`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 a memory-fitting 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

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

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

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 $P_\Omega : \mathbf{R}^{n_1 \times n_2} \to \mathbf{R}^{n_1 \times n_2}$ 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

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

Unfortunately, this optimization problem has been shown to be NP-hard and all know 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 exists. One, the "tightest convex relaxation of $\mathbf{rank}(X)$" [Faz02] is the following problem

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

where $\sigma_i(X)$ denotes the ith 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

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

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 $f : \mathbf{R}^{n_1 \times r} \times \mathbf{R}^{r \times n_2} \to \mathbf{R}$ be defined as

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

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] :

$$\nabla f_Z(Y) = -(P_\Omega(M) - P_\Omega(YZ))Z^T$$

$$\nabla f_Y(Z) = -Y^T(P_\Omega(M) - P_\Omega(YZ)).$$

$$t_y = \frac{\|\nabla f_Z(Y)\|_F^2}{\|P_\Omega(\nabla f_Z(Y)Z\|_F^2}$$

$$t_z = \frac{\|\nabla f_Y(Z)\|_F^2}{\|P_\Omega(Y\nabla f_Y(Z))\|_F^2}$$

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

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

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

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

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:

$\nabla f_{Z_i}(Y_i) = -(P_\Omega(M) - P_\Omega(Y_i Z_i))Z_i^T$

$d_{y_i} = -\nabla f_{Z_i}(Y_i)(Z_i Z_i^T)^{-1}, t_{y_i} = \frac{-\nabla f_{Z_i}(Y_i) \cdot d_{y_i}}{\|P_\Omega(d_{y_i} Z_i)\|_F^2}$

$Y_{i+1} = Y_i + t_{y_i} d_{y_i}$

$\nabla f_{Y_{i+1}}(Z_i) = -Y_{i+1}^T(P_\Omega(M) - P_\Omega(Y_{i+1} Z_i))$

$d_{z_i} = (Y_{i+1}^T Y_{i+1}^{-1} \nabla f_{Y_{i+1}}(Z_i), t_{z_i} = \frac{-\nabla f_{Y_{i+1}}(Z_i) \cdot d_{z_i}}{\|P_\Omega(Y_{i+1} d_{z_i})\|_F^2}$

$Z_{i+1} = Z_i + t_{z_i} d_{z_i}$

$i = i + 1$

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 language of linear algebra and the typical optimizations using gradient methods require access to all of the data at once for updating the parameters, since its dependent on calculating inverse of matrices or solving systems of linear equations. However, given the scale of data that is common in today's digital world, the need for more efficient and memory friendly algorithms arises.

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:

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

With stochastic gradient descent, a variation of the standard batch gradient optimization, instead of calculating the gradient for the user feature vector using all the item feature vectors that the user has rated as would be done usually, we approximate the user's gradient using the current item vector for the current rating and vice versa for updates of the item vectors.

Thus we are essentially exchanging the luxury of using less memory and computational resources for more time and iterations spent with stochastic gradient descent.[1]

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 algorithms performance is highly dependent on the step size (α) parameter of the update equations. Too small of a alpha will result in convergence taking a very long time and lower the chance of finding the global objective. Too large of a alpha 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

[1] We 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.

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, 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 pipe-delimited file consisting of roughly 210,000 user-movie ratings. The test set has roughly 10,000 records. There are 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 SGD algorithm on these two datasets comparing it 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.

The benchmark 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. A quick run of Mahout yields 1.17 and CompleteThat's sgd model yielded a 1.19. So our model yielded a very similar RMSE as the MAHOUT package. Note that for both of these instances we used a factorization of rank equal to seven and the Complete that sgd algorithm ran slightly faster than Mahout's (.537 minutes vs 1.4 minutes).

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 in 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 know 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 objetive function as the problems above (using the Frobenious 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.

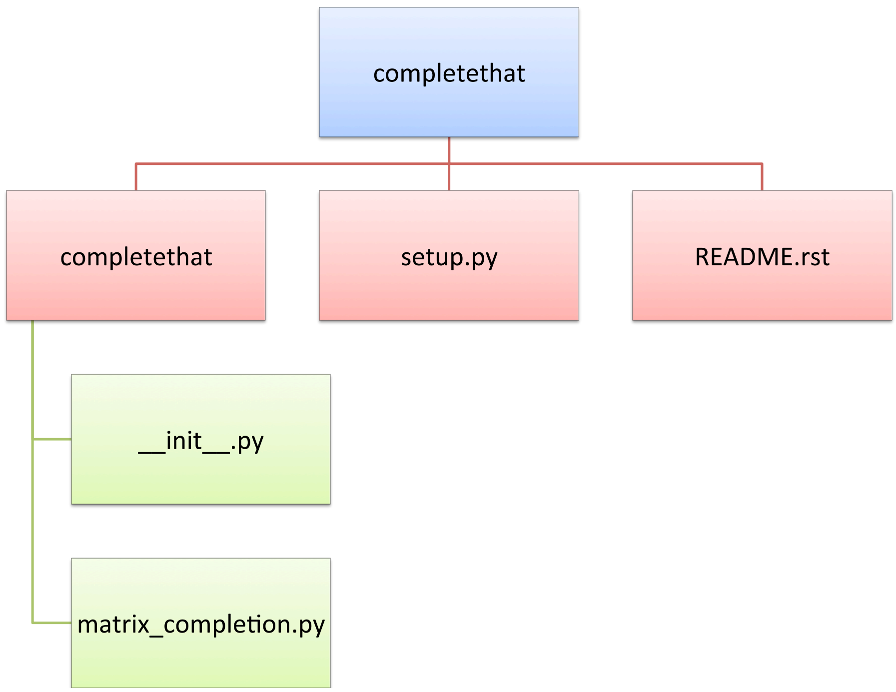

Figure 2: CompleteThat structure

---

completethat/matrix_completion.py

```
import numpy as np
from scipy.sparse import csc_matrix
from scipy.sparse import linalg as linalg_s
import os, random, time
from scipy import linalg

class MatrixCompletion:
    """ A general class to represent a matrix completion problem

    Data members
    ==================
    M:= data matrix (numpy array).
    X:= optimized data matrix (numpy array)
```

```
    out_info:= output information for the optimization (list)


    Class methods
    ====================
    complete_it():= method to complete the matrix
    get_optimized_matrix():= method to get the solution to the problem
    get_matrix():= method to get the original matrix
    get_out():= method to get extra information on the optimization (iter
    number, convergence, objective function)

    """
    def __init__(self, X,*args, **kwargs):
        """ Constructor for the problem instance

            Inputs:
             1) X: known data matrix. Numpy array with np.nan on the unknow entries.
                example:
                    X = np.random.randn(5, 5)
                    X[1][3] = np.nan
                    X[0][0] = np.nan
                    X[4][4] = np.nan

        """
        # Initialization of the members
        self._M = X
        self._X = np.array(X, copy = True) #Initialize with ini data matrix
        self._out_info = []

    def get_optimized_matrix(self):
        """ Getter function to return the optimized matrix X

            Ouput:
             1) Optimized matrix
        """
        return self._X

    def get_matrix(self):
        """ Getter function that returns the original matrix M

            Output:
            1) Original matrix M
        """
        return self._M

    def get_out(self):
        """ Getter function to return the output information
            of the optimization

            Output:
```

```
            1) List of length 2: number of iterations and relative residual

        """
        return self._out_info


    def _ASD(self, M, r = None, reltol=1e-5, maxiter=5000):
        """
        Alternating Steepest Descent (ASD)
        Taken from Low rank matrix completion by alternating steepest descent methods
        Jared Tanner and Ke Wei
        SIAM J. IMAGING SCIENCES (2014)

        We have a matrix M with incomplete entries,
        and want to estimate the full matrix

        Solves the following relaxation of the problem:
        minimize_{X,Y} \frac{1}{2} ||P_{\Omega}(Z^0) - P_\{Omega}(XY)||_F^2
        Where \Omega represents the set of m observed entries of the matrix M
        and P_{\Omega}() is an operator that represents the observed data.

        Inputs:
         M := Incomplete matrix, with NaN on the unknown matrix
         r := hypothesized rank of the matrix

        Usage:
         Just call the function _ASD(M)
        """

        # Get shape and Omega
        m, n = M.shape
        if r == None:
            r = min(m, n, 50)

        # Set relative error
        Omega = ~np.isnan(M)
        frob_norm_data = linalg.norm(M[Omega])
        relres = reltol * frob_norm_data

        # Initialize
        I, J = np.where(Omega)
        M_omega =  csc_matrix((M[Omega], (I, J)), shape=M.shape)
        U, s, V = linalg_s.svds(M_omega, r)
        S = np.diag(s)
        X = np.dot(U, S)
        Y = V
        itres = np.zeros((maxiter+1, 1))

        XY = np.dot(X, Y)
        diff_on_omega = M[Omega] - XY[Omega]
```

```
        res = linalg.norm(diff_on_omega)
        iter = 0
        itres[iter] = res/frob_norm_data

        while iter < maxiter and res >= relres:

            # Gradient for X
            diff_on_omega_matrix = np.zeros((m,n))
            diff_on_omega_matrix[Omega] = diff_on_omega
            grad_X = np.dot(diff_on_omega_matrix, np.transpose(Y))

            # Stepsize for X
            delta_XY = np.dot(grad_X, Y)
            tx = linalg.norm(grad_X,'fro')**2/linalg.norm(delta_XY)**2

            # Update X
            X = X + tx*grad_X;
            diff_on_omega = diff_on_omega-tx*delta_XY[Omega]

            # Gradient for Y
            diff_on_omega_matrix = np.zeros((m,n))
            diff_on_omega_matrix[Omega] = diff_on_omega
            Xt = np.transpose(X)
            grad_Y = np.dot(Xt, diff_on_omega_matrix)

            # Stepsize for Y
            delta_XY = np.dot(X, grad_Y)
            ty = linalg.norm(grad_Y,'fro')**2/linalg.norm(delta_XY)**2

            # Update Y
            Y = Y + ty*grad_Y
            diff_on_omega = diff_on_omega-ty*delta_XY[Omega]

            res = linalg.norm(diff_on_omega)
            iter = iter + 1
            itres[iter] = res/frob_norm_data

        M_out = np.dot(X, Y)

        out_info = [iter, itres]

        return M_out, out_info

    def _sASD(self, M, r = None, reltol=1e-5, maxiter=10000):
        """
        Scaled Alternating Steepest Descent (ScaledASD)
        Taken from:
        Low rank matrix completion by alternating steepest descent methods
        Jared Tanner and Ke Wei
        SIAM J. IMAGING SCIENCES (2014)

```

```
    We have a matrix M with incomplete entries,
    and want to estimate the full matrix

    Solves the following relaxation of the problem:
    minimize_{X,Y} \frac{1}{2} ||P_{\Omega}(Z^0) - P_\{Omega}(XY)||_F^2
    Where \Omega represents the set of m observed entries of the matrix M
    and P_{\Omega}() is an operator that represents the observed data.

    Inputs:
     M := Incomplete matrix, with NaN on the unknown matrix
     r := hypothesized rank of the matrix

    Usage:
     Just call the function _sASD(M)
    """



    # Get shape and Omega
    m, n = M.shape
    if r == None:
        r = min(m, n, 50)

    # Set relative error
    Omega = ~np.isnan(M)
    frob_norm_data = linalg.norm(M[Omega])
    relres = reltol * frob_norm_data

    # Initialize
    identity = np.identity(r);
    I, J = np.where(Omega)
    M_omega =  csc_matrix((M[Omega], (I, J)), shape=M.shape)
    U, s, V = linalg_s.svds(M_omega, r)
    S = np.diag(s)
    X = np.dot(U, S)
    Y = V
    itres = np.zeros((maxiter+1, 1))

    XY = np.dot(X, Y)
    diff_on_omega = M[Omega] - XY[Omega]
    res = linalg.norm(diff_on_omega)
    iter = 0
    itres[iter] = res/frob_norm_data

    while iter < maxiter and res >= relres:

        # Gradient for X
        diff_on_omega_matrix = np.zeros((m,n))
        diff_on_omega_matrix[Omega] = diff_on_omega
        grad_X = np.dot(diff_on_omega_matrix, np.transpose(Y))

        # Scaled gradient
```

```python
            scale = linalg.solve(np.dot(Y, np.transpose(Y)), identity)
            dx = np.dot(grad_X, scale)

            delta_XY = np.dot(dx, Y)
            tx = np.trace(np.dot(np.transpose(dx),grad_X))/linalg.norm(delta_XY[Omega])**2

            # Update X
            X = X + tx*dx
            diff_on_omega = diff_on_omega-tx*delta_XY[Omega]

            # Gradient for Y
            diff_on_omega_matrix = np.zeros((m,n))
            diff_on_omega_matrix[Omega] = diff_on_omega
            Xt = np.transpose(X)
            grad_Y = np.dot(Xt, diff_on_omega_matrix)

            # Scaled gradient
            scale = linalg.solve(np.dot(Xt, X), identity)
            dy = np.dot(scale, grad_Y)

            # Stepsize for Y
            delta_XY = np.dot(X, dy)
            ty = np.trace(np.dot(dy,np.transpose(grad_Y)))/linalg.norm(delta_XY[Omega])**2

            # Update Y
            Y = Y + ty*dy
            diff_on_omega = diff_on_omega-ty*delta_XY[Omega]

            # Update iteration information
            res = linalg.norm(diff_on_omega)
            iter = iter + 1
            itres[iter] = res/frob_norm_data

        M_out = np.dot(X, Y)

        out_info = [iter, itres]

        return M_out, out_info

    def complete_it(self, algo_name, r = None, reltol=1e-5, maxiter=5000):

        """ Function to solve the optimization with the choosen algorithm

            Input:
             1) algo_name: Algorithm name (ASD, sASD, ect)
             2) r: rank of the matrix if performing alternating algorithm
        """
        if algo_name == "ASD":
            self._X, self._out_info = self._ASD(self._M, r, reltol, maxiter)
```

```
        elif algo_name == "sASD":
            self._X, self._out_info = self._sASD(self._M, r, reltol, maxiter)
        else:
            raise NameError("Algorithm name not recognized")

class MatrixCompletionBD:
    """
    A general class for matrix factorization via stochastic gradient descent

    Class members
    ====================
    file: three column file of user, item, and value to build models


    Class methods
    ====================
    train_sgd():= method to complete the matrix via sgd
    shuffle_file():= method to 'psuedo' shuffle input file in chunks
    file_split():= method to split input file into training and test set
    save_model():= save user and items parameters to text file
    validate_sgd():= validate sgd model on test set
    build_matrix():= for smaller data build complete matrix in pandas df or numpy matrix?
    """


    def __init__(self,file_path,delimitter='\t',*args, **kwargs):
         """
             Object constructor
             Initialize Matrix Completion BD object
         """
         self._file = file_path
         self._delimitter = '\t'
         self._users = dict()
         self._items = dict()

    def shuffle_file(self,batch_size=50000):
        """

        Shuffle line of file for sgd method, improves performance/convergence

        """
        data = open(self._file)
        temp_file=open('temp_shuffled.txt','w')
        try:
            temp=open('backup_data_file.txt')
            temp.close()
        except:
            os.system('cp ' +self._file + ' backup_data_file.txt')

        temp_array=[]
```

```
        counter=0
        for line in data:
            counter+=1
            temp_array.append(line)
            if counter==batch_size :
                random.shuffle(temp_array)
                for entry in temp_array:
                    temp_file.write(entry)
                temp_array=[]
                counter=0

        if len(temp_array)>0:
            random.shuffle(temp_array)
            for entry in temp_array:
                temp_file.write(entry)

        data.close()
        temp_file.close()
        system_string='mv temp_shuffled.txt ' + self._file
        os.system(system_string)

    def file_split(self,percent_train=.80, train_file='data_train.csv', test_file='data_test.csv'):
        """

        split input file randomly into training and test set for cross validation

        """
        train=open(train_file,'w')
        test=open(test_file,'w')
        temp_file=open(self._file)
        for line in temp_file:
            if np.random.rand()<percent_train:
                train.write(line)
            else:
                test.write(line)

        train.close()
        test.close()
        print('test file written as ' + train_file)
        print('test file written as ' + test_file)
        temp_file.close()

    def train_sgd(self,dimension=6,init_step_size=.01,min_step=1e-5,reltol=.05,rand_init_scalar=1, maxiter=100,batch_size_sgd=50000,shuffle=True,print_output=False):

        init_time=time.time()
        alpha=init_step_size
        iteration=0
```

```
        delta_err=1
        new_mse=reltol+10
        counter=0
        ratings=[]

        while iteration != maxiter and delta_err > reltol :

            data=open(self._file)
            total_err=[0]
            if alpha>=min_step: alpha*=.3
            else: alpha=min_step

            for line in data:

                record=line[0:len(line)-1].split(self._delimitter)
                record[2]=float(record[2])
                # format : user, movie,5-point-ratings
                ratings.append(record[2])
                #if record[0] in self.users and record[1] in self.items :
                try:
                    # do some updating
                    # updates
                    error=record[2]-np.dot(self._users[record[0]],self._items[record[1]])
                    self._users[record[0]]=self._users[record[0]]+alpha*2*error*self._items[record[1]]
                    self._items[record[1]]=self._items[record[1]]+alpha*2*error*self._users[record[0]]
                    total_err.append(error**2)
                except:
                    #else:
                    counter+=1
                    if record[0] not in self._users:
                        self._users[record[0]]=np.random.rand(dimension)*rand_init_scalar
                    if record[1] not in self._items:
                        self._items[record[1]]=np.random.rand(dimension)*rand_init_scalar

            data.close()
            if shuffle:
                self.shuffle_file(batch_size=batch_size_sgd)
            iteration+=1
            old_mse=new_mse
            new_mse=sum(total_err)*1.0/len(total_err)
            delta_err=abs(old_mse-new_mse)
            if print_output and iteration%10==0:
                print ('Delta Error: %f ' % delta_err)

        #Printing Final Output
        if print_output:
```

```
            print ('Iterations: %f ' % iteration)
            print ('MSE: %f ' % new_mse)
            minutes=(time.time()-init_time)/60
            print ('Total Minutes to Run: %f' % minutes)


    def save_model(self,user_out='user_params.txt',item_out='item_params.txt'):
        """
        save model user and item parameters to text file
        user_key, user_vector entries
        item_key, item_vector entries
        """
        users=open(user_out,'w')
        items=open(item_out,'w')
        for key in self._users:
            user_string= key+ self._delimitter + self._delimitter.join(map(str,list(self._users[key]))) + '\n'
            users.write(user_string)

        for key in self._items:
            item_string=key+ self._delimitter + self._delimitter.join(map(str,list(self._items[key]))) + '\n'
            items.write(item_string)

        users.close()
        items.close()

    ## read saved model, particularly useful for fitting very large files!
    def read_model(self,dimension=6,saved_user_params='user_params.txt',saved_item_params='item_params.txt'):
        """

        Read the saved user and item parameters from text files to the item and user dictionaries

        """
        #populate users:
        user_data=open(saved_user_params)
        for line in user_data:
            record=line[0:len(line)-1].split(self._delimitter)
            key=record.pop(0)
            params=np.array(map(float,record))
            self._users[key]=params

        user_data.close()

        #populate items:
        item_data=open(saved_item_params)
        for line in item_data:
            record=line[0:len(line)-1].split(self._delimitter)
```

```
            key=record.pop(0)
            params=np.array(map(float,record))
            self._items[key]=params

        item_data.close()

    def clear_model(self):
        """

        clear the user and item parameters

        """
        del self._items, self._users
        self._items=dict()
        self._users=dict()

    def validate_sgd(self,test_file_path):
        """

        run model on test/validation set, returns MSE

        """
        mse=[]
        counter=0
        test_set=open(test_file_path)
        for line in test_set:
            record=line[0:len(line)-1].split(self._delimitter)
            record[2]=float(record[2])
            try:
                error=record[2]-np.dot(self._users[record[0]],self._items[record[1]])
                mse.append(error**2)
            except:
                counter+=1

        if counter>0: print('Items/Users Key Errors: %f ' % counter )
        # returns Mean Squared Error
        return sum(mse)/len(mse)

    def build_matrix(self):
        pass
```

---

completethat/matrix__init__.py

```
""" 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.
```

```

Mathematically, the package solves a relaxation (using the nuclear norm or the Frobenius norm of the objective matrix) of the following problem:
    minimize_{X} ||X||
    st. X(i,j) = M(i,j) \forall (i,j)\in \Omega,

Where, M represents the data matrix and \Omega represents the set of p observed entries of M

    Usage:
    # MatrixCompletion
    >>> from completethat import MatrixCompletion
    >>> problem = MatrixCompletion(M)
    >>> problem.complete_it(algo_name)
    >>> X = problem.get_matrix()
    >>> out_info = problem.get_out() #Extra info (iterations, ect)

    # MatrixCompletionBD
    >>> from completethat import MatrixCompletionBD
    >>> temp=MatrixCompletionBD('input_data.txt')
    >>> 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):
    >>> temp.validate_sgd('test_data.txt')
    >>> temp.save_model()

"""
from matrix_completion import MatrixCompletion
from matrix_completion import MatrixCompletionBD
```

---

setup.py

```
from setuptools import setup

def readme():
    with open('README.rst') as f:
        return f.read()

setup(
      name='completethat',
      version='0.1dev',
      description='A package to solve low rank matrix completion problems',
      long_description=readme(),
      author='Joshua Edgerton, Esteban Fajardo',
      author_email='ef2451@columbia.edu, jae2154@columbia.edu',
      license='BSD',
      packages=['completethat'],
      install_requires=[
```

```
            'scipy', 'numpy'
        ],
        classifiers=[
            'Development Status :: 3 - Alpha',
            'License :: OSI Approved :: BSD License',
            'Programming Language :: Python :: 2.7',
            'Topic :: Scientific/Engineering :: Mathematics',
            'Topic :: Utilities'
        ],
        include_package_data=True,
        zip_safe=False
)
```

README.rst

```
CompleteThat (v0.1dev)
======================

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.

Mathematically, the package solves a relaxation (using the nuclear norm or the
Frobenius norm of the objective matrix) of the following problem:

  minimize_{X} ||X||
  st. X(i,j) = M(i,j) \forall (i,j)\in \Omega,
  where, M represents the data matrix and \Omega represents the set of p
  observed entries of M

Usage
-----

>>> from completethat import MatrixCompletion
>>> problem = MatrixCompletion(M)
>>> problem.complete_it(algo_name)
>>> X = problem.get_matrix()
>>> out_info = problem.get_out()

>>> 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()

Authors
```

```
————

This package was written by Joshua Edgerton and Esteban Fajardo

Acknowledgments
————

This package is the result of the final project for the class EEOR E4650: Convex
Optimization at Columbia University, Fall 2014. We would like to thank the
authors of the different algorithms used in the package to solve the problem.
```