

CompleteThat

A python package for low-rank matrix completion

EEOR E4650 Course Project

Joshua Edgerton

Esteban Fajardo

January 5, 2014

Abstract

We have developed a Python package (`CompleteThat`) to perform low rank matrix completion. Given a low rank matrix with partial entries we implemented different methods to solve the matrix completion problem: First, for “small” problems we implemented two memory-based algorithms following the work by Tanner and Wei [TW14] to find, given a number r , the matrix with rank r that best fits the data using the Frobenius norm. Second, when the problem does not fit into memory, we implemented an out-of-core algorithm (stochastic gradient descent) following the approach in Zhang [Zha04] and Bottou [Bot12]. The package is available at <https://pypi.python.org/pypi/completethat/0.1dev>.

Contents

1	Introduction	3
2	Matrix Completion Problem	3
3	Implementation	4
4	Case Studies	9
5	Future Work	13
6	Source code	15

1 Introduction

Matrix factorization has recently seen a large growth in popularity within the mathematics, statistics, and computer science communities as industry continues to apply machine learning techniques on a wide array of problems and scenarios. For our convex optimization project, we decided to take some of the more interesting and applicable topics and algorithms of our class and develop a python package to implement them. We briefly discuss the theory behind matrix factorizing, the models and approaches we choose to use and the algorithms for the optimization. Later, we walk through two case studies illustrating the usefulness and applicability in the context of image processing and a recommendation system for Yahoo music and movie data.

2 Matrix Completion Problem

The matrix completion problem can be formally stated as follows. We are interested in recovering a matrix $M \in \mathbf{R}^{n_1 \times n_2}$ but only get to observe a number $m \ll n_1 n_2$ of its entries. Thus, we want to find a solution to the following optimization problem

$$\begin{aligned} & \text{minimize} && \mathbf{rank}(X) \\ & \text{subject to} && X_{ij} = M_{ij} \quad (i, j) \in \Omega \end{aligned} \tag{1}$$

where $X \in \mathbf{R}^{n_1 \times n_2}$ is the decision variable and $\mathbf{rank}(X)$ is equal to the rank of the matrix X . The problem (1) seeks the simplest matrix fitting the observed data. Of course, if there were only one low-rank object fitting the data, this would recover M . In order to fix the notation correctly define the projector $P_\Omega : \mathbf{R}^{n_1 \times n_2} \rightarrow \mathbf{R}^{n_1 \times n_2}$ as

$$P_\Omega(A) = \begin{cases} A_{ij} & : (i, j) \in \Omega \\ 0 & : \text{otherwise} \end{cases}$$

Using the projector, Problem (1) can be rewritten as

$$\begin{aligned} & \text{minimize} && \mathbf{rank}(X) \\ & \text{subject to} && P_\Omega(X) = P_\Omega(M) \end{aligned} \tag{2}$$

Unfortunately, this optimization problem has been shown to be NP-hard and all known algorithms which provide exact solutions require time doubly exponential in the dimension of the matrix both in theory and in practice [CR09]. Also, the $\mathbf{rank}(X)$ makes the problem non-convex.

Several approximations to the problem exist. One, the “tightest convex relaxation of $\mathbf{rank}(X)$ ” [Faz02] is the following problem

$$\begin{aligned} & \text{minimize} && \|X\|_* := \sum_{i=1}^r \sigma_i(X) \\ & \text{subject to} && P_\Omega(X) = P_\Omega(M) \end{aligned} \tag{3}$$

where $\sigma_i(X)$ denotes the i th largest singular value of X and $\|X\|_*$ is the called the nuclear norm. The main point of this relaxation is that the nuclear norm is a convex function and thus can be optimized efficiently via semidefinite programming or by iterative soft thresholding algorithms [CCS10] [GM11].

Alternative to nuclear norm minimization there have been many algorithms which are designed to target the following optimization problem

$$\text{minimize}_{Y,Z} \quad \frac{1}{2} \|P_\Omega(YZ) - P_\Omega(M)\|_F^2 \quad (4)$$

where $X = YZ$ is the completed matrix and $Y \in \mathbf{R}^{n_1 \times r}$, $Z \in \mathbf{R}^{r \times n_2}$ and r represents the (hopefully small) rank of the matrix X . The main point of this relaxation is the use of an alternating minimization scheme or Gauss-Seidel or 2-block coordinate descent method, where first Y is fixed and then the problem is reduced to a convex, standard least squares problem on Z . Later, Z is fixed and then the problem is reduced to a convex, standard least squares problem on Y .

3 Implementation

CompleteThat is a python package that solves the low rank matrix completion problem. Given a low rank matrix with partial entries the package solves an optimization problem to estimate the missing entries. We allow the user to choose between several optimization algorithms including two in memory based algorithms, alternating steepest descent and scaled alternating steepest descent, and one out of memory fitting procedure using stochastic gradient descent. We use extensively the numerical libraries `spicy` and `numpy`, and the current implementation includes two classes, `MatrixCompletion` for in memory based algorithms and `MatrixCompletionBD` for the memory fitting procedure. The package is available worldwide and can be downloaded from <https://pypi.python.org/pypi/completethat/0.1dev>.

CVX

Problem (3) can be easily solved directly using CVX, a package for specifying and solving convex programs [GB14], [GB08], with the following code:

```
index = find(~isnan(M));
cvx_begin
    variable X(size(M));
    minimize norm_nuc(X)
    % s.t.
    X(index) == M(index);
cvx_end
```

Where M is a MATLAB matrix with `nan` on the missing entries and `X(index) == M(index)` corresponds to the $P_\Omega(X) = P_\Omega(M)$ restriction. However, the computation is

very slow and for any matrix greater than 100x100 the computational time is too high. Since this is clearly unsatisfactory for practical purposes, specific algorithms were implemented on the package.

Algorithms

Low rank matrix completion by alternating steepest descent methods

The alternating steepest descent methods implemented on `CompleteThat` solve the Problem (4) above. Let the objective function of the optimization problem $f : \mathbf{R}^{n_1 \times r} \times \mathbf{R}^{r \times n_2} \rightarrow \mathbf{R}$ be defined as

$$f(Y, Z) := \frac{1}{2} \|P_\Omega(YZ) - P_\Omega(M)\|_F^2. \quad (5)$$

To solve efficiently the above problem, Tanner and Wei [TW14] use a single step of simple line-search along the gradient descent directions. Let us write $f(Y, Z)$ as $f_Z(Y)$ when Z is held constant and $f_Y(Z)$ when Y is held constant and let t_y, t_z be the steepest descent step sizes for descent directions.

The Alternating Steepest Descent Method (ASD) applies steepest gradient descent to $f(Y, Z)$ in (5) alternatively with respect to Y and Z . It can be shown that the directions of gradient ascent and the steepest descent step sizes are [TW14] :

$$\begin{aligned} \nabla f_Z(Y) &= -(P_\Omega(M) - P_\Omega(YZ))Z^T \\ \nabla f_Y(Z) &= -Y^T(P_\Omega(M) - P_\Omega(YZ)). \\ t_y &= \frac{\|\nabla f_Z(Y)\|_F^2}{\|P_\Omega(\nabla f_Z(Y)Z)\|_F^2} \\ t_z &= \frac{\|\nabla f_Y(Z)\|_F^2}{\|P_\Omega(Y\nabla f_Y(Z))\|_F^2} \end{aligned}$$

Algorithm 1 Alternating Steepest Descent (ASD) [TW14]

Input: $r \in \mathbf{R}, P_\Omega(M), Y_0 \in \mathbf{R}^{n_1 \times r}, Z_0 \in \mathbf{R}^{r \times n_2}$

Repeat:

$$\begin{aligned} \nabla f_{Z_i}(Y_i) &= -(P_\Omega(M) - P_\Omega(Y_i Z_i))Z_i^T, t_{y_i} = \frac{\|\nabla f_{Z_i}(Y_i)\|_F^2}{\|P_\Omega(\nabla f_{Z_i}(Y_i)Z_i)\|_F^2} \\ Y_{i+1} &= Y_i - t_{y_i} \nabla f_{Z_i}(Y_i) \\ \nabla f_{Y_{i+1}}(Z_i) &= -Y_{i+1}^T(P_\Omega(M) - P_\Omega(Y_{i+1} Z_i)), t_{z_i} = \frac{\|\nabla f_{Y_{i+1}}(Z_i)\|_F^2}{\|P_\Omega(Y_{i+1} \nabla f_{Y_{i+1}}(Z_i))\|_F^2} \\ Z_{i+1} &= Z_i - t_{z_i} \nabla f_{Y_{i+1}}(Z_i) \\ i &= i + 1 \end{aligned}$$

Until: termination criteria is reached

Output: $X := Y_i Z_i \in \mathbf{R}^{n_1 \times n_2}$

The Scaled Alternating Steepest Descent Method (sASD) is an accelerated version of ASD and uses a scaled gradient descent direction with exact line-search. Its main motivation for

the scaled factors is the explicit expression for the Newton directions for the problem (5) if all the entries in the matrix M were know:

$$(M - YZ)Z^T(ZZ^T)^{-1}$$

$$(X^T X)^{-1}X^T(M - YZ)$$

which are the gradient descent directions scaled by $(ZZ^T)^{-1}$ and $(X^T X)^{-1}$.

Algorithm 2 Scaled Alternating Steepest Descent (sASD) [TW14]

Input: $r \in \mathbf{R}, P_\Omega(M), Y_0 \in \mathbf{R}^{n_1 \times r}, Z_0 \in \mathbf{R}^{r \times n_2}$

Repeat:

$$\begin{aligned} \nabla f_{Z_i}(Y_i) &= -(P_\Omega(M) - P_\Omega(Y_i Z_i))Z_i^T \\ d_{y_i} &= -\nabla f_{Z_i}(Y_i)(Z_i Z_i^T)^{-1}, t_{y_i} = \frac{-\nabla f_{Z_i}(Y_i) \cdot d_{y_i}}{\|P_\Omega(d_{y_i} Z_i)\|_F^2} \\ Y_{i+1} &= Y_i + t_{y_i} d_{y_i} \\ \nabla f_{Y_{i+1}}(Z_i) &= -Y_{i+1}^T (P_\Omega(M) - P_\Omega(Y_{i+1} Z_i)) \\ d_{z_i} &= (Y_{i+1}^T Y_{i+1})^{-1} \nabla f_{Y_{i+1}}(Z_i), t_{z_i} = \frac{-\nabla f_{Y_{i+1}}(Z_i) \cdot d_{z_i}}{\|P_\Omega(Y_{i+1} d_{z_i})\|_F^2} \\ Z_{i+1} &= Z_i + t_{z_i} d_{z_i} \\ i &= i + 1 \end{aligned}$$

Until: termination criteria is reached

Output: $X := Y_i Z_i \in \mathbf{R}^{n_1 \times n_2}$

Low rank matrix completion by stochastic gradient descent

The formal problem as defined in the model section is stated in the standard language of linear algebra and the typical optimizations using gradient methods require the calculation of the gradient based on the sum of all the error terms, meaning that when updating our latent feature vectors, we calculate the gradient based on all the data in our ratings file. Given the scale of data that is common in today's digital world, the need for more efficient and memory friendly algorithms arises as calculating the gradient on a large enough file will be computationally expensive and time consuming.

Let U be the set of users and I the set of items. Let $K \subset U \times I$ be the set of known ratings and r_{ij} the rating that user i gave to item $j, \forall (i, j) \in K$. Let p_i be the latent feature vector for user i of length equal to hypothesized rank. Let q_j be the latent feature vector for item j of length equal to hypothesized rank. The stochastic gradient descent method solves the following optimization problem:

$$\begin{aligned} &\text{minimize}_{r_{ij} \in K} \quad f := (r_{ij} - \hat{r}_{ij})^2 = e_{ij}^2 \\ &\text{subject to} \quad r_{ij} = p_i^T q_j \end{aligned} \tag{6}$$

With Stochastic gradient descent, a variation of the standard batch gradient optimization, instead of calculating the gradient using all of the ratings data, we approximate the gradient

of our objective function using just the current record and update just the latent feature vectors of the user and item pertaining to the current rating. Thus one of the highlights of our package is that we simply read the file in line by line, running the algorithm in place. After each update, we simply read in the next line of the file and repeat.¹

Our RAM requirements for the algorithm decrease significantly with this sort of out-of-memory implementation. For example, with the Yahoo Music Data we only needed to store the parameters for each user and item, rather than the entire 115 million records of our file plus the parameters. This results in storing 2 million parameters in a python dictionary (a sort of built in hash function and table) rather than 115 million records in memory. Thus we are essentially exchanging the luxury of using less memory and computational resources for more time and iterations spent with stochastic gradient descent.

After the algorithm converges or the maximum number of iterations are met we have a set of user and item latent vectors which can be used to compute the estimated rating r_{ij} by simply taking the dot product of the two vectors. Similarly the entire ratings matrix could easily be built with the vectors although we have neglected to do so for practicality reasons.

Algorithm 3 Stochastic Gradient Descent (SGD)

Input: $r_{ij} \in \mathbf{R}, \forall (i, j) \in K$

Initialize Parameters: $\forall i \in U$ randomly initialize $p_i \in \mathbf{R}^r$. $\forall j \in I$ randomly initialize $q_j \in \mathbf{R}^r$

Repeat:

 Read rating r_{ij}

 Compute error $e_{ij} = r_{ij} - p_i^T q_j$

 Compute gradients:

$$\frac{\partial f}{\partial p_i} = -2e_{ij}q_j$$

$$\frac{\partial f}{\partial q_j} = -2e_{ij}p_i$$

 Update parameters:

$$p_i := p_i + 2\alpha e_{ij}q_j$$

$$q_j := q_j + 2\alpha e_{ij}p_i$$

 Update MSE: $\text{MSE} := \text{MSE} + e_{ij}$

Until: termination criteria is reached

Shuffle Data Set

Output: $p_i \ \forall i \in U; q_j \ \forall j \in I$

Notes

- **Step Size:** The algorithm's performance is highly dependent on the step size (α) parameter of the update equations. Too small of an alpha will result in convergence taking a very long time and lower the chance of finding the global objective. Too large of an alpha

¹We note that our formulation is not convex but, due to shuffling of the data and the way the gradients are estimated, the algorithm will converge to at least a local minimum.

will result in convergence not happening as the the algorithm continually 'overshoots' the optimal value. Thus an essential part of using this algorithm will be experimentation with the step size. Currently, the step size (alpha parameter) is fully adjustable, with the default being a .01 decreasing 30% each pass over the data until it reaches 1e-6 whereby it will remain constant until the stopping criterion of the algorithm are reached.

- **Shuffle:** When dealing with large data files we needed to be clever with how we shuffle our data. Stochastic gradient descent performs much better with shuffling of the data after each pass over the file thus shuffling is essential; yet shuffling is no trivial task for files too big to fit entirely into the computers random access memory. We explored various shuffling methods for both in-memory and out-of-memory but ultimately decided with using a batch pseudo-shuffle. Our batch shuffle method shuffles the file in chunks. So we read in some large portion (the batch size) of the file that will fit in memory, we shuffle the ratings of that batch using built in python functions, and lastly we write the shuffled ratings back to a text file, and continue sequentially until the all of the file has been shuffled. Following this method we say 'pseudo-shuffled' since the file is not truly shuffled. It is shuffled within the batch/portion of the file it belongs to but still not truly shuffled within the entire file. The larger the batches, the better the shuffled ratings, until of course we read the entire file into memory and shuffle it, the ideal situation. Hopefully by choosing a large enough batch size for reading in, our file will be shuffled well enough. We also explored a more robust method of shuffling the data by additionally shuffling the shuffled batches whereas before we wrote the shuffled ratings back to a file sequentially. However ultimately the noticeably bigger time complexity of moving and appending large files together via Linux did not prove worthwhile given almost negligible returns in convergence

Code Example

For matrices that fit into memory use the `MatrixCompletion` module. Otherwise, use the `MatrixCompletionBD` module.

MatrixCompletion

Given a numpy matrix M with `numpy.nan` on the missing entries, the matrix completion problem can be solved (using ASD, for example) as:

```
>>> from completethat import MatrixCompletion
>>> problem = MatrixCompletion(M)
>>> problem.complete_it("ASD")
>>> X = problem.get_matrix() #Desired matrix
>>> out_info = problem.get_out() #Extra information
```


MatrixCompletionBD

Given a csv file with the input data with three fields (user id, item id , rating) the matrix completion problem can be solved as:

```
>>> from completethat import MatrixCompletionBD
>>> problem = MatrixCompletionBD('input_data.txt')
>>> problem.train_sgd(dimension=6,init_step_size=.01,min_step=.000001,
                    reltol=.001,rand_init_scale=10, maxiter=1000,
                    batch_size_sgd=50000,shuffle=True)
>>> problem.validate_sgd('test_data.txt')
>>> problem.save_model()
```

4 Case Studies

Yahoo movies and music reviews database

The Yahoo music and movie data sets are publicly available datasets published by Yahoo for use to the public for recreational and research purposes. The music training data set is a small 4 mb tab-delimited file consisting of roughly 210,000 user-movie ratings. The test set has roughly 10,000 records. There are actually two ratings schemas available, one on a 5-point scale and the other on a 13-point scale and we arbitrarily chose to use the 5-point scale. The Yahoo music data is a larger 2.2 gb file of 115 million user-song ratings. The ratings are on a 100 point scale.

We look to evaluate the performance of our algorithms on these two datasets, comparing the various algorithms implemented versus a naive benchmark estimate where we used the mean of the training set as our estimator for all records in the test set. Then we compared how well the algorithms performed versus the benchmark as well as Mahout. Apache Mahout is a mainstream scalable and fast machine learning library implemented in java which has a recommendation engine component similar to MatrixCompletionBD but uses alternating least squares for the optimization. Thus given the origins and popularity of Mahout, we are also interested in comparing it to our python package's performance. The naive estimator for the yahoo music data is 4.088. This is the mean value of the ratings in the training set so we will compute the MSE on the test set using the benchmark, which yields an RMSE of 1.10 (Note we use RMSE here only because Mahout uses RMSE). A quick run of Mahout yields 1.17 and CompleteThat's SGD model yielded a 1.19 (we used rank 7 but larger ranks will outperform the naive baseline as we show in our table). So our model yielded a pretty similar RMSE as the MAHOUT package. Note that for both of these instances we used a factorization of rank equal to seven and the CompleteThat stochastic gradient descent algorithm ran slightly faster than Mahout's, with our SGD algorithm running on 220,000 ratings and converging in less than a minute (.537 minutes versus 1.4 minutes)!

The Yahoo Music data is a lot larger and provided a much more interesting scenario but unfortunately due to the time constraints and how long it took to converge on the Yahoo

Music data, we didn't get as much testing done here. It does take a very long time for the stochastic gradient descent algorithm to converge, but does run. In experimentation with features of rank six, we saw it converging in approximately 9 hours, with results roughly on par relative to the naive baseline using the training set. Unfortunately, we weren't able to get Apache Mahout working on our local machines with that scale of data, thus have no real comparison to be made in terms of how well our package does scale, but if nothing else we can say that our algorithm is fairly invariant to the size of the data.

Table 1 shows the performance of the SGD on the Yahoo music with varying the rank of the user and item latent features. We see that the algorithm performs best on the test set using a rank around 15, and it is evident that too large a rank is detrimental to performance.

Step size	Iterations	Minutes	Train MSE	Test RMSE
1	8	0.585	1.411	1.757
2	8	0.598	1.304	1.550
3	8	0.628	1.235	1.483
4	8	0.617	1.180	1.397
5	7	0.538	1.136	1.351
6	7	0.534	1.100	1.244
7	7	0.534	1.071	1.194
8	7	0.541	1.046	1.127
9	7	0.537	1.027	1.128
10	7	0.529	1.010	1.102
15	7	0.551	0.971	1.063
20	7	0.540	0.976	1.075
30	8	0.614	1.045	1.359

Table 1: RMSE vs Step Size vs Rank

Columbia University photographs

In this section, we demonstrate the applicability of the package to image processing problems. Grayscale pictures can be transform into a rectangular matrix where each element of the matrix determines the intensity of the corresponding pixel. For convenience, most of the current digital files use integer numbers between 0 and 255. By randomly erasing, or setting to zero, a predefined proportion of the pixels of an image the problem of completing the image can be cast as a matrix completion problem, where the missing entries are the those with zero on it.

For illustration we used three 512 x 512 grayscale photographs taken by one of the authors of the Columbia University campus on December, 2014. The original, erased and reconstructed images are shown on Figure 1, where we erased 35% of the pixels of each image. In addition, to illustrate the ease of use of the package, we also present the (very short) script used to recover the photographs:

```
import numpy as np
from scipy import misc
from completethat import MatrixCompletion

def blur(imgpath, delta = 0.65):
    """
        Function to blur the image
    """
    photo = scipy.misc.imread(imgpath, flatten=True).astype(float) #Read as
    grayscale
    m, n = photo.shape
    p = round(delta * (m * n - 1)) #Number of non-blanks
    A = np.zeros((m * n, 1), dtype=bool)
    A[0:p] = True
    ind = np.random.permutation(range(m*n))
    A = A[ind]
    A = A.reshape((m,n))
    return photo, A

if __name__ == '__main__':

    # Read image and randomly erase 35% of the pixels
    photo, A = blur('./columbia_1.png')
    M = np.copy(photo)
    M[~A] = np.nan
    problem = MatrixCompletion(M)

    # Solve the problem
    problem.complete_it('ASD')
    X = np.copy(problem.get_optimized_matrix())
```

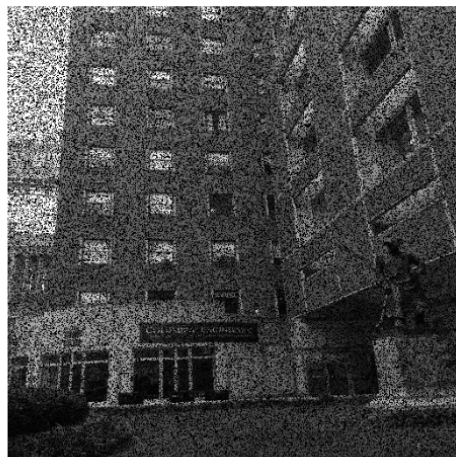
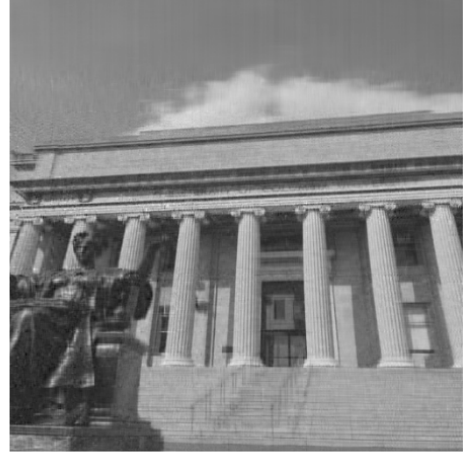


Figure 1: ASD method applied three different Columbia University photographs using random sampling

5 Future Work

We have plans for various improvements as we continue working with the `CompleteThat` python package. Overall, we would like to test the software and integrate automated testing cases into the software developing cycle. Also, documentation for the package and its functions, including examples, is high on our priority list. In addition, we hope to incorporate more rigorous error checking and make available more customization to the user.

For the memory-based algorithms it is well known that noise in the data introduces overfitting on the estimated matrix. Following the approach taken by Mazumder et al. [MHT10] where they solve the same objective function as the problems above (using the Frobenius norm) but introducing the nuclear norm as regularizer to account for overfitting, we would like to introduce a regularization option into the matrix completion procedure.

For the stochastic gradient descent method, we plan to add a lambda penalty feature for combatting overfitting as well as considering options for more advanced versions of our basic latent factor model as demonstrated by Koren [Kor08] in his paper on the prize-winning Netflix models. For all algorithms, we would like to explore and research various hardware and software optimization techniques for faster code.

Finally, given the similarities between matrix completion and robust principal component analysis one further extension to the functionality of the package could be implementing the robust-PCA procedure outlined by Candés et al. [CLMW11].

References

- [Bot12] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.
- [CCS10] Jian-Feng Cai, Emmanuel J Candès, and Zuowei Shen. A singular value thresholding algorithm for matrix completion. *SIAM Journal on Optimization*, 20(4):1956–1982, 2010.
- [CLMW11] Emmanuel J Candès, Xiaodong Li, Yi Ma, and John Wright. Robust principal component analysis? *Journal of the ACM (JACM)*, 58(3):11, 2011.
- [CR09] E. J. Candès and B. Recht. Exact matrix completion via convex optimization. *Foundations of Computational mathematics*, 9(6):717–772, 2009.
- [Faz02] M. Fazel. *Matrix rank minimization with applications*. PhD thesis, Stanford University, 2002.
- [GB08] Michael Grant and Stephen Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent Advances in Learning and Control*, Lecture Notes in Control and Information Sciences, pages 95–110. Springer-Verlag Limited, 2008. http://stanford.edu/~boyd/graph_dcp.html.
- [GB14] Michael Grant and Stephen Boyd. CVX: Matlab software for disciplined convex programming, version 2.1. <http://cvxr.com/cvx>, March 2014.
- [GM11] Donald Goldfarb and Shiqian Ma. Convergence of fixed-point continuation algorithms for matrix rank minimization. *Foundations of Computational Mathematics*, 11(2):183–210, 2011.
- [Kor08] Yehuda Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–434. ACM, 2008.
- [MHT10] Rahul Mazumder, Trevor Hastie, and Robert Tibshirani. Spectral regularization algorithms for learning large incomplete matrices. *The Journal of Machine Learning Research*, 11:2287–2322, 2010.
- [TW14] J. Tanner and K. Wei. Low rank matrix completion by alternating steepest descent methods. Technical report, SIAM, SIAM J. Imaging Sciences, 2014.
- [Zha04] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *Proceedings of the twenty-first international conference on Machine learning*, page 116. ACM, 2004.

6 Source code

On the whole, the package directory structure looks like Figure 2. In what follows we present the source code of the package.

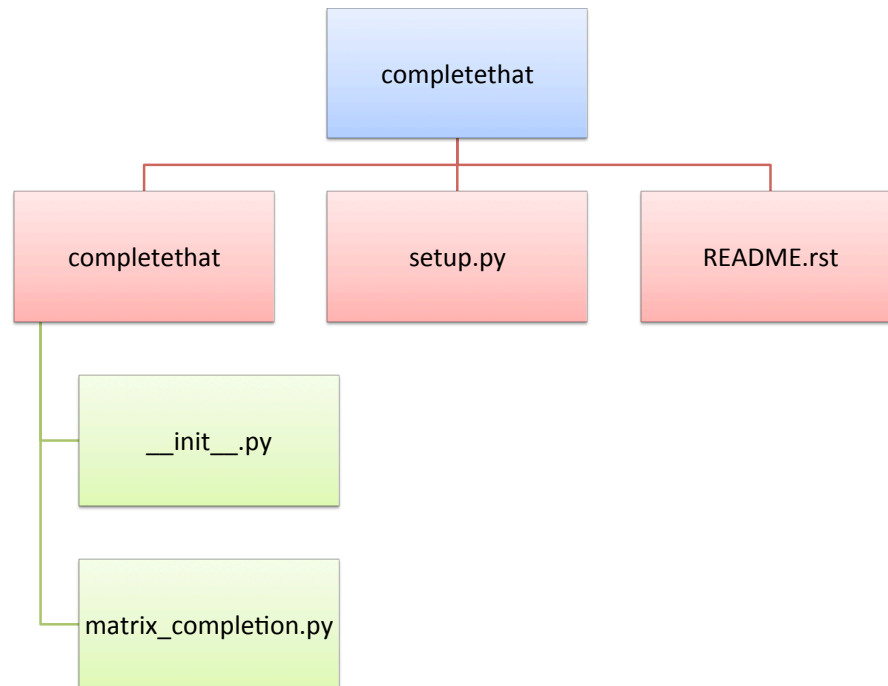


Figure 2: CompleteThat structure

completethat/matrix_completion.py

```
1 import numpy as np
2 from scipy.sparse import csc_matrix
3 from scipy.sparse import linalg as linalg_s
4 import os, random, time
5 from scipy import linalg
6
7 class MatrixCompletion:
8     """ A general class to represent a matrix completion problem
9
10     Data members
11     """
12     M= data matrix (numpy array).
13     X= optimized data matrix (numpy array)
```

```

14 out_info:= output information for the optimization (list)
15
16
17 Class methods
18 =====
19 complete_it():= method to complete the matrix
20 get_optimized_matrix():= method to get the solution to the problem
21 get_matrix():= method to get the original matrix
22 get_out():= method to get extra information on the optimization (iter
23 number, convergence, objective function)
24
25 """
26 def __init__(self, X,*args, **kwargs):
27     """ Constructor for the problem instance
28
29     Inputs:
30     1) X: known data matrix. Numpy array with np.nan on the unknow
entries.
31
32     example:
33         X = np.random.randn(5, 5)
34         X[1][3] = np.nan
35         X[0][0] = np.nan
36         X[4][4] = np.nan
37
38     """
39     # Initialization of the members
40     self.M = X
41     self.X = np.array(X, copy = True) #Initialize with ini data matrix
42     self._out_info = []
43
44 def get_optimized_matrix(self):
45     """ Getter function to return the optimized matrix X
46
47     Ouput:
48     1) Optimized matrix
49
50     """
51     return self._X
52
53 def get_matrix(self):
54     """ Getter function that returns the original matrix M
55
56     Output:
57     1) Original matrix M
58
59     """
60     return self._M
61
62 def get_out(self):
63     """ Getter function to return the output information
of the optimization
64
65     Output:

```



```

64         1) List of length 2: number of iterations and relative residual
65
66         """
67         return self._out_info
68
69
70     def _ASD(self, M, r = None, reltol=1e-5, maxiter=5000):
71         """
72         Alternating Steepest Descent (ASD)
73         Taken from Low rank matrix completion by alternating steepest descent
74         methods
75         Jared Tanner and Ke Wei
76         SIAM J. IMAGING SCIENCES (2014)
77
78         We have a matrix M with incomplete entries ,
79         and want to estimate the full matrix
80
81         Solves the following relaxation of the problem:
82         minimize_{X,Y} \frac{1}{2} ||P_{\Omega}(Z^0) - P_{\Omega}(XY)||_F^2
83         Where \Omega represents the set of m observed entries of the matrix M
84         and P_{\Omega}() is an operator that represents the observed data.
85
86         Inputs:
87         M := Incomplete matrix, with NaN on the unknown matrix
88         r := hypothesized rank of the matrix
89
90         Usage:
91         Just call the function _ASD(M)
92         """
93
94         # Get shape and Omega
95         m, n = M.shape
96         if r == None:
97             r = min(m, n, 50)
98
99         # Set relative error
100         Omega = ~np.isnan(M)
101         frob_norm_data = linalg.norm(M[Omega])
102         relres = reltol * frob_norm_data
103
104         # Initialize
105         I, J = np.where(Omega)
106         M_omega = csc_matrix((M[Omega], (I, J)), shape=M.shape)
107         U, s, V = linalg.s.svds(M_omega, r)
108         S = np.diag(s)
109         X = np.dot(U, S)
110         Y = V
111         itres = np.zeros((maxiter+1, 1))
112
113         XY = np.dot(X, Y)
114         diff_on_omega = M[Omega] - XY[Omega]

```

```

114     res = linalg.norm(diff_on_omega)
115     iter = 0
116     itres[iter] = res/frob_norm_data
117
118     while iter < maxiter and res >= relres:
119
120         # Gradient for X
121         diff_on_omega_matrix = np.zeros((m,n))
122         diff_on_omega_matrix[Omega] = diff_on_omega
123         grad_X = np.dot(diff_on_omega_matrix, np.transpose(Y))
124
125         # Stepsize for X
126         delta_XY = np.dot(grad_X, Y)
127         tx = linalg.norm(grad_X, 'fro')**2/linalg.norm(delta_XY)**2
128
129         # Update X
130         X = X + tx*grad_X;
131         diff_on_omega = diff_on_omega-tx*delta_XY[Omega]
132
133         # Gradient for Y
134         diff_on_omega_matrix = np.zeros((m,n))
135         diff_on_omega_matrix[Omega] = diff_on_omega
136         Xt = np.transpose(X)
137         grad_Y = np.dot(Xt, diff_on_omega_matrix)
138
139         # Stepsize for Y
140         delta_XY = np.dot(X, grad_Y)
141         ty = linalg.norm(grad_Y, 'fro')**2/linalg.norm(delta_XY)**2
142
143         # Update Y
144         Y = Y + ty*grad_Y
145         diff_on_omega = diff_on_omega-ty*delta_XY[Omega]
146
147         res = linalg.norm(diff_on_omega)
148         iter = iter + 1
149         itres[iter] = res/frob_norm_data
150
151     M_out = np.dot(X, Y)
152
153     out_info = [iter, itres]
154
155     return M_out, out_info
156
157 def _sASD(self, M, r = None, reltol=1e-5, maxiter=10000):
158     """
159     Scaled Alternating Steepest Descent (ScaledASD)
160     Taken from:
161     Low rank matrix completion by alternating steepest descent methods
162     Jared Tanner and Ke Wei
163     SIAM J. IMAGING SCIENCES (2014)
164

```

```

165 We have a matrix M with incomplete entries ,
166 and want to estimate the full matrix
167
168 Solves the following relaxation of the problem:
169 minimize_{X,Y} \frac{1}{2} || P_{\Omega}(Z^0) - P_{\Omega}(XY) ||_F^2
170 Where \Omega represents the set of m observed entries of the matrix M
171 and P_{\Omega}() is an operator that represents the observed data.
172
173 Inputs:
174 M := Incomplete matrix , with NaN on the unknown matrix
175 r := hypothesized rank of the matrix
176
177 Usage:
178 Just call the function _sASD(M)
179 """
180
181
182 # Get shape and Omega
183 m, n = M.shape
184 if r == None:
185     r = min(m, n, 50)
186
187 # Set relative error
188 Omega = ~np.isnan(M)
189 frob_norm_data = linalg.norm(M[Omega])
190 relres = reltol * frob_norm_data
191
192 # Initialize
193 identity = np.identity(r);
194 I, J = np.where(Omega)
195 M_omega = csc_matrix((M[Omega], (I, J)), shape=M.shape)
196 U, s, V = linalg.s.svds(M_omega, r)
197 S = np.diag(s)
198 X = np.dot(U, S)
199 Y = V
200 itres = np.zeros((maxiter+1, 1))
201
202 XY = np.dot(X, Y)
203 diff_on_omega = M[Omega] - XY[Omega]
204 res = linalg.norm(diff_on_omega)
205 iter = 0
206 itres[iter] = res/frob_norm_data
207
208 while iter < maxiter and res >= relres:
209
210     # Gradient for X
211     diff_on_omega_matrix = np.zeros((m,n))
212     diff_on_omega_matrix[Omega] = diff_on_omega
213     grad_X = np.dot(diff_on_omega_matrix, np.transpose(Y))
214
215     # Scaled gradient

```

```

216         scale = linalg.solve(np.dot(Y, np.transpose(Y)), identity)
217         dx = np.dot(grad_X, scale)
218
219         delta_XY = np.dot(dx, Y)
220         tx = np.trace(np.dot(np.transpose(dx), grad_X))/linalg.norm(
delta_XY[Omega])**2
221
222         # Update X
223         X = X + tx*dx
224         diff_on_omega = diff_on_omega-tx*delta_XY[Omega]
225
226         # Gradient for Y
227         diff_on_omega_matrix = np.zeros((m,n))
228         diff_on_omega_matrix[Omega] = diff_on_omega
229         Xt = np.transpose(X)
230         grad_Y = np.dot(Xt, diff_on_omega_matrix)
231
232         # Scaled gradient
233         scale = linalg.solve(np.dot(Xt, X), identity)
234         dy = np.dot(scale, grad_Y)
235
236         # Stepsize for Y
237         delta_XY = np.dot(X, dy)
238         ty = np.trace(np.dot(dy, np.transpose(grad_Y)))/linalg.norm(
delta_XY[Omega])**2
239
240         # Update Y
241         Y = Y + ty*dy
242         diff_on_omega = diff_on_omega-ty*delta_XY[Omega]
243
244         # Update iteration information
245         res = linalg.norm(diff_on_omega)
246         iter = iter + 1
247         itres[iter] = res/frob_norm_data
248
249         M_out = np.dot(X, Y)
250
251         out_info = [iter, itres]
252
253         return M_out, out_info
254
255     def complete_it(self, algo_name, r = None, reltol=1e-5, maxiter=5000):
256
257         """ Function to solve the optimization with the choosen algorithm
258
259         Input:
260         1) algo_name: Algorithm name (ASD, sASD, ect)
261         2) r: rank of the matrix if performing alternating algorithm
262         """
263         if algo_name == "ASD":
264             self._X, self._out_info = self._ASD(self._M, r, reltol, maxiter)

```

```

265         elif algo_name == "sASD":
266             self._X, self._out_info = self._sASD(self._M, r, reltol, maxiter)
267         else:
268             raise NameError("Algorithm name not recognized")
269
270 class MatrixCompletionBD:
271     """
272     A general class for matrix factorization via stochastic gradient descent
273
274     Class members
275     =====
276     file: three column file of user, item, and value to build models
277
278
279     Class methods
280     =====
281     train_sgd():= method to complete the matrix via sgd
282     shuffle_file():= method to 'psuedo' shuffle input file in chunks
283     file_split():= method to split input file into training and test set
284     save_model():= save user and items parameters to text file
285     validate_sgd():= validate sgd model on test set
286     build_matrix():= for smaller data build complete matrix in pandas df or
287     numpy matrix?
288     """
289
290     def __init__(self, file_path, delimiter='\t', *args, **kwargs):
291         """
292         Object constructor
293         Initialize Matrix Completion BD object
294         """
295         self._file = file_path
296         self._delimiter = '\t'
297         self._users = dict()
298         self._items = dict()
299
300     def shuffle_file(self, batch_size=50000):
301         """
302
303         Shuffle line of file for sgd method, improves performance/convergence
304
305         """
306         data = open(self._file)
307         temp_file=open('temp-shuffled.txt', 'w')
308         try:
309             temp=open('backup_data_file.txt')
310             temp.close()
311         except:
312             os.system('cp ' +self._file + ' backup_data_file.txt')
313
314         temp_array=[]

```

```

315         counter=0
316         for line in data:
317             counter+=1
318             temp_array.append(line)
319             if counter==batch_size :
320                 random.shuffle(temp_array)
321                 for entry in temp_array:
322                     temp_file.write(entry)
323                 temp_array=[]
324                 counter=0
325
326         if len(temp_array)>0:
327             random.shuffle(temp_array)
328             for entry in temp_array:
329                 temp_file.write(entry)
330
331         data.close()
332         temp_file.close()
333         system_string='mv temp.shuffled.txt ' + self._file
334         os.system(system_string)
335
336     def file_split(self, percent_train=.80, train_file='data_train.csv',
337                   test_file='data_test.csv'):
338         """
339         split input file randomly into training and test set for cross
340         validation
341         """
342         train=open(train_file, 'w')
343         test=open(test_file, 'w')
344         temp_file=open(self._file)
345         for line in temp_file:
346             if np.random.rand()<percent_train:
347                 train.write(line)
348             else:
349                 test.write(line)
350
351         train.close()
352         test.close()
353         print('test file written as ' + train_file)
354         print('test file written as ' + test_file)
355         temp_file.close()
356
357     def train_sgd(self, dimension=6, init_step_size=.01, min_step=1e-5, reitol
358                  =.05, rand_init_scalar=1, maxiter=100, batch_size_sgd=50000, shuffle=True,
359                  print_output=False):
360         init_time=time.time()
361         alpha=init_step_size
362         iteration=0

```

```

362         delta_err=1
363         new_mse=reltol+10
364         counter=0
365         ratings=[]
366
367         while iteration != maxiter and delta_err > reltol :
368
369             data=open( self._file )
370             total_err=[0]
371             if alpha>=min_step: alpha*=.3
372             else: alpha=min_step
373
374             for line in data:
375
376                 record=line[0:len(line)-1].split( self._delimiter )
377                 record[2]=float( record[2] )
378                 # format : user , movie,5-point-ratings
379                 ratings.append( record[2] )
380                 #if record[0] in self.users and record[1] in self.items :
381                 try:
382                     # do some updating
383                     # updates
384                     error=record[2]-np.dot( self._users[ record[0] ] , self._items[
record[1]] )
385                     self._users[ record[0] ] = self._users[ record[0] ] + alpha*2*
error*self._items[ record[1] ]
386                     self._items[ record[1] ] = self._items[ record[1] ] + alpha*2*
error*self._users[ record[0] ]
387                     total_err.append( error**2 )
388                 except:
389                     #else:
390                     counter+=1
391                     if record[0] not in self._users:
392                         self._users[ record[0] ] = np.random.rand( dimension ) *
rand_init_scalar
393                     if record[1] not in self._items:
394                         self._items[ record[1] ] = np.random.rand( dimension ) *
rand_init_scalar
395
396                 data.close()
397                 if shuffle:
398                     self.shuffle_file( batch_size=batch_size_sgd )
399                 iteration+=1
400                 old_mse=new_mse
401                 new_mse=sum( total_err ) * 1.0 / len( total_err )
402                 delta_err=abs( old_mse-new_mse )
403                 if print_output and iteration%10==0:
404                     print ( 'Delta Error: %f ' % delta_err )
405
406             #Printing Final Output
407             if print_output:

```

```

408         print ( 'Iterations: %f ' % iteration)
409         print ( 'MSE: %f ' % new_mse)
410         minutes=(time.time()-init_time)/60
411         print ( 'Total Minutes to Run: %f' % minutes)
412
413
414     def save_model(self , user_out='user_params.txt' , item_out='item_params.txt' )
415     :
416         """
417         save model user and item parameters to text file
418         user_key , user_vector entries
419         item_key , item_vector entries
420         """
421         users=open( user_out , 'w')
422         items=open( item_out , 'w')
423         for key in self._users:
424             user_string= key+ self._delimiter + self._delimiter.join( map(str
425             , list( self._users[key] ) ) ) + '\n'
426             users.write( user_string )
427
428         for key in self._items:
429             item_string=key+ self._delimiter + self._delimiter.join( map(str ,
430             list( self._items[key] ) ) ) + '\n'
431             items.write( item_string )
432
433         users.close()
434         items.close()
435
436     ## read saved model, particularly useful for fitting very large files!
437     def read_model( self , dimension=6, saved_user_params='user_params.txt' ,
438     saved_item_params='item_params.txt' ):
439         """
440         Read the saved user and item parameters from text files to the item
441         and user dictionaries
442         """
443         #populate users:
444         user_data=open( saved_user_params )
445         for line in user_data:
446             record=line[0: len( line ) -1].split( self._delimiter )
447             key=record.pop(0)
448             params=np. array( map( float , record ) )
449             self._users[ key ]=params
450
451         user_data.close()
452
453         #populate items:
454         item_data=open( saved_item_params )
455         for line in item_data:
456             record=line[0: len( line ) -1].split( self._delimiter )

```



```

454         key=record.pop(0)
455         params=np.array(map(float,record))
456         self._items[key]=params
457
458     item_data.close()
459
460     def clear_model(self):
461         """
462
463         clear the user and item parameters
464
465         """
466         del self._items, self._users
467         self._items=dict()
468         self._users=dict()
469
470     def validate_sgd(self, test_file_path):
471         """
472
473         run model on test/validation set, returns MSE
474
475         """
476         mse=[]
477         counter=0
478         test_set=open(test_file_path)
479         for line in test_set:
480             record=line[0:len(line)-1].split(self._delimiter)
481             record[2]=float(record[2])
482             try:
483                 error=record[2]-np.dot(self._users[record[0]], self._items[
record[1]])
484                 mse.append(error**2)
485             except:
486                 counter+=1
487
488             if counter>0: print('Items/Users Key Errors: %f ' % counter)
489             # returns Mean Squared Error
490             return sum(mse)/len(mse)
491
492     def build_matrix(self):
493         pass

```

completethat/matrix__init__.py

```

1 """ CompleteThat is a python package that solves the low rank matrix
   completion
2 problem. Given a low rank matrix with partial entries the package solves an
3 optimization problem to estimate the missing entries.

```

```

4
5 Mathematically, the package solves a relaxation (using the nuclear norm or the
6 Frobenius norm of the objective matrix) of the following problem:
7     minimize- $\{X\}$   $\|X\|$ 
8     st.  $X(i,j) = M(i,j) \setminus \text{forall } (i,j) \setminus \text{in } \Omega$ ,
9
10 Where,  $M$  represents the data matrix and  $\Omega$  represents the set of  $p$ 
    observed entries of  $M$ 
11
12 Usage:
13 # MatrixCompletion
14 >>> from completethat import MatrixCompletion
15 >>> problem = MatrixCompletion(M)
16 >>> problem.complete_it(algo_name)
17 >>> X = problem.get_matrix()
18 >>> out_info = problem.get_out() #Extra info (iterations, ect)
19
20 # MatrixCompletionBD
21 >>> from completethat import MatrixCompletionBD
22 >>> temp=MatrixCompletionBD('input_data.txt')
23 >>> temp.train_sgd(dimension=6,init_step_size=.01,min_step=.000001, reltol
    =.001,rand_init_scale=10, maxiter=1000,batch_size_sgd=50000,shuffle=True
    ):
24 >>> temp.validate_sgd('test_data.txt')
25 >>> temp.save_model()
26
27 """
28 from matrix_completion import MatrixCompletion
29 from matrix_completion import MatrixCompletionBD

```

setup.py

```

1 from setuptools import setup
2
3 def readme():
4     with open('README.rst') as f:
5         return f.read()
6
7 setup(
8     name='completethat',
9     version='0.1dev',
10    description='A package to solve low rank matrix completion problems',
11    long_description=readme(),
12    author='Joshua Edgerton, Esteban Fajardo',
13    author_email='ef2451@columbia.edu, jae2154@columbia.edu',
14    license='BSD',
15    packages=['completethat'],
16    install_requires=[

```

```

17         'scipy', 'numpy'
18     ],
19     classifiers=[
20         'Development Status :: 3 - Alpha',
21         'License :: OSI Approved :: BSD License',
22         'Programming Language :: Python :: 2.7',
23         'Topic :: Scientific/Engineering :: Mathematics',
24         'Topic :: Utilities'
25     ],
26     include_package_data=True,
27     zip_safe=False
28 )

```

README.rst

```

1 CompleteThat (v0.1 dev)
2 =====
3
4 CompleteThat is a python package that solves the low rank matrix completion
5 problem. Given a low rank matrix with partial entries the package solves an
6 optimization problem to estimate the missing entries.
7
8 Mathematically, the package solves a relaxation (using the nuclear norm or the
9 Frobenius norm of the objective matrix) of the following problem:
10
11     minimize_{X} ||X||
12     st. X(i,j) = M(i,j) \forall (i,j) \in \Omega,
13     where, M represents the data matrix and \Omega represents the set of p
14     observed entries of M
15
16 Usage
17 -----
18
19 >>> from completethat import MatrixCompletion
20 >>> problem = MatrixCompletion(M)
21 >>> problem.complete_it(algo_name)
22 >>> X = problem.get_matrix()
23 >>> out_info = problem.get_out()
24
25 >>> from completethat import MatrixCompletionBD
26 >>> problem = MatrixCompletionBD('input_data.txt')
27 >>> problem.train_sgd(dimension=6, init_step_size=.01, min_step=.000001, retol
    =.001, rand_init_scale=10, maxiter=1000, batch_size_sgd=50000, shuffle=True
    )
28 >>> problem.validate_sgd('test_data.txt')
29 >>> problem.save_model()
30
31 Authors

```

32

33

34 This package was written by Joshua Edgerton and Esteban Fajardo

35

36 Acknowledgments

37

38

39 This package is the result of the final project for the class EEOR E4650:
Convex

40 Optimization at Columbia University, Fall 2014. We would like to thank the

41 authors of the different algorithms used in the package to solve the problem.