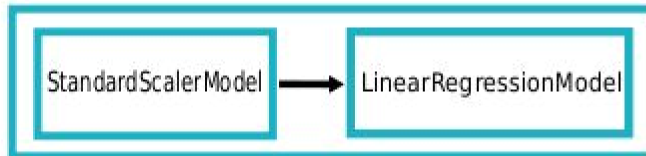
Pipeline

Pipeline

Pipeline



PipelineModel



Two packages
spark.mllib, spark.ml

Transformer
A Transformer is a class which can transform one DataFrame into another DataFrame
E.g. HashingTF, LogisticRegressionModel, Binarizer

Estimator
An Estimator is a class which can take a DataFrame and produce a Transformer

E.g. LogisticRegression, StandardScaler, Pipeline

Some notes on Spark codes
```
sc.range(1, 7, 2).collect()
[1, 3, 5]
```

Logistic Regression and Click-through Rate Prediction

Efficient ads matching
Idea: Predict probability that user will click each ad and choose ads to maximize probability
- Estimate P(click|predictive features)

Predictive features
- Ad's historical performance
- Advertiser and ad content info
- Publisher info
- User info (e.g. search/click history)

Publishers get billions of impressions per day
Data is high-dimensional, sparse, and skewed
- Hundreds of millions of online users
- Millions of unique publisher pages to display ads
- Millions of unique ads to display
- Very few ads get clicked by users

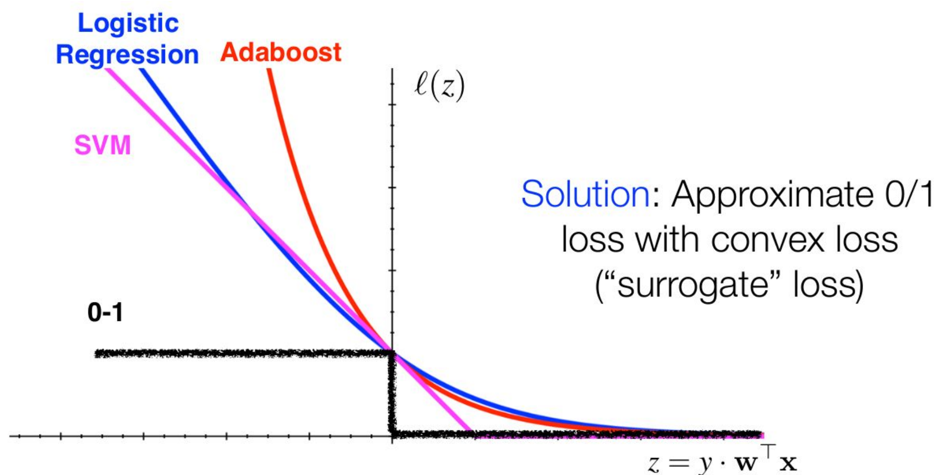
Massive datasets are crucial to tease out signal
Goal: Estimate P(click|user,ad,publisher info)
Given: Massive amounts of labeled data

Classification
Goal: learn a mapping from observations to discrete labels given a set of training examples (supervised learning)
Example: Click-through Rate Prediction
- Observations are user-ad-publisher triples
- Labels are {not-click, click}
- Given a set of labeled observations, we want to predict whether a new user-ad-publisher triple will result in a click

Evaluating predictions
- Regression: can measure 'closeness' between labels and prediction
- classification: class predictions are discrete
0-1 loss: Penalty is 0 for correct prediction, and 1 otherwise

How can we learn model (w)?
Assume we have n training points, where x^i denotes the ith point
Idea: Find w that minimize average 0-1 loss over training points:
$\min_w\sum_{i=1}^n \ell_{0/1}(y_i \cdot w^\top x^i)$
We use 0-1 loss: $\ell_{0/1}(z)$

The original 0/1 loss minimization is hard optimization, not convex
Approximate 0/1 loss
SVM(hinge), logistic regression (logistic), adaboost (exponential)



Solution: approximate 0/1 loss with convex loss

Logistic loss (logloss): $l_{log}(z) = log(1 + e^{-z})$

Goal: Find w^* that minimizes
$$f(w) = \sum_{i=1}^{n} l_{log}(y^i \cdot w^T x^i)$$
Can solve visa Gradient Descent
Update rule: $w_{i+1} = w_i - \alpha \nabla f(w)$
$\sum_{j=1}^n [1- \frac{1}{1+\exp(-y^i w_i^\top x^j)}] (-y^i x^i)$

Logistic Regression: Learn mapping (w) that minimize logistic loss on training data
$$min_w \sum_{i=1}^{n} l_{log}(y^{(i)} \cdot w^T x^{(i)})$$
- Convex
- Closed form solution doesn't exist
- Can add regularization term as in ridge regression

Logistic regression: Probabilistic interpretation

Goal: Model conditional probability: P(y=1|x)
Example: Predict click from ad's historical performance, user's click frequency, and publisher page's relevance
P[y=click | h= GOOD, f = HIGH, r = HIGH] = .1]
P[y=click | h= BAD, f = LOW, r = HIGH] = .05]

Logistic regression uses logistic function to model this conditional probability
- P(y=1|x) = \sigma(w^\top x)
- P(y=0|x) = 1 - \sigma(w^\top x)

$$\min_{\mathbf{w}} \sum_{i=1}^{n} \overbrace{\ell_{0/1}\left(y^{(i)} \cdot \mathbf{w}^\top \mathbf{x}^{(i)}\right)}^{\text{Training LogLoss}} + \overbrace{\lambda ||\mathbf{w}||_2^2}^{\text{Model Complexity}}$$

Decision boundary: w^\top x = 0
- P(y=1|x) = \sigma(w^\top x) > .5 => \hat{y} = 1

Categorical Data and one-hot-encoding

Data is assumed to be numerical

One idea: Create single numerical feature to represent non-numeric one
Creating single numerical feature introduces relationships between categories that don't other otherwise exist

One-hot-encoding: Creating dummy features does not introduce spurious relationships

Feature hashing
Problem: Number of dummy features equals number of categories => high dimensionality

Feature hashing
- Use hashing principles to reduce feature dimension
- Obviates need to compute expensive OHE dictionary
- Preserves sparsity
- Theoretical underpinning

Hash function: Maps an object to one of m buckets
- Should be efficient and distribute objects across buckets

Reasonable
Hash feature have nice theoretical properties
- Good approximations of inner products of OHE features under certain conditions

- Many learning methods (including linear/logistic regression) can be viewed solely in terms of inner products
- Good empirical performance

Distributed computation
trainHash = train.map(applyHashFunction)
Step 1: Apply hash function on raw data
- Local computation and hash function are usually fast
- No need to compute OHE features or communication

Step 2: Store hashed features in sparse representation
- Local computation
- Saves storage and speeds up computation

Distributed PCA

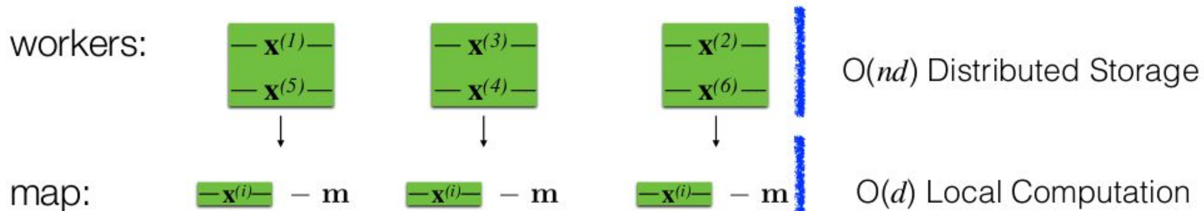
Brain
~50,000 neurons per cubic millimeter

Computing PCA solution
Given: n x d matrix of uncentered raw data
Goal: compute k << d dimensional representation

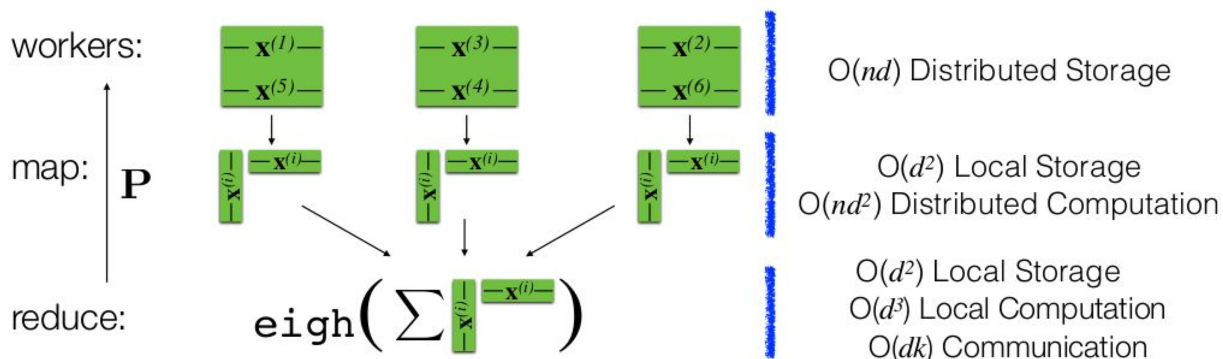
PCA steps
Step 1: Center Data
Step 2: Compute covariance or scatter matrix
- $\frac{1}{n}(X^T X)^{-1} X^T X$
Step 3: Eigendecomposition
Step 4: compute PCA Scores

PCA at scale
Case 1: Big n and small d
- O(d^2) local storage, O(d^3) local computation, O(dk) communication
- Similar strategy as closed-form linear regression

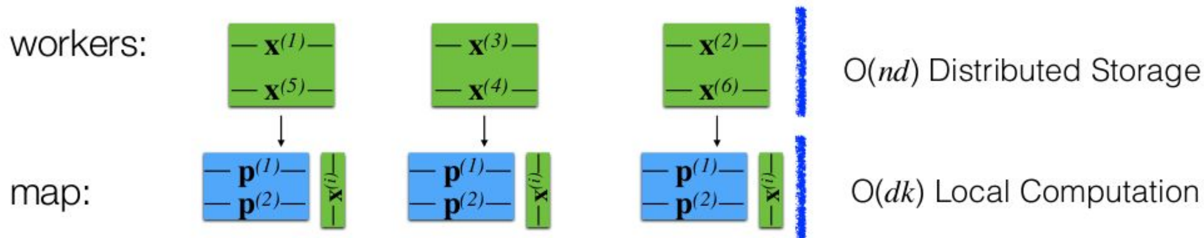
Step 1: Center Data
- Compte d feature means, m\in R^d
- communicate m to all workers
- Subtract m from each data point



Step 2: Compute covariance or scatter matrix
- Compute matrix product via outer products
Step 3: Eigendecomposition
- Perform locally since d is small
- Communicate k principal components (P \in R^{dxk}) to workers



Step 4: Compute PCA scores
- Multiply each point by principal components, P



Case 2: Big n and big d
- O(d) local storage and computation on workers, O(dk) communication
- Iterative algorithm

Step 1: Center Data
Rely on a sequence of matrix-vector products to compute top k eigenvectors (P)
- Krylov subspace or random random projection methods

Krylov subspace methods iteratively compute X^\top Xv for some v\in R^d provided by the method
- Requires O(k) passes over data, O(d) local storage on workers

- No need to compute the covariance matrix

Step 2: Compute covariance or scatter matrix

- $\frac{1}{n}X^T X$ vs $X^T X$

Repeat for O(k) iterations:

1. Communicate $v_i \in R^d$ to all workers
2. Compute $q_i = X^T X v_i$ in a distributed fashion

- Step 1: $b_i = X v_i$
- Step 2: $q_i = X^T b_i$
- Perform in single map-reduce

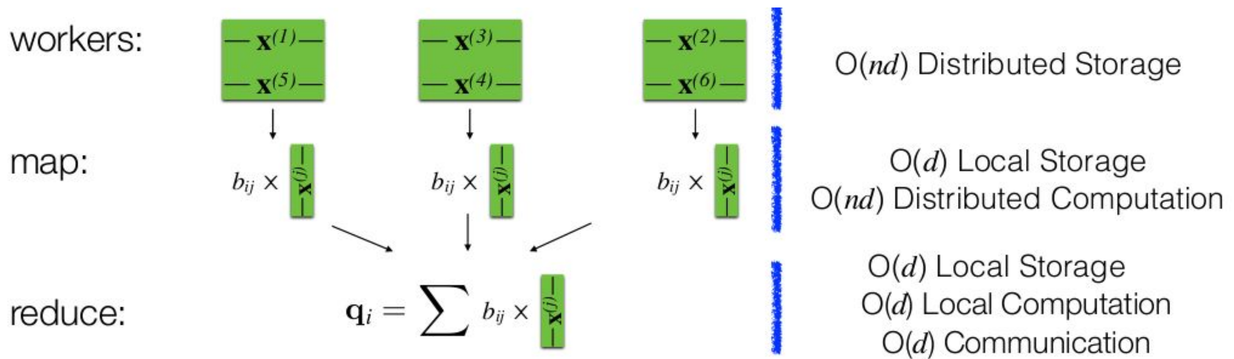
3. Drive uses q_i to update estimate of P

$b_{ij} v_i^T x^j$: each component is dot product

- q_i is a sum of rescaled data points, i.e. $q_i = \sum_{j=1}^{n} b_{ij} x^j$

Compute q_i = X^\top X v_i in a distributed fashion

- $b_{ij} = v_i^T x^j$ and $q_i = \sum_{j=1}^{n} b_{ij} x^j$
- Locally compute each dot product and rescale each point before summing all rescaled points in reduced step



Code:

q = trainData.map(rescaleByBi).reduce(sumVectors)

Step 3: Eigendecomposition

Step 4: compute PCA Scores