DATABRICKS

Sparse data support in MLlib

Xiangrui Meng

Spark MLlib

MLlib is an Apache Spark component focusing on machine learning:

- initial contribution from AMPLab, UC Berkeley
- shipped with Spark since version 0.8 (Sep 2013)
- 50 contributors

Algorithms

- **classification**: logistic regression, linear support vector machine (SVM), naive Bayes, classification tree

- **regression**: generalized linear models (GLMs), regression tree

- **collaborative filtering**: alternating least squares (ALS), non-negative matrix factorization (NMF)

- **clustering**: k-means

- **decomposition**: singular value decomposition (SVD), principal component analysis (PCA)

What's new in v1.0

- new user guide and code examples
- API stability
- sparse data support
- regression and classification tree
- distributed matrices
- tall-and-skinny PCA and SVD
- L-BFGS
- binary classification model evaluation

Sparse data support

“large-scale ~~sparse~~ problems”

Sparsity is almost everywhere

Sparse datasets appear almost everywhere in the world of big data, where the sparsity may come from many sources, e.g.,

- feature transformation:
  one-hot encoding, interaction, and bucketing,
- large feature space: n-grams,
- missing data: rating matrix,
- low-rank structure: images and signals.

One-hot encoding

Converts a categorical feature to numerical, e.g.,

- country: {Germany, Brazil, Argentina, …}
- Germany -> 0, Brazil -> 1, Argentina -> 2, …
- Germany -> [1, 0, 0, 0, …], Brazil -> [0, 1, 0, 0, …], Argentina -> [0, 0, 1, 0, …], …
- density: 1/#categories

Bucketing

Converts a numerical feature to categorical, e.g.,

- second of day: [0, 24*3600)
- hour of day: [0, 24)
- 4:33am -> [0, 0, 0, 0, 1, 0, …]
- density: 1/#buckets

Sparsity is almost everywhere

The Netflix Prize:

- number of users: 480,189
- number of movies: 17,770
- number of observed ratings: 100,480,507
- density = 1.17%

Sparsity is almost everywhere

rcv1.binary (test):

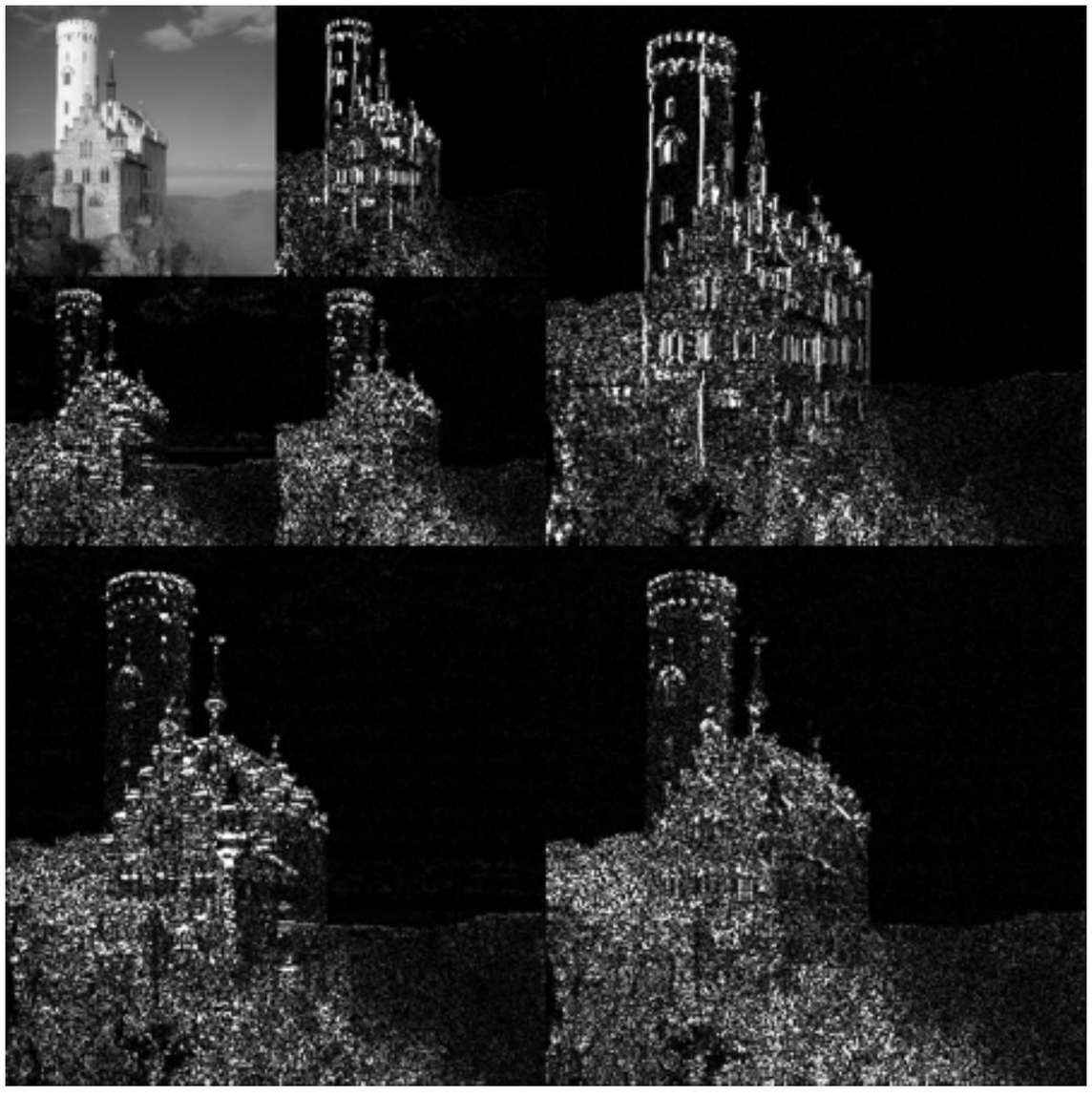
- number of examples: 677,399
- number of features: 47,236
- density: 0.15%
- storage: 270GB (dense) or 600MB (sparse)

Sparsity in a broader sense

real-world data = sparse — or — low-rank + noise

A dense image …

… is sparse under wavelet basis

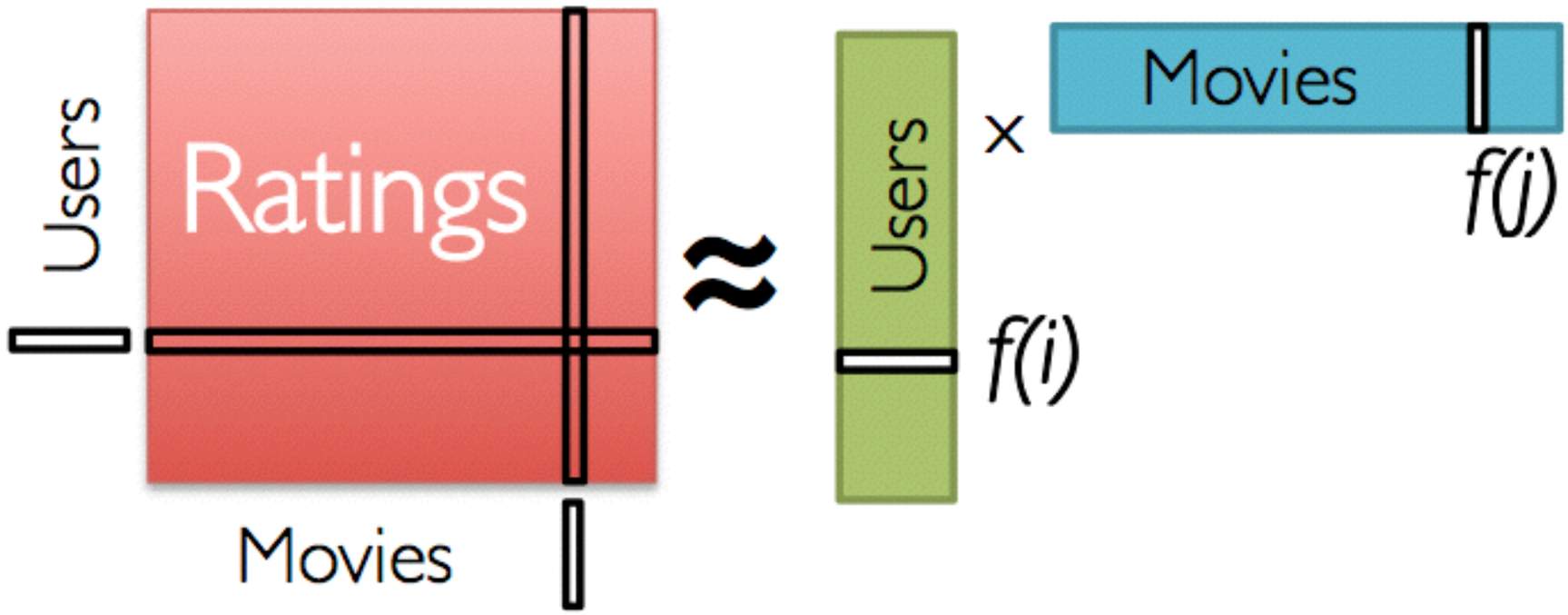


A huge rating matrix …

Amazon reviews:

- Number of users: 6,643,669
- Number of products: 2,441,053
- Number of reviews: 34,686,770

… is approximately low-rank



Exploiting sparsity

- As a user
  - recognize sparsity
- As a developer
  - utilize sparsity

Exploiting sparsity

- In Spark v1.0, MLlib adds support for sparse input data in Scala, Java, and Python.

- MLlib takes advantage of sparsity in both storage and computation in
  - collaborative filtering,
  - linear methods (linear SVM, logistic regression, etc),
  - naive Bayes,
  - k-means,
  - summary statistics,
  - singular value decomposition.

Open-source linear algebra packages

We benchmarked several JVM-based open-source linear algebra packages on the operations we need to implement sparse data support.

- breeze
- matrix-toolkits-java
- mahout-math
- commons-math3
- jblas

Sparse representation of features

$$\text{dense}: \underbrace{1.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{0.}\ \underbrace{3.}$$

$$\text{sparse}: \begin{cases} \text{size}: 7 \\ \text{indices}: \underbrace{0}\ \underbrace{6} \\ \text{values}: \underbrace{1.}\ \underbrace{3.} \end{cases}$$

Create/save/load sparse data

Create a sparse vector representing [1., 0., 3.]

- in Scala: `Vectors.sparse(3, Array(0, 2), Array(1., 3.))`
- in Java: `Vectors.sparse(3, new int[] {0, 2}, new double[] {1., 3.})`
- in Python: `Vectors.sparse(3, [0, 2], [1., 3.])`

Create a labeled point with a sparse feature vector

- `Scala/Java/Python: LabeledPoint(label, sparseVector)`

Create/save/load sparse data

Save a sparse training data set (`RDD[LabeledPoint]`)

- in LIBSVM format
  - `MLUtils.saveAsLibSVMFile(rdd, dir) -> 1 1:1.0 3:3.0`
- in MLlib's format (v1.1)
  - `rdd.saveAsTextFile(dir) -> (3,[0,2],[1.,3.])`

Create/save/load sparse data

Load a sparse training dataset

- in LIBSVM format
  - `MLUtils.loadLibSVMFile(sc, path)`
- in MLlib's format (v1.1)
  - `MLUtils.loadLabeledData(sc, path)`

$$\mathcal{O}(\mathrm{nnz})$$

Exploiting sparsity in k-means

Training set:

- number of examples: 12 million
- number of features: 500
- density: 10%

	dense	sparse
storage	47GB	7GB
time	240s	58s

Not only did we save 40GB of storage by switching to the sparse format, but we also received a 4x speedup.

Implementation of sparse k-means

Algorithm:

- For each point, find its closest center.

$$l_i = \arg\min_j \|x_i - c_j\|_2^2$$

- Update cluster centers.

$$c_j = \frac{\sum_{i, l_i = j} x_j}{\sum_{i, l_i = j} 1}$$

Implementation of sparse k-means

The points are usually sparse, but the centers are most likely to be dense. Computing the distance takes O(d) time. So the time complexity is O(n d k) per iteration. We don't take any advantage of sparsity on the running time. However, we have

$$\|x - c\|_2^2 = \|x\|_2^2 + \|c\|_2^2 - 2\langle x, c\rangle$$

Computing the inner product only needs non-zero elements. So we can cache the norms of the points and of the centers, and then only need the inner products to obtain the distances. This reduce the running time to O(nnz k + d k) per iteration.

Exploiting sparsity in linear methods

$$L(w; x, y) = f(x^T w; y)$$
$$g(w; x, y) = f'(x^T w; y) \cdot x$$

- The essential part of the computation in a gradient-based method is computing the sum of gradients.

- For linear methods, the gradient is sparse if the feature vector is sparse.

Exploiting sparsity in linear methods

Proposal:

```
g = points.map(p => grad(w, p)).reduce(_ + _)
```

Cons:

- Creating many small objects.
- Adding sparse vectors.

Exploiting sparsity in linear methods

MLlib's implementation

- does not add sparse vectors together,
- instead, adds sparse vectors directly to a dense vector for each partition and then computes the sum
  - fast random access
  - no temporary object creation

Exploiting sparsity in summary statistics

Multivariate summary statistics:

- count / mean / max / min / nnz / variance

MLlib's implementation

- computes the variance accurately in a single pass,
- ignores the zero values during the pass.

Singular value decomposition

- Top singular values can be computed via the eigenvalue decomposition of the Gram matrix.

$$\sigma_j(A) = \sqrt{\lambda_j(A^T A)}$$

- Lanczos algorithm computes eigenvalue decomposition, which only needs a routine that multiplies a matrix with a vector.

$$(A^T A)v = \sum_i (a_i^T v) a_i$$

Exploiting sparsity in SVD

$$(A^T A)v = \sum_i (a_i^T v) a_i$$

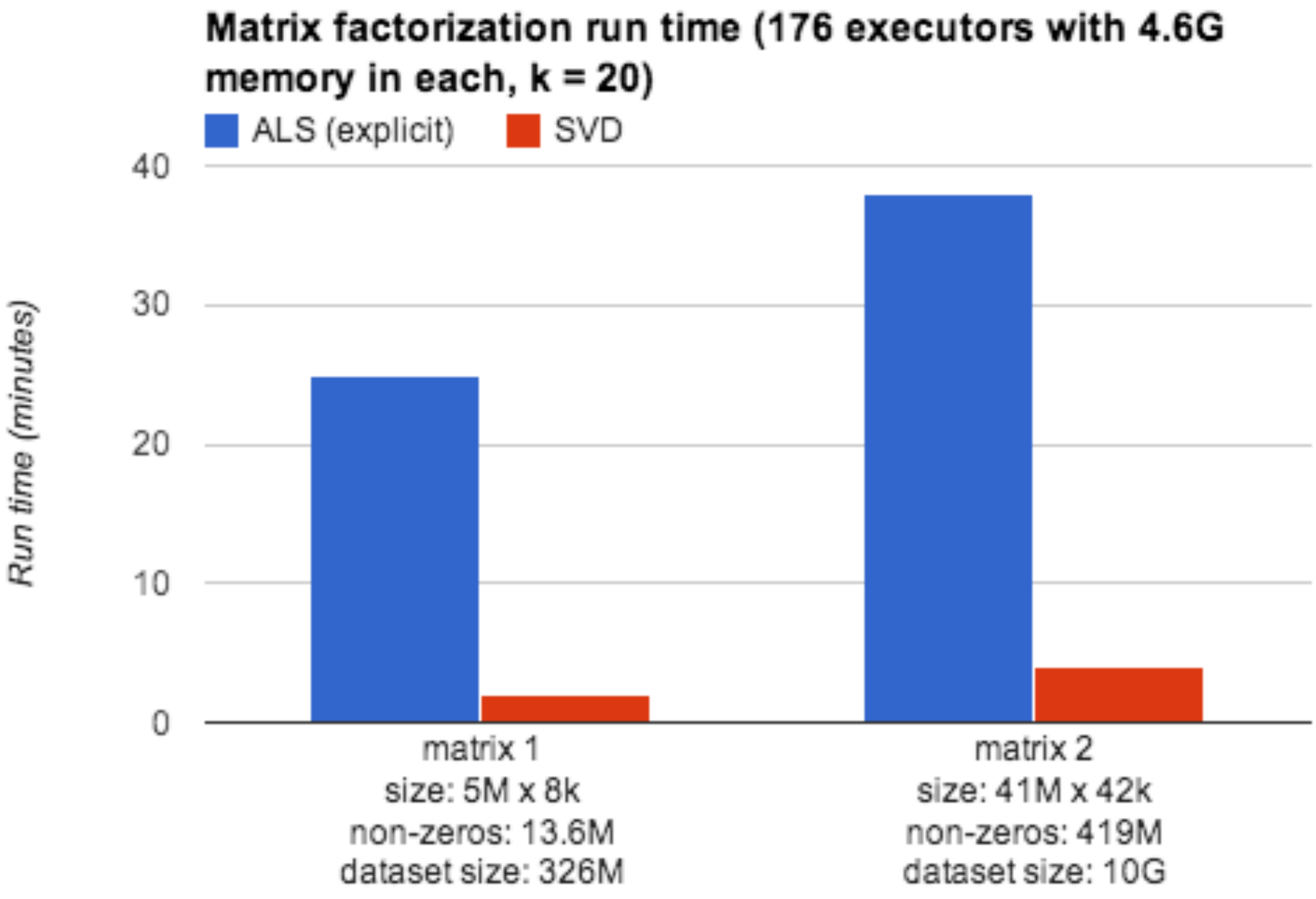
- v is dense in general, while {a_i} are sparse.
- The inner product can be computed quickly.
- Computing the sum is the same as in linear methods.

Make the right choice

Sparse vs. dense

- Storage
  - sparse format: 12 nnz + 4 bytes.
  - dense format: 8n bytes
- Speed
  - problem dependent

Pick algorithms that fit your data



ALS vs. SVD

Both algorithms compute low-rank matrix factorizations.

- ALS
  - is scalable on both directions
  - ignores unobserved entries (explicit feedback)
- SVD
  - is scalable on one direction
  - treats unobserved entries as zeros

Acknowledgement

- David Hall (breeze)
- Sam Halliday (netlib-java)
- Xusen Yin (summary statistics)
- Reza Zadeh (tall-and-skinny SVD & PCA)
- Li Pu (SVD via Lanczos)
- Tor Myklebust (NMF)

Summary

- Real-world data = sparse/low-rank + noise
- MLlib supports sparse data in
  - linear methods / naive Bayes / k-means / collaborative filtering / summary statistics / singular value decomposition
- As a user, recognize sparsity.
- As a developer, utilize sparsity.

DATABRICKS

Thank You!