Distributed CUR Decomposition for Bi-Clustering

Kevin Shaw, Stephen Kline
{keshaw, sakline}@stanford.edu

June 5, 2016
Stanford University, CME 323 Final Project

Abstract

We describe a distributed algorithm for calculating the CUR decomposition of a sparse matrix. Using p machines, with m rows and n columns, we analyze the theoretical run-time of both algorithms. Further, we compare the use of CUR decomposition for bi-clustering algorithms which originally rely on the SVD decomposition. We find that using a CUR decomposition speeds up the run-time of a bi-clustering algorithm by roughly a factor of 2x on sample data sets with a consensus score of roughly 0.7 when compared to using SVD. In certain settings, this may be an acceptable penalty for the speed increase.

1 Introduction

A matrix decomposition is a factorization of a matrix into a product of smaller matrices. There are many different matrix decompositions, and each has various applications. In our data analysis, the data matrix is a matrix with m rows and n columns, where m represents a sample of item-entities (e.g. people) and n represents another sample of item-entities (e.g. movies) related to each other with specific weights (e.g. ratings.) For various machine learning algorithms, matrix decomposition can be used on such a data matrix in order to cluster, or group, the samples by a measure of similarity.

2 Background of CUR

The CUR matrix decomposition is a low-rank approximation of a given matrix A, made up of three matrices: C, U and R. When multiplied together, these three matrices approximate the original matrix A. CUR decompositions are modeled after SVD decompositions and carry different benefits and costs when compared to the SVD decomposition.

Formally, for a CUR decomposition of matrix A, the C matrix is made up of actual columns of the original matrix. Likewise, the R matrix is made of up actual rows. This gives the added benefit that matrices C and R represent actual observations from the data, unlike the SVD decomposition.[1] Lastly, the U matrix is then constructed in such a way that the product of C, U and R closely approximates the original matrix A.

Beyond interpret-ability, another benefit of the CUR decomposition is the storage space. Since C and R represent actual columns and rows of the original matrix A, these matrices will also be sparse (given that A is sparse). Further, though the U matrix is dense, it is quite small, and thus does not require large storage space.[2] This provides an additional benefit over SVD decompositions, as the U and V matrices are often large and very dense, while Σ is small and diagonal. Figure 1 and Figure 2 provide visual representations for both the SVD and CUR decomposition, respectively.

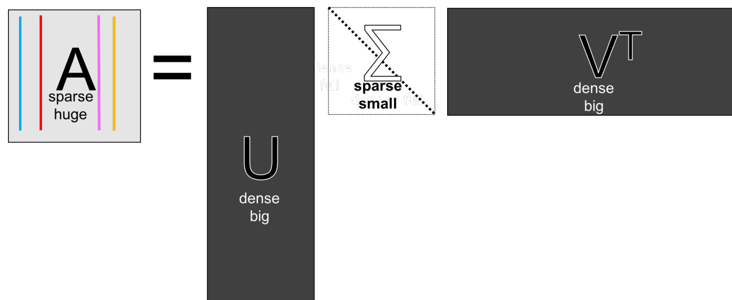

Figure 1: Example of SVD Matrix Decomposition

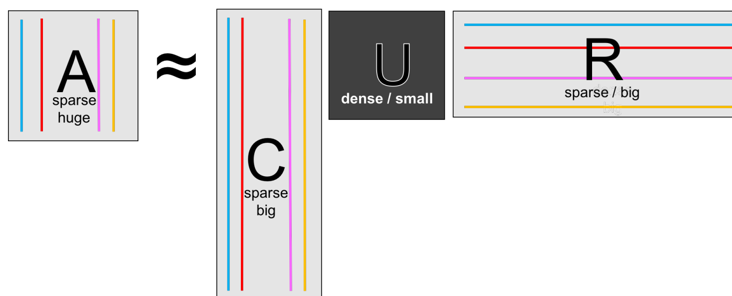

Figure 2: Example of CUR Matrix Decomposition

[1] There are other CUR decompositions that use right and left singular values for the C and R matrix. We will not be analyzing those in this paper.

[2] The U matrix is r-by-c, where r and c are often a small numbers in relation to the dimension of the original matrix.

3 CUR Algorithms

Over the past few years, there have been various serial algorithms proposed for CUR matrix decompositions. For our analysis, we chose to focus on one that provides a fast run-time while still providing a good approximation to the SVD decomposition. For instance, some CUR algorithms provide better accuracy by constructing the C and R matrices in a different procedure, though pay for this in terms of run-time. The algorithm we describe and analyze below was proposed in a paper by Petros Drineas, Ravi Kannan and Michael W. Mahoney in 2006 [1]. This paper presents both a linear-time CUR algorithm (scales linearly) and a constant-time CUR algorithm. We chose the linear-time CUR algorithm for our analysis and implementation. It makes a better approximation but requires greater computation.

3.1 Serial CUR

3.1.1 Algorithm/Implementation of CUR

Below is the linear-time algorithm described by Drineas, Kannan and Mahoney from their 2006 paper [1].

Data: A $\in \mathbb{R}^{m\times n}$, r, c, $k \in \mathbb{Z}^+$ such that $1 \leq r \leq m$, $1 \leq c \leq n$, and $1 \leq k \leq \min(r, c)$, $\{p_i\}_i^m$ such that $p_i \geq 0$ and $\Sigma_{i=1}^m \ p_i = 1$, and $\{q_j\}_j^n$ such that $q_j \geq 0$ and $\Sigma_{j=1}^n \ q_i = 1$.

Result: C, U and R matrices

for t = *1 to* c **do**

  Pick $j_t \in$ {1,...,n} with $\mathbf{Pr}[j_t = \alpha] = q_\alpha$, α = 1,...,n

  Set $C^{(t)} = A^{(j_t)} \ / \ \sqrt{cq_{j_t}}$

end

for t = *1 to* r **do**

  Pick $i_t \in$ {1,...,n} with $\mathbf{Pr}[i_t = \alpha] = p_\alpha$, α = 1,...,n

  Set $R_{(t)} = A_{(i_t)} \ / \ \sqrt{rp_{ji_t}}$

  Set $Y_{(t)} = C_{(i_t)} \ / \ \sqrt{rp_{ji_t}}$

end

Compute C^TC and its SVD. Say $C^TC = \Sigma_{t=1}^c \ \sigma_t^2 \ (C) y^t {y^t}^T$

$X = \Sigma_{t=1}^k \ \frac{1}{\sigma_t^2(C)} y^t {y^t}^T$

$U = XY^T$

Algorithm 1: Serial CUR Algorithm

3.1.2 Analysis of Run Time

For the analysis of this algorithm, we choose to focus on the construction of the C and R matrices.[3] Assume we have a sparse matrix A which has m rows, n columns and b non-zero entries. We will consider A to be sparse. Further,

[3] This part of the algorithm provides opportunity for distribution amongst a cluster of machines.

assume r, c, and k are constants as described above in Algorithm 1.

Though the algorithm above assumes the user is already provided the probability measures for both the columns and rows of A, this may not be the case. Thus, it is important to also analyze the run-time to develop these figures. The probabilities are based off the $L2$ norm of each column and row. The $L2$ norm is calculated as:

$$|x| = \sqrt{\Sigma_{k=1}^{n} x_k^2}$$

For each column c_j, we calculate its $L2$ norm. A similar procedure is done for each row r_i. Next, we must loop through each c_j again to divide its $L2$ norm by the Frobenius norm, calculated as:

$$\|A\|_F^2 = \Sigma_{i=1}^{m} \Sigma_{j=1}^{n} A_{ij}^2$$

This provides our probability measure for each column and row of the matrix. This procedures takes $O(mn)$, as calculating the Frobenius, as well as the $L2$ norm of columns and $L2$ norm of the rows, each take $O(mn)$.

Given the probability measures for both the columns and rows of A, we must sample c columns and r rows. To do this, we convert our probability measures for each column into a cumulative sum. We first draw c random numbers between 0 and 1, and, using a binary-search, determine the corresponding column of A to which the random number is associated with. Lastly, we must scale each number of the selected column. This process takes $O(n + c \log n + cn)$, or more simply, $O(cn)$, due to the scaling procedure. A similar analysis shows that the run-time for sampling r rows from A takes $O(rm)$.

As discussed above, the remainder of the algorithm does not provide significant opportunity to distribute. Thus, we have chosen to not analyze the run-time.

3.2 Distributed CUR

3.2.1 Algorithm/Implementation of CUR

From our analysis above for the Serial CUR algorithm, it is clear there are areas of the algorithm which we can improve from a run-time perspective using distributed computing. The following is our proposed distributed algorithm for CUR.[4]

[4] Our algorithm does not assume probability measures have been provided to the user.

Data: A $\in \mathbb{R}^{m \times n}$, provided in coordinate form (i.e. (x, y, v)) with b non-zero entries, r, c, $k \in \mathbb{Z}^+$ such that $1 \leq r \leq m$, $1 \leq c \leq n$, and $1 \leq k \leq \min(r, c)$, and p machines. Note, a d subscript (i.e. x_d) implies x is an RDD.
Result: C, U and R matrices
R_d = partition of A by rows, with key-value pairs = $(x, (y, v))$
C_d = partition of A by columns, with key-value pairs = $(y, (x, v))$
Compute RN_d as reduce among keys of R_d
Compute CN_d as reduce among keys of C_d
Compute T as reduce of RN_d
Compute RP_d probability measures of RN_d with T as scaling factor
Compute CP_d probability measures of CN_d with T as scaling factor
Sample r rows locally from RP_d and collect data
Sample c columns locally from CP_d and collect data
Follow Serial CUR Algorithm to compute U

Algorithm 2: Proposed Distributed CUR Algorithm

3.2.2 Analysis of Run Time

Similar to Serial CUR, for the analysis of this algorithm, we choose to focus on the construction of the C and R matrices. We again assume we have a sparse matrix A which has m rows, n columns and b non-zero entries. We will consider A to be sparse. Further, assume r, c, and k are constants as described above in Algorithm 2.

First, the algorithm calls to duplicate the A matrix coordinate data, and partition one by columns and the other by rows.[5] This partition will require two all-to-all communications, each with a potential cost of $O(mn)$. However, as noted in the assumptions, there are b non-zero entries, so we can be more precise and note the cost at $O(b)$. By paying a potentially high up-front cost to move the data, we will save ourselves multiple all-to-all communications in subsequent steps.

Next, we compute the column and row norms. This is similar to our $L2$ calculation in the Serial CUR algorithm. However, since we have two partitioned data sets, one by rows and the other by columns, this becomes an embarrassingly parallel calculation. There is no required communication, and the run-time is calculated as the last reducer to finish. In this case, this is determined by the row or column with most non-zero entries. Given this is a sparse matrix, we assume this cost will be small.

After calculating each column and row $L2$ norm, we must calculate the Frobenius norm of A. While calculating the $L2$ norm of each column and row, we also calculate the sum of each entry squared. Thus, for a given key r_i, our RN_d

[5] This is not an uncommon practice, and is implemented in other distributed algorithms.

contains:

$$\left(r_i, \left(\sqrt{\Sigma_{j=1}^{n} A_{ij}^2}, \Sigma_{j=1}^{n} A_{ij}^2\right)\right)$$

Therefore, by reducing the RN_d, we can also calculate the Frobenius norm. This reduce will require an all-to-one communication. Here, we take advantage of the tree pattern, which says at each step i, $\frac{p}{2^i}$ machines communicate with $\frac{p}{2^i}$ other machines. Since we are only sending a constant amount of numbers (in this case, just one number) between each machine, the computation cost is $O(p)$ and the communication time is $O(\log p)$.

To calculate each column and row's probability measure, we must collect each column and row norm, and divide it by the Frobenius norm, or T. This requires an all-to-1 communication. Again, we will take advantage of the tree pattern. Once we have all of the norms located on the driver, the simple division is assumed to be $O(1)$.

With the probability measures calculated, we can now sample the c columns and r rows to create C and R matrices. Similarly to the Serial CUR algorithm, we calculate the cumulative sum, and draw c random numbers between 0 and 1. Then, using a binary-search, we determine the corresponding column of A associated with each random number. This process takes $O(n + c \log n)$. Since c is assumed to be a small number, we report this as $O(n)$. A similar analysis shows the run-time for sampling r rows is $O(m)$.

Finally, we must create our C and R matrices. We broadcast the selected columns, or c, and selected rows, r, to each machine. Each column or row determines if it has been selected, and if so, is communicated back to the driver. The initial broadcast is an all-to-one communication, with computation cost of $O((c + r)p)$ and communication time of $O((c + r) \log p)$. To collect the selected rows and columns, we need to grab data from at most $r + c$ machines. We can still implement a tree pattern of communication, but it will involve less machines. This will resemble an all-to-one communication, with a computation cost of $O(max(m, n)(r+c))$ and communication time of $O(max(m, n) \log(r+c))$.

Lastly, with C and R now stored locally, we follow the Serial CUR Algorithm to compute U.

3.2.3 Benefits of Duplicating and Partitioning Data

The up-front cost paid for duplicating and partitioning the data is easily outweighed by the savings farther along the algorithm. Without duplicating and partitioning, we would need to incur an additional all-to-all communications with potentially large shuffle sizes. First, computing the CN_d and RN_d would both potentially require an all-to-all communication. There is no guarantee that

rows or columns would be on the same machine, thus calculating any norms would require moving them to be local first. Second, by duplicating the data, we can have one partitioned by rows and another partitioned by columns. This allows us to easily process calculations for each in a efficient manner without restructuring the data each time. Further, since we assume A is sparse, the increase in storage size is assumed to be small, and thus not a huge cost.

4 Applications of CUR: Bi-clustering

4.1 Introduction to Bi-Clustering

Bi-clustering was initially developed from an applications perspective for identifying structure in DNA microarray data. In this context, the search is for groups of genes ("marker genes") that identify certain sets of conditions (e.g. tumor types). The concept of bi-clustering is to simultaneously cluster the genes and tumor samples so as to let the data determine which genes expressed most prominently with which tumors and there by group both genes and tumors simultaneously. In the context of movie ratings, this might be seen as looking for movie groupings (e.g. "playlists") that target a particular "taste profile" (group of users.) The benefit of this approach (e.g. vs. collaborative filtering) may be a greater contextual awareness for the user. A playlist might be suggested to a group of users along with an explanation of their machine-generated "taste profile".

4.2 Previous Algorithms Using SVD

There are a number of approaches to achieve the biclustered result. Our approach in this paper focuses on spectral biclustering in particular - namely, the use of the SVD as a core component. Specifically we followed the approach as presented in Kluger, et al. (2003) [3]. The main steps are:

1. Normalize the data
2. Compute the SVD
3. Select best vectors and k-means cluster

1) Normalize the data

Given the sparsity of our data set (MovieLens data), we chose a bistochastic approach which normalizes the rows and columns of the matrix simultaneously in an iterative manner so all rows sum to one constant and all columns sum to a different constant.

2) Compute the SVD

The algorithm only requires the finding of singular vectors (not values) and uses a randomized approach or ARPACK. Given the randomization already part of the CUR algorithm, we chose ARPACK which uses the underlying sklearn implementation at sklearn.utils.arpack.svds. The underlying implementation uses the Lanczos algorithm (an adaptation of power methods) to find the most useful vectors.

3) Select best vectors and k-means cluster

Select a subset of singular vectors and pick the best subset x out of y total singular vectors – i.e. that are best approximated according to Euclidean distance by piecewise constant vectors representing the clusters. The piecewise constant vectors are found using k-means clustering.* Finally the columns are rows are projected onto these singular vectors and the result is clustered.

* For the k-means clustering there are options to use k-means++ (default), randomly initialize or initialize with a specific array. We stuck with the k-mean++ default. The number of k-means initializations can be specified as well. For our experiments thus far, we left the default at 10. Finally, there is also a mini-batch k-means option which attempts several initializations to determine the best and the run the full algorithm only once.

4.3 Possible benefits of CUR

A CUR version of the algorithm above focuses on step 2 from Section 4.2 above, the calculation of the SVD which is the most time consuming step (based on our observation of repeated runs via DataBricks.) As described in Section 3, instead of singular vectors (in rotated space), CUR uses actual rows and columns attempting to approximate these singular vectors. These rows and columns are selected via a random algorithm that can run very quickly. We can replace the CUR for SVD within the biclustering algorithm easily because we only need the singular vectors (not the singular values). As discussed in Section 3, we have come up with a distributed approach enabling all the benefits of the CUR algorithm in a highly scalable form - i.e. when the A matrix (e.g. ratings matrix) does not fit in memory on a single machine.

4.4 Experiment and Results

In our implementation, we started with an existing implementation of the algorithm above in sklearn.cluster.bicluster.[6] Then, we modified this algorithm to use CUR by breaking existing code into sections and parameterizing as separate function calls (i.e. inserting "hooks") according to the 3 primary algorithm steps above enabling us to effectively replace SVD with CUR. For the baseline

[6]Specifically, we implemented "SpectralBiclustering", which is available at: http://scikit-learn.org/stable/modules/biclustering.html.

serial CUR algorithm we started with an implementation from the Python Matrix Factorization (PyMF)[7] which uses Drineas, et al. (2006). [1] In particular, we implemented the linearly scaling CUR. (There is also a constant-time CUR which further sacrifices accuracy for scalability.)

For our experiment, we used the MovieLens data. We ran the test on three different sizes: 100K, 500K and 1M.[8] We first ran the out of box algorithm in a serial fashion as a baseline. This was important for both SVD and CUR versions in order to match the results of our distributed algorithm to be confident we were producing the same result - e.g. counts per cluster in for SVD (exact) and CUR (approximate). Although these data sets were known to be sparse (Table 1 below), we also found that they skipped IDs - i.e. the IDs for users and movies was not consecutive and affected the matrix build. We had to implement a data preparation stage which effectively re-indexed the data so as to avoid any empty rows or columns.

Dataset	Users	Movies	Observations	Sparsity Measure
100K	943	1682	100000	6.30%
500K	3048	3645	502849	4.53%
1M	6040	3706	1000209	4.47%

Table 1: Dimensions of datasets (MovieLens)

All of our code was written in Python, tested locally, and then implemented in Spark via PySpark where serial was first tested then sections re-implemented in a distributed fashion.

For each experimental test, we ran with the following fixed parameters:

- Data cluster count: 3 x 3 (users x movies clusters)
- SVD components: 6, best: 3 (defaults)
- CUR components: 24, best: 12
- Machine cluster count: 8 (AWS Memory-optimized) - Spark 8 Node, 270 GB On-demand, Spark 1.6.1 (Hadoop 1), 23.2 GB per machine (1 driver, 8 masters)[9]
- SVD runs before CUR, using shared memory

[7] The code is available at: https://pypi.python.org/pypi/PyMF.

[8] The MovieLens data provides data sets for 100k and 1M. We also created a 500K row data set by sub-sampling users from the 1M row set. The data is available at: http://grouplens.org/datasets/movielens/.

[9] For an 8 node cluster established via Databricks, we observed that it spun up 5 instances in AWS (Amazon Web Services) and so some nodes were sharing the same machine.

- Spark cluster restarted between runs to clear memory, as well as Databricks notebook detached/reattached (to clear any local memory)

Run	Dataset	SVD Time	CUR Time	Speed Up	Consensus Score
1	100K	12.444	3.848	3.234	0.716
2	100K	12.537	3.882	3.229	0.740
3	100K	12.653	3.880	3.261	0.710
4	500K	25.916	7.204	3.597	0.674
5	500K	29.925	6.736	4.442	0.793
6	500K	30.781	6.599	4.664	0.657
7	1M	30.670	10.491	2.923	0.733
8	1M	33.532	9.946	3.371	0.750
9	1M	32.754	17.691	1.851	0.698

Table 2: Experimental results on increasing dataset size (MovieLens)

The SVD and CUR times in Table 2 above are in seconds. The Consensus Score above was calculated via sklearn.metrics.consensus_score using the defaults which employ a Jaccard similarity coefficient. The sklearn function takes 4 vectors identifying the rows (users) and columns (movie) cluster memberships i.e. both the SVD-based cluster result and the CUR-based cluster result. The similarity is then calculated as size of the intersection of these row/col sets divided by the size of the union of the row/column sets for each cluster. Consider the two sets as a bipartite graph, the best matching between row/column sets is found using the "Hungarian algorithm", a combinatorial optimization algorithm.[2] The final score is then the sum the similarities divided by the size of the larger set.

Figure 3 and Figure 4 below provide visual representations of the tabular data above. These were created in R with ggplot and jitter to ensure that points do not obscure each other.

5 Conclusions

As can be seen in the results above we saw a decreased runtime in a distributed biclustering context. We gain the benefits of distribution (e.g. working with data that is not possible to fit on a single machine) and still keep the scalability and speed benefits of CUR. As discussed, CUR approximates the SVD with an actual data sample. In the course of our experimentation, we found this was indeed the case, especially very "tall and skinny" matrices (e.g. by limiting the total number of rated movies). However, the "curse of dimensionality" plays a role as the dimension of each observation (e.g. total number of movies rated) becomes high, making the comparison to SVD more difficult because the the singular vector is harder to approximate.

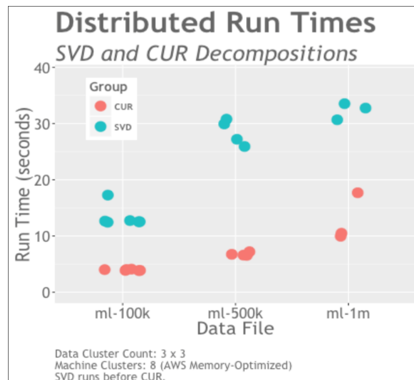

Figure 3: Distributed Run Times

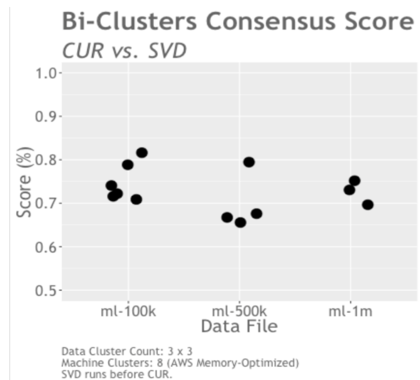

Figure 4: Biclusters Consensus Score

6 Further Research Opportunities

We believe there are many opportunities for further research to both improve the run-time of our proposed Distributed CUR algorithm, and continue to test CUR as an alternative to SVD for bi-clustering.

Though we believe our proposed algorithm provides a nice outline to improve the run-time of calculating the CUR decomposition, there are possible bottlenecks. For instance, to compute the row and column norms, are algorithm performs a simple serial sum, which is linear in the number of non-zero items. However, given a very dense row or column (for simplicity, call this r_l), this become a bottleneck to the algorithm.

One potential solution is to distribute r_l among more than one machine. For instance, for a given matrix A, if row r_l is very dense, while every other row r_i only has 1 entry, then partitioning by rows may not be effective. Thus, we would like to partition by rows while requiring each partition to contain a roughly equal number of observations. This may require some prior knowledge about the data beforehand to determine which rows to partition across multiple machines, and which to keep intact.

We would also like to continue to test our distributed algorithm on larger data sets. Though we feel our results reflect the true speed increase of distributed CUR vs SVD, in practical use today, data sets are much larger in size. MovieLens data offers sample data of larger sizes, including 10M, 20M and 22M. These would provide us with a nice range of data sets to use.

References

[1] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. "Fast Monte Carlo Algorithms For Matrices III: Computing a Compressed Approximate Matrix Decomposition". In: *Society for Industrial and Applied Mathematics* 36.1 (2006), pp. 184–206. DOI: 10.1137/S0097539704442702.

[2] Sepp Hochreiter et al. "FABIA: factor analysis for bicluster acquisition." In: *Bioinformatics* 26.12 (2010), pp. 1520–1527. DOI: 10.1093/bioinformatics/btq227.

[3] Yuval Kluger et al. "Spectral Biclustering of Microarray Cancer Data: Coclustering Genes and Conditions". In: *Genome Research* 13 (2003), pp. 703–716. DOI: 10.1.1.135.1608.