Implementing Randomized Matrix Algorithms in Parallel and Distributed Environments

Michael W. Mahoney

Stanford University

(For more info, see: http://cs.stanford.edu/people/mmahoney/or Google on “Michael Mahoney”)

October 2013
Outline

1. General thoughts
2. Randomized regression in RAM
3. Solving $\ell_2$ regression using MPI
4. Solving $\ell_1$ regression on MapReduce
Goal: very large-scale “vector space analytics”

Small-scale and medium-scale:
- Model data by graphs and matrices
- Compute eigenvectors, correlations, etc. in RAM

Very large-scale:
- Model data with flat tables and the relational model
- Compute with join/select and other “counting” in, e.g., Hadoop

Can we “bridge the gap” and do “vector space computations” at very large scale?
- Not obviously yes: exactly computing eigenvectors, correlations, etc. is subtle and uses lots of communication.
- Not obviously no: lesson from random sampling algorithms is you can get $\epsilon$-approximation of optimal with very few samples.
Why randomized matrix algorithms?

Traditional matrix algorithms (direct & iterative methods, interior point, simplex, etc.) are designed to work in RAM and their performance is measured in floating-point operations per second (FLOPS).

- **Traditional algorithms** are NOT well-suited for:
  - problems that are very large
  - distributed or parallel computation
  - when communication is a bottleneck
  - when the data must be accessed via “passes”

- **Randomized matrix algorithms** are:
  - faster: better theory
  - simpler: easier to implement
  - implicitly regularize: noise in the algorithm avoids overfitting
  - inherently parallel: exploiting modern computer architectures
  - more scalable: modern massive data sets
Traditional algorithms

- **for $\ell_2$ regression:**
  - direct methods: QR, SVD, and normal equation ($O(mn^2 + n^2)$ time)
    - ✤ Pros: high precision & implemented in LAPACK
    - ✤ Cons: hard to take advantage of sparsity & hard to implement in parallel environments
  - iterative methods: CGLS, LSQR, etc.
    - ✤ Pros: low cost per iteration, easy to implement in some parallel environments, & capable of computing approximate solutions
    - ✤ Cons: hard to predict the number of iterations needed

- **for $\ell_1$ regression:**
  - linear programming
  - interior-point methods (or simplex, ellipsoid? methods)
  - re-weighted least squares
  - first-order methods

Nearly all traditional algorithms for low-rank matrix problems, continuous optimization problems, etc. boil down to variants of these methods.
An $\ell_p$ regression problem is specified by a design matrix $A \in \mathbb{R}^{m \times n}$, a response vector $b \in \mathbb{R}^m$, and a norm $\| \cdot \|_p$:

$$\text{minimize}_{x \in \mathbb{R}^n} \quad \|Ax - b\|_p.$$ 

Assume $m \gg n$, i.e., many more “constraints” than “variables.” Given an $\epsilon > 0$, find a $(1 + \epsilon)$-approximate solution $\hat{x}$ in relative scale, i.e.,

$$\|A\hat{x} - b\|_p \leq (1 + \epsilon)\|Ax^* - b\|_p,$$

where $x^*$ is a/the optimal solution.

- $p = 2$: Least Squares Approximation: Very widely-used, but highly non-robust to outliers.
- $p = 1$: Least Absolute Deviations: Improved robustness, but at the cost of increased complexity.
Large-scale environments and how they scale

- **Shared memory**
  - cores: \([10, 10^3]\)^*
  - memory: \([100GB, 100TB]\)

- **Message passing**
  - cores: \([200, 10^5]\)^†
  - memory: \([1TB, 1000TB]\)
  - CUDA cores: \([5 \times 10^4, 3 \times 10^6]\)^‡
  - GPU memory: \([500GB, 20TB]\)

- **MapReduce**
  - cores: \([40, 10^5]\)^§
  - memory: \([240GB, 100TB]\)
  - storage: \([100TB, 100PB]\)^¶

- **Distributed computing**
  - cores: \([-, 3 \times 10^5]\)^∥

---

† http://www.top500.org/list/2011/11/100
‡ http://i.top500.org/site/50310
¶ http://hortonworks.com/blog/an-introduction-to-hdfs-federation/
∥ http://fah-web.stanford.edu/cgi-bin/main.py?qtype=osstats
Outline

1. General thoughts

2. Randomized regression in RAM

3. Solving $\ell_2$ regression using MPI

4. Solving $\ell_1$ regression on MapReduce
Two important notions: leverage and condition

(Mahoney, “Randomized Algorithms for Matrices and Data,” FnTML, 2011.)

**Statistical leverage.** (Think: eigenvectors. Important for low-precision.)
- The *statistical leverage scores* of $A$ (assume $m \gg n$) are the diagonal elements of the projection matrix onto the column span of $A$.
- They equal the $\ell_2$-norm-squared of any orthogonal basis spanning $A$.
- They measure:
  - how well-correlated the singular vectors are with the canonical basis
  - which constraints have largest “influence” on the LS fit
  - a notion of “coherence” or “outlierness”
- Computing them exactly is as hard as solving the LS problem.

**Condition number.** (Think: eigenvalues. Important for high-precision.)
- The *$\ell_2$-norm condition number* of $A$ is $\kappa(A) = \sigma_{\text{max}}(A)/\sigma_{\text{min}}^+(A)$.
- $\kappa(A)$ bounds the number of iterations; for ill-conditioned problems (e.g., $\kappa(A) \approx 10^6 \gg 1$), the convergence speed is very slow.
- Computing $\kappa(A)$ is generally as hard as solving the LS problem.

These are for the $\ell_2$-norm. Generalizations exist for the $\ell_1$-norm.
Meta-algorithm for $\ell_2$-norm regression

(Drineas, Mahoney, etc., 2006, 2008, etc., starting with SODA 2006; Mahoney FnTML, 2011.)

1: Using the $\ell_2$ statistical leverage scores of $A$, construct an importance sampling distribution $\{p_i\}_{i=1}^m$.

2: Randomly sample a small number of constraints according to $\{p_i\}_{i=1}^m$ to construct a subproblem.

3: Solve the $\ell_2$-regression problem on the subproblem.

A naïve version of this meta-algorithm gives a $1 + \epsilon$ relative-error approximation in roughly $O(mn^2/\epsilon)$ time (DMM 2006, 2008). (Ugh.)
We can make this meta-algorithm “fast” in RAM:**
- This meta-algorithm runs in $O(mn \log n/\epsilon)$ time in RAM if:
  - we perform a Hadamard-based random projection and sample uniformly in a randomly rotated basis, or
  - we quickly computing approximations to the statistical leverage scores and using those as an importance sampling distribution.

We can make this meta-algorithm “high precision” in RAM:††
- This meta-algorithm runs in $O(mn \log n \log(1/\epsilon))$ time in RAM if:
  - we use the random projection/sampling basis to construct a preconditioner and couple with a traditional iterative method.

---

** (Sarlós 2006; Drineas, Mahoney, Muthu, Sarlós 2010; Drineas, Magdon-Ismail, Mahoney, Woodruff 2011.)
†† (Rokhlin & Tygert 2008; Avron, Maymounkov, & Toledo 2010; Meng, Saunders, & Mahoney 2011.)
‡‡ (Mahoney, “Randomized Algorithms for Matrices and Data,” FnTML, 2011.)
Conclusions:

- **Randomized algorithms** “beats Lapack’s direct dense least-squares solver by a large margin on essentially any dense tall matrix.”

- **These results** “suggest that random projection algorithms should be incorporated into future versions of Lapack.”
Computing large rectangular regressions/SVDs/CUR decompositions:

- In commodity hardware (e.g., a 4GB RAM, dual-core laptop), using MatLab 7.0 (R14), the computation of the SVD of the dense 2,240 × 447,143 matrix A takes about 20 minutes.
- Computing this SVD is not a one-liner—we can not load the whole matrix in RAM (runs out-of-memory in MatLab).
- Instead, compute the SVD of AAT.
- In a similar experiment, compute 1,200 SVDs on matrices of dimensions (approx.) 1,200 × 450,000 (roughly, a full leave-one-out cross-validation experiment).
A retrospective

Randomized matrix algorithms:

- BIG success story in high precision scientific computing applications and *large-scale* statistical data analysis!

- Can they *really* be implemented in parallel and distributed environments for *LARGE-scale* statistical data analysis?
Outline

1. General thoughts
2. Randomized regression in RAM
3. Solving $\ell_2$ regression using MPI
4. Solving $\ell_1$ regression on MapReduce
Algorithm LSRN (for strongly over-determined systems)

(Meng, Saunders, and Mahoney 2011)

1: Choose an oversampling factor $\gamma > 1$, e.g., $\gamma = 2$. Set $s = \lceil \gamma n \rceil$.
2: Generate $G = \text{randn}(s, m)$, a Gaussian matrix.
3: Compute $\tilde{A} = GA$.
4: Compute $\tilde{A}$’s economy-sized SVD: $\tilde{U} \tilde{\Sigma} \tilde{V}^T$.
5: Let $N = \tilde{V} \tilde{\Sigma}^{-1}$.
6: Iteratively compute the min-length solution $\hat{y}$ to

$$\min_{y \in \mathbb{R}^r} \| A Ny - b \|_2.$$

7: Return $\hat{x} = N \hat{y}$. 
Why we choose Gaussian random projection

(Meng, Saunders, and Mahoney 2011)

Gaussian random projection
- has the best theoretical result on conditioning,
- can be generated super fast,
- uses level 3 BLAS on dense matrices,
- speeds up automatically on sparse matrices and fast operators,
- still works (with an extra “allreduce” operation) when $A$ is partitioned along its bigger dimension.

So, although it is “slow” (compared with “fast” Hadamard-based projections i.t.o. FLOPS), it allows for better communication properties.
Theoretical properties of LSRN

(Meng, Saunders, and Mahoney 2011)

- In exact arithmetic, $\hat{x} = x^*$ almost surely.
- The distribution of the spectrum of $AN$ is the same as that of the pseudoinverse of a Gaussian matrix of size $s \times r$.
- $\kappa(AN)$ is independent of all the entries of $A$ and hence $\kappa(A)$.
- For any $\alpha \in (0, 1 - \sqrt{r/s})$, we have

$$P \left( \kappa(AN) \leq \frac{1 + \alpha + \sqrt{r/s}}{1 - \alpha - \sqrt{r/s}} \right) \geq 1 - 2e^{-\alpha^2 s/2},$$

where $r$ is the rank of $A$.

So, if we choose $s = 2n \geq 2r$, we have $\kappa(AN) < 6$ w.h.p., and hence we only need around 100 iterations to reach machine precision.
Implementation of LSRN

(Meng, Saunders, and Mahoney 2011)

- Shared memory (C++ with MATLAB interface)
  - Multi-threaded ziggurat random number generator (Marsaglia and Tsang 2000), generating $10^9$ numbers in less than 2 seconds using 12 CPU cores.
  - A naïve implementation of multi-threaded dense-sparse matrix multiplications.

- Message passing (Python)
  - Multi-threaded BLAS/LAPACK for SVD.
  - Using the Chebyshev semi-iterative method (Golub and Varga 1961) instead of LSQR.
### Solving real-world problems

**Table:** Real-world problems and corresponding running times. DGELSD doesn’t take advantage of sparsity. Though MATLAB’s backslash may not give the min-length solutions to rank-deficient or under-determined problems, we still report its running times. Blendenpik either doesn’t apply to rank-deficient problems or runs out of memory (OOM). LSRN’s running time is mainly determined by the problem size and the sparsity.

<table>
<thead>
<tr>
<th>matrix</th>
<th>m</th>
<th>n</th>
<th>nnz</th>
<th>rank</th>
<th>cond</th>
<th>DGELSD</th>
<th>$A \backslash b$</th>
<th>Blendenpik</th>
<th>LSRN</th>
</tr>
</thead>
<tbody>
<tr>
<td>landmark</td>
<td>71952</td>
<td>2704</td>
<td>1.15e6</td>
<td>2671</td>
<td>1.0e8</td>
<td>29.54</td>
<td>0.6498 *</td>
<td>-</td>
<td>17.55</td>
</tr>
<tr>
<td>rail4284</td>
<td>4284</td>
<td>1.1e6</td>
<td>1.1e7</td>
<td>full</td>
<td>400.0</td>
<td>&gt; 3600</td>
<td>1.203 *</td>
<td>OOM</td>
<td>136.0</td>
</tr>
<tr>
<td>tning_1</td>
<td>951</td>
<td>1e6</td>
<td>2.1e7</td>
<td>925</td>
<td>-</td>
<td>630.6</td>
<td>1067 *</td>
<td>-</td>
<td>36.02</td>
</tr>
<tr>
<td>tning_2</td>
<td>1000</td>
<td>2e6</td>
<td>4.2e7</td>
<td>981</td>
<td>-</td>
<td>1291</td>
<td>&gt; 3600 *</td>
<td>-</td>
<td>72.05</td>
</tr>
<tr>
<td>tning_3</td>
<td>1018</td>
<td>3e6</td>
<td>6.3e7</td>
<td>1016</td>
<td>-</td>
<td>2084</td>
<td>&gt; 3600 *</td>
<td>-</td>
<td>111.1</td>
</tr>
<tr>
<td>tning_4</td>
<td>1019</td>
<td>4e6</td>
<td>8.4e7</td>
<td>1018</td>
<td>-</td>
<td>2945</td>
<td>&gt; 3600 *</td>
<td>-</td>
<td>147.1</td>
</tr>
<tr>
<td>tning_5</td>
<td>1023</td>
<td>5e6</td>
<td>1.1e8</td>
<td>full</td>
<td>-</td>
<td>&gt; 3600</td>
<td>&gt; 3600 *</td>
<td>OOM</td>
<td>188.5</td>
</tr>
</tbody>
</table>
Iterating with LSQR

(Paige and Saunders 1982)

Code snippet (Python):

```python
u = A.matvec(v) - alpha*u
beta = sqrt(comm.allreduce(np.dot(u,u)))
...
v = comm.allreduce(A.rmatvec(u)) - beta*v
```

Cost per iteration:
- two matrix-vector multiplications
- two cluster-wide synchronizations
Iterating with Chebyshev semi-iterative (CS) method

(Golub and Varga 1961)

The strong concentration results on $\sigma_{\text{max}}(AN)$ and $\sigma_{\text{min}}(AN)$ enable use of the CS method, which requires an accurate bound on the extreme singular values to work efficiently.

Code snippet (Python):

```python
v = comm.allreduce(A.rmatvec(r)) - beta*v
x += alpha*v
r -= alpha*A.matvec(v)
```

Cost per iteration:

- two matrix-vector multiplications
- one cluster-wide synchronization
## LSQR vs. CS on an Amazon EC2 cluster

(Meng, Saunders, and Mahoney 2011)

<table>
<thead>
<tr>
<th>solver</th>
<th>$N_{\text{nodes}}$</th>
<th>$N_{\text{processes}}$</th>
<th>$m$</th>
<th>$n$</th>
<th>nnz</th>
<th>$N_{\text{iter}}$</th>
<th>$T_{\text{iter}}$</th>
<th>$T_{\text{total}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSRN w/ CS</td>
<td>2</td>
<td>4</td>
<td>1024</td>
<td>4e6</td>
<td>8.4e7</td>
<td>106</td>
<td>34.03</td>
<td>170.4</td>
</tr>
<tr>
<td>LSRN w/ LSQR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>84</td>
<td>41.14</td>
<td>178.6</td>
</tr>
<tr>
<td>LSRN w/ CS</td>
<td>5</td>
<td>10</td>
<td>1024</td>
<td>1e7</td>
<td>2.1e8</td>
<td>106</td>
<td>50.37</td>
<td>193.3</td>
</tr>
<tr>
<td>LSRN w/ LSQR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>84</td>
<td>68.72</td>
<td>211.6</td>
</tr>
<tr>
<td>LSRN w/ CS</td>
<td>10</td>
<td>20</td>
<td>1024</td>
<td>2e7</td>
<td>4.2e8</td>
<td>106</td>
<td>73.73</td>
<td>220.9</td>
</tr>
<tr>
<td>LSRN w/ LSQR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>84</td>
<td>102.3</td>
<td>249.0</td>
</tr>
<tr>
<td>LSRN w/ CS</td>
<td>20</td>
<td>40</td>
<td>1024</td>
<td>4e7</td>
<td>8.4e8</td>
<td>106</td>
<td>102.5</td>
<td>255.6</td>
</tr>
<tr>
<td>LSRN w/ LSQR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>84</td>
<td>137.2</td>
<td>290.2</td>
</tr>
</tbody>
</table>

**Table:** Test problems on an Amazon EC2 cluster and corresponding running times in seconds. Though the CS method takes more iterations, it actually runs faster than LSQR by making only one cluster-wide synchronization per iteration.
Outline

1. General thoughts
2. Randomized regression in RAM
3. Solving $\ell_2$ regression using MPI
4. Solving $\ell_1$ regression on MapReduce
“Everything generalizes” from $\ell_2$ regression to $\ell_1$ regression

(But “everything generalizes messily” since $\ell_1$ is “worse” than $\ell_2$.)

- A matrix $U \in \mathbb{R}^{m \times n}$ is $(\alpha, \beta, p = 1)$-conditioned if $|U|_1 \leq \alpha$ and $\|x\|_\infty \leq \beta \|Ux\|_1$, $\forall x$; and $\ell_1$-well-conditioned basis if $\alpha, \beta = \text{poly}(n)$.

- Define the $\ell_1$ leverage scores of an $m \times n$ matrix $A$, with $m > n$, as the $\ell_1$-norms-squared of the rows of an $\ell_1$-well-conditioned basis of $A$.

- Define the $\ell_1$-norm condition number of $A$, denoted by $\kappa_1(A)$, as:

$$\kappa_1(A) = \frac{\sigma_{1}^{\text{max}}(A)}{\sigma_{1}^{\text{min}}(A)} = \frac{\max_{\|x\|_2 = 1} \|Ax\|_1}{\min_{\|x\|_2 = 1} \|Ax\|_1}.$$  

This implies: $\sigma_{1}^{\text{min}}(A) \|x\|_2 \leq \|Ax\|_1 \leq \sigma_{1}^{\text{max}}(A) \|x\|_2$, $\forall x \in \mathbb{R}^n$.  

Meta-algorithm for $\ell_1$-norm regression

1. Using an $\ell_1$-well-conditioned basis for $A$, construct an importance sampling distribution $\{p_i\}_{i=1}^m$ from the $\ell_1$-leverage scores.
2. Randomly sample a small number of constraints according to $\{p_i\}_{i=1}^m$ to construct a subproblem.
3. Solve the $\ell_1$-regression problem on the subproblem.

A naïve version of this meta-algorithm gives a $1 + \epsilon$ relative-error approximation in roughly $O(mn^5/\epsilon^2)$ time (DDHKM 2009). (Ugh.)

But, as with $\ell_2$ regression:

- We can make this algorithm run much faster in RAM by
  - approximating the $\ell_1$-leverage scores quickly, or
  - performing an “$\ell_1$ projection” to uniformize them approximately.
- We can make this algorithm work at higher precision in RAM at large-scale by coupling with an iterative algorithm.
Conditioning: finding an $\ell_1$ well-conditioned basis

Recall, given an $n \times d$ matrix $A$ and $p \in [1, \infty]$, we want to find a low-distortion embedding $\Pi \in \mathbb{R}^{s \times n}$ s.t. $s = \mathcal{O}(\text{poly}(d))$ and

$$\frac{1}{\mathcal{O}(\text{poly}(d))} \cdot \|Ax\|_p \leq \|\Pi Ax\|_p \leq \mathcal{O}(\text{poly}(d)) \cdot \|Ax\|_p, \quad \forall x \in \mathbb{R}^d.$$ 

There are two main ways:

**Lemma (Conditioning via QR on low-distortion embedding)**

*Given a low-distortion embedding matrix $\Pi$ of $A_p$, let $R$ be the “$R$” matrix from the QR decomposition of $\Pi A$. Then, $AR^{-1}$ is $\ell_p$-well-conditioned.*

**Lemma (Conditioning via ellipsoidal rounding)**

*Given an $n \times d$ matrix $A$ and $p \in [1, \infty]$, it takes at most $\mathcal{O}(nd^3 \log n)$ time to find a matrix $R \in \mathbb{R}^{d \times d}$ such that $\kappa_p(AR^{-1}) \leq 2d$.***
Making $\ell_1$ regression work to low and high precision

Finding a good basis (to get a \textit{low-precision} solution):

<table>
<thead>
<tr>
<th>name</th>
<th>running time</th>
<th>$\kappa$</th>
<th>type</th>
<th>passes (for soln)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCT [SW11]</td>
<td>$O(mn^2 \log n)$</td>
<td>$O(n^{5/2} \log^{3/2} m)$</td>
<td>QR</td>
<td>2</td>
</tr>
<tr>
<td>FCT [CDMMMW13]</td>
<td>$O(mn \log n)$</td>
<td>$O(n^{7/2} \log^{5/2} m)$</td>
<td>QR</td>
<td>2</td>
</tr>
<tr>
<td>Ellipsoid rounding [Cla05]</td>
<td>$O(mn^5 \log m))$</td>
<td>$n^{3/2}(n + 1)^{1/2}$</td>
<td>ER</td>
<td>$n^4$</td>
</tr>
<tr>
<td>Fast ER [CDMMMW13]</td>
<td>$O(mn^3 \log m))$</td>
<td>$2n^2$</td>
<td>ER</td>
<td>$n^2$</td>
</tr>
<tr>
<td>SPC1 [MM13]</td>
<td>$O(\text{nnz}(A) \cdot \log m)$</td>
<td>$O(n^{13/2} \log n)$</td>
<td>QR</td>
<td>2</td>
</tr>
<tr>
<td>SPC2 [MM13]</td>
<td>$O(\text{nnz}(A) \cdot \log m) + \text{ER_small}$</td>
<td>$6n^2$</td>
<td>QR+ER</td>
<td>3</td>
</tr>
<tr>
<td>SPC3 [YMM13]</td>
<td>$O(\text{nnz}(A) \cdot \log m) + \text{QR_small}$</td>
<td>$O(n^{19/4} \log^{11/4} n)$</td>
<td>QR+QR</td>
<td>3</td>
</tr>
</tbody>
</table>

Iteratively solving (to get a \textit{medium- to high-precision} solution):

<table>
<thead>
<tr>
<th>method</th>
<th>passes</th>
<th>extra work per pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>subgradient (Clarkson 2005)</td>
<td>$O\left(n^4/\epsilon^2\right)$</td>
<td>—</td>
</tr>
<tr>
<td>gradient (Nesterov 2009)</td>
<td>$O\left(m^{1/2}/\epsilon\right)$</td>
<td>—</td>
</tr>
<tr>
<td>ellipsoid (Nemirovski and Yudin 1972)</td>
<td>$O\left(n^2 \log (\kappa_1/\epsilon)\right)$</td>
<td>—</td>
</tr>
<tr>
<td>inscribed ellipsoids (Tarasov, Khachiyan, and Erlikh 1988)</td>
<td>$O\left(n \log (\kappa_1/\epsilon)\right)$</td>
<td>$O\left(n^{7/2} \log n\right)$</td>
</tr>
</tbody>
</table>
Prior work and evaluations
Evaluate on real and simulated data:
- Simulated data, size ca. $10^9 \times 10^2$, designed to have “bad” nonuniformities.
- Real US Census data, size ca. $10^7 \times 10$, or “stacked” to size ca. $10^{10} \times 10$.

State of the art (due to Portnoy-Koenker, 1997):
- Standard solver for $\ell_1$ regression is interior-point method ipm, applicable for $10^6 \times 50$-sized problems.
- Best previous sampling algorithm for $\ell_1$ regression, prqfn, uses an interior-point method on a smaller randomly-constructed subproblem.
A MapReduce implementation

- **Inputs:** $A \in \mathbb{R}^{m \times n}$ and $\kappa_1$ such that
  \[
  \|x\|_2 \leq \|Ax\|_1 \leq \kappa_1 \|x\|_2, \quad \forall x,
  \]
  
  $c \in \mathbb{R}^n$, sample size $s$, and number of subsampled solutions $n_x$.

- **Mapper:**
  1. For each row $a_i$ of $A$, let $p_i = \min\{s \|a_i\|_1 / (\kappa_1 n^{1/2}), 1\}$.
  2. For $k = 1, \ldots, n_x$, emit $(k, a_i/p_i)$ with probability $p_i$.

- **Reducer:**
  1. Collect row vectors associated with key $k$ and assemble $A_k$.
  2. Compute $\hat{x}_k = \arg \min_{c^T x = 1} \|A_k x\|_1$ using interior-point methods.
  3. Return $\hat{x}_k$.

Note that multiple subsampled solutions can be computed in a single pass.
Evaluation on large-scale $\ell_1$ regression problem

First (solid) and the third (dashed) quartiles of entry-wise absolute errors (on synthetic data that has “bad” nonuniformities).

<table>
<thead>
<tr>
<th>Method</th>
<th>$\frac{|x - x^<em>|_1}{|x^</em>|_1}$</th>
<th>$\frac{|x - x^<em>|_2}{|x^</em>|_2}$</th>
<th>$\frac{|x - x^<em>|_\infty}{|x^</em>|_\infty}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CT (Cauchy)</td>
<td>[0.008, 0.0115]</td>
<td>[0.00895, 0.0146]</td>
<td>[0.0113, 0.0211]</td>
</tr>
<tr>
<td>GT (Gaussian)</td>
<td>[0.0126, 0.0168]</td>
<td>[0.0152, 0.0232]</td>
<td>[0.0184, 0.0366]</td>
</tr>
<tr>
<td>NOCD</td>
<td>[0.0823, 22.1]</td>
<td>[0.126, 70.8]</td>
<td>[0.193, 134]</td>
</tr>
<tr>
<td>UNIF</td>
<td>[0.0572, 0.0951]</td>
<td>[0.089, 0.166]</td>
<td>[0.129, 0.254]</td>
</tr>
</tbody>
</table>

First and the third quartiles of relative errors in 1-, 2-, and $\infty$-norms on a data set of size $10^{10} \times 15$. CT (and FCT) clearly performs the best. GT is worse but follows closely. NOCD and UNIF are much worse. (Similar results for size $10^{10} \times 100$ if SPC2 is used.)
The Method of Inscribed Ellipsoids (MIE)

MIE works similarly to the bisection method, but in a higher dimension.

Why do we choose MIE?

- Least number of iterations
- Initialization using all the subsampled solutions
- Multiple queries per iteration

At each iteration, we need to compute (1) a function value and (2) a gradient/subgradient.

- For each subsampled solution, we have a hemisphere that contains the optimal solution.
- We use all these solution hemispheres to construct initial search region.
Constructing the initial search region

Given any feasible $\hat{x}$, let $\hat{f} = \|A\hat{x}\|_1$ and $\hat{g} = A^T \text{sign}(A\hat{x})$. we have

$$\|x^* - \hat{x}\|_2 \leq \|A(x^* - \hat{x})\|_1 \leq \|Ax^*\|_1 + \|A\hat{x}\|_1 \leq 2\hat{f},$$

and, by convexity,

$$\|Ax^*\|_1 \geq \|A\hat{x}\|_1 + \hat{g}^T (x^* - \hat{x}),$$

which implies $\hat{g}^T x^* \leq \hat{g}^T \hat{x}$.

Hence, for each subsampled solution, we have a hemisphere that contains the optimal solution.

We use all these hemispheres to construct the initial search region $S_0$. 
Computing multiple $f$ and $g$ in a single pass

On MapReduce, the IO cost may dominate the computational cost, which requires algorithms that could do more computation in a single pass.

- **Single query:**

  $$f(x) = \|Ax\|_1, \quad g(x) = A^T \text{sign}(Ax).$$

- **Multiple queries:**

  $$F(X) = \text{sum}(|AX|, 0), \quad G(X) = A^T \text{sign}(AX).$$

An example on a 10-node Hadoop cluster:

- $A : 10^8 \times 50$, 118.7GB.
- A single query: 282 seconds.
- 100 queries in a single pass: 328 seconds.
MIE with sampling initialization and multiple queries

Comparing different MIE methods on large/LARGE $\ell_1$ regression problem.

(d) size: $10^6 \times 20$

(e) size: $5.24e9 \times 15$
Conclusion

Randomized regression in parallel & distributed environments: different design principles for high-precision versus low-precision

- Least Squares Approximation
- Least Absolute Deviations
- Extensions to Quantile Regression, Kernel-based Learning, Etc.

Algorithms require more computation than traditional matrix algorithms, but they have better communication profiles.

- On MPI: Chebyshev semi-iterative method vs. LSQR.
- On MapReduce: Method of inscribed ellipsoids with multiple queries.
- Look beyond FLOPS in parallel and distributed environments.