# ST446 Distributed Computing for Big Data

Lecture 7

## Scalable machine learning
## Part I


Milan Vojnovic

https://github.com/lse-st446/lectures2021

# Goals of this lecture

- Learn about distributed computing for training large-scale machine learning models
- Learn about iterative optimization algorithms and their convergence and scalability properties (gradient descent, stochastic gradient descent, BFGS, L-BFGS)
- Learn about distributed systems for training large-scale machine learning models

# Topics of this lecture

- Introduction to loss function minimization
- Gradient descent algorithm
- Stochastic gradient descent algorithm
- Newton and quasi-Newton methods

# Introduction to loss function minimization

# Motivating example: click through rate prediction

- Prediction task: predict whether a user will click on an ad
- Example: Criteo **1TB** click log example

```
<label>, <int feature 1>, …, <int feature 13>, <categorial feature 1>, …, <categorical feature 26>
```

- Supervised learning task with many training examples
  - **24 days**, subsampled, positive (click) and negative (no click) examples
  - Day 0: **15.1GB** compressed, **49.77GB** raw data size
    **196M** examples, **6.31M** positive examples (**3.21%**)
- Need to scale out the computation
- Dataset available from: https://labs.criteo.com/2013/12/download-terabyte-click-logs/

# Example: click through rate prediction (cont'd)

```
LSE021353:criteo-1TB vojnovic$ curl -O http://azuremlsampleexperiments.blob.core.windows.net/criteo/day_0.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 15.1G  100 15.1G    0     0  2846k      0  1:33:16  1:33:16 --:--:-- 3345k
```

```
LSE021353:criteo-1TB vojnovic$ more day_0
1       5       110             16              1       0       14      7       1               306             62770d79
e21f5d58        afea442f        945c7fcf        38b02748        6fcd6dcb        3580aa21        28808903        46dedfa6
2e027dc1        0c7c4231        95981d1f        00c5ffb7        be4ee537        8a0b74cc        4cdc3efa        d20856aa
b8170bba        9512c20b        c38e2f28        14f65a5d        25b1b089        d7c1fc0b        7caf609c        30436bfc
ed10571d
0       32      3       5               1       0       0       61      5       0       1       3157    5       e5f3fd8d
a0aaffa6        6faa15d5        da8a3421        3cd69f23        6fcd6dcb        ab16ed81        43426c29        1df5e154
7de9c0a9        6652dc64        99eb4e27        00c5ffb7        be4ee537        f3bbfe99        4cdc3efa        d20856aa
a1eb1511        9512c20b        febfd863        a3323ca1        c8e1ee56        1752e9e8        75350c8a        991321ea
b757e957
0               233     1       146     1       0       0       99      7       0       1       3101    1       62770d79
ad984203        62bec60d        386c49ee        e755064d        6fcd6dcb        b5f5eb62        d1f2cc8b        2e4e821f
2e027dc1        0c7c4231        12716184        00c5ffb7        be4ee537        f70f0d0b        4cdc3efa        d20856aa
628f1b8d        9512c20b        c38e2f28        14f65a5d        25b1b089        d7c1fc0b        34a9b905        ff654802
ed10571d
```

# Example: click through rate prediction (cont'd)

- Number of examples

```
LSE021353:criteo-1TB vojnovic$ wc -l day_0
 195841983 day_0
```

- Number of positive examples

```
LSE021353:criteo-1TB vojnovic$ grep '^1' day_0 | wc -l
 6286525
```

# Logistic regression

- Training examples: $(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_m, y_m)$
  where $\boldsymbol{x}_i \in \mathbf{R}^n$ is a feature vector and $y_i \in \{0,1\}$ is a label

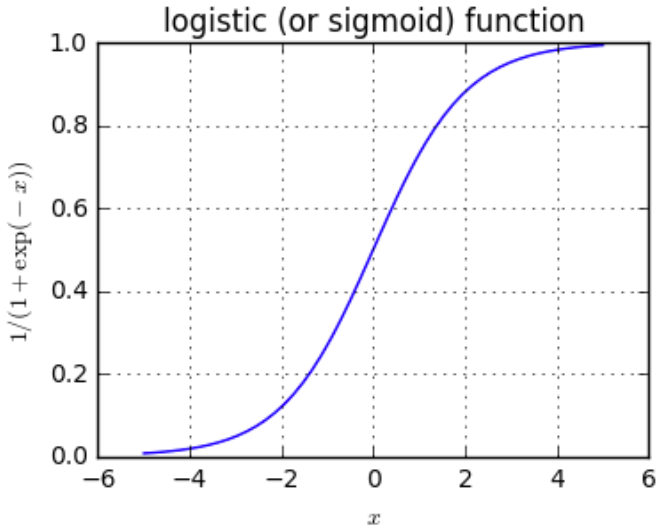
- Prediction problem: learn the function $g(\boldsymbol{x}; \boldsymbol{w}) = \mathbf{P}[\, y_i = 1 \mid \boldsymbol{x}_i = \boldsymbol{x} \,]$
  where $\boldsymbol{w} \in \mathbf{R}^n$ is a parameter vector

- Logistic regression model:

$$g(\boldsymbol{x}; \boldsymbol{w}) = \sigma(\boldsymbol{x}^\top \boldsymbol{w})$$

where $\sigma(x) = 1/(1 + \exp(-x))$ is the sigmoid function

Linear classification rule: $\log\left(\frac{\mathbf{P}[y_i=1 \mid \boldsymbol{x}_i=\boldsymbol{x}]}{\mathbf{P}[y_i=0 \mid \boldsymbol{x}_i=\boldsymbol{x}]}\right) = \boldsymbol{x}^\top \boldsymbol{w}$



# Loss function: negative log-likelihood

- Log-likelihood function:

$$\ell(\boldsymbol{w}) = \sum_{i=1}^{m}[y_i \log\left(\sigma\left(\boldsymbol{x}_i^\top \boldsymbol{w}\right)\right) + (1 - y_i)\log(1 - \sigma\left(\boldsymbol{x}_i^\top \boldsymbol{w}\right))]$$

$$= \sum_{i=1}^{m}[y_i \boldsymbol{x}_i^\top \boldsymbol{w} - \log(1 + e^{\boldsymbol{x}_i^\top \boldsymbol{w}})]$$

- Loss function: negative log-likelihood $f(\boldsymbol{w}) = -\ell(\boldsymbol{w})$

- For logistic regression, $f$ is a convex function
  - No local minima problems

- Not all loss functions are convex
  - Ex. Matrix factorization models for collaborative filtering (next lecture)

# Background: convex functions

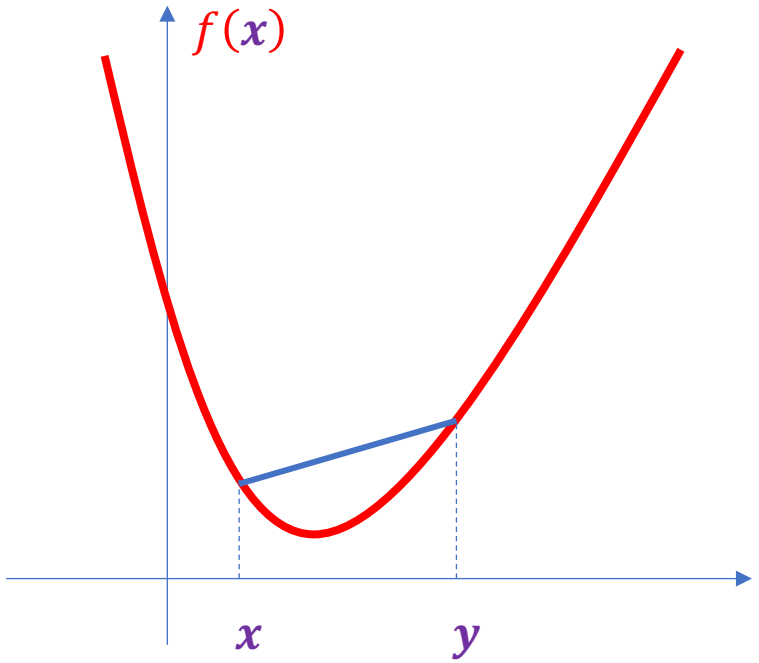
- A function $f: \mathbf{R}^n \to \mathbf{R}$ is convex on $X \subseteq \mathbf{R}^n$ if every chord of $f$ lies above it, i.e. for all $x, y \in X$

  $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$ for all $\lambda \in [0,1]$

  $f$ is strictly convex if the inequality is strict for all $\lambda \in (0,1)$

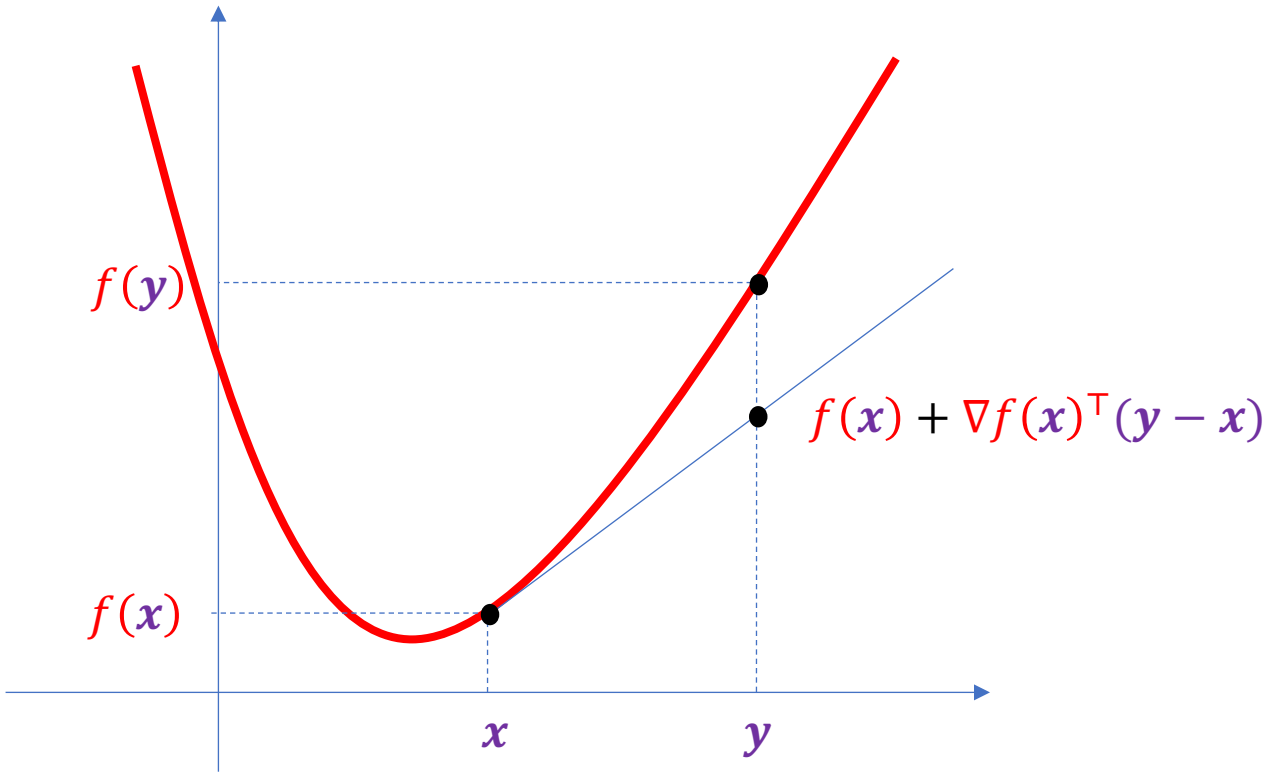
- Important property for convex functions: every local minima is globally optimal

- If $f$ is strictly convex, then it has a unique global minima on any compact (closed and bounded) convex set



# Background: convex differentiable functions

- $f$ is convex on $X$ if $f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^{\top}(\boldsymbol{y} - \boldsymbol{x})$ for all $\boldsymbol{x}, \boldsymbol{y} \in X$



# Background: convex twice-differentiable functions

- If $f$ is twice-differentiable, $f$ is convex on $X$ if, and only if, the Hessian matrix $\nabla^2 f(\boldsymbol{x})$ is positive semidefinite for all $\boldsymbol{x} \in X$
  - Hessian: $\nabla^2 f(\boldsymbol{x}) = \left[\frac{\partial^2}{\partial x_i \partial x_j} f(\boldsymbol{x})\right]$
- A real symmetric matrix $\boldsymbol{A}$ is positive semidefinite (PSD) if all its eigenvalues are *non-negative*, and it is positive definite (PD) if all its eigenvalues are *positive*
  - $\boldsymbol{A}$ PSD $\Leftrightarrow \boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} \geq 0$ for all $\boldsymbol{x} \neq \boldsymbol{0}$
  - $\boldsymbol{A}$ PD $\Leftrightarrow \boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x} > 0$ for all $\boldsymbol{x} \neq \boldsymbol{0}$

# Gradient descent algorithm

# Gradient descent algorithm

- Gradient descent algorithm: for given initial value $\boldsymbol{w}^{(0)} \in \mathbf{R}^n$, defined by the iterative update rule, for $t \geq 0$,

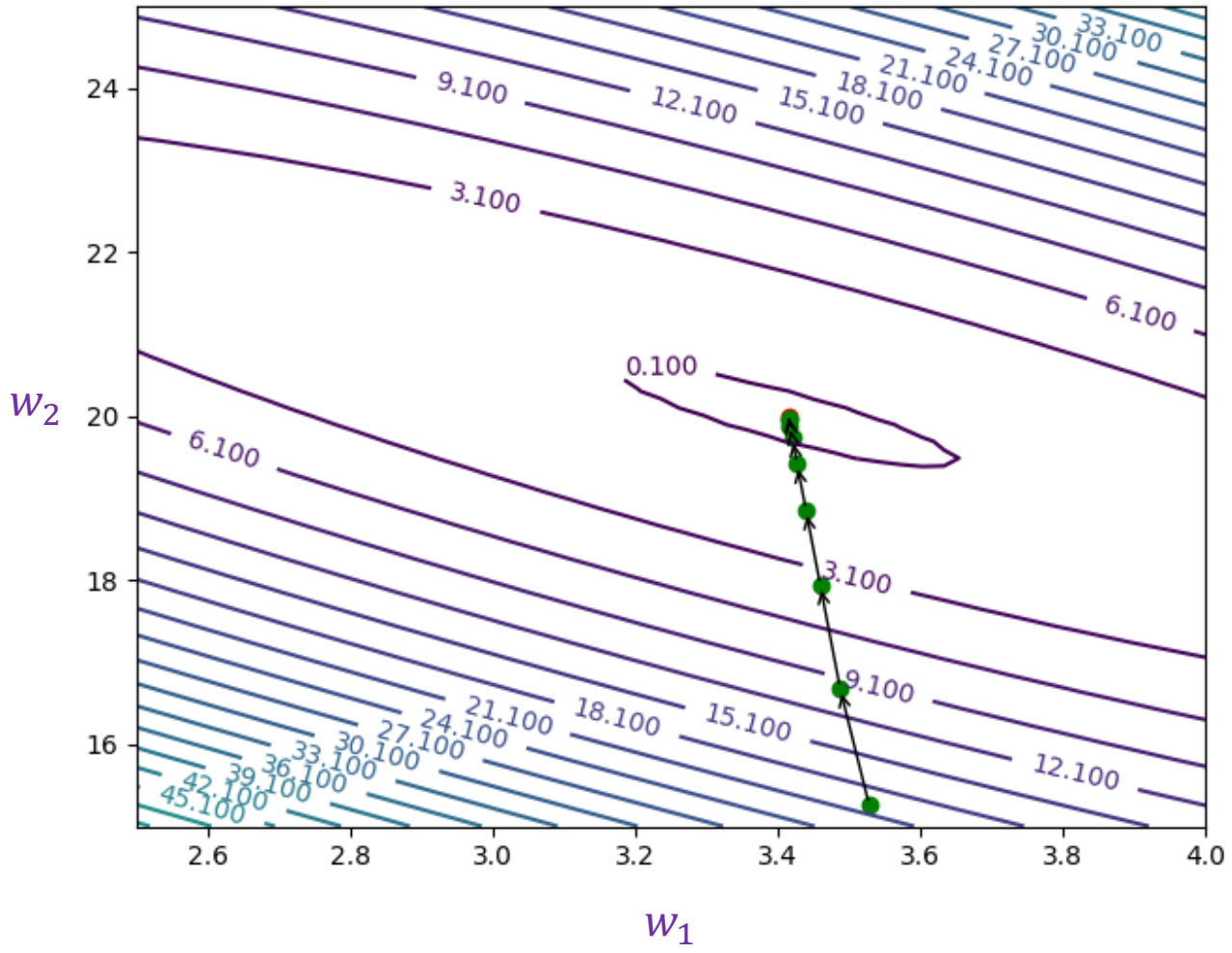
$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta_t \nabla f\left(\boldsymbol{w}^{(t)}\right)$$

  where $\eta_t$ is the step size (either constant or slowly decreasing in $t$, e.g. $\eta_t = \min\{\frac{c}{\sqrt{t}}, 1\}$ or $\eta_t = \min\{\frac{c}{t}, 1\}$, for some constant $c > 0$) and $\nabla f(\boldsymbol{w}^{(t)})$ is the gradient vector

- Ex. for logistic regression, the gradient descent update rule is as follows:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \eta_t \sum_{i=1}^{m} \left(y_i - \sigma\left(\boldsymbol{x}_i^\top \boldsymbol{w}^{(t)}\right)\right) \boldsymbol{x}_i$$

# Convergence illustration



# Convergence rates for gradient descent

- A point $\boldsymbol{w} \in W$ is said to be an $\epsilon$-optimal point on $W$ if $f(\boldsymbol{w}) - f(\boldsymbol{w}^*) \leq \epsilon$, where $\boldsymbol{w}^*$ is a minima point of $f$ on $W$
- Let $t_\epsilon(\boldsymbol{w}^{(0)})$ be the smallest number of iterations $t$ such that $f\left(\boldsymbol{w}^{(t)}\right) - f(\boldsymbol{w}^*) \leq \epsilon$
- Let $t_\epsilon(R) = \max_{\boldsymbol{w}^{(0)}:\, \|\boldsymbol{w}^{(0)} - \boldsymbol{w}^*\| \leq R} t_\epsilon(\boldsymbol{w}^{(0)})$

- Then, we have
  - If $f$ is convex and smooth gradient: $t_\epsilon(R) = O\left(\frac{1}{\epsilon^2}\right)$
  - If $f$ is convex and smooth: $t_\epsilon(R) = O\left(\frac{1}{\epsilon}\right)$
  - If $f$ is strongly convex and smooth: $t_\epsilon(R) = O\left(\log\left(\frac{1}{\epsilon}\right)\right)$

- For $f$ twice-differentiable:
  - $f$ is $\beta$-smooth means that the largest eigenvalue of $\nabla^2 f(\boldsymbol{w})$ is $\leq \beta$
  - $f$ is $\alpha$-strongly convex means that the smallest eigenvalue of $\nabla^2 f(\boldsymbol{w})$ is $\geq \alpha$

# Projected gradient descent

- Projected gradient descent update rule:

$$\boldsymbol{w}^{(t+1)} = \Pi_W \left( \boldsymbol{w}^{(t)} - \eta_t \nabla f\left(\boldsymbol{w}^{(t)}\right) \right)$$

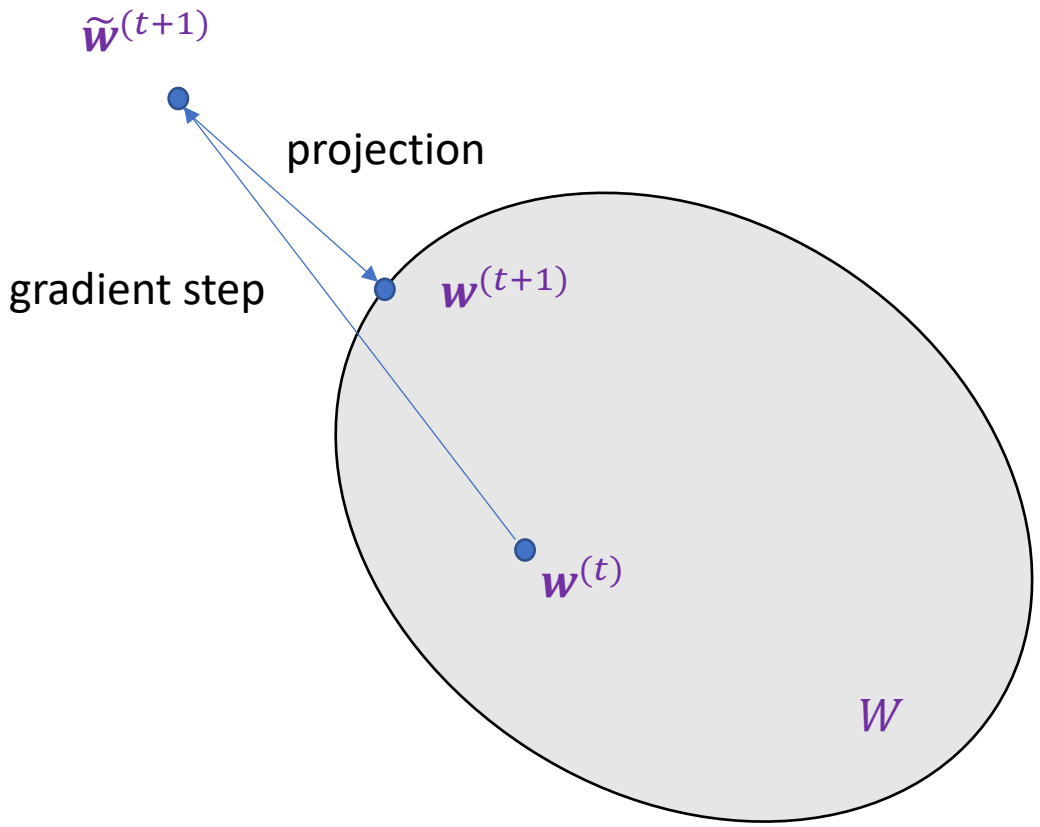
where $W$ is a convex and compact case and

$$\Pi_W(\boldsymbol{w}) = \arg\min_{\boldsymbol{w}' \in W} \|\boldsymbol{w} - \boldsymbol{w}'\|$$

$\Pi_W(\boldsymbol{w})$ is a projection transformation ensuring that the update lies in $W$ by projecting back (if necessary) onto it

- Notation: $\widetilde{\boldsymbol{w}}^{(t+1)} := \boldsymbol{w}^{(t)} - \eta_t \nabla f(\boldsymbol{w}^{(t)})$

# Projected gradient descent (cont'd)



# GD convergence rate: convex, smooth gradient

- Thm. Assume that $W$ is in the Euclidean ball with center $\boldsymbol{w}^{(0)}$ and radius $R$ and that $f$ is such that there exists $L > 0$ such that $\|\nabla f(\boldsymbol{w})\| \leq L$ for all $\boldsymbol{w} \in W$, and the step size is constant $\eta_t = \eta > 0$ for all $t \geq 0$.

  The projected-GD with $\eta = R/(L\sqrt{t})$ satisfies

$$f\left(\frac{1}{t}\sum_{s=0}^{t-1}\boldsymbol{w}_s\right) - f(\boldsymbol{w}^\star) \leq \frac{RL}{\sqrt{t}}$$

- Hence, for $f\left(\frac{1}{t}\sum_{s=0}^{t-1}\boldsymbol{w}_s\right) - f(\boldsymbol{w}^\star) \leq \epsilon$ to hold it suffices that $t = O(\frac{1}{\epsilon^2})$
- Tight: in order to reach an $\epsilon$-optimal point one needs $\Omega(\frac{1}{\epsilon^2})$ iterations a worst-case (Bubeck, Section 3.5)

# Proof sketch*

$$f\left(\boldsymbol{w}^{(s)}\right) - f(\boldsymbol{w}^\star) \leq \nabla f\left(\boldsymbol{w}^{(s)}\right)^\top \left(\boldsymbol{w}^{(s)} - \boldsymbol{w}^\star\right)$$

$$= \frac{1}{\eta}\left(\boldsymbol{w}^{(s)} - \widetilde{\boldsymbol{w}}^{(s+1)}\right)^\top \left(\boldsymbol{w}^{(s)} - \boldsymbol{w}^\star\right)$$

$$= \frac{1}{2\eta}\left(\left\|\boldsymbol{w}^{(s)} - \boldsymbol{w}^\star\right\|^2 + \left\|\boldsymbol{w}^{(s)} - \widetilde{\boldsymbol{w}}^{(s+1)}\right\|^2 - \left\|\widetilde{\boldsymbol{w}}^{(s+1)} - \boldsymbol{w}^\star\right\|^2\right)$$

$$= \frac{1}{2\eta}\left(\left\|\boldsymbol{w}^{(s)} - \boldsymbol{w}^\star\right\|^2 - \left\|\widetilde{\boldsymbol{w}}^{(s+1)} - \boldsymbol{w}^\star\right\|^2\right) + \frac{\eta}{2}\left\|\nabla f\left(\boldsymbol{w}^{(s)}\right)\right\|^2$$

$$\leq \frac{1}{2\eta}\left(\left\|\boldsymbol{w}^{(s)} - \boldsymbol{w}^\star\right\|^2 - \left\|\widetilde{\boldsymbol{w}}^{(s+1)} - \boldsymbol{w}^\star\right\|^2\right) + \frac{\eta}{2}L^2$$

$$\leq \frac{1}{2\eta}\left(\left\|\boldsymbol{w}^{(s)} - \boldsymbol{w}^\star\right\|^2 - \left\|\boldsymbol{w}^{(s+1)} - \boldsymbol{w}^\star\right\|^2\right) + \frac{\eta}{2}L^2 \qquad \text{(Bubeck Lemma 3.1)}$$

# Proof sketch (cont'd)*

- Summing up, we have

$$\sum_{s=0}^{t-1}(f(\boldsymbol{w}^{(s)}) - f(\boldsymbol{w}^{\star})) \leq \frac{R^2}{2\eta} + \frac{\eta L^2 t}{2}$$

- Plug in the value of $\eta$ and use, by Jensen's inequality,

$$f\left(\frac{1}{t}\sum_{s=0}^{t-1}\boldsymbol{w}^{(s)}\right) \leq \frac{1}{t}\sum_{s=0}^{t-1} f(\boldsymbol{w}^{(s)})$$

- To probe further: Jensen's inequality see here

# Smooth convex case

- A function $f$ is said to be $\beta$-smooth if $\|\nabla f(\boldsymbol{y}) - \nabla f(\boldsymbol{x})\| \leq \beta \|\boldsymbol{y} - \boldsymbol{x}\|$
  - If $f$ is twice-differentiable, $\nabla^2 f(\boldsymbol{x}) \preccurlyeq \beta \mathbf{I}$

- Thm. Suppose $f$ is convex and $\beta$-smooth and the step size is such that $\eta = 1/\beta$. Then, gradient descent algorithm satisfies

$$f\left(\boldsymbol{w}^{(t)}\right) - f(\boldsymbol{w}^\star) \leq 2\beta \left\|\boldsymbol{w}^{(0)} - \boldsymbol{w}^\star\right\|^2 \frac{1}{t-1}$$

- Hence, the number of iterations $O\left(\frac{1}{\epsilon}\right)$

- To probe further: Bubeck Thm 3.3

# Smooth strongly convex case

- A function is said to be $\alpha$-strongly convex if $f(\boldsymbol{y}) \geq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^{\mathsf{T}}(\boldsymbol{y} - \boldsymbol{x}) + \frac{\alpha}{2}\|\boldsymbol{y} - \boldsymbol{x}\|^2$
  - If $f$ is twice-differentiable $\nabla^2 f(\boldsymbol{x}) \succcurlyeq \alpha \mathbf{I}$

- Thm. Suppose $f$ is $\alpha$-strongly convex and $\beta$-smooth and $\eta = 2/(\alpha + \beta)$. Then, gradient descent algorithm satisfies

$$f\left(\boldsymbol{w}^{(t+1)}\right) - f(\boldsymbol{w}^\star) \leq \frac{\beta}{2}\left\|\boldsymbol{w}^{(0)} - \boldsymbol{w}^\star\right\|^2 \exp\left(-4\frac{\alpha}{\alpha + \beta}t\right)$$

- Hence, the number of iterations $O(\log(1/\epsilon))$

- To probe further: Bubeck Thm 3.12

# Stochastic gradient descent algorithm

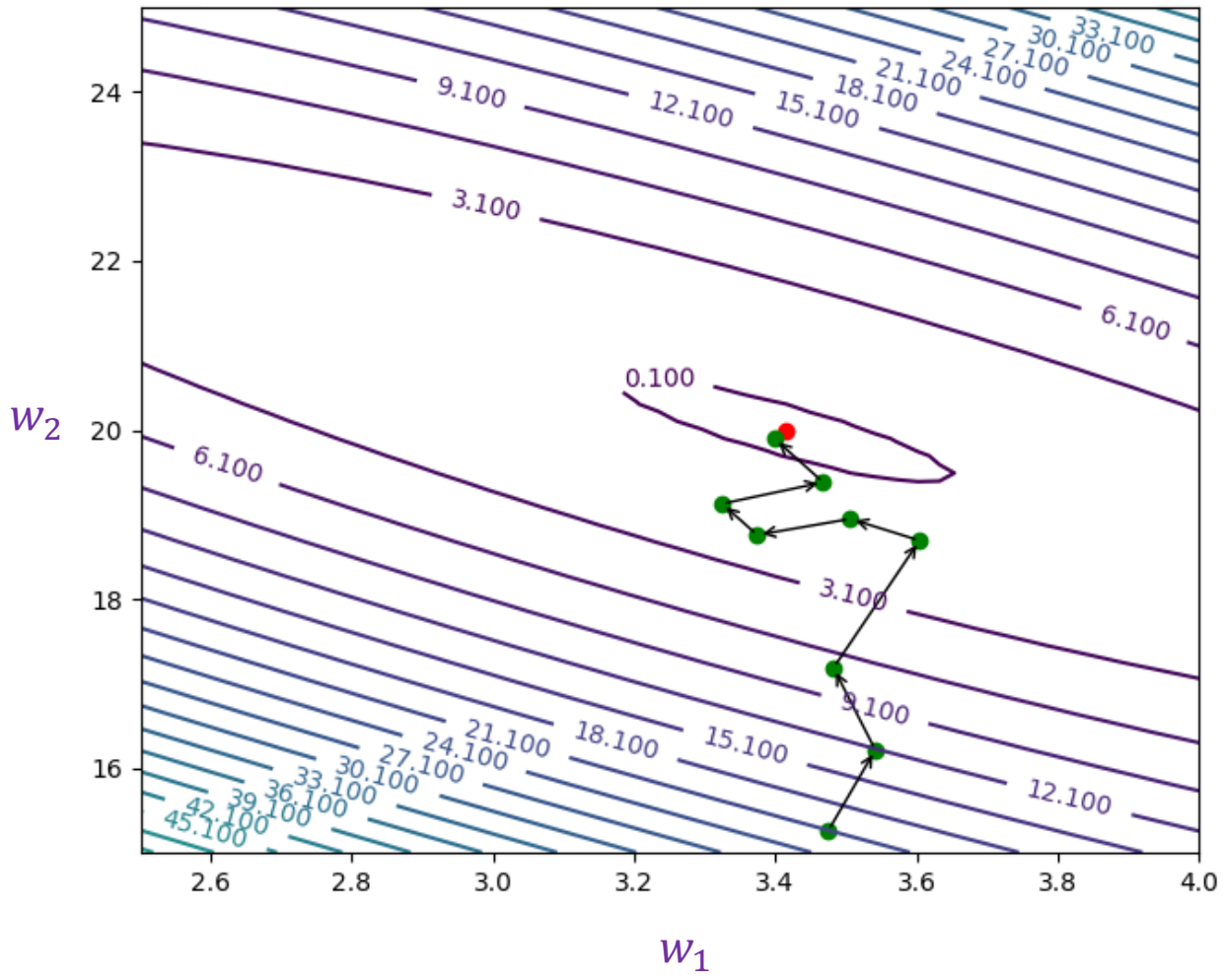
# Stochastic Gradient Descent (SGD)

- Computing the gradient is expensive as it requires summing gradients of loss functions associated with all training examples

$$\nabla f(\boldsymbol{w}; \boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{m} \nabla f_i(\boldsymbol{w}; \boldsymbol{x}_i, y_i)$$

- Stochastic Gradient Descent (SGD): use stochastic gradient vector $\widehat{\boldsymbol{g}}_t$ at iteration $t$
- Desiderata:
  - Unbiased estimator: $\mathbf{E}[\widehat{\boldsymbol{g}}_t(\boldsymbol{w})] = \nabla f(\boldsymbol{w})$
  - Small variance: $\sigma_t^2 = \mathbf{E}[\|\widehat{\boldsymbol{g}}_t(\boldsymbol{w}) - \nabla f(\boldsymbol{w})\|^2]$
- SGD methods:
  - One sample: $\widehat{\boldsymbol{g}}_t(\boldsymbol{w}) = \nabla f_{I_t}(\boldsymbol{w}; \boldsymbol{x}_{I_t}, y_{I_t})$ where $I_t$ is a randomly sampled example
  - Mini-batch SGD: $\widehat{\boldsymbol{g}}_t(\boldsymbol{w}) = \frac{1}{k}\sum_{i \in S_t} \nabla f_i(\boldsymbol{w}; \boldsymbol{x}_i, y_i)$ where $S_t$ is a random sample of $k$ examples from the training set

# Convergence illustration



# Convergence rate of SGD

- Thm. Suppose $f$ is convex and $\beta$-smooth, $W$ is a convex set contained in the ball of radius $R$ with center $\boldsymbol{w}^{(0)}$ and that there exists $\sigma > 0$ such that $\mathbf{E}[\|\widehat{\boldsymbol{g}}_t(\boldsymbol{w}) - \nabla f(\boldsymbol{w})\|^2] \leq \sigma^2$ for all $\boldsymbol{w} \in W$.

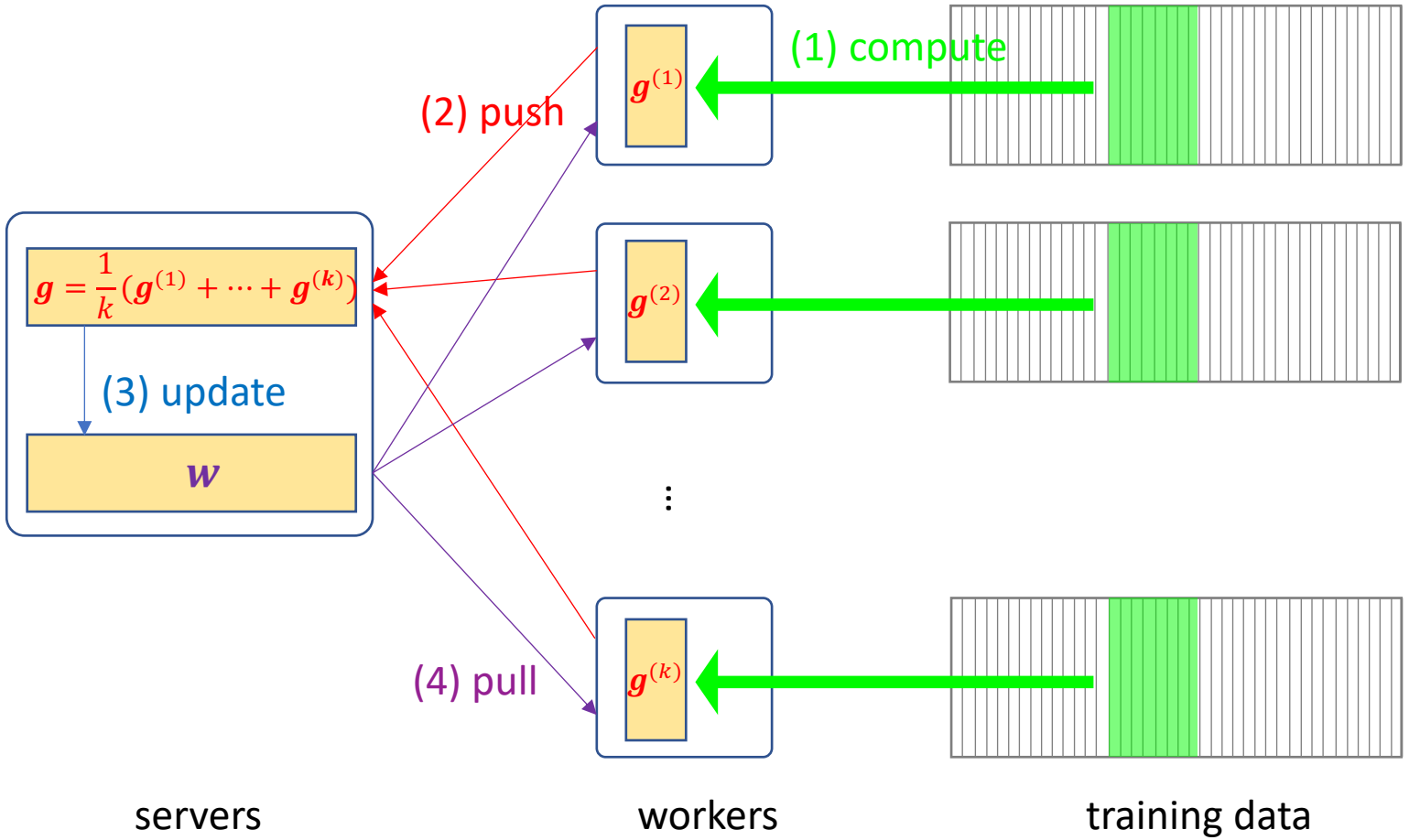
  Then, the SGD with step size $\eta_t = 1/(\beta + \sigma/R\sqrt{2}\sqrt{t})$, it holds

$$\mathbf{E}\left[f\left(\frac{1}{t}\sum_{s=1}^{t}\boldsymbol{w}^{(s)}\right)\right] - f(\boldsymbol{w}^*) \leq \frac{R\sigma\sqrt{2}}{\sqrt{t}} + \frac{\beta R^2}{t}$$

- This implies the number of iterations $O(1/\epsilon^2)$

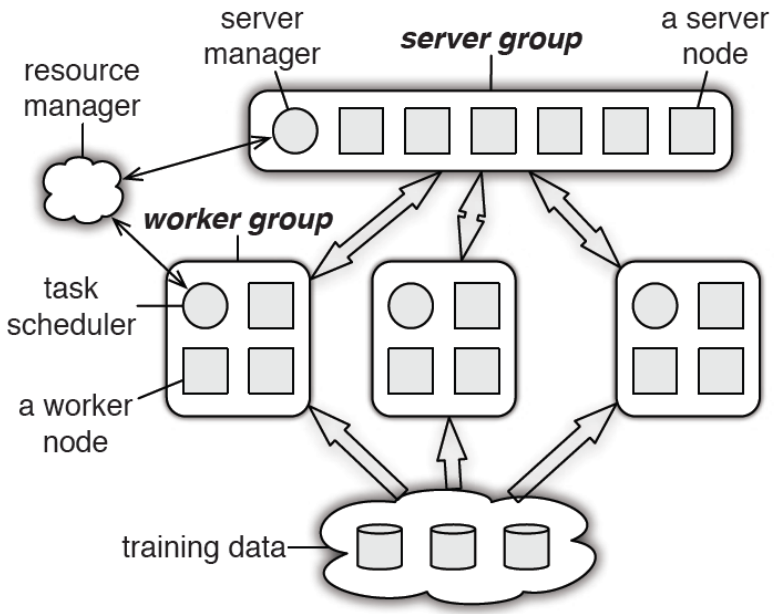
# Distributed GD / SGD

- Data parallel model: training data partitioned among all the worker nodes
  - Computing the loss function gradients is divided among all workers



# Parameter server

- Parameter server: system architecture introduced for training topic modelling algorithms (VLDB 2010)
- Server nodes: maintain globally shared parameters, represented as dense or sparse vectors and matrices
- Worker nodes: have access to data (training examples) and are responsible for computing gradients of the loss function



# Task scheduler

```
Issue loadData() to all workers

for t = 0, 1, …, T do

      Issue workerIterate(t) to all workers

end for
```

# Worker w

**function** loadData()

Load $(x_i, y_i)$ for $i \in N_w$ // load part of training data

Pull $w^{(0)}$ from server

**function** workerIterate(t)

$g_t^{(w)} = \sum_{i \in N_w} \nabla f(w^{(t)}; x_i, y_i)$ // compute gradient vector update

Push $g_t^{(w)}$ to server

Pull $w^{(t+1)}$ from server

# Server

**function** serverIterate(t)

$\boldsymbol{g}_t = \sum_{w=1}^{k} \boldsymbol{g}_t^{(w)}$ // aggregate gradient updates

$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \eta(\boldsymbol{g}_t + \lambda \nabla \Omega(\boldsymbol{w}^{(t)}))$ // update the parameter vector

Regularizer (ex. L1 or L2)

# Logistic regression in Spark

```
from pyspark.ml.classification import LogisticRegression

# Load training data
training = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.tx
t")

lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Fit the model
lrModel = lr.fit(training)

# Print the coefficients and intercept for logistic regression
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))

# We can also use the multinomial family for binary classification
mlr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8, family
="multinomial")

# Fit the model
mlrModel = mlr.fit(training)

# Print the coefficients and intercepts for logistic regression with multinomia
l family
print("Multinomial coefficients: " + str(mlrModel.coefficientMatrix))
print("Multinomial intercepts: " + str(mlrModel.interceptVector))
```

- To probe further: https://spark.apache.org/docs/2.3.0/ml-classification-regression.html#logistic-regression

# SGD in Spark

```scala
/**
 * :: DeveloperApi ::
 * Top-level method to run gradient descent.
 */
@DeveloperApi
object GradientDescent extends Logging {
  /**
   * Run stochastic gradient descent (SGD) in parallel using mini batches.
   * In each iteration, we sample a subset (fraction miniBatchFraction) of the total data
   * in order to compute a gradient estimate.
   * Sampling, and averaging the subgradients over this subset is performed using one standard
   * spark map-reduce in each iteration.
   *
   * @param data Input data for SGD. RDD of the set of data examples, each of
   *             the form (label, [feature values]).
   * @param gradient Gradient object (used to compute the gradient of the loss function of
   *                 one single data example)
   * @param updater Updater function to actually perform a gradient step in a given direction.
   * @param stepSize initial step size for the first step
   * @param numIterations number of iterations that SGD should be run.
   * @param regParam regularization parameter
   * @param miniBatchFraction fraction of the input data set that should be used for
   *                          one iteration of SGD. Default value 1.0.
   * @param convergenceTol Minibatch iteration will end before numIterations if the relative
   *                       difference between the current weight and the previous weight is less
   *                       than this value. In measuring convergence, L2 norm is calculated.
   *                       Default value 0.001. Must be between 0.0 and 1.0 inclusively.
   * @return A tuple containing two elements. The first element is a column matrix containing
   *         weights for every feature, and the second element is an array containing the
   *         stochastic loss computed for every iteration.
   */
  def runMiniBatchSGD(
      data: RDD[(Double, Vector)],
      gradient: Gradient,
      updater: Updater,
      stepSize: Double,
      numIterations: Int,
      regParam: Double,
      miniBatchFraction: Double,
      initialWeights: Vector,
      convergenceTol: Double): (Vector, Array[Double]) = {

    // convergenceTol should be set with non minibatch settings
    if (miniBatchFraction < 1.0 && convergenceTol > 0.0) {
      logWarning("Testing against a convergenceTol when using miniBatchFraction " +
        "< 1.0 can be unstable because of the stochasticity in sampling.")
    }
```

```scala
    var regVal = updater.compute(
      weights, Vectors.zeros(weights.size), 0, 1, regParam)._2

    var converged = false // indicates whether converged based on convergenceTol
    var i = 1
    while (!converged && i <= numIterations) {
      val bcWeights = data.context.broadcast(weights)
      // Sample a subset (fraction miniBatchFraction) of the total data
      // compute and sum up the subgradients on this subset (this is one map-reduce)
      val (gradientSum, lossSum, miniBatchSize) = data.sample(false, miniBatchFraction, 42 + i)
        .treeAggregate((BDV.zeros[Double](n), 0.0, 0L))(
          seqOp = (c, v) => {
            // c: (grad, loss, count), v: (label, features)
            val l = gradient.compute(v._2, v._1, bcWeights.value, Vectors.fromBreeze(c._1))
            (c._1, c._2 + l, c._3 + 1)
          },
          combOp = (c1, c2) => {
            // c: (grad, loss, count)
            (c1._1 += c2._1, c1._2 + c2._2, c1._3 + c2._3)
          })
      bcWeights.destroy()

      if (miniBatchSize > 0) {
        /**
         * lossSum is computed using the weights from the previous iteration
         * and regVal is the regularization value computed in the previous iteration as well.
         */
        stochasticLossHistory += lossSum / miniBatchSize + regVal
        val update = updater.compute(
          weights, Vectors.fromBreeze(gradientSum / miniBatchSize.toDouble),
          stepSize, i, regParam)
        weights = update._1
        regVal = update._2

        previousWeights = currentWeights
        currentWeights = Some(weights)
        if (previousWeights != None && currentWeights != None) {
          converged = isConverged(previousWeights.get,
            currentWeights.get, convergenceTol)
        }
      } else {
        logWarning(s"Iteration ($i/$numIterations). The size of sampled batch is zero")
      }
      i += 1
    }

    logInfo("GradientDescent.runMiniBatchSGD finished. Last 10 stochastic losses %s".format(
      stochasticLossHistory.takeRight(10).mkString(", ")))
```

- https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/optimization/GradientDescent.scala

# treeAggregate

```scala
/**
 * Aggregates the elements of this RDD in a multi-level tree pattern.
 * This method is semantically identical to [[org.apache.spark.rdd.RDD#aggregate]].
 *
 * @param depth suggested depth of the tree (default: 2)
 */
def treeAggregate[U: ClassTag](zeroValue: U)(
    seqOp: (U, T) => U,
    combOp: (U, U) => U,
    depth: Int = 2): U = withScope {
  require(depth >= 1, s"Depth must be greater than or equal to 1 but got $depth.")
  if (partitions.length == 0) {
    Utils.clone(zeroValue, context.env.closureSerializer.newInstance())
  } else {
    val cleanSeqOp = context.clean(seqOp)
    val cleanCombOp = context.clean(combOp)
    val aggregatePartition =
      (it: Iterator[T]) => it.aggregate(zeroValue)(cleanSeqOp, cleanCombOp)
    var partiallyAggregated: RDD[U] = mapPartitions(it => Iterator(aggregatePartition(it)))
    var numPartitions = partiallyAggregated.partitions.length
    val scale = math.max(math.ceil(math.pow(numPartitions, 1.0 / depth)).toInt, 2)
    // If creating an extra level doesn't help reduce
    // the wall-clock time, we stop tree aggregation.

    // Don't trigger TreeAggregation when it doesn't save wall-clock time
    while (numPartitions > scale + math.ceil(numPartitions.toDouble / scale)) {
      numPartitions /= scale
      val curNumPartitions = numPartitions
      partiallyAggregated = partiallyAggregated.mapPartitionsWithIndex {
        (i, iter) => iter.map((i % curNumPartitions, _))
      }.foldByKey(zeroValue, new HashPartitioner(curNumPartitions))(cleanCombOp).values
    }
    val copiedZeroValue = Utils.clone(zeroValue, sc.env.closureSerializer.newInstance())
    partiallyAggregated.fold(copiedZeroValue)(cleanCombOp)
  }
}
```

- https://github.com/apache/spark/blob/master/core/src/main/scala/org/apache/spark/rdd/RDD.scala#L1189

# Newton and quasi-Newton methods

# Newton method

- Suppose $f$ is twice-differentiable and consider the following quadratic approximation:

$$f(\boldsymbol{w}) \approx q_t(\boldsymbol{w}) \coloneqq f\left(\boldsymbol{w}^{(t)}\right) + \left(\boldsymbol{w} - \boldsymbol{w}^{(t)}\right)^\top \boldsymbol{g}_t + \frac{1}{2}\left(\boldsymbol{w} - \boldsymbol{w}^{(t)}\right)^\top \boldsymbol{H}_t \left(\boldsymbol{w} - \boldsymbol{w}^{(t)}\right)$$

where

$\boldsymbol{g}_t = \nabla f\left(\boldsymbol{w}^{(t)}\right)$ is the gradient vector, and

$\boldsymbol{H}_t = \nabla^2 f\left(\boldsymbol{w}^{(t)}\right)$ is the Hessian matrix

- Suppose we want to set $\boldsymbol{w}^{(t+1)}$ to be the minimizer of $q_t(\boldsymbol{w})$
- If $\boldsymbol{H}_t$ is positive semidefinite, then $\boldsymbol{w}^{(t+1)}$ such that $\nabla q_t\left(\boldsymbol{w}^{(t+1)}\right) = 0$ is a global optimum
- Since $\nabla q_t\left(\boldsymbol{w}^{(t+1)}\right) = 0 \Leftrightarrow \boldsymbol{g}_t + \boldsymbol{H}_t\left(\boldsymbol{w}^{(t+1)} - \boldsymbol{w}^{(t)}\right) = 0$, we have the direction for update

$$\boldsymbol{w}^{(t+1)} - \boldsymbol{w}^{(t)} = -\boldsymbol{H}_t^{-1}\boldsymbol{g}_t$$

# Newton-Raphson iterative algorithm

- Iterate until convergence:

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - a^{\star}\boldsymbol{H}_t^{-1}\boldsymbol{g}_t$$

  where

$$a^{\star} = \arg\min_{a>0} f(\boldsymbol{w}^{(t)} - a\boldsymbol{H}_t^{-1}\boldsymbol{g}_t)$$

- Remarks
  - Second-order method: uses not only the gradient vector but also the Hessian matrix
  - Can be much faster than first-order methods (ex. GD or SGD)

# Convergence rate of Newton method

- Thm. Suppose $f$ is twice-differentiable, $\alpha$-strongly convex, and $\|\nabla^2 f(\boldsymbol{x}) - \nabla^2 f(\boldsymbol{y})\| \leq L\|\boldsymbol{x} - \boldsymbol{y}\|$, for some $L > 0$. Then, there exist $0 < \eta \leq \alpha^2/L$ and $\gamma > 0$ such that
  - If $\left\|\nabla f\left(\boldsymbol{w}^{(t)}\right)\right\| \geq \eta$ then $f\left(\boldsymbol{w}^{(t+1)}\right) - f\left(\boldsymbol{w}^{(t)}\right) \leq -\gamma$
  - Otherwise, if $\left\|\nabla f\left(\boldsymbol{w}^{(t)}\right)\right\| < \eta$ then $\frac{L}{2\alpha^2}\left\|\nabla f\left(\boldsymbol{w}^{(t+1)}\right)\right\| \leq \left(\frac{L}{2\alpha^2}\left\|\nabla f\left(\boldsymbol{w}^{(t)}\right)\right\|\right)^2$ (quadratic convergence)
- To achieve $f\left(\boldsymbol{w}^{(t)}\right) - f(\boldsymbol{w}^\star) \leq \epsilon$ it suffices that

$$t = O\left(\frac{f\left(\boldsymbol{w}^{(0)}\right) - f(\boldsymbol{w}^\star)}{\gamma} + \log\left(\log\left(\frac{\delta}{\epsilon}\right)\right)\right)$$

  where $\delta = 2\alpha^3/L^2$

- To probe further: Ch 9.5, Boyd and Vandenberghe

# Computation complexity issues

- Newton method requires to compute the inverse of the Hessian matrix
- This is computationally too expensive for high-dimensional models
- Rarely used in practice for large-scale optimization problems
- Instead, quasi-Newton methods are used: approximate the inverse Hessian matrix

# Quasi-Newton methods

- **Input**: initial point $\boldsymbol{w}^{(0)}$, a positive definite matrix $\boldsymbol{B}_0$

- For $t = 0, 1, \ldots$

  Compute quasi-Newton direction $\delta \boldsymbol{w}_t = -\boldsymbol{B}_t \nabla f\left(\boldsymbol{w}^{(t)}\right)$
  Determine step size $\alpha_t$
  Update $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \alpha_t \delta \boldsymbol{w}_t$
  Compute $\boldsymbol{B}_{t+1}$

- Different update rules have different methods for updating $\boldsymbol{B}_{t+1}$

# BFGS


Source: http://aria42.com/blog/2014/12/understanding-lbfgs

# BFGS update

- BFGS update

$$\boldsymbol{H}_{t+1} = \boldsymbol{H}_t + \rho_t \boldsymbol{y}_t \boldsymbol{y}_t^\top - \frac{\boldsymbol{H}_t \boldsymbol{s}_t \boldsymbol{s}_t^\top \boldsymbol{H}_t}{\boldsymbol{s}_t^\top \boldsymbol{H}_t \boldsymbol{s}_t}$$

where

$\boldsymbol{y}_t = \boldsymbol{g}_{t+1} - \boldsymbol{g}_t$ (gradient change)

$\boldsymbol{s}_t = \boldsymbol{w}^{(t+1)} - \boldsymbol{w}^{(t)}$ (parameter estimate change)

$\rho_t = 1/\boldsymbol{y}_t^\top \boldsymbol{s}_t$

- Inverse update:

$$\boldsymbol{H}_{t+1}^{-1} = (\boldsymbol{I} - \rho_t \boldsymbol{s}_t \boldsymbol{y}_t^\top) \boldsymbol{H}_t^{-1} (I - \rho_t \boldsymbol{y}_t \boldsymbol{s}_t^\top) + \rho_t \boldsymbol{s}_t \boldsymbol{s}_t^\top$$

- Cost of update or inverse update is $O(n^2)$ operations
- Note $\boldsymbol{y}_t^\top \boldsymbol{s}_t = \left(\nabla f\left(\boldsymbol{w}^{(t+1)}\right) - \nabla f\left(\boldsymbol{w}^{(t)}\right)\right)^\top \left(\boldsymbol{w}^{(t+1)} - \boldsymbol{w}^{(t)}\right) > 0$ if $f$ is strictly convex

* Abuse of notation: $\boldsymbol{H}_t$ and $\boldsymbol{H}_t^{-1}$ denote *approximate* Hessian and inverse Hessian

# Approximating the Hessian

- The BFGS update satisfies the *secant condition*:

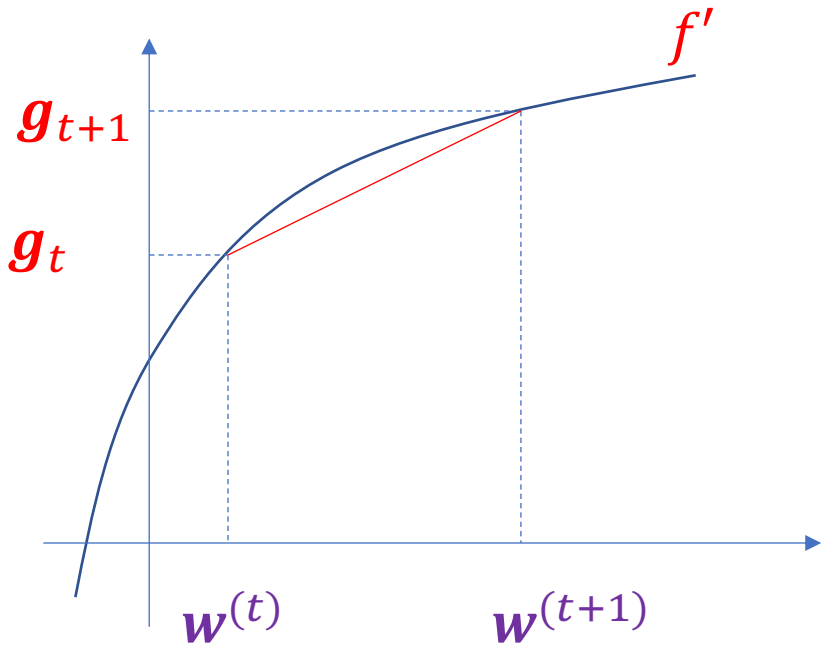
$$\boldsymbol{H}_{t+1}\boldsymbol{s}_t = \boldsymbol{y}_t$$

- For the quadratic approximation $q_{t+1}$ of $f$ at point $\boldsymbol{w}^{(t+1)}$, we have

$$\nabla q_{t+1}\left(\boldsymbol{w}^{(t+1)}\right) = \boldsymbol{g}_{t+1}$$

- Secant condition implies $\nabla q_{t+1}\left(\boldsymbol{w}^{(t)}\right) = \boldsymbol{g}_t$

$$\nabla q_{t+1}\left(\boldsymbol{w}^{(t)}\right) = \boldsymbol{g}_{t+1} + \boldsymbol{H}_{t+1}\left(\boldsymbol{w}^{(t)} - \boldsymbol{w}^{(t+1)}\right)$$
$$= \boldsymbol{g}_{t+1} - \boldsymbol{y}_t$$
$$= \boldsymbol{g}_t$$



# BFGS update (cont'd)

- Approximate Hessian update

$$\boldsymbol{H}_{t+1} = \boldsymbol{H}_t + \boldsymbol{U}_t + \boldsymbol{V}_t \qquad \text{(AHU)}$$

  where $\boldsymbol{U}_t$ and $\boldsymbol{V}_t$ are real symmetric, rank-one matrices

- Assume $\boldsymbol{U}_t = a\,\boldsymbol{u}\boldsymbol{u}^\top$ and $\boldsymbol{V}_t = b\,\boldsymbol{v}\boldsymbol{v}^\top$ with $\boldsymbol{u} = \boldsymbol{y}_t$ and $\boldsymbol{v} = \boldsymbol{H}_t\boldsymbol{s}_t$

- Then, by taking $a = 1/\boldsymbol{y}_t^\top\boldsymbol{s}_t$ and $b = -1/\boldsymbol{s}_t^\top\boldsymbol{H}_t\boldsymbol{s}_t$, it follows that the secant condition $\boldsymbol{H}_{t+1}\boldsymbol{s}_t = \boldsymbol{y}_t$ holds

- Note: the update (AHU) updates by a rank-two matrix

# Optimality of BFGS update

- $\boldsymbol{H}_{t+1}$ solves the following convex optimization problem, for a positive definite matrix $\boldsymbol{W}$

$$\begin{aligned} \text{minimize} \quad & \|\boldsymbol{X} - \boldsymbol{H}_t\|_{\boldsymbol{W}} \\ \text{subject to} \quad & \boldsymbol{X}\boldsymbol{s}_t = \boldsymbol{y}_t \\ & \boldsymbol{X} = \boldsymbol{X}^\top \end{aligned}$$

where $\|\boldsymbol{A}\|_{\boldsymbol{W}}^2 = \left\|\boldsymbol{W}^{1/2}\boldsymbol{A}\boldsymbol{W}^{1/2}\right\|_F^2 = \operatorname{trace}(\boldsymbol{W}\boldsymbol{A}^\top\boldsymbol{W}\boldsymbol{A})$

and $\|\boldsymbol{A}\|_F^2 = \sum_{i,j} A_{i,j}^2$ is the squared Frobenius norm

- Proof is conceptually simple but tedious using KKT conditions
  - See Chapter 3, Fletcher 1987

# Optimality of BFGS update (cont'd)

- $\boldsymbol{H}_{t+1}$ solves the following convex optimization problem

$$\begin{aligned} \text{minimize} \quad & \mathrm{KL}(N(\mathbf{0}, \boldsymbol{X}) \,||\, N(\mathbf{0}, \boldsymbol{H}_t)) \\ \text{subject to} \quad & \boldsymbol{X}\boldsymbol{s}_t = \boldsymbol{y}_t \\ & \boldsymbol{X} = \boldsymbol{X}^\top \end{aligned}$$

where $\mathrm{KL}(N(\mathbf{0}, \boldsymbol{A}) \,||\, N(\mathbf{0}, \boldsymbol{B}))$is the Kullback-Leibler divergence (relative entropy) between two zero-mean Gaussian distributions with $n \times n$ covariance matrices $\boldsymbol{A}$ and $\boldsymbol{B}$

$$\mathrm{KL}(N(\mathbf{0}, \boldsymbol{A}) \,||\, N(\mathbf{0}, \boldsymbol{B})) = \frac{1}{2}\left(\mathrm{tr}(\boldsymbol{B}^{-1}\boldsymbol{A}) - \log\left(\det\left(\boldsymbol{B}^{-1}\boldsymbol{A}\right)\right) - n\right)$$

- To probe more about the KL divergence: see here

# L-BFGS: Limited memory BFGS

- Disadvantage of BFGS method is the need to *store the inverse Hessian matrix*
- Limited-memory BFGS: do not store $\boldsymbol{H}_t^{-1}$ explicitly
- Store up to $m$ values $\boldsymbol{s}_{t-m+1}, \ldots, \boldsymbol{s}_t$ and $\boldsymbol{y}_{t-m+1}, \ldots, \boldsymbol{y}_t$
- Evaluate $\boldsymbol{H}_\tau^{-1}\boldsymbol{g}_\tau$ recursively by using

$$\boldsymbol{H}_{t+1}^{-1} = (\boldsymbol{I} - \rho_t \boldsymbol{s}_t \boldsymbol{y}_t^\top)\boldsymbol{H}_t^{-1}(I - \rho_t \boldsymbol{y}_t \boldsymbol{s}_t^\top) + \rho_t \boldsymbol{s}_t \boldsymbol{s}_t^\top$$

  for some assumed initial value $\boldsymbol{H}_{\tau-m}^{-1}$ (ex. identity matrix)

- Cost per iteration is $\Theta(mn)$ for computation and storage
  - Substantially smaller than $\theta(n^2)$ for fixed $m$ and large $n$

# Spark L-BFGS: Python code example

```python
"""
Logistic Regression With LBFGS Example.
"""
from __future__ import print_function

from pyspark import SparkContext
# $example on$
from pyspark.mllib.classification import LogisticRegressionWithLBFGS, LogisticRegressionModel
from pyspark.mllib.regression import LabeledPoint
# $example off$

if __name__ == "__main__":

    sc = SparkContext(appName="PythonLogisticRegressionWithLBFGSExample")

    # $example on$
    # Load and parse the data
    def parsePoint(line):
        values = [float(x) for x in line.split(' ')]
        return LabeledPoint(values[0], values[1:])

    data = sc.textFile("data/mllib/sample_svm_data.txt")
    parsedData = data.map(parsePoint)

    # Build the model
    model = LogisticRegressionWithLBFGS.train(parsedData)

    # Evaluating the model on training data
    labelsAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))
    trainErr = labelsAndPreds.filter(lambda lp: lp[0] != lp[1]).count() / float(parsedData.count())
    print("Training Error = " + str(trainErr))

    # Save and load model
    model.save(sc, "target/tmp/pythonLogisticRegressionWithLBFGSModel")
    sameModel = LogisticRegressionModel.load(sc,
                                             "target/tmp/pythonLogisticRegressionWithLBFGSModel")
    # $example off$
```

- https://github.com/apache/spark/blob/master/examples/src/main/python/mllib/logistic_regression_with_lbfgs_example.py

# L-BFGS in Spark

**runLBFGS**

```
public static scala.Tuple2<Vector,double[]> runLBFGS(RDD<scala.Tuple2<Object,Vector>> data,
                                                     Gradient gradient,
                                                     Updater updater,
                                                     int numCorrections,
                                                     double convergenceTol,
                                                     int maxNumIterations,
                                                     double regParam,
                                                     Vector initialWeights)
```

Run Limited-memory BFGS (L-BFGS) in parallel. Averaging the subgradients over different partitions is performed using one standard spark map-reduce in each iteration.

**Parameters:**

data - - Input data for L-BFGS. RDD of the set of data examples, each of the form (label, [feature values]).

gradient - - Gradient object (used to compute the gradient of the loss function of one single data example)

updater - - Updater function to actually perform a gradient step in a given direction.

numCorrections - - The number of corrections used in the L-BFGS update.

convergenceTol - - The convergence tolerance of iterations for L-BFGS which is must be nonnegative. Lower values are less tolerant and therefore generally cause more iterations to be run.

maxNumIterations - - Maximal number of iterations that L-BFGS can be run.

regParam - - Regularization parameter

initialWeights - (undocumented)

**Returns:**

A tuple containing two elements. The first element is a column matrix containing weights for every feature, and the second element is an array containing the loss computed for every iteration.

https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/mllib/optimization/LBFGS.html

# Spark code

```scala
/**
 * CostFun implements Breeze's DiffFunction[T], which returns the loss and gradient
 * at a particular point (weights). It's used in Breeze's convex optimization routines.
 */
private class CostFun(
  data: RDD[(Double, Vector)],
  gradient: Gradient,
  updater: Updater,
  regParam: Double,
  numExamples: Long) extends DiffFunction[BDV[Double]] {

  override def calculate(weights: BDV[Double]): (Double, BDV[Double]) = {
    // Have a local copy to avoid the serialization of CostFun object which is not serializable.
    val w = Vectors.fromBreeze(weights)
    val n = w.size
    val bcW = data.context.broadcast(w)
    val localGradient = gradient

    val seqOp = (c: (Vector, Double), v: (Double, Vector)) =>
      (c, v) match {
        case ((grad, loss), (label, features)) =>
          val denseGrad = grad.toDense
          val l = localGradient.compute(features, label, bcW.value, denseGrad)
          (denseGrad, loss + l)
      }

    val combOp = (c1: (Vector, Double), c2: (Vector, Double)) =>
      (c1, c2) match { case ((grad1, loss1), (grad2, loss2)) =>
        val denseGrad1 = grad1.toDense
        val denseGrad2 = grad2.toDense
        axpy(1.0, denseGrad2, denseGrad1)
        (denseGrad1, loss1 + loss2)
      }

    val zeroSparseVector = Vectors.sparse(n, Seq.empty)
    val (gradientSum, lossSum) = data.treeAggregate((zeroSparseVector, 0.0))(seqOp, combOp)

    // broadcasted model is not needed anymore
    bcW.destroy()

    /**
     * regVal is sum of weight squares if it's L2 updater;
     * for other updater, the same logic is followed.
     */
    val regVal = updater.compute(w, Vectors.zeros(n), 0, 1, regParam)._2

    val loss = lossSum / numExamples + regVal
    /**
     * It will return the gradient part of regularization using updater.
     *
     * Given the input parameters, the updater basically does the following,
     *
     * w' = w - thisIterStepSize * (gradient + regGradient(w))
     * Note that regGradient is function of w
     *
     * If we set gradient = 0, thisIterStepSize = 1, then
     *
     * regGradient(w) = w - w'
     *
     * TODO: We need to clean it up by separating the logic of regularization out
     *       from updater to regularizer.
     */
    // The following gradientTotal is actually the regularization part of gradient.
    // Will add the gradientSum computed from the data with weights in the next step.
    val gradientTotal = w.copy
    axpy(-1.0, updater.compute(w, Vectors.zeros(n), 1, 1, regParam)._1, gradientTotal)

    // gradientTotal = gradientSum / numExamples + gradientTotal
    axpy(1.0 / numExamples, gradientSum, gradientTotal)

    (loss, gradientTotal.asBreeze.asInstanceOf[BDV[Double]])
  }
}
```

- LBFGS [spark/mllib/src/main/scala/org/apache/spark/mllib/optimization/LBFGS.scala]

# Breeze: a library for numerical processing in Scala

```scala
object LBFGS {
  case class ApproximateInverseHessian[T](
      m: Int,
      private[LBFGS] val memStep: IndexedSeq[T] = IndexedSeq.empty,
      private[LBFGS] val memGradDelta: IndexedSeq[T] = IndexedSeq.empty)(
      implicit space: MutableInnerProductModule[T, Double])
      extends NumericOps[ApproximateInverseHessian[T]] {

    import space._

    def repr: ApproximateInverseHessian[T] = this

    def updated(step: T, gradDelta: T) = {
      val memStep = (step +: this.memStep).take(m)
      val memGradDelta = (gradDelta +: this.memGradDelta).take(m)

      new ApproximateInverseHessian(m, memStep, memGradDelta)
    }

    def historyLength = memStep.length

    def *(grad: T) = {
      val diag = if (historyLength > 0) {
        val prevStep = memStep.head
        val prevGradStep = memGradDelta.head
        val sy = prevStep.dot(prevGradStep)
        val yy = prevGradStep.dot(prevGradStep)
        if (sy < 0 || sy.isNaN) throw new NaNHistory
        sy / yy
      } else {
        1.0
      }
      val dir = space.copy(grad)
      val as = new Array[Double](m)
      val rho = new Array[Double](m)

      for (i <- 0 until historyLength) {
        rho(i) = memStep(i).dot(memGradDelta(i))
        as(i) = (memStep(i).dot(dir)) / rho(i)
        if (as(i).isNaN) {
          throw new NaNHistory
        }
        axpy(-as(i), memGradDelta(i), dir)
      }

      dir *= diag

      for (i <- (historyLength - 1) to 0 by (-1)) {
        val beta = (memGradDelta(i).dot(dir)) / rho(i)
        axpy(as(i) - beta, memStep(i), dir)
      }

      dir *= -1.0
      dir
    }
  }

  implicit def multiplyInverseHessian[T](
      implicit vspace: MutableInnerProductModule[T, Double]): OpMulMatrix.Impl2[ApproximateInverseHessian[T], T, T] = {
    new OpMulMatrix.Impl2[ApproximateInverseHessian[T], T, T] {
      def apply(a: ApproximateInverseHessian[T], b: T): T = a * b
    }
  }
}
```

- https://github.com/scalanlp/breeze/blob/master/math/src/main/scala/breeze/optimize/LBFGS.scala

# References


- S. Bubeck, Convex Optimization: Algorithms and Complexity, Now Publishers, 2015, online copy here
- S. Boyd and L. Vandenberghe, Convex Optimization, Cambridge University Press, online copy here, Ch 9.5 Newton method
- L. Bottou and Y. L. Cun, Large Scale Online Learning, NIPS 2003
- L. Bottou et al, Optimization Methods for Large-Scale Machine Learning, Arxiv, 2016
- Liu et al, Recent Advances in Distributed Machine Learning, AAAI 2017
- Xing, E. P. and Ho, Q., A New Look at the System, Algorithm and Theory Foundations of Distributed Machine Learning, ACM KDD 2015 Tutorial

# References (cont'd)


- D. C. Liu and J. Nocedal, On the limited memory BFGS method for large scale optimization, Mathematical Programming, 45, 1989, pp 503-528
- J. Nocedal, Updating Quasi-Newton Matrices with Limited Storage, Mathematics of Computation, Vol 35, No 151, 1980
- R. Fletcher, Practical Methods of Optimization, 2nd Edition, John-Willey and Sons, 1987, online copy here
- J. Nocedal and S. J. Wright, Numerical Optimization, Second Edition, Springer, 2000
- A. D. Haghighi, Understanding L-BFGS, Blog, 2014
- (OWL-QN: Orthant-Wise Limited-memory Quasi-Newton) G. Andrew and J. Gao, Scalable Training of L1-Regularized Log-Linear Models, ICML 2007

# References (cont'd)

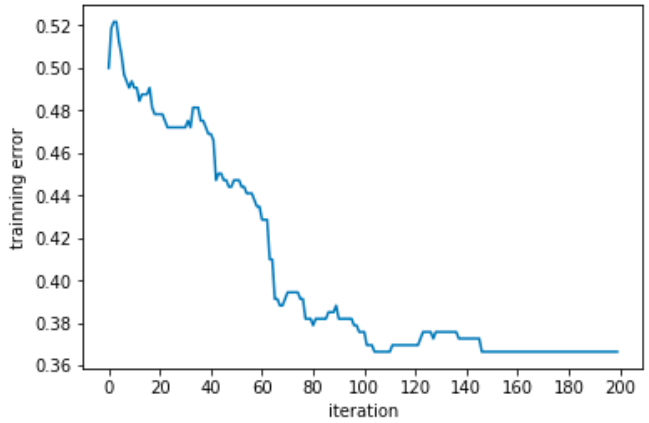

- A. Smola and S. Narayanamurthy, An Architecture for Parallel Topic Models, VLDB 2010
- Li et al, Scaling Distributed Machine Learning with the Parameter Server, OSDI 2014
- E. P. Xing et al, Petuum: A New Platform for Distributed Machine Learning on Big Data, ACM KDD 2015
- M. Li et al, Communication Efficient Distributed Machine Learning with the Parameter Server, NIPS 2014
- Parameter server open source ps-lite

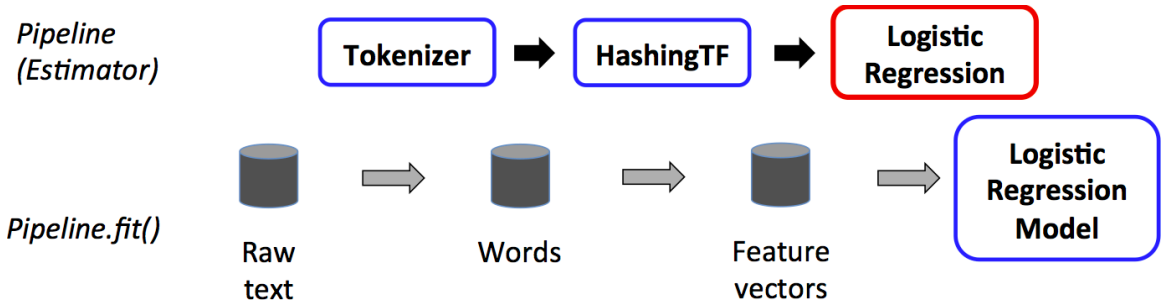
# References (cont'd)


- Spark Optimization – RDD-based API
- Apache Spark MLlib
- X. Meng et al, MLlib: Machine Learning in Apache Spark, Vol 17, JMLR 2016
- Machine Learning Library (MLlib) programming guide
- R. Zadeh, Distributed Machine Learning on Spark, Strata 2015
- R. Zadeh et al, Matrix Computations and Optimization in Apache Spark, ACM KDD 2016
- X. Meng, MLlib: Scalable Machine Learning on Spark
- Spark summer school 2013

# Seminar class

- Batch gradient descent
- Comparison of L-BFGS and SGD
- Use of DataFrame API and pipelines




https://github.com/lse-st446/lectures2021/blob/master/Week08/class/README.md