PART B. GEAR SESSIONS
SESSION 2: MACHINE LEARNING FOR BIG DATA

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

FAQs

• Lossy Algorithm
Topics of Today's Class

- Programming Assignment #2 Lossy Algorithm
- GEAR Session 2. Machine Learning for Big Data
- Lecture 2.
  - Distributed Optimization Problem in Machine Learning

Programming Assignment 2
Lossy Counting Algorithm
Solving frequent element

Motwani, R; Manku, G.S (2002). "Approximate frequency counts over data streams". VLDB '02 Proceedings of the 28th international conference on Very Large Data Bases: 346–357

Algorithm

- Divide the incoming stream into buckets of \( w = \frac{1}{\varepsilon} \)
- Each buckets are labeled with integer starting from 1
- Current bucket number = \( b_{\text{current}} \)
- \( b_{\text{current}} = \frac{N}{w} \)
- True frequency of an element \( e = f_e \)

- Data structure
  - \((e, f, \Delta)\)
  - \( e \) is an element in the stream
  - \( f \) is an integer representing its estimated frequency
  - \( \Delta \) is a maximum possible error in \( f \)
• When an element arrives
  • Lookup to see if there is an entry for that element already exists
    • If there is an entry, increase its frequency \( f \) by one
    • Otherwise, create a new entry of the form \((e, f, \Delta) = (e, f, b_{current} - 1)\)

• When the new elements fill up the bucket
  • \( N \mod w = 0 \)
  • Prune elements
    • \((e,f,\Delta)\) is deleted if \( f + \Delta \leq b_{current} \)

• When user request a list of item with threshold \( s \)
  • Outputs are items that \( f \geq (s-\epsilon)N \)

---

**Example (\( \epsilon = 0.2, w = 1/\epsilon = 5 \), 1st bucket)**

\( \epsilon = 0.2 \)
\( w = 1/\epsilon = 5 \)  (5 items per “bucket”)

<table>
<thead>
<tr>
<th>Bucket 1</th>
<th>Bucket 2</th>
<th>Bucket 3</th>
<th>Bucket 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,4,3,4</td>
<td>3,4,5,4,6</td>
<td>7,3,3,6,1</td>
<td>1,3,2,4,7</td>
</tr>
</tbody>
</table>

\( b_{current} = 1 \)  inserted: 1 2 4 3 4

**Insert phase:**
D (before removing): (x=1;f=1;\( \Delta = 0 \)) (x=2;f=1;\( \Delta = 0 \)) (x=4;f=2;\( \Delta = 0 \)) (x=3;f=1;\( \Delta = 0 \))

**Delete phase**: delete elements with \( f + \Delta \leq b_{current} \) (=1)
D (after removing): (x=4;f=2;\( \Delta = 0 \))

**NOTE**: elements with frequencies \( \leq 1 \) are deleted
New elements added has maximum count error of 0
## Example ($\varepsilon = 0.2, \ w = 1/\varepsilon = 5$), 2nd bucket

$\varepsilon = 0.2$

$w = 1/\varepsilon = 5$  (5 items per "bucket")

<table>
<thead>
<tr>
<th>Bucket 1</th>
<th>Bucket 2</th>
<th>Bucket 3</th>
<th>Bucket 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,4,3,4</td>
<td>3,4,5,4,6</td>
<td>7,3,3,6,1</td>
<td>1,3,2,4,7</td>
</tr>
</tbody>
</table>

[Bucket 2]
- $b_{current} = 2, \ b_{maximum} = 3$
- Inserted: 3,4,5,4,6

**Insert phase:**
- D (before removing): $(x=4; f=4; \Delta = 0)$
- $(x=3; f=1; \Delta = 1)$
- $(x=5; f=1; \Delta = 1)$
- $(x=6; f=1; \Delta = 1)$

**Delete phase:**
- delete elements with $f + \Delta \leq b_{current} (=2)$
  - D (after removing): $(x=4; f=4; \Delta = 0)$

**NOTE:** elements with frequencies $\leq 2$ are deleted

New elements added has maximum count error of 1

## Example ($\varepsilon = 0.2, \ w = 1/\varepsilon = 5$), 3rd bucket

$\varepsilon = 0.2$

$w = 1/\varepsilon = 5$  (5 items per "bucket")

<table>
<thead>
<tr>
<th>Bucket 1</th>
<th>Bucket 2</th>
<th>Bucket 3</th>
<th>Bucket 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,4,3,4</td>
<td>3,4,5</td>
<td>7,3,3,6,1</td>
<td>1,3,2,4,7</td>
</tr>
</tbody>
</table>

[Bucket 3]
- $b_{current} = 3, \ b_{maximum} = 3$
- Inserted: 7,3,3,6,1

**Insert phase:**
- D (before removing): $(x=7; f=1; \Delta = 2)$
- $(x=3; f=2; \Delta = 2)$
- $(x=4; f=4; \Delta = 0)$
- $(x=6; f=1; \Delta = 2)$
- $(x=1; f=1; \Delta = 2)$

**Delete phase:**
- delete elements with $f + \Delta \leq b_{current} (=3)$
  - D (after removing): $(x=4; f=4; \Delta = 0)$
  - $(x=3; f=2; \Delta = 2)$

**NOTE:** elements with frequencies $\leq 3$ are deleted

New elements added has maximum count error of 2
### Example ($\varepsilon = 0.2$, $w = 1/\varepsilon = 5$), 4th bucket

$\varepsilon = 0.2$

$w = 1/\varepsilon = 5$ (5 items per "bucket")

<table>
<thead>
<tr>
<th>bucket 1</th>
<th>bucket 2</th>
<th>bucket 3</th>
<th>bucket 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2,4,3,4</td>
<td>3,4,5,4,6</td>
<td>7,3,3,6,1</td>
<td>1,3,2,4,7</td>
</tr>
</tbody>
</table>

[Bucket 4]

$b_{current} = 4$ inserted: 1 3 2 4 7

**Insert phase:**
- $D$ (before removing): ($x=4;f=5;\Delta=0$) ($x=3;f=3;\Delta=2$) ($x=1;f=1;\Delta=3$) ($x=2;f=1;\Delta=3$) ($x=7;f=1;\Delta=3$)

**Delete phase:** delete elements with $f + \Delta \leq b_{current} = 4$

$D$ (after removing): ($x=4;f=5;\Delta=0$) ($x=3;f=3;\Delta=2$)

**NOTE:** elements with frequencies $\leq 4$ are deleted
New elements added has maximum count error of 3

### Example ($\varepsilon = 0.2$, $w = 1/\varepsilon = 5$), Output

$\varepsilon = 0.2$

$w = 1/\varepsilon = 5$ (5 items per "bucket")

| 1,2,4,3,4 | 3,4,5,4,6 | 7,3,3,6,1 | 1,3,2,4,7 |

$D$ : ($x=4;f=5;\Delta=0$) ($x=3;f=3;\Delta=2$)

For the threshold $s = 0.3$ (so far, $N=20$)

$(s-\varepsilon) N = (0.3-0.2) \times 20 = 2$

There are only two elements available:

<table>
<thead>
<tr>
<th>Item</th>
<th>$f_{estimated}$</th>
<th>$f_{actual}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

If $s = 0.5$?
No element will be returned
Why does it work?

• Lemma 1.
  \( b_{current} \) is at a bucket boundary
  Where the most recently started new bucket
  The approximate value of \( b_{current} = \varepsilon \times N \)

• Lemma 2.
  • If an entity \((e; f; \Delta)\) is deleted in the delete phase of the algorithm when \( b_{current} = k \) then
    • The number of occurrences of \( e \) (actual count \( f_e \)) is less than or equal to \( k \)
    • \( f_e \leq b_{current} \)

Infrequent Items are NOT included in \( D \)

• Lemma 3.
  • If an item \( e \) is not included \( D \), then \( f_e \leq \varepsilon \times N \)
    • i.e., the true frequency count of \( e \) is less than or equal to \( \varepsilon \times N \)

• Case 1. trivial case
  
  If \( e \) does not appear in the input stream, then trivially, the entry \((e, f, \Delta)\) was never entered into \( D \) and hence, \((e, f, \Delta) \notin D \)
  
  We have then:
  
  \[ f_e = 0 \]
  and trivially:

  \[ f_e \ (= 0) \leq \varepsilon \times N \]
  
  is true.
Lemma 3: continued

- Case 2:
  - If $e$ was in the input stream, and the entry $(e, f, \Delta)$ is not in the output set $D$, then $(e, f, \Delta)$ was deleted in some bucket.

  - The maximum actual frequency of $e$ is $f_e = f + \Delta$
  - According to lemma 2,
    - Because $(e, f, \Delta)$ is deleted in bucket $b_{current}$, the actual count at that moment $f_e \leq b_{current}$

  - Since Lemma 3 is true, (If $(e, f, \Delta) \notin D$, when the algorithm terminates then, the actual frequency of item $e$: $f_e \leq \epsilon \times N$)
  - By rules of negation,
    - If the actual frequency of item $e$: $f_e > \epsilon \times N$ then, $(e, f, \Delta) \in D$, when the algorithm terminates
Difference between true frequency count and approximate frequency count

- Lemma 4.
- If \((e, f, \Delta) \in D\), then: \(f \leq f_e \leq f + \varepsilon \times N\)

**Proof.**
- Part 1. \(f \leq f_e\)
  - Since the value \(f\) (variable in the algorithm) count the item \(e\) in the input after the entry \((e, f, \Delta)\) has been inserted in \(D\), and the entry \((e, f, \Delta)\) may have been deleted before, it is obvious that \(f \leq f_e\)

Lemma 4: continued

- Part 2. \(f_e \leq f + \varepsilon \times N\)

  - The only occurrences of \(e\) that the algorithm fails to count are those that appeared *prior* to the bucket \(\Delta + 1\).
Lemma 4: continued

• The maximum number of missing count (worst case scenario) happens when the entry \((e, f, \Delta)\) was deleted in the bucket just prior to the bucket \(\Delta+1\) (in which \((e, f=1, \Delta)\) was entered into \(D\))

• By Lemma 2, at the moment of deletion, the actual frequency count of item \(e\) is at most:
  \[ f_e \leq b_{current} \]
  - With Lemma 1, \(f_e \leq b_{current} = \varepsilon \times N^*\)
  - where \(N^*\) is the number to items processed at the end of bucket \(\Delta\)
  - Therefore, \(f_e \leq b_{current} = \varepsilon \times N^* \leq \varepsilon \times N\)
  - Thus, \(f_e \leq \varepsilon \times N\)

GEAR Session 2. Machine Learning for Big Data
Lecture 2. Distributed Deep Learning Models
What is the optimization problem in ML?
What is “optimization”? 

• Finding one or more minimizer of a function subject to constraints 
  • Most of machine learning problems are optimization problems in computing

• For example, in $k$-Means clustering 
  • Looks for $k$-clusters in which each observation belongs to the cluster with the nearest mean 
  • In this case, “optimization” is the process to find:
    
    $$\arg \min_{\mu_1, \mu_2, \ldots, \mu_k} J(\mu) = \sum_{i=1}^{k} \sum_{j \in C_j} \| x_i - \mu_j \|^2$$

Sometimes, optimization is NOT straightforward 

• Minimize $f(x)$?
Convex optimization

- Convex function
- Definition
  - A function $f: \mathbb{R}^n \to \mathbb{R}$ is convex if for $x, y \in \text{dom } f$ and any $a \in [0, 1],
  \[ f(ax + (1-a)y) \leq af(x) + (1-a)f(y) \]

Convexity optimization

- Theorem
  - If $x$ is a local minimizer of a convex optimization problem, it is a global minimizer
Optimizations in Apache Spark

- Spark supports
  - Gradient descent
  - Stochastic gradient descent (SGD)
  - Limited-memory BFGS (L-BFGS)
Gradient Descent

• The simplest method to solve optimization problems
  • Achieve $\min_{w \in \mathbb{R}^d} f(w)$
  • Suitable for large scale and distributed computation

• Finds a local minimum of a function by iteratively taking steps in the direction of steepest descent
  • Negative of the derivative (gradient) of the function at the current point

Fitting the linear regression model

• Linear regression model
  $$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 \ldots$$

• Example: Predict the student’s science score based on the math score
  $$h_\theta(x) = \theta_0 + \theta_1 x$$

How big is the error of the fitted model? We would like to minimize this error
Objective function (Cost function)

- For a given training set, how do we pick, or learn the parameter $\theta$?

- Make $h(x)$ close to $y$
  - Make your prediction close to the real observation

- We define the objective (cost) function
  - Using Mean Squared Error and multiplying $\frac{1}{2}$ for convenience

$$J(\theta) = \frac{1}{2m} \sum_{i=0}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Minimization problem

- We have a function $J(\theta_0, \theta_1)$
- We want to find $\min_{\theta_0, \theta_1} J(\theta_0, \theta_1)$

- Goal: Find parameters to minimize the cost (output of the objective function)

- Outline of our approach:
  - Start with some $\theta_0, \theta_1$
  - Keep changing $\theta_0, \theta_1$ to reduce $J(\theta_0, \theta_1)$ until we end up at a minimum
Gradient descent algorithm

Repeat until convergence {

\[ \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1) \]

(for j=0 and j=1)

\[ \theta_0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) \]

\[ \theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) \]

Decreasing and increasing \( \theta_j \)
Decreasing $\theta_1$

- **Positive** slope

\[ \theta_1 = \theta_0 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) \]

Increasing $\theta_1$

- **Negative** slope

\[ \theta_1 = \theta_0 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) \]
GEAR Session 2. Machine Learning for Big Data
Lecture 2. Distributed Deep Learning Models
Stochastic Gradient Descent

Stochastic Gradient Descent (SGD)

• Batch methods
  • Full training set to compute the next update to parameters at each iteration tend to converge very well

• Advantage
  • Straight forward to get working provided a good off the shelf implementation
    • Very few hyper-parameters to tune

• Disadvantages
  • Computing the cost and gradient for the entire training set can be very slow
    • Intractable on a single machine if the dataset is too big to fit in main memory
  • No easy way to incorporate new data in an 'online'
Stochastic Gradient Descent (SGD)

- Stochastic Gradient Descent (SGC)
  - Follows the negative gradient of the objective after seeing only a single or a few training examples
  - The use of SGD in the neural network setting is motivated by the high cost of running back propagation over the full training set.
  - Fast convergence

Stochastic Gradient Descent

- The standard gradient descent algorithm updates the parameters $\theta$ of the objective $J(\theta)$ as,

$$\theta = \theta - \alpha \nabla \theta E[J(\theta)]$$

, where the function evaluates the cost and gradient over the full training set

- Stochastic Gradient Descent (SGD) uses only a single or a few training examples

$$\theta = \theta - \alpha \nabla \theta J(\theta; x^{(i)}, y^{(i)})$$

- with a pair $(x^{(i)}, y^{(i)})$ from the training set
SGD used in supervised machine learning with Spark

\[ f(w) = \lambda R(w) + \frac{1}{n} \sum_{i=1}^{n} L(w; x_i, y_i). \tag{1} \]

- where \( f(W) \) is gradient descent
- Optimization formulation used in Spark
- The loss is written as an average of the individual losses coming from each data point

- A stochastic subgradient is a randomized choice of a vector
  - Selects one datapoint \( i \in [1..n] \) uniformly at random, to obtain a stochastic subgradient of (1) with respect to \( w \) as follows:
    \[ f'_{w,i} = L'_{w,i} + \lambda R'_w \]
  - Where \( L'_{w,i} \) is a sub-gradient of the part of the loss function determined by \( i-th \) data point
  - \( R'_w \) is a sub-gradient of the regularizer \( R(w) \)
  \[ R'_w \in \frac{\partial}{\partial w} R(w) \]

- Running SGD is now walking in the direction of the negative stochastic sub-gradient \( f'_{w,i} \)
  \[ w^{(t+1)} = w^{(t)} - \gamma f'_{w,i} \]
  - \( \gamma \) is the step size
  - Default implementation is decreasing with the squarer root of the iteration counter

http://www.cs.colostate.edu/~cs535
SGD used in supervised machine learning with Spark: Update schemes for Distributed SGD

- SGD uses a simple (distributed) sampling of the data examples
- Recall the SGD optimization problem-(1):

\[ f(w) = \lambda R(w) + \frac{1}{n} \sum_{i=1}^{n} L(w; x_i, y_i). \quad \text{--(1)} \]

- Here, the loss part of the optimization problem

\[ \frac{1}{n} \sum_{i=1}^{n} L(w; x_i, y_i) \]

- Therefore, the true sub-gradient:

\[ \frac{1}{n} \sum_{i=1}^{n} L'_{w,i} \]

- This will require access to the full dataset

---

In Apache Spark, the parameter miniBatchFraction specifies which fraction of the full data

- The average of the gradients within this subset

\[ \frac{1}{|S|} \sum_{i \in S} L'_{w,i} \]

- Will be an actual stochastic gradient
- Here, |S| is the sample subset size

- In each iteration, Spark performs sampling in its RDDs
SGD used in supervised machine learning with Spark: Update schemes for Distributed SGD

- $|S|$: size of the sampled subset
- $|S| = \text{miniBatchFraction} \times n$
- If $|S| = 1$, it is equivalent to the standard SGD
- If $\text{miniBatchFraction} = 1$, it is equivalent to the batch SGD
GEAR Session 2. Machine Learning for Big Data
Lecture 2. Distributed Deep Learning Models
Limited Memory BFGS

Limited-memory BFGS (L-BFGS)

- BFGS (Broyden–Fletcher–Goldfarb–Shanno algorithm)
- Iterative method for solving unconstrained nonlinear optimization problems
  - Objective functions are non-linear

- A type of quasi-Newton methods
**Limited-memory BFGS (L-BFGS)**

- L-BFGS algorithm approximates BFGS algorithm using limited amount of memory
- Stores last $M$ value/gradient pairs and uses them to build positive definite Hessian approximation
  - This approximate Hessian matrix is used to make quasi-Newton step
- If quasi-Newton step does not lead to sufficient decrease of the value/gradient,
  - The algorithm makes line search along direction of this step
- Only last $M$ function/gradient pairs are used
  - $M$ is moderate number smaller than problem size $N$, often as small as 3-10
  - Very cheap iterations, which cost just $O(N \cdot M)$ operations.

**Choosing an optimization method**

- Linear methods use optimization internally
  - Linear SVM, logistic regression, regressions (Linear least squares, Lasso)
- Some linear methods in `spark.mllib` support both SGD and L-BFGS
- Different optimization methods can have different convergence guarantees
  - depending on the properties of the objective function
- In general, when L-BFGS is available, we recommend using it instead of SGD since L-BFGS tends to converge faster (in fewer iterations)
GD and SGD: Implementation in MLlib

- Gradient descent methods including stochastic sub-gradient descent (SGD) as included as a low-level primitive in MLlib

- `Vector optimize(RDD<scala.Tuple2<Object, Vector>>data.Vector
initialWeights)`

- The SGD class `GradientDescent` sets the following parameters:
  - **Gradient**
    - A class that computes the stochastic gradient of the function being optimized, i.e., with respect to a single training example, at the current parameter value
    - MLlib includes gradient classes for common loss functions
    - e.g., hinge, logistic, least-squares
    - The gradient class takes as input a training example, its label, and the current parameter value.

GD and SGD: Implementation in MLlib

- **Updater**
  - A class that performs the actual gradient descent step
  - i.e. updating the weights in each iteration, for a given gradient of the loss part
  - The updater is also responsible to perform the update from the regularization part.
  - MLlib includes updaters for cases without regularization, as well as L1 and L2 regularizers

- **stepSize**
  - A scalar value denoting the initial step size for gradient descent. All updaters in MLlib use a step size at the \( t \)-th step equal to \( \frac{\text{stepSize}}{\sqrt{t}} \)

- **numIterations**
  - The number of iterations to run.
GD and SGD: Implementation in MLlib

- **regParam**
  - The regularization parameter when using L1 or L2 regularization

- **miniBatchFraction**
  - The fraction of the total data that is sampled in each iteration, to compute the gradient direction.
  - Sampling still requires a pass over the entire RDD, so decreasing miniBatchFraction may not speed up optimization much. Users will see the greatest speedup when the gradient is expensive to compute, for only the chosen samples are used for computing the gradient.

Questions?