

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>, <categorical 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
```

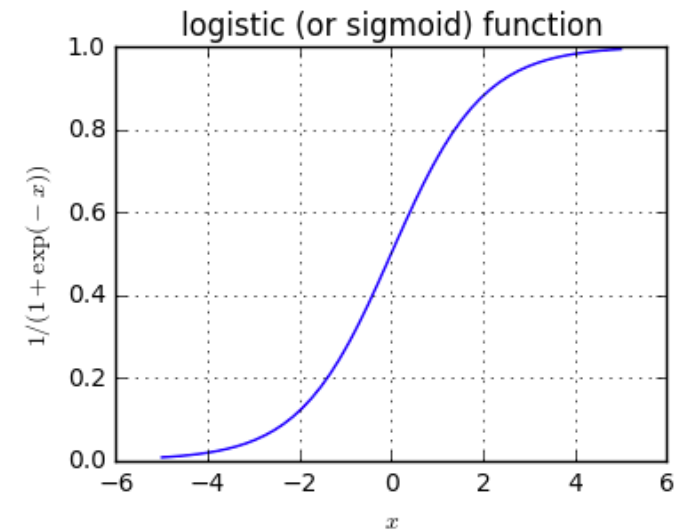
Logistic regression

- Training examples: $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)$
where $\mathbf{x}_i \in \mathbf{R}^n$ is a feature vector and $y_i \in \{0,1\}$ is a label
- Prediction problem: learn the function $g(\mathbf{x}; \mathbf{w}) = \mathbf{P}[y_i = 1 \mid \mathbf{x}_i = \mathbf{x}]$
where $\mathbf{w} \in \mathbf{R}^n$ is a parameter vector
- Logistic regression model:

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

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

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



Loss function: negative log-likelihood

- Log-likelihood function:

$$\begin{aligned}\ell(\mathbf{w}) &= \sum_{i=1}^m [\mathbf{y}_i \log(\sigma(\mathbf{x}_i^\top \mathbf{w})) + (1 - \mathbf{y}_i) \log(1 - \sigma(\mathbf{x}_i^\top \mathbf{w}))] \\ &= \sum_{i=1}^m [\mathbf{y}_i \mathbf{x}_i^\top \mathbf{w} - \log(1 + e^{\mathbf{x}_i^\top \mathbf{w}})]\end{aligned}$$

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

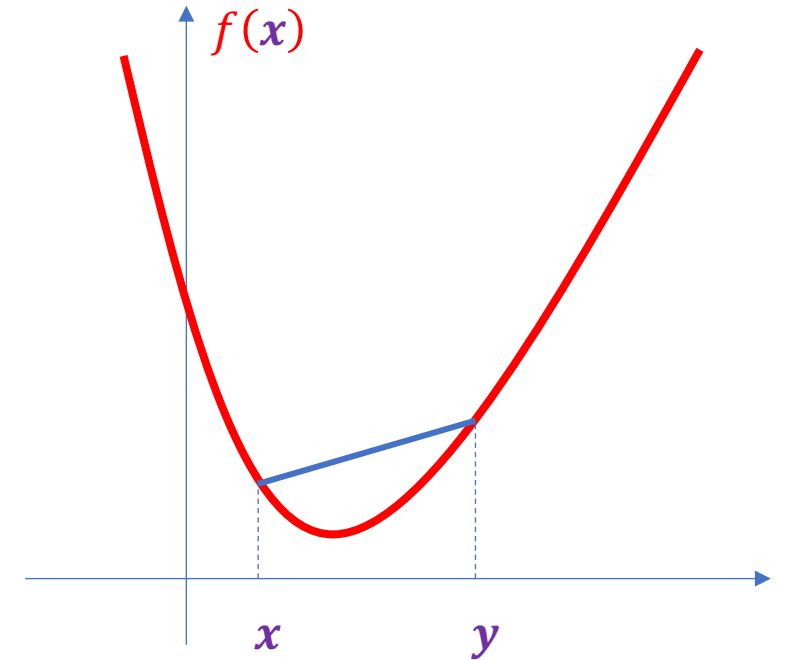
Background: convex functions

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

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

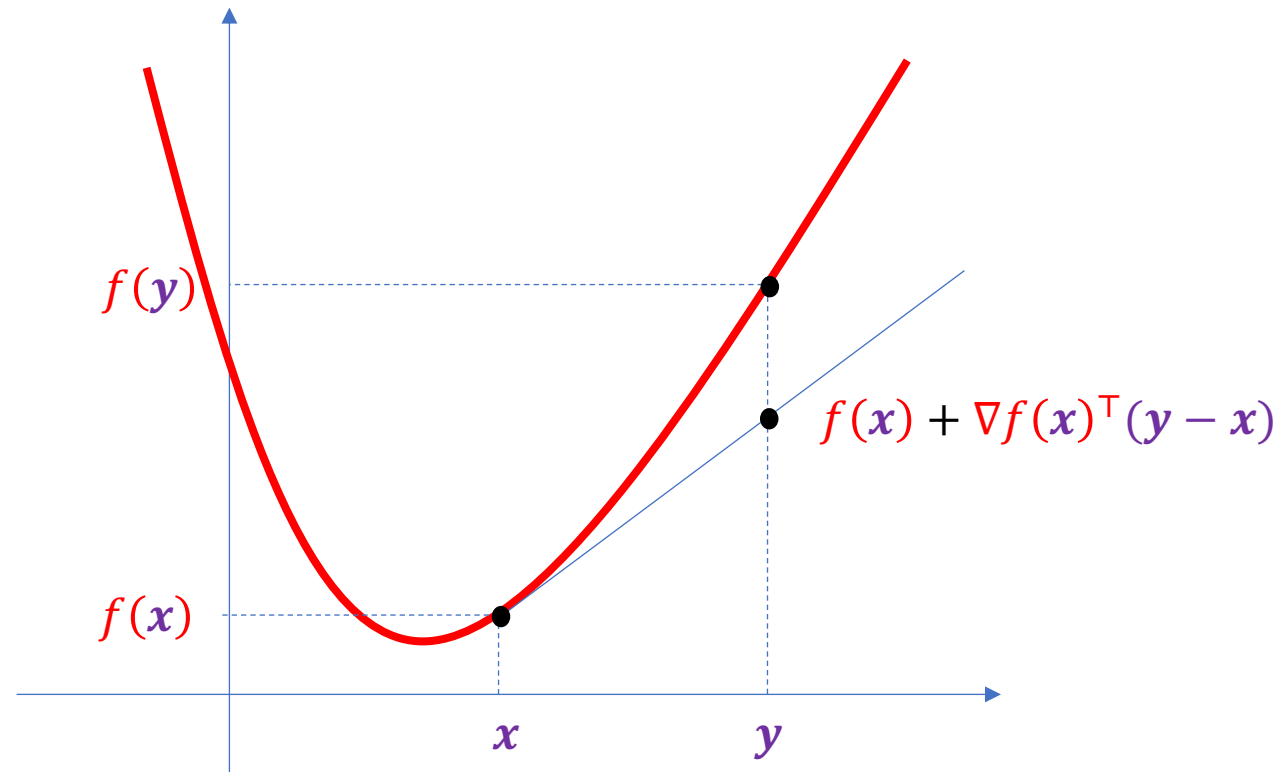
f is **strictly convex** if the inequality is strict for all $\lambda \in (0,1)$

- 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 $f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x})$ for all $\mathbf{x}, \mathbf{y} \in X$



Background: convex twice-differentiable functions

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

Gradient descent algorithm

Gradient descent algorithm

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

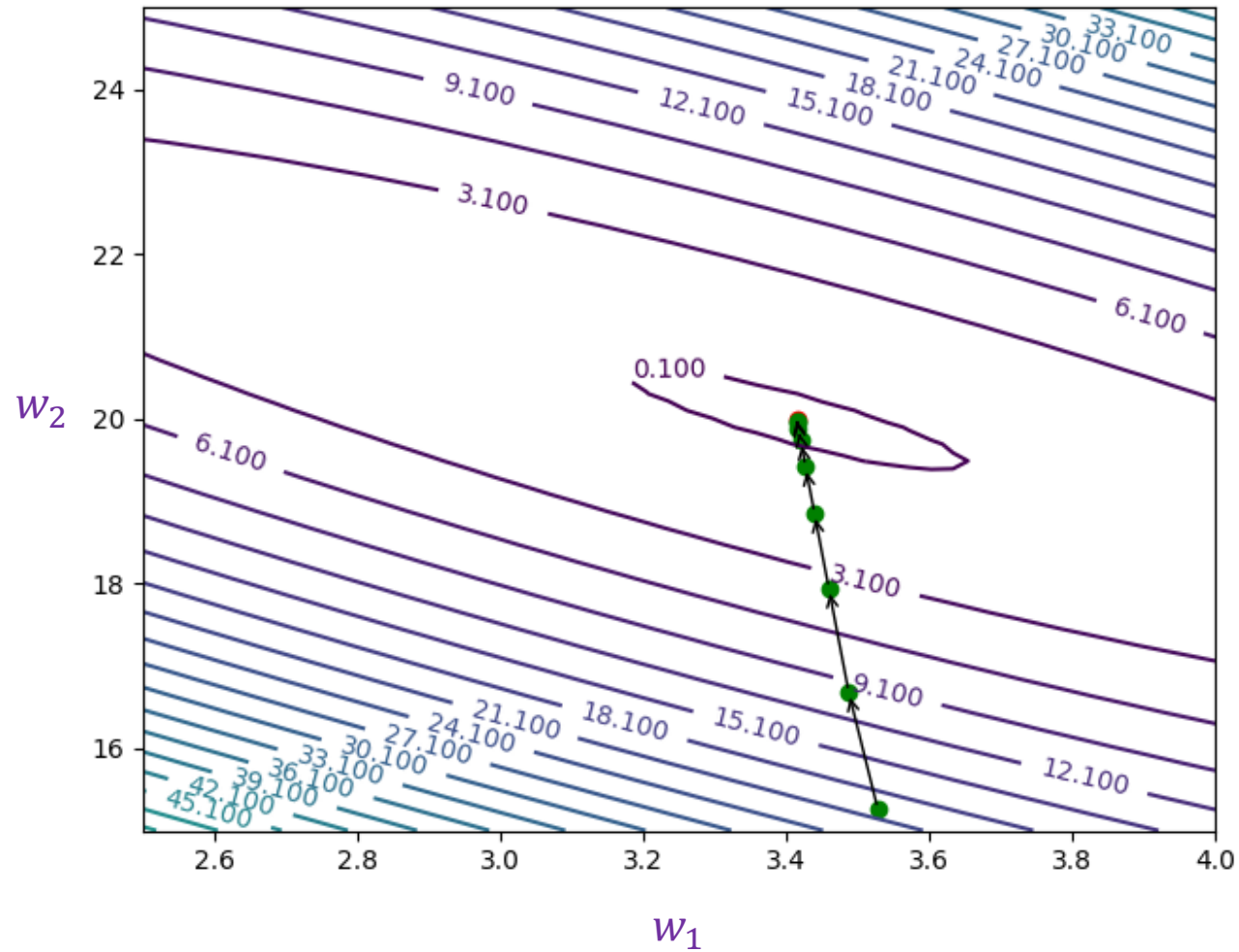
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta_t \nabla f(\mathbf{w}^{(t)})$$

where η_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 $\nabla f(\mathbf{w}^{(t)})$ is the gradient vector

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

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

Convergence illustration



Convergence rates for gradient descent

- A point $\mathbf{w} \in W$ is said to be an ϵ -optimal point on W if $f(\mathbf{w}) - f(\mathbf{w}^*) \leq \epsilon$, where \mathbf{w}^* is a minima point of f on W
- Let $t_\epsilon(\mathbf{w}^{(0)})$ be the smallest number of iterations t such that $f(\mathbf{w}^{(t)}) - f(\mathbf{w}^*) \leq \epsilon$
- Let $t_\epsilon(R) = \max_{\mathbf{w}^{(0)}: \|\mathbf{w}^{(0)} - \mathbf{w}^*\| \leq R} t_\epsilon(\mathbf{w}^{(0)})$
- 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 β -smooth means that the largest eigenvalue of $\nabla^2 f(\mathbf{w})$ is $\leq \beta$
 - f is α -strongly convex means that the smallest eigenvalue of $\nabla^2 f(\mathbf{w})$ is $\geq \alpha$

Projected gradient descent

- Projected gradient descent update rule:

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

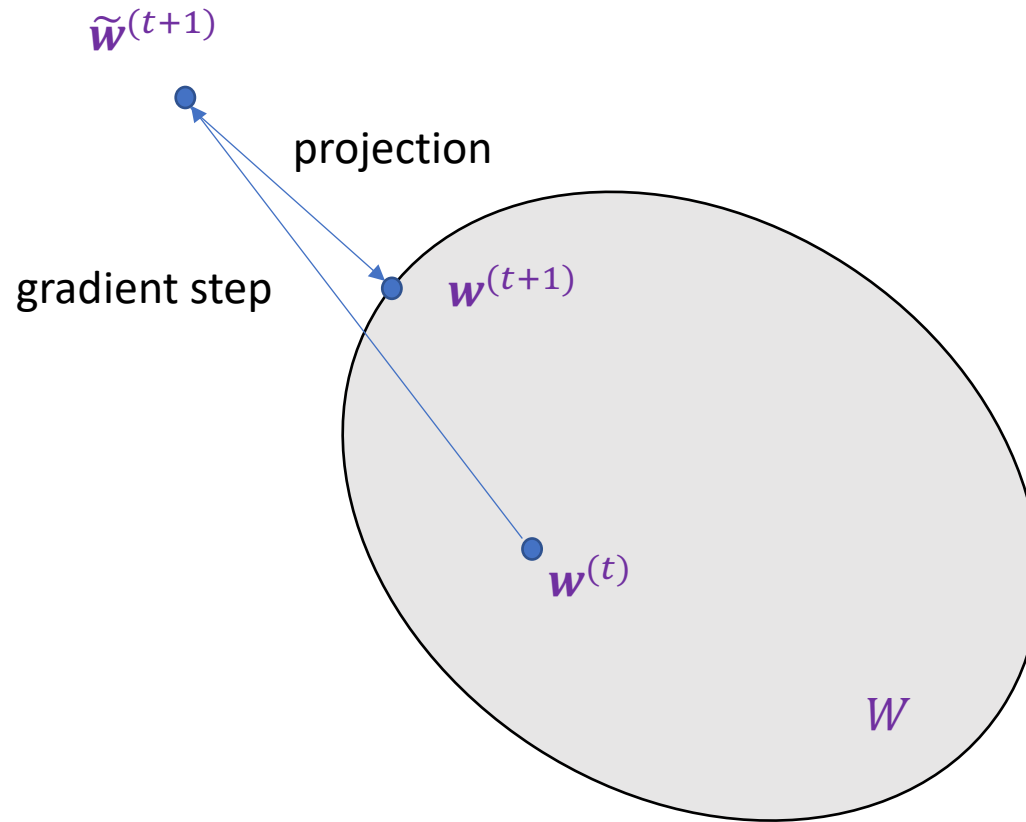
where W is a convex and compact set and

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

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

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

Projected gradient descent (cont'd)



GD convergence rate: convex, smooth gradient

- **Thm.** Assume that W is in the Euclidean ball with center $\mathbf{w}^{(0)}$ and radius R and that f is such that there exists $L > 0$ such that $\|\nabla f(\mathbf{w})\| \leq L$ for all $\mathbf{w} \in W$, 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

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

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

Proof sketch*

$$\begin{aligned} f(\mathbf{w}^{(s)}) - f(\mathbf{w}^*) &\leq \nabla f(\mathbf{w}^{(s)})^\top (\mathbf{w}^{(s)} - \mathbf{w}^*) \\ &= \frac{1}{\eta} (\mathbf{w}^{(s)} - \tilde{\mathbf{w}}^{(s+1)})^\top (\mathbf{w}^{(s)} - \mathbf{w}^*) \\ &= \frac{1}{2\eta} \left(\|\mathbf{w}^{(s)} - \mathbf{w}^*\|^2 + \|\mathbf{w}^{(s)} - \tilde{\mathbf{w}}^{(s+1)}\|^2 - \|\tilde{\mathbf{w}}^{(s+1)} - \mathbf{w}^*\|^2 \right) \\ &= \frac{1}{2\eta} \left(\|\mathbf{w}^{(s)} - \mathbf{w}^*\|^2 - \|\tilde{\mathbf{w}}^{(s+1)} - \mathbf{w}^*\|^2 \right) + \frac{\eta}{2} \|\nabla f(\mathbf{w}^{(s)})\|^2 \\ &\leq \frac{1}{2\eta} \left(\|\mathbf{w}^{(s)} - \mathbf{w}^*\|^2 - \|\tilde{\mathbf{w}}^{(s+1)} - \mathbf{w}^*\|^2 \right) + \frac{\eta}{2} L^2 \\ &\leq \frac{1}{2\eta} \left(\|\mathbf{w}^{(s)} - \mathbf{w}^*\|^2 - \|\mathbf{w}^{(s+1)} - \mathbf{w}^*\|^2 \right) + \frac{\eta}{2} L^2 \quad (\text{Bubeck Lemma 3.1}) \end{aligned}$$

Proof sketch (cont'd)*

- Summing up, we have

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

- Plug in the value of η and use, by Jensen's inequality,

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

- To probe further: Jensen's inequality see [here](#)

Smooth convex case

- A function f is said to be β -smooth if $\|\nabla f(\mathbf{y}) - \nabla f(\mathbf{x})\| \leq \beta \|\mathbf{y} - \mathbf{x}\|$
 - If f is twice-differentiable, $\nabla^2 f(\mathbf{x}) \preceq \beta \mathbf{I}$
- **Thm.** Suppose f is convex and β -smooth and the step size is such that $\eta = 1/\beta$. Then, gradient descent algorithm satisfies

$$f(\mathbf{w}^{(t)}) - f(\mathbf{w}^*) \leq 2\beta \|\mathbf{w}^{(0)} - \mathbf{w}^*\|^2 \frac{1}{t-1}$$

- 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 α -strongly convex if $f(\mathbf{y}) \geq f(\mathbf{x}) + \nabla f(\mathbf{x})^\top (\mathbf{y} - \mathbf{x}) + \frac{\alpha}{2} \|\mathbf{y} - \mathbf{x}\|^2$
 - If f is twice-differentiable $\nabla^2 f(\mathbf{x}) \succeq \alpha \mathbf{I}$
- **Thm.** Suppose f is α -strongly convex and β -smooth and $\eta = 2/(\alpha + \beta)$. Then, gradient descent algorithm satisfies

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

- Hence, the number of iterations $O(\log(1/\epsilon))$
- To probe further: Bubeck Thm 3.12

Stochastic gradient descent algorithm

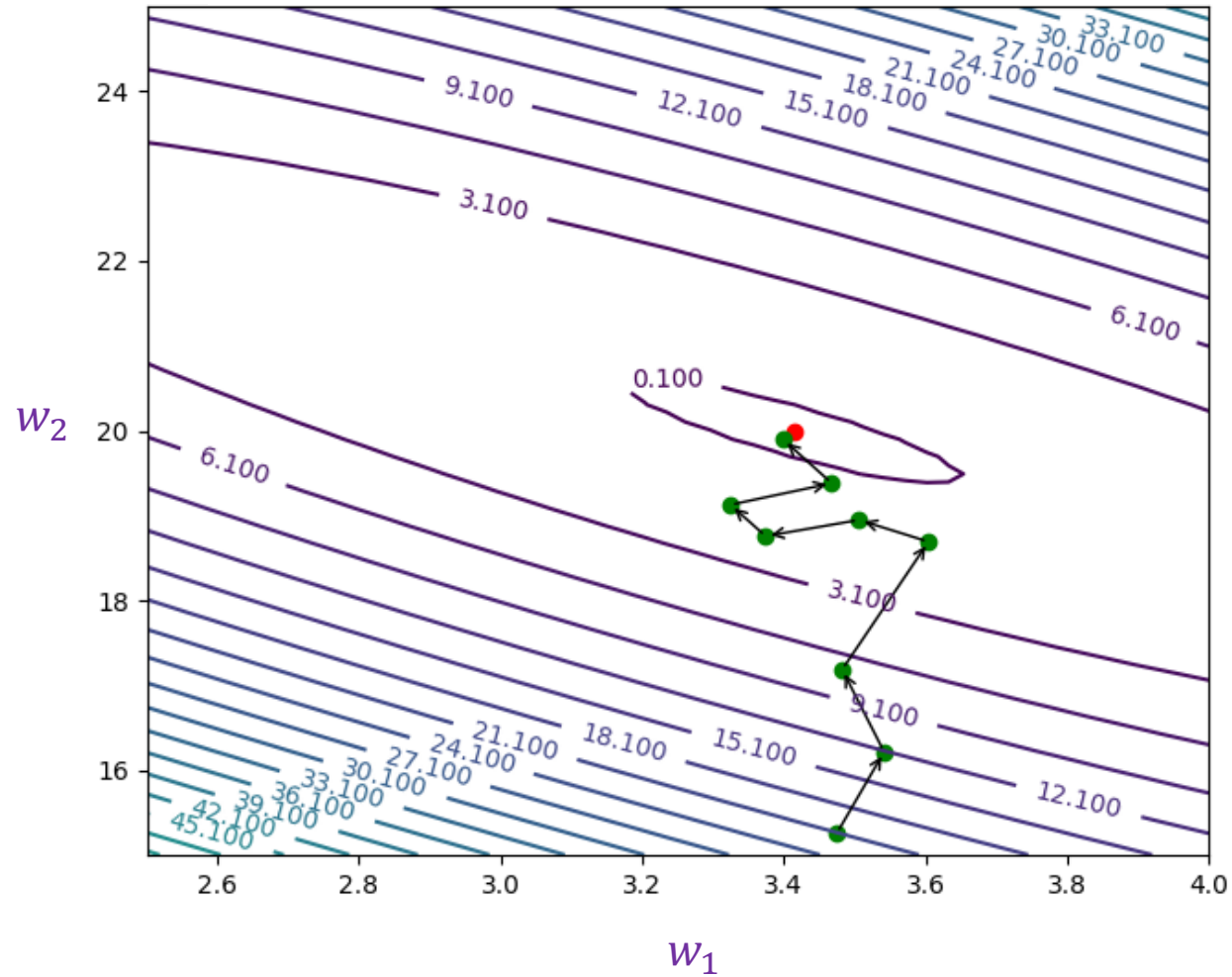
Stochastic Gradient Descent (SGD)

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

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

- **Stochastic Gradient Descent (SGD)**: use stochastic gradient vector $\hat{\mathbf{g}}_t$ at iteration t
- Desiderata:
 - Unbiased estimator: $\mathbf{E}[\hat{\mathbf{g}}_t(\mathbf{w})] = \nabla f(\mathbf{w})$
 - Small variance: $\sigma_t^2 = \mathbf{E}[\|\hat{\mathbf{g}}_t(\mathbf{w}) - \nabla f(\mathbf{w})\|^2]$
- **SGD methods**:
 - One sample: $\hat{\mathbf{g}}_t(\mathbf{w}) = \nabla f_{I_t}(\mathbf{w}; \mathbf{x}_{I_t}, \mathbf{y}_{I_t})$ where I_t is a randomly sampled example
 - Mini-batch SGD: $\hat{\mathbf{g}}_t(\mathbf{w}) = \frac{1}{k} \sum_{i \in S_t} \nabla f_i(\mathbf{w}; \mathbf{x}_i, \mathbf{y}_i)$ 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 β -smooth, W is a convex set contained in the ball of radius R with center $\mathbf{w}^{(0)}$ and that there exists $\sigma > 0$ such that $\mathbf{E}[\|\hat{\mathbf{g}}_t(\mathbf{w}) - \nabla f(\mathbf{w})\|^2] \leq \sigma^2$ for all $\mathbf{w} \in W$.

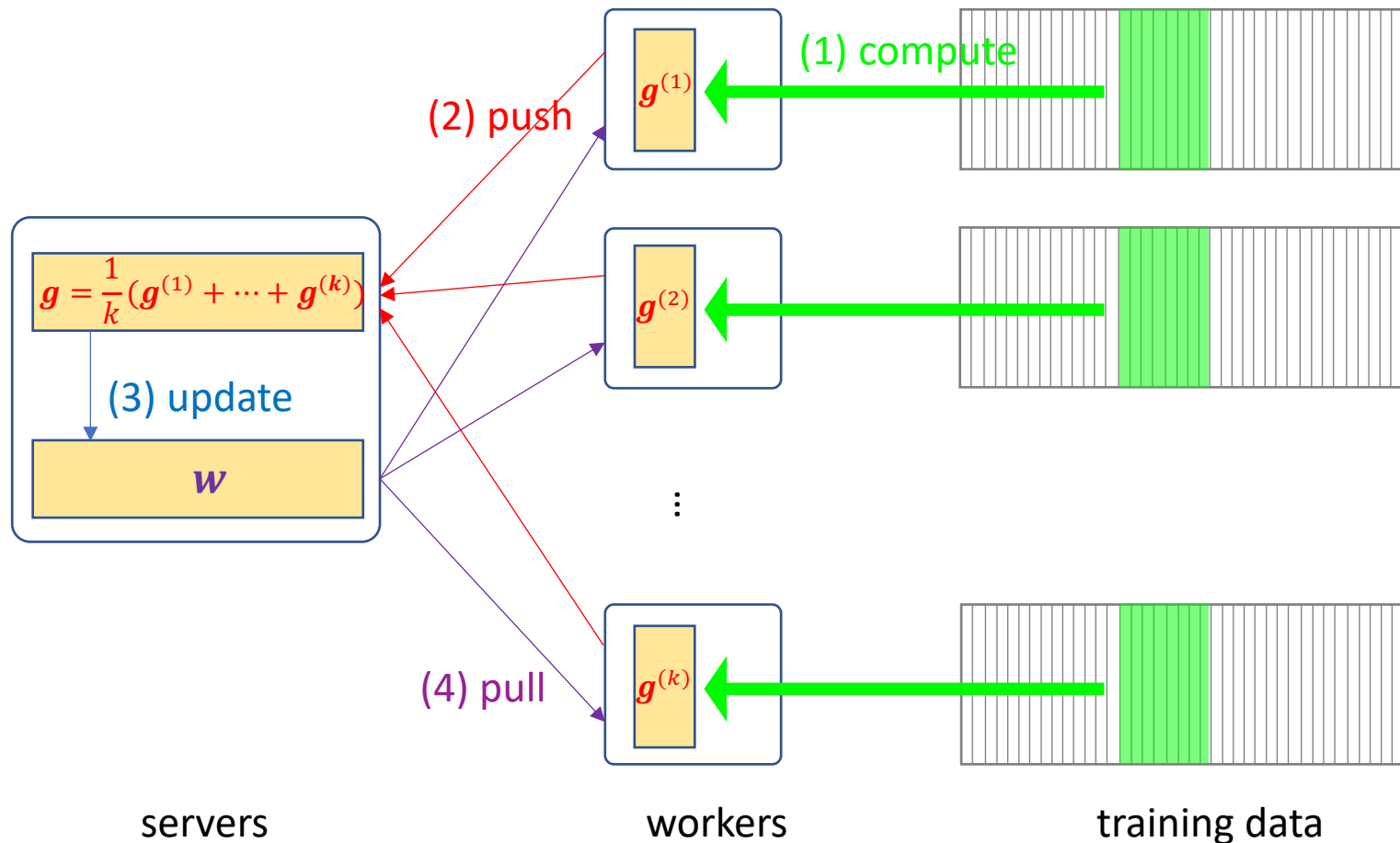
Then, the SGD with step size $\eta_t = 1/(\beta + \sigma/R\sqrt{2}\sqrt{t})$, it holds

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

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

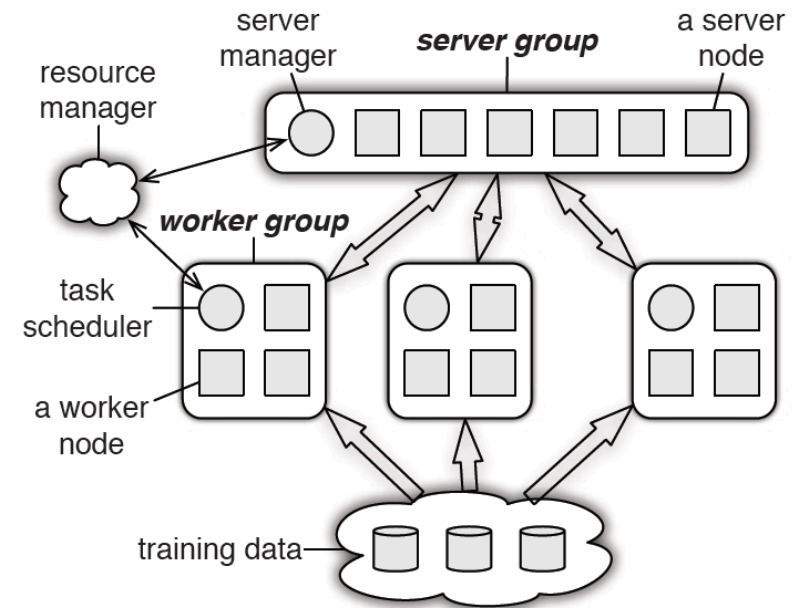
Distributed GD / SGD

- **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 $t = 0, 1, \dots, T$ **do**

 Issue `workerIterate(t)` to all workers

end for

Worker w

```
function loadData()
```

```
    Load  $(\mathbf{x}_i, \mathbf{y}_i)$  for  $i \in N_w$  // load part of training data
```

```
    Pull  $\mathbf{w}^{(0)}$  from server
```

```
function workerIterate(t)
```

```
     $\mathbf{g}_t^{(w)} = \sum_{i \in N_w} \nabla f(\mathbf{w}^{(t)}; \mathbf{x}_i, \mathbf{y}_i)$  // compute gradient vector update
```

```
    Push  $\mathbf{g}_t^{(w)}$  to server
```


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

Server

```
function serverIterate(t)
```

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

```
 $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta(\mathbf{g}_t + \lambda \nabla \Omega(\mathbf{w}^{(t)}))$  // 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.txt")

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 multinomial 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

```
150 /**
151  * :: DeveloperApi ::
152  * Top-level method to run gradient descent.
153  */
154 @DeveloperApi
155 object GradientDescent extends Logging {
156   /**
157    * Run stochastic gradient descent (SGD) in parallel using mini batches.
158    * In each iteration, we sample a subset (fraction miniBatchFraction) of the total data
159    * in order to compute a gradient estimate.
160    * Sampling, and averaging the subgradients over this subset is performed using one standard
161    * spark map-reduce in each iteration.
162    *
163    * @param data Input data for SGD. RDD of the set of data examples, each of
164    *           the form (label, [feature values]).
165    * @param gradient Gradient object (used to compute the gradient of the loss function of
166    *           one single data example)
167    * @param updater Updater function to actually perform a gradient step in a given direction.
168    * @param stepSize initial step size for the first step
169    * @param numIterations number of iterations that SGD should be run.
170    * @param regParam regularization parameter
171    * @param miniBatchFraction fraction of the input data set that should be used for
172    *           one iteration of SGD. Default value 1.0.
173    * @param convergenceTol Minibatch iteration will end before numIterations if the relative
174    *           difference between the current weight and the previous weight is less
175    *           than this value. In measuring convergence, L2 norm is calculated.
176    *           Default value 0.001. Must be between 0.0 and 1.0 inclusively.
177    * @return A tuple containing two elements. The first element is a column matrix containing
178    *         weights for every feature, and the second element is an array containing the
179    *         stochastic loss computed for every iteration.
180    */
181   def runMiniBatchSGD(
182     data: RDD[(Double, Vector)],
183     gradient: Gradient,
184     updater: Updater,
185     stepSize: Double,
186     numIterations: Int,
187     regParam: Double,
188     miniBatchFraction: Double,
189     initialWeights: Vector,
190     convergenceTol: Double): (Vector, Array[Double]) = {
191
192     // convergenceTol should be set with non minibatch settings
193     if (miniBatchFraction < 1.0 && convergenceTol > 0.0) {
194       logWarning("Testing against a convergenceTol when using miniBatchFraction " +
195         "< 1.0 can be unstable because of the stochasticity in sampling.")
196     }
197   }
```

```
229   var regVal = updater.compute(
230     weights, Vectors.zeros(weights.size), 0, 1, regParam)._2
231
232   var converged = false // indicates whether converged based on convergenceTol
233   var i = 1
234   while (!converged && i <= numIterations) {
235     val bcWeights = data.context.broadcast(weights)
236     // Sample a subset (fraction miniBatchFraction) of the total data
237     // compute and sum up the subgradients on this subset (this is one map-reduce)
238     val (gradientSum, lossSum, miniBatchSize) = data.sample(false, miniBatchFraction, 42 + i)
239     .treeAggregate(BDV.zeros[Double](n), 0.0, 0L)((
240       seqOp = (c, v) => {
241         // c: (grad, loss, count), v: (label, features)
242         val l = gradient.compute(v._2, v._1, bcWeights.value, Vectors.fromBreeze(c._1))
243         (c._1, c._2 + l, c._3 + 1)
244       },
245       combOp = (c1, c2) => {
246         // c: (grad, loss, count)
247         (c1._1 += c2._1, c1._2 + c2._2, c1._3 + c2._3)
248       })
249     bcWeights.destroy()
250
251     if (miniBatchSize > 0) {
252       /**
253        * lossSum is computed using the weights from the previous iteration
254        * and regVal is the regularization value computed in the previous iteration as well.
255        */
256       stochasticLossHistory += lossSum / miniBatchSize + regVal
257       val update = updater.compute(
258         weights, Vectors.fromBreeze(gradientSum / miniBatchSize.toDouble),
259         stepSize, i, regParam)
260       weights = update._1
261       regVal = update._2
262
263       previousWeights = currentWeights
264       currentWeights = Some(weights)
265       if (previousWeights != None && currentWeights != None) {
266         converged = isConverged(previousWeights.get,
267           currentWeights.get, convergenceTol)
268       }
269     } else {
270       logWarning(s"Iteration ($i/$numIterations). The size of sampled batch is zero")
271     }
272     i += 1
273   }
274
275   logInfo("GradientDescent.runMiniBatchSGD finished. Last 10 stochastic losses %s".format(
276     stochasticLossHistory.takeRight(10).mkString(", ")))
277 }
```

treeAggregate

```
1189  /**
1190   * Aggregates the elements of this RDD in a multi-level tree pattern.
1191   * This method is semantically identical to [[org.apache.spark.rdd.RDD#aggregate]].
1192   *
1193   * @param depth suggested depth of the tree (default: 2)
1194   */
1195  def treeAggregate[U: ClassTag](zeroValue: U)(
1196    seqOp: (U, T) => U,
1197    combOp: (U, U) => U,
1198    depth: Int = 2): U = withScope {
1199    require(depth >= 1, s"Depth must be greater than or equal to 1 but got $depth.")
1200    if (partitions.length == 0) {
1201      Utils.clone(zeroValue, context.env.closureSerializer.newInstance())
1202    } else {
1203      val cleanSeqOp = context.clean(seqOp)
1204      val cleanCombOp = context.clean(combOp)
1205      val aggregatePartition =
1206        (it: Iterator[T]) => it.aggregate(zeroValue)(cleanSeqOp, cleanCombOp)
1207      var partiallyAggregated: RDD[U] = mapPartitions(it => Iterator(aggregatePartition(it)))
1208      var numPartitions = partiallyAggregated.partitions.length
1209      val scale = math.max(math.ceil(math.pow(numPartitions, 1.0 / depth)).toInt, 2)
1210      // If creating an extra level doesn't help reduce
1211      // the wall-clock time, we stop tree aggregation.
1212
1213      // Don't trigger TreeAggregation when it doesn't save wall-clock time
1214      while (numPartitions > scale + math.ceil(numPartitions.toDouble / scale)) {
1215        numPartitions /= scale
1216        val curNumPartitions = numPartitions
1217        partiallyAggregated = partiallyAggregated.mapPartitionsWithIndex {
1218          (i, iter) => iter.map((i % curNumPartitions, _))
1219        }.foldByKey(zeroValue, new HashPartitioner(curNumPartitions))(cleanCombOp).values
1220      }
1221      val copiedZeroValue = Utils.clone(zeroValue, sc.env.closureSerializer.newInstance())
1222      partiallyAggregated.fold(copiedZeroValue)(cleanCombOp)
1223    }
1224  }
```

- <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:

$$f(\mathbf{w}) \approx q_t(\mathbf{w}) := f(\mathbf{w}^{(t)}) + (\mathbf{w} - \mathbf{w}^{(t)})^\top \mathbf{g}_t + \frac{1}{2} (\mathbf{w} - \mathbf{w}^{(t)})^\top \mathbf{H}_t (\mathbf{w} - \mathbf{w}^{(t)})$$

where

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

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

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

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

Newton-Raphson iterative algorithm

- Iterate until convergence:

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

where

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

- 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 the Newton method

- **Thm.** Assume that f is twice-differentiable, α -strongly convex, and $\|\nabla^2 f(\mathbf{x}) - \nabla^2 f(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\|$, for some $L > 0$.

Then, there exist $0 < \eta \leq \alpha^2/L$ and $\gamma > 0$ such that

- If $\|\nabla f(\mathbf{w}^{(t)})\| \geq \eta$ then $f(\mathbf{w}^{(t+1)}) - f(\mathbf{w}^{(t)}) \leq -\gamma$
 - Otherwise, if $\|\nabla f(\mathbf{w}^{(t)})\| < \eta$ then $\frac{L}{2\alpha^2} \|\nabla f(\mathbf{w}^{(t+1)})\| \leq \left(\frac{L}{2\alpha^2} \|\nabla f(\mathbf{w}^{(t)})\|\right)^2$ (quadratic convergence!)
- To achieve $f(\mathbf{w}^{(t)}) - f(\mathbf{w}^*) \leq \epsilon$ it suffices that

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

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 $\mathbf{w}^{(0)}$, a positive definite matrix \mathbf{B}_0

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

Compute quasi-Newton direction $\delta \mathbf{w}_t = -\mathbf{B}_t \nabla f(\mathbf{w}^{(t)})$

Determine step size α_t

Update $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \alpha_t \delta \mathbf{w}_t$

Compute \mathbf{B}_{t+1}

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

BFGS

Broyden, Fletcher, Goldfarb, Shanno



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

BFGS update

- BFGS update:

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

where

$$\begin{aligned} \mathbf{y}_t &= \mathbf{g}_{t+1} - \mathbf{g}_t && \text{(gradient change)} \\ \mathbf{s}_t &= \mathbf{w}^{(t+1)} - \mathbf{w}^{(t)} && \text{(parameter estimate change)} \\ \rho_t &= 1 / \mathbf{y}_t^\top \mathbf{s}_t \end{aligned}$$

- Note $\mathbf{y}_t^\top \mathbf{s}_t = \left(\nabla f(\mathbf{w}^{(t+1)}) - \nabla f(\mathbf{w}^{(t)}) \right)^\top (\mathbf{w}^{(t+1)} - \mathbf{w}^{(t)}) > 0$ if f is strictly convex

BFGS update (cont'd)

- Inverse matrix update:

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

(follows from the BFGS update and the Sherman-Morrison formula for inverse of the sum of an invertible matrix and the outer product of two vectors)

- The inverse matrix update can be written as:

$$\mathbf{H}_{t+1}^{-1} = \mathbf{H}_t^{-1} + \frac{1}{(\mathbf{y}_t^\top \mathbf{s}_t)^2} (\mathbf{s}_t \mathbf{y}_t^\top + \mathbf{y}_t^\top \mathbf{H}_t^{-1} \mathbf{y}_t) \mathbf{s}_t \mathbf{s}_t^\top + \frac{1}{\mathbf{y}_t^\top \mathbf{s}_t} (\mathbf{H}_t^{-1} \mathbf{y}_t \mathbf{s}_t^\top + \mathbf{s}_t \mathbf{y}_t^\top \mathbf{H}_t^{-1})$$

- Cost of update or inverse update is $O(n^2)$ operations

Secant condition

- The BFGS update satisfies the *secant condition*:

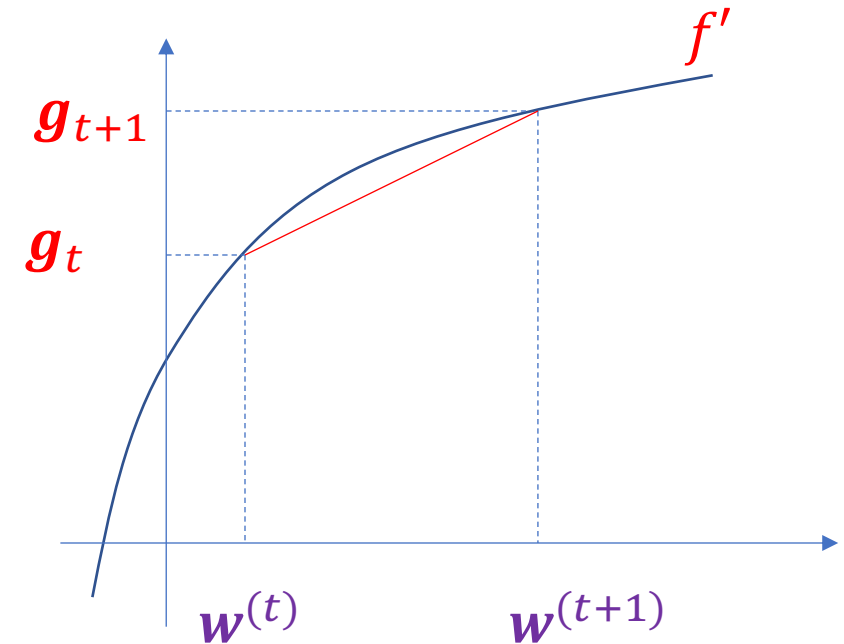
$$\mathbf{H}_{t+1} \mathbf{s}_t = \mathbf{y}_t \quad (\text{easy to check})$$

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

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

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

$$\begin{aligned} \nabla q_{t+1}(\mathbf{w}^{(t)}) &= \mathbf{g}_{t+1} + \mathbf{H}_{t+1}(\mathbf{w}^{(t)} - \mathbf{w}^{(t+1)}) \\ &= \mathbf{g}_{t+1} - \mathbf{y}_t \\ &= \mathbf{g}_t \end{aligned}$$



BFGS update (cont'd)

- Approximate Hessian update

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

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

- Assume $\mathbf{U}_t = a \mathbf{u} \mathbf{u}^\top$ and $\mathbf{V}_t = b \mathbf{v} \mathbf{v}^\top$ with $\mathbf{u} = \mathbf{y}_t$ and $\mathbf{v} = \mathbf{H}_t \mathbf{s}_t$
- By taking $a = 1/\mathbf{y}_t^\top \mathbf{s}_t$ and $b = -1/\mathbf{s}_t^\top \mathbf{H}_t \mathbf{s}_t$, we obtain the BFGS update
- Note: (AHU) is such that the update is by a rank-two matrix

Optimality of BFGS update

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

$$\text{minimize} \quad \|\mathbf{X} - \mathbf{H}_t\|_{\mathbf{W}}$$

$$\text{subject to} \quad \mathbf{X}\mathbf{s}_t = \mathbf{y}_t \\ \mathbf{X} = \mathbf{X}^\top$$

$$\text{where } \|\mathbf{A}\|_{\mathbf{W}}^2 = \|\mathbf{W}^{1/2}\mathbf{A}\mathbf{W}^{1/2}\|_F^2 = \text{trace}(\mathbf{W}\mathbf{A}^\top\mathbf{W}\mathbf{A})$$

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

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

Optimality of BFGS update (cont'd)

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

$$\text{minimize} \quad \text{KL}(N(\mathbf{0}, \mathbf{X}) \parallel N(\mathbf{0}, \mathbf{H}_t))$$

$$\text{subject to} \quad \begin{aligned} \mathbf{X}\mathbf{s}_t &= \mathbf{y}_t \\ \mathbf{X} &= \mathbf{X}^\top \end{aligned}$$

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

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

- 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 \mathbf{H}_t^{-1} explicitly
- Store up to m values $\mathbf{s}_{t-m+1}, \dots, \mathbf{s}_t$ and $\mathbf{y}_{t-m+1}, \dots, \mathbf{y}_t$
- Evaluate $\mathbf{H}_\tau^{-1} \mathbf{g}_\tau$ recursively by using

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

for some assumed initial value $\mathbf{H}_{\tau-m}^{-1}$ (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

```
18 """
19 Logistic Regression With LBFGS Example.
20 """
21 from __future__ import print_function
22
23 from pyspark import SparkContext
24 # $example on$
25 from pyspark.mllib.classification import LogisticRegressionWithLBFGS, LogisticRegressionModel
26 from pyspark.mllib.regression import LabeledPoint
27 # $example off$
28
29 if __name__ == "__main__":
30
31     sc = SparkContext(appName="PythonLogisticRegressionWithLBFGSExample")
32
33     # $example on$
34     # Load and parse the data
35     def parsePoint(line):
36         values = [float(x) for x in line.split(' ')]
37         return LabeledPoint(values[0], values[1:])
38
39     data = sc.textFile("data/mllib/sample_svm_data.txt")
40     parsedData = data.map(parsePoint)
41
42     # Build the model
43     model = LogisticRegressionWithLBFGS.train(parsedData)
44
45     # Evaluating the model on training data
46     labelsAndPreds = parsedData.map(lambda p: (p.label, model.predict(p.features)))
47     trainErr = labelsAndPreds.filter(lambda lp: lp[0] != lp[1]).count() / float(parsedData.count())
48     print("Training Error = " + str(trainErr))
49
50     # Save and load model
51     model.save(sc, "target/tmp/pythonLogisticRegressionWithLBFGSModel")
52     sameModel = LogisticRegressionModel.load(sc,
53                                             "target/tmp/pythonLogisticRegressionWithLBFGSModel")
54     # $example off$
```

- 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.

Spark code

```
226  /**
227   * CostFun implements Breeze's DiffFunction[T], which returns the loss and gradient
228   * at a particular point (weights). It's used in Breeze's convex optimization routines.
229   */
230  private class CostFun(
231    data: RDD[(Double, Vector)],
232    gradient: Gradient,
233    updater: Updater,
234    regParam: Double,
235    numExamples: Long) extends DiffFunction[BDV[Double]] {
236
237    override def calculate(weights: BDV[Double]): (Double, BDV[Double]) = {
238      // Have a local copy to avoid the serialization of CostFun object which is not serializable.
239      val w = Vectors.fromBreeze(weights)
240      val n = w.size
241      val bcW = data.context.broadcast(w)
242      val localGradient = gradient
243
244      val seqOp = (c: (Vector, Double), v: (Double, Vector)) =>
245        (c, v) match {
246          case ((grad, loss), (label, features)) =>
247            val denseGrad = grad.toDense
248            val l = localGradient.compute(features, label, bcW.value, denseGrad)
249            (denseGrad, loss + l)
250        }
251
252      val combOp = (c1: (Vector, Double), c2: (Vector, Double)) =>
253        (c1, c2) match { case ((grad1, loss1), (grad2, loss2)) =>
254          val denseGrad1 = grad1.toDense
255          val denseGrad2 = grad2.toDense
256          axpy(1.0, denseGrad2, denseGrad1)
257          (denseGrad1, loss1 + loss2)
258        }
259
260      val zeroSparseVector = Vectors.sparse(n, Seq.empty)
261      val (gradientSum, lossSum) = data.treeAggregate((zeroSparseVector, 0.0))(seqOp, combOp)
262
263      // broadcasted model is not needed anymore
264      bcW.destroy()
265    }
```

```
266  /**
267   * regVal is sum of weight squares if it's L2 updater;
268   * for other updater, the same logic is followed.
269   */
270  val regVal = updater.compute(w, Vectors.zeros(n), 0, 1, regParam)._2
271
272  val loss = lossSum / numExamples + regVal
273  /**
274   * It will return the gradient part of regularization using updater.
275   *
276   * Given the input parameters, the updater basically does the following,
277   *
278   *  $w' = w - \text{thisIterStepSize} * (\text{gradient} + \text{regGradient}(w))$ 
279   * Note that regGradient is function of w
280   *
281   * If we set gradient = 0, thisIterStepSize = 1, then
282   *
283   *  $\text{regGradient}(w) = w - w'$ 
284   *
285   * TODO: We need to clean it up by separating the logic of regularization out
286   *       from updater to regularizer.
287   */
288  // The following gradientTotal is actually the regularization part of gradient.
289  // Will add the gradientSum computed from the data with weights in the next step.
290  val gradientTotal = w.copy
291  axpy(-1.0, updater.compute(w, Vectors.zeros(n), 1, 1, regParam)._1, gradientTotal)
292
293  // gradientTotal = gradientSum / numExamples + gradientTotal
294  axpy(1.0 / numExamples, gradientSum, gradientTotal)
295
296  (loss, gradientTotal.asBreeze.asInstanceOf[BDV[Double]])
297  }
298  }
299  }
```

- LBFGS [spark/mllib/src/main/scala/org/apache/spark/mllib/optimization/LBFGS.scala](https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/optimization/LBFGS.scala)

Breeze: a library for numerical processing in Scala

```
90 object LBFGS {
91   case class ApproximateInverseHessian[T](
92     m: Int,
93     private[LBFGS] val memStep: IndexedSeq[T] = IndexedSeq.empty,
94     private[LBFGS] val memGradDelta: IndexedSeq[T] = IndexedSeq.empty)(
95     implicit space: MutableInnerProductModule[T, Double])
96     extends NumericOps[ApproximateInverseHessian[T]] {
97
98     import space._
99
100    def repr: ApproximateInverseHessian[T] = this
101
102    def updated(step: T, gradDelta: T) = {
103      val memStep = (step +: this.memStep).take(m)
104      val memGradDelta = (gradDelta +: this.memGradDelta).take(m)
105
106      new ApproximateInverseHessian(m, memStep, memGradDelta)
107    }
108
109    def historyLength = memStep.length
110
111    def *(grad: T) = {
112      val diag = if (historyLength > 0) {
113        val prevStep = memStep.head
114        val prevGradStep = memGradDelta.head
115        val sy = prevStep.dot(prevGradStep)
116        val yy = prevGradStep.dot(prevGradStep)
117        if (sy < 0 || sy.isNaN) throw new NaNHistory
118        sy / yy
119      } else {
120        1.0
121      }
122    }
```

```
123     val dir = space.copy(grad)
124     val as = new Array[Double](m)
125     val rho = new Array[Double](m)
126
127     for (i <- 0 until historyLength) {
128       rho(i) = memStep(i).dot(memGradDelta(i))
129       as(i) = (memStep(i).dot(dir)) / rho(i)
130       if (as(i).isNaN) {
131         throw new NaNHistory
132       }
133       axpy(-as(i), memGradDelta(i), dir)
134     }
135
136     dir *= diag
137
138     for (i <- (historyLength - 1) to 0 by (-1)) {
139       val beta = (memGradDelta(i).dot(dir)) / rho(i)
140       axpy(as(i) - beta, memStep(i), dir)
141     }
142
143     dir *= -1.0
144     dir
145   }
146 }
147
148 implicit def multiplyInverseHessian[T](
149   implicit vspace: MutableInnerProductModule[T, Double]): OpMulMatrix.Impl2[ApproximateInverseHessian[T], T, T] = {
150   new OpMulMatrix.Impl2[ApproximateInverseHessian[T], T, T] {
151     def apply(a: ApproximateInverseHessian[T], b: T): T = a * b
152   }
153 }
154 }
```

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-Wiley 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)

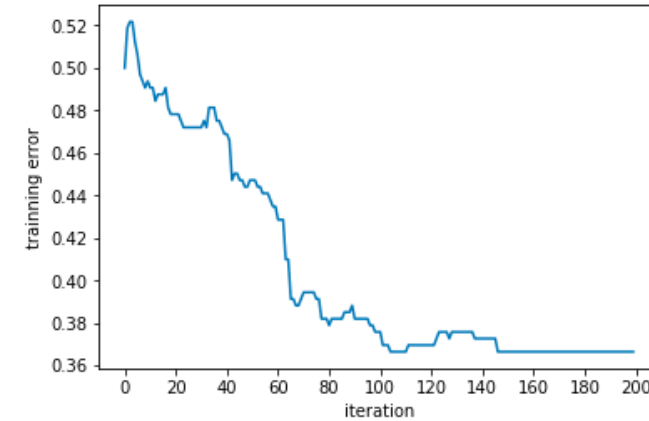
- A. Smola and S. Narayanamurthy, [An Architecture for Parallel Topic Models](#), VLDB 2010
- Li et al, [Scaling Distributed Machine Learning with the Parameter Server](#), OSDI 2014
- E. P. Xing et al, [Petuum: A New Platform for Distributed Machine Learning on Big Data](#), ACM KDD 2015
- M. Li et 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



*Pipeline
(Estimator)*

Pipeline.fit()

Tokenizer

HashingTF

Logistic
Regression



Raw
text



Words



Feature
vectors



Logistic
Regression
Model

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