Contents

Introduction

This is a report of the data science capstone project. The project implements the SVT algorithm introduced by [2] in Python. SVT is an algorithm which completes the unknown entries of a matrix, by assuming the actual matrix is low rank. This problem shows up whenever our data is assumed to be linearly dependent on a small number of 'hidden' factors.

This report first analyzes the SVT algorithm in the context of convex optimization algorithms. Then an explnation of why the SVT is efficient will be given, while evaluating the SVT algorithm on artificially generated data. Afterwards, we compare SVT with another method, which is gradient descent on the latent factors of the data by [4]. Then, we apply SVT on the MovieLens dataset, and evaluate whether SVT is really a good method on predicting users' movie preferences. Finally, we highlight some technical aspects in implementation of SVT and document the Python codes in this project.

Notation and Algorithm

We mainly follow the notation in [2]. M will refer to a matrix, $\|M\|_*, \|M\|_F$ would refer to the nuclear norm and Frobenius norm of matrices, and the set $\Omega \subset \{(i,j) : 1 \leq i,j \leq n\}$ is the sampled entries of the matrix (we have to predict the entries in the complement of Ω). $P_\Omega : \mathbb{R}^{n\times n} \to \mathbb{R}^{n\times n}$ means the projection operator by setting the entries outside of Ω to zero.

The thresholding (or shrinkage) operator is denoted $\mathcal{D}_\tau$, where $\mathcal{D}_\tau(M)$ obtains the SVD of $M = U\Sigma V$ and decreases the entries in the diagonal of Σ by τ, setting to 0 if it is negative, and then reconstructs the matrix. This means

$$\mathcal{D}_\tau(M) = U \max\{(\Sigma - \tau I), 0\} V$$

The SVT algorithm in [2] takes the form

$$X^{k+1} = \mathcal{D}_\tau(Y^k)$$
$$Y^{k+1} = Y^k + \delta_k P_\Omega(M - X^k)$$

with stopping criterion the relative error $\|P_\Omega(X^k - M)\|_F / \|P_\Omega(M)\|_F \leq \epsilon$ for tolerance $\epsilon > 0$.

In the Python code, Ω is usually represented by the variable 'locations', where the matrices would be represented by various variables. The thresholding operator $\mathcal{D}_\tau$ is given by the function shrinkage_operator in svt.py or svtcpu.py.

SVT in context of Convex Optimization

A brief explanation of the SVT algorithm would be given here with respect to the summary of [1] of common convex optimization algorithms. Such an explanation is also given in the SVT paper [2]. As it was hinted to us in the presentations that the SVT algorithm is either an ADMM algorithm or some other convex optimization algorithm, this section simply servers to show some of my understanding. There is nothing novel in this section.

Convex optimization algorithms

We will not talk about whether or on what conditions the stated algorithms would converge. Such discussions appear in [1]. The algorithms are briefly stated as a context for the SVT algorithm. All algorithms below appear in [1].

The setup for our problem is to minimize $f(x)$ subject to $Ax = b$, where f is convex. f has Lagrangian $L(x, y) = f(x) + y^T(Ax - b)$ with dual function $g(y) = \inf_x L(x, y)$.

Gradient descent This is probably the simplest algorithm, the update is given by $x_{n+1} = x_n - \delta_k \nabla f(x_n)$.

Dual ascent Instead of minimizing f directly, the dual function is maximized. According to [1],

$$\begin{aligned} x^{k+1} &= \text{argmin}_x L(x, y^k) \\ y^{k+1} &= y^k + \alpha^k(Ax^{k+1} - b) \end{aligned}$$

This means after updating x to be the value that attains $g(y)$, we update y by the gradient of g.

Dual decomposition It was mentioned in [1] that if $f(x) = \sum_i f_i(x_i)$ where $x_i \in \mathbb{R}^{n_i}, x \in \mathbb{R}^n$ with $\sum_i n_i = n$, then the minimization of L can be done in parallel. This is simply by observing the Lagrangian L can also be decomposed into a sum of functions $L_i(x_i, y)$ (as $y^T(Ax - b)$ is linear in x), and clearly for any fixed y the L is minimized whenever L_i are minimized separately. By [1] the algorithm is

$$\begin{aligned} x_i^{k+1} &= \text{argmin}_{x_i} L_i(x_i, y^k) && \text{(For } i = 1, \cdots, n\text{)} \\ y^{k+1} &= y^k + \alpha^k(Ax^{k+1} - b) \end{aligned}$$

Method of multipliers According to [1], introducing a penalty term would increase the robustness of the dual ascent algorithm, making it converge under less strict assumptions. For some penalty parameter $\rho > 0$, we minimize $f_\rho(x) = f(x) + (\rho/2)\|Ax - b\|_2^2$, subject to the same conditions. And then the dual ascent algorithm would be applied to $f_\rho(x)$.

Notice that this changes the Lagrangian, but however, does not change the minimial points, since $f_\rho(x) = f(x)$ on all x in our domain (because of the conditions $\{x \in \mathbb{R}^n : Ax = b\}$).

ADMM The book [1] has mentioned that the goal of ADMM is to benefit from both the effectiveness due to parallelization of Dual Decomposition, and the better convergence of Method of Multipliers. However, even if f is decomposable, it would be hard to decompose $\|Ax - b\|_2^2$ in the method of multipliers.

Following the notation in [1], we consider a decomposable problem, minimizing $F(x, z) = f(x) + g(z)$ subject to $Ax + Bz = c$ (in original notation, F, f, g

corresponds to f, f_1, f_2 and x, z corresponds to x_1, x_2). Then the augmented function

$$F_\rho(x,z) = f(x) + g(z) + \frac{\rho}{2}\|Ax + Bz - c\|_2^2$$

and the augmented Lagrangian is

$$L_\rho(x,z,y) = f(x) + g(z) + \frac{\rho}{2}\|Ax + Bz - c\|_2^2 + y^T(Ax + Bz - c)$$

Then [1] states that the ADMM algorithm does the following iteratively:

$$\begin{aligned} x^{k+1} &= \text{argmin}_x L_\rho(x, z^k, y^k) \\ z^{k+1} &= \text{argmin}_z L_\rho(x^{k+1}, z, y^k) \\ y^{k+1} &= y^k + \rho(Ax^{k+1} + Bz^{k+1} - c) \end{aligned}$$

SVT as one of the algorithms

In the SVT paper [2] Section 2.4, the authors pointed out it is a Lagrange multiplier method, and that minimizing the Lagrangian $L(X,Y) = f_\tau(X) + \langle Y, P_\Omega(M - X)\rangle$ with fixed Y is equivalent to setting $X = \mathcal{D}_\tau(P_\Omega(Y))$. Viewing the SVT algorithm with the shrinkage operator $D_\tau(Y^k)$ replaced with the minimizer of the Lagrangian, we see that this is exactly a dual ascent method applied to the problem of minimizing $f_\tau(X) = \tau\|X\|_* + \frac{1}{2}\|X\|_F^2$ subject to $P_\Omega(X) = P_\Omega(M)$.

It seems SVT algorithm is nothing more than the dual ascent method, as the penalty $\frac{1}{2}\|X\|_F^2$ is not related to the linear constraint $P_\Omega(X) = P_\Omega(M)$ (so it is not Method of Multipliers), and there is no dual decomposition of the target function. It can be said to be related to the Method of Multipliers, in that it choose a penalty term (although not exactly like the Method of Multipliers). To conclude, the SVT algorithm is a dual ascent method applied to $f_\tau(X)$, which aims to minimize the nuclear norm plus a penalty term.

SVT on artifically generated data

The notebooks are matrix_completion.ipynb, matrix_completion_cpu.ipynb. In this section, the data are generated by a product $M = XY$, where X, Y are random $n \times r, r \times n$ matrices with i.i.d entries. So M is $n \times n$ and with rank r in almost all cases. In Python implementation, this is implemented in utils.py generate_matrix_rank.

The entries in X, Y are either sampled from a standard normal distribution, or a uniform distribution in $[0,1]$. There could be a scaling factor for the randomly generated matrix, i.e $M = KXY$ for some scaling $K > 0$.

Using the generate_locations method in utils.py, we randomly select $\Omega \subset \{(i,j) : 1 \leq i,j \leq n\}$ with fixed size $|\Omega|$, and obtain a sampled matrix $S = P_\Omega(M)$ (where entries in Ω equal to M, zero otherwise). Using $P_\Omega(M)$, we use the SVT algorithm to try recover M. We then compare SVT($P_\Omega(M)$) against M to see whether the recovery is good.

Metrics for evaluation

We evaluate the accuracy of prediction of SVT using the following metrics. Let M be the actual matrix, and $\tilde{M} = \text{SVT}(P_\Omega(M))$ be the predicted matrix.

$$\text{max\_diff} = \max_{i,j} \left[P_\Omega(M - \tilde{M})\right]_{ij}$$

$$\text{rel\_error} = \frac{\|M - \tilde{M}\|_F}{\|M\|_F}$$

This means max diff is the maximum difference among the sampled entries Ω, and the relative error is the Frobenius norm of the error relative for the Frobenius norm of M.

Performance of SVT

The stopping criterion (tolerance) is $\epsilon = 0.001$. Here are the metrics for SVT implementation with GPU for matrix multiplication:

Size (n):	1000	1000	1000	1000	1000	5000	5000
rank (r):	10	50	100	10	50	10	10
distribution:	Normal	Normal	Normal	Uniform	Uniform	Normal	Uniform
$\|\Omega\|$:	120000	390000	570000	120000	390000	600000	600000
time elapsed (s):	10.8	44.8	101.8	40.8	602.0	171	667
max_diff:	0.0446	0.0534	0.0586	0.02294	0.076	0.0582	0.0276
rel_error:	0.0016	0.00159	0.00165	0.00151	0.0015	0.00166	0.00153
rank of prediction:	10	50	100	10	50	10	10
trials:	5	5	5	5	5	5	5

The first 4 rows are the parameters of generating the random matrix, and the $|\Omega|$ the size of sampled entries. The configurations are borrowed from the SVT paper [2]. The next 4 rows are the average metrics running over 5 trials each.

It would seem that SVT is quite robust on artifical data, the relative error is low. Notice that the time required for higher rank matrices is more since the corresponding $|\Omega|$ is larger, so the algorithms have to deal with a less sparse matrix. Interestingly, the SVT algorithm is slower on uniformly generated matrices than on Gaussian generated matrices. The reason should be that uniformly generated matrices have larger entries than normally generated ones, as the entries are always positive and cannot cancel out.

As the trials are low, the time elapsed is for reference only. It obviously also depends on the state of the computer.

Running time of SVT with CPU vs GPU

The major bottlenecks in the SVT algorithm are in the shrinkage operator, namely SVD and matrix multiplication. Matrix multiplication can be parallelized, so running it over GPU will be faster. The results of running time of SVT with CPU matrix multiplication (matrix_completion_cpu.ipynb) and GPU matrix multiplication (matrix_completion.ipynb) show SVT with GPU is faster than CPU.

SVT vs Gradient Descent on Latent Factors

Another method of low rank matrix completion is given in [4]. Starting with sampled entries $P_\Omega(M)$, the algorithm in [4] fixes r, with randomly initialized 'latent' factors $A \in \mathbb{R}^{n\times r}, B \in \mathbb{R}^{r\times n}$, the algorithm tries to minimize $f(A, B) = \|P_\Omega(M - AB)\|_F^2$ by doing gradient descent on (A, B). This is implemented in descent.py via PyTorch, with Adam optimizer.

An obvious drawback of this method is that it requires to store both A, B, where the SVT algorithm only has to store a sparse matrix with entries in Ω theoretically. Also, this method requires a priori knowledge of the expected rank of the actual matrix, as the algorithm requires r as a hyperparameter.

In the below, the SVT and gradient descent method will be ran using the same actual matrix M, and the same sampled matrix $P_\Omega(M)$ and compared. This is done also in the notebook matrix_completion.ipynb. Here the stopping criterion is $\epsilon = 0.01$, so we expect the SVT algorithm to be faster than previous tests. For gradient descent $\|P_\Omega(M - AB)\|_F/\|P_\Omega(M)\|_F \leq \epsilon$ the same stopping criterion is used.

Size (n):	1000	1000	1000	1000	1000	1000	5000	5000
rank (r):	10	10	50	50	100	100	10	10
distribution:	Normal	Normal	Normal	Normal	Normal	Normal	Normal	Normal
$\lvert\Omega\rvert$:	120000	120000	390000	390000	570000	570000	600000	600000
time elapsed (s):	5.2	5.2	22.4	316.2	52.2	483.0	87	2175
max_diff:	0.3447	0.5345	0.437	0.654	0.535	0.940	0.390	0.537
rel_error:	0.0155	0.016	0.0154	0.0154	0.0157	0.0155	0.0155	0.0164
rank of prediction:	10	10	50	50	100	100	10	10
trials:	5	5	5	5	5	5	3	3
Algorithm:	SVT	descent	SVT	descent	SVT	descent	SVT	descent

Using GPU for matrix multiplication for SVT

In this case, the relative error of the SVT is similar to that of the gradient descent on latent factors method, so the accuracy of the algorithm seems to be robust. However, the time required for the gradient descent algorithm (using device=GPU on PyTorch) is much slower than SVT.

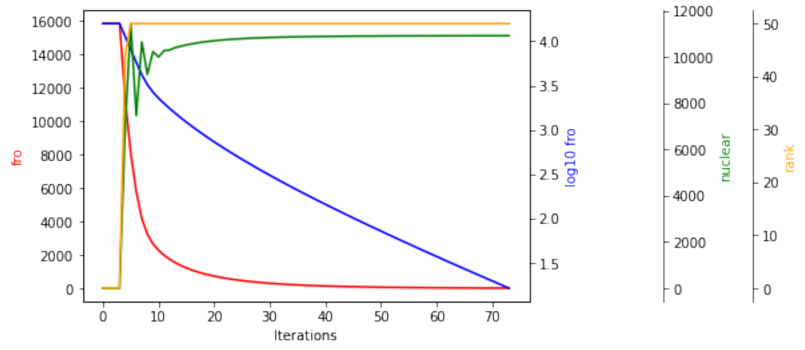
The learning rate of the gradient descent algorithm is determined by testing different values empirically, it would seem for larger values of lr the gradient descent algorithm is unable to decrease the relative error under 0.01. A larger learning rate would mean faster convergence, but with less accuracy.

Still, the above paragraph is anecdotal, as the convergence with respect to the learning rate is not provided. Also, a different optimizer / learning method can be used.

Nonetheless, this shows that the SVT algorithm has superior convergence properties, according to [2] the algorithm converges whenever the step size < 2, but the gradient descent algorithm requires toying around with the learning rate.

SVT on artificial large matrices

This test is on a 10000×10000 matrix, $|\Omega| = 5000000$ and rank 50, single trial. The relative error 0.0016 is still quite small, and the algorithm took 2144 seconds to run.



With similar observations as the SVT paper [2], the rank of the matrix is non-decreasing throughout, and the Frobenius norm strictly decreasing.

SVT on MovieLens Dataset

Introduction

Here we apply SVT on the MovieLens Dataset [5] (ml-25m.zip). This is application_to_movielens.ipynb, or application_to_movielens_cpu.ipynb for CPU version. We use the data from ml-25m/ratings.csv, which contains users ratings on movies, valued $\{1.0, 1.5, \cdots, 4.5, 5\}$. We restrict our attention onto particular 3000 unique users and 3000 unique movies (details explained later), and do cross validation on the dataset to see whether SVT is a good choice.

The reason that low rank matrix completion problem is applied to MovieLens is mainly because of the assumption that user's movie preferences depends only on a relatively small amount of factors (linearly). If we arrange each user's ratings in columns, then a user's preference to all movies is a vector.

Lets consider Action movies, so we have a column vector $\vec{v}_{\text{Action}}$ which $= 1$ if the movie entry is an action movie, and -1 if not. We expect the coefficient for users who like action to be $+ve$, and for users who dislike action movies to be $-ve$.

Considering all preferences p of the users (like Action, Adventure, Sci-Fi and so on), the ratings for the each user would be $\sum_p c_p \vec{v}_p$, where p is expected to be relatively small by assumption. Of course the users would not have watched all the movies, which gives the setting that only a subsample $P_\Omega(M)$ is known from the 'true' user ratings M of all the movies. This background justifies a possibility of the low rank matrix completion problem to the movie recommendation problem.

SVT setup for MovieLens

In the step size discussion of the SVT paper, section 5.1.2 [2], the algorithm converges whenever the step size $\delta < 2$. However, the authors observed a 'trick' that allows for larger step size $\delta = 1.2n^2/|\Omega|$ in their experiments (in general much larger than 2), so that the algorithm runs faster.

The discussion in section 5.1.2 of [2] hinges on the assumption that the actual matrix M is of rank not too large. So SVT might not converge for large δ if the rank of M is large. In the MovieLens data, we do not expect the ratings of users to be perfectly linearly depends on a small number of factors, as we expect it to be noisy. For instance, the rating depends on the mood of the user in the day of watching the movie.

The noisiness of the data poses a problem, given entries of $P_\Omega(M)$ where M is noisy, the matrix M might not be low rank! So the large step size 'trick' might not work. Indeed, running the SVT algorithm with the step size 'trick' on MovieLens dataset fails to converge, the Frobenius norm increases violently suddenly, but the expected behavior is that it is nonincreasing [2] Lemma 4.1.

For SVT to be run on MovieLens successfully, we can pick $\delta = 1.99$ for guaranteed convergence and being as large as possible (note: $\delta = 1.9999$ probably works but there might be numerical errors if δ is too close to 2). However, this seems to be still not fast enough. By empirical testing, $\delta = 4$ works for convergence, so we use that. If the data is changed, a different δ step size might be required.

Some preprocessing on data

The original MovieLens [5] ratings dataset is given in rows, where each row contains userId, movieId and rating (we do not care about other info). We do a simple two-pass filtering on users and movies. A filtering step is selecting the top M the movies (or users resp.) with the most data, for some integer M.

By filtering movies, then users, with $M = 10000, 3000$ sequentially, we arrive at a smaller dataset with 3000 unique movies and users. There are around 3050000 entries (available ratings) in total, so the available samples ratio is around 0.33966355555555555. Let Ω be the available samples.

For cross validation, we have to give the SVT algorithm a further subset of data, and see whether SVT does good on predicting the rest of the data. Here, we partition Ω into equally sized subsets $\Omega_1, \cdots, \Omega_{10}$. Then the prediction algorithm would be given the data Ω_k for some k, and validated on $\bigcup_{j \neq k} \Omega_j$.

Each Ω_k has around 305000 entries, and we are dealing with 3000×3000 data, which makes the available sample density around 3.4%.

Stopping criterion

The stopping criterion is set to $P_{\Omega_k}(M - \overline{M}) < 125$, where $\overline{M}$ is the prediction in current iteration (rather than relative error). This is because, heuristically, the ratings are discrete with difference 0.5, so difference in each entry to be 0.25 the ratings are exactly the same. Notice that $\sqrt{305000 * 0.25^2} \approx 125$.

Prediction & Results with SVT

Recall that the ratings from MovieLens [5] are $\{1, 1.5, \cdots, 4.5, 5\}$. For convenience, we scale it by a factor of 2 to make it integers from $1 \leq k \leq 10$ when we compare the predictions vs the actual ratings.

For prediction, of course we feed in $P_{\Omega_k}(M)$, into the SVT, with M the original matrix. Notice that M is non-zero only on Ω since we do not have the data elsewhere. With the result $\mathrm{SVT}(P_{\Omega_k}(M))$, we threshold it entrywise so it is between 1 and 10, and round to the nearest integer.

This gives a prediction of rating for each pair of movie and user. The predicted rating for user j watching movie i would be $\left[T(\mathrm{SVT}(P_{\Omega_k}(M)))\right]_{ij}/2$ where T is the thresholding and rounding operator. Afterwards we compare $T(\mathrm{SVT}(P_{\Omega_k}(M)))$ with the actual M (with entries in Ω) on the training set Ω_k, and also on the validation set $\bigcup_{j \neq k} \Omega_j$. Note that the validation set is 9x size the training set.

Metrics Consider the training set Ω_k and validation set $\tilde{\Omega}_k = \bigcup_{j \neq k} \Omega_j$. In the below, we always consider the x2 version of the ratings, $\{1, 2, \cdots, 10\}$. For convenience, let $\overline{M} = T(\mathrm{SVT}(P_{\Omega_k}(M)))$ be the predicted matrix.

$$\begin{aligned}
\text{Correct} &= \text{Number of predicted ratings in } \tilde{\Omega}_k \text{ exactly equal to actual} \\
\text{Correct ratio} &= \text{Correct}/|\tilde{\Omega}_k| \\
\text{Off by one} &= \text{Number of predicted ratings in } \tilde{\Omega}_k \text{ with distance to actual by at most one} \\
\text{Off by one ratio} &= \text{Correct}/|\tilde{\Omega}_k| \\
L2 &= \|P_{\tilde{\Omega}_k}(\overline{M} - M)\|_F / \|P_{\tilde{\Omega}_k}(M)\|_F
\end{aligned}$$

This means L2 is the relative error of Frobenius norm restricted to the entries $\tilde{\Omega}_k$, the validation set. Recalling the ratings are x2, this means off by one are actually ± 0.5 (or exactly equal) in the actual ratings. We also validate $\overline{M}$ with Ω_k itself, and the formulas above are the same by replacing $\tilde{\Omega}_k$ with Ω_k.

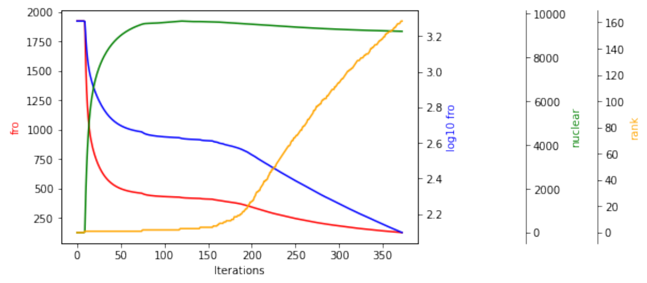
Results These are drawn from the GPU version (application_to_movielens.ipynb). The results are similar in the CPU version anyway. Let $\tilde{\Omega}$ be the validation set, and Ω be the training set (which training set is explicitly indicated below). The column $k =$ indicates which Ω_k is used for training, $|\tilde{\Omega}|$ means the size of the validation set, Used set means whether the training set itself or validation set is used for validation (for Train Ω_k itself, otherwise $\bigcup_{j \neq k} \Omega_j$).

Results of SVT prediction pipeline on MovieLens

$k =$	$\|\tilde{\Omega}\|$	Used set	Correct	Ratio	Off by one	Ratio	L2
0	305697	Train	229025	0.7491	304630	0.9965	0.073
0	2751273	Validate	713284	0.259	1821198	0.6619	0.243
1	305697	Train	229111	0.749	304621	0.9964	0.073
1	2751273	Validate	715054	0.259	1821671	0.662	0.243
2	305697	Train	228705	0.7481	304704	0.9967	0.073
2	2751273	Validate	715828	0.260	1823013	0.6626	0.243
3	305697	Train	228421	0.7472	304651	0.9966	0.073
3	2751273	Validate	714456	0.2596	1820838	0.6618	0.243
4	305697	Train	229066	0.7493	304652	0.9965	0.073
4	2751273	Validate	716551	0.2604	1825025	0.6633	0.243
5	305697	Train	228635	0.7479	304643	0.9965	0.073
5	2751273	Validate	714406	0.2596	1820191	0.6615	0.243
6	305697	Train	228773	0.7483	304545	0.9962	0.073
6	2751273	Validate	714059	0.2595	1820869	0.6618	0.243
7	305697	Train	228770	0.7483	304636	0.9965	0.073
7	2751273	Validate	712951	0.2591	1821447	0.662	0.243
8	305697	Train	228700	0.7481	304627	0.9964	0.073
8	2751273	Validate	715290	0.2599	1822299	0.6623	0.243
9	305697	Train	229182	0.7497	304660	0.9966	0.073
9	2751273	Validate	715712	0.2601	1823842	0.6629	0.243

We see that both the correct ratio, off by one ratio, L2 indicates more accuracy for validating on the training set Ω_k rather than the validation set $\tilde{\Omega}_k = \bigcup_{j \neq k} \Omega_j$. This is expected as the SVT algorithm minimizes $P_{\Omega_k}(\tilde{M} - M) \to 0$ as the iterations $\to \infty$, so there are more correct values by validating via training set than the validation set.

Notice that the correct ratios, off by one ratios, and L2 are quite consistent across training with different subsets Ω_k. It may be interesting to inquire whether further decreasing the stopping criterion into $\|P_{\Omega_k}(\tilde{M} - M)\|_F < \epsilon$ where ϵ is much less than 125 yields better results. According to the training metrics of one particular instance,



the rank has significantly shot up after some iterations. This is probably due to the noise in the user ratings as mentioned before, so that the noisy matrix M has high rank, but actually $M = N + E$ where N has low rank and the error E random. From the graph we can infer the 'actual rank' is around 10-20, but corrupted with noise so the rank shoots up after $|P_{\Omega_k}(M - \tilde{M})|_F$ is sufficiently small.

Conclusion of prediction with SVT

From the above discussion, it seems impossible to further improve the predictions by choosing a stricter stopping criterion because of the noise. However, the training results seem to do well at prediction. In the validation set, there are 26% of ratings exactly correct as prediction, and 66% predicted ratings off by at most 0.5 to the actual value.

If the prediction were chosen at random, there are 9 discrete choices of prediction values, so we expect the correct ratio to be $\frac{1}{9}$, and off by one ratio $< \frac{3}{9}$ (if the actual rating is 1 or 5, there is $\frac{2}{9}$ chance of being correct). So the SVT algorithm achieves at least two times accuracy than guessing the ratings completely at random. The probability of the guessed ratings off by at most 0.5 is 66%, which would seem somewhat reliable on predicting user's preferences roughly. The L2 is not large (≈ 0.24), by these metrics we conclude SVT does well on predictions.

Some empirical observations of running time

In the SVT paper [2], the running time of SVT is efficient, since it only has to deal with sparse matrices and the SVD can be done with the first k singular values. However, in the graph above for running SVT on MovieLens, the rank shoots up after a specific threshold on rank, due to the noise. This is bad for running time, as more singular values have to be computed, and a larger matrix multiplication have to be done to reconstruct the matrix from SVD.

For faster results, choose an larger stopping criterion so that the matrix remains low rank throughout, and less iterations are made. Notice that the CPU version takes around 1200s to complete, while the GPU version around 900s to complete. This is because matrix multiplication benefit a lot with GPU, and a large matrix multiplication is needed when the rank shoots up in the SVT iterations.

The above time are just rough estimates, since it depends on the system and the sets $\Omega_1, \cdots, \Omega_{10}$ are randomly sampled from Ω, so the CPU and GPU version are ran with different partitions.

Further possibilities

If we use GPU for matrix multiplication, the major bottleneck in the SVT algorithm is the SVD step, since it cannot be parallelized easily. To further optimize running time, we could implement another variant of SVT which does not involve the SVD. This is explored in another paper [3].

For the prediction on MovieLens data, we only selected particular users which watches movies alot on the platform, and particular movies which is being watched alot. To do general prediction, it may be possible to try augmenting the 3000×3000 matrix with a small number of users / movies with not much known ratings, and see if SVT works well on those.

Appendix - Documentation of Python Code

Python libraries used

They are: numpy cupy pandas torch scipy matplotlib jupyter_notebook

numpy	1.23.3
cupy-cuda117	10.6.0
pandas	1.2.2
torch	1.13.0+cu117
scipy	1.9.1
matplotlib	3.3.4
jupyter	1.0.0

Other versions differing not too much probably still work. cupy is not necessary for the CPU versions of the Python notebooks.

svt.py and svtcpu.py

They are implementations of the GPU and CPU version (for matrix multiplication) of the SVT algorithm. Main Functions:

svt_algorithm (M, locations, step_size, tolerance, tau, increment, log, tolerance_absolute)

M is the partial sparse matrix.
locations are the Ω of available samples.
step_size is the δ in SVT algorithm.
tolerance is the stopping criterion ϵ.
τ is the hyperparameter in SVT.
increment is how much the no. of singular values should increase in the iteration of shrinkage_operator (refer to 5.1.1 of [2]). **log** determines whether logging is enabled.
tolerance_absolute determines whether to use relative L2 $\frac{\|P_\Omega(\overline{M}-M)\|_F}{\|P_\Omega(M)\|_F}$ or absolute L2 $\|P_\Omega(\overline{M}-M)\|_F$ as stopping criterion.

The other functions are just helper functions or SVT with some predefined parameters.

descent.py

The main function is gradient_descent_completion. The arguments are the same as SVT, but with rank in addition. The rank is the expected rank of the matrix. This file is a pretty standard gradient descent implementation with PyTorch, and should automatically detect whether GPU or CPU is used as device.

utils.py

This file contains a bunch of miscellaneous helper functions. This file contains all the random generation functions, such as random low rank matrix, random set of locations, and so on. It also has a filter locations function, which cuts down a matrix, retaining only the value in specified locations, setting the others to zero. Each function is documented in the code.

matrix_completion.ipynb & matrix_completion_cpu.ipynb

These contain the GPU and CPU versions of testing with artificial data, and comparison with the Latent Factor gradient descent method [4]. All the cells separated by a large header in Markdown (lines starting with #) can be run separately.

For each sequence of cells under the same large header, they have to be run sequentially from top to bottom. The details of how the algorithms are tested with artifical data are given above.

application_to_movielens.ipynb & application_to_movielens_cpu.ipynb

These are the GPU and CPU versions of SVT applying to predict ratings in MovieLens. The cells must be run from top to bottom sequentially.

It is necessary to create a folder "ml-25m" in the same directory as the ipynb file, and the "ml-25m" folder should contain a file ratings.csv, which are the ratings from MovieLens [5].

References

[1] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. *Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers*. Now Foundations and Trends, 2011.

[2] Jian-Feng Cai, Emmanuel J. Candes, and Zuowei Shen. A singular value thresholding algorithm for matrix completion. *SIAM Journal on Optimization*, 20:1956–1982, 2010.

[3] Jian-Feng Cai and Stanley Osher. Fast singular value thresholding without singular value decomposition. *Methods and Applications of Analysis*, 20:335–352, 2013.

[4] Denise Chen. Recommender system - singular value decomposition (svd) & truncated svd. `https://towardsdatascience.com/recommender-system-singular-value-decomposition-svd-truncated-svd-97096338f361`, 08 2000. Accessed: 2022-10-20.

[5] Group Lens. Movie lens. `https://grouplens.org/datasets/movielens/`, 2022. Accessed: 2022-10-2.