DATA 612: Project 3: Matrix Factorization

Derek G. Nokes

2019-06-25

Introduction

Matrix factorizations are ubiquitous in modern data science. Our need to convert big data into a smaller set of latent factors grows as the amount of data we collect increases. In this project, we provide a brief overview of singular value decomposition (SVD) and a set SVD-like matrix factorizations. We implement the simple power method to do SVD on a dense matrix, then we move to a recommender system application on a sparse dataset using probabilistic matrix factorization (with and without biases).

Theory: Singular Value Decomposition (i.e., Spectral Decomposition) and Related Matrix Factorizations

Singular Value Decomposition (SVD)

The singular value decomposition (SVD) of a matrix A with m rows and n columns is

$$A = U\Sigma V^T$$

where

A = $m \times n$ input data matrix ($A \in \mathbb{R}^{m \times n}$)

r = rank of matrix A (i.e., the number of linearly independent columns of A)

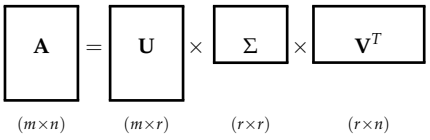
U = $m \times r$ matrix of left singular vectors

Σ = $r \times r$ diagonal matrix

V = $n \times r$ matrix of right singular vectors. The superscript T indicates the transpose of matrix V.

Rank r is the number of linearly independent columns of A. In most applications, the rank is equal to the number of columns (i.e., $r = n$).

If A has rank r, then U is $m \times r$, Σ is an $r \times r$ diagonal matrix with non-negative, non-increasing entries (i.e., sorted from largest to smallest), V is $n \times r$ and V^T is $r \times n$.



It is always possible to decompose a *real* matrix A into $U\Sigma V^T$ where U, Σ, and V are unique, U and V are column orthonormal, and Σ is a diagonal matrix where the singular values found along the diagonal are positive and sorted in decreasing order (i.e., $\sigma_1 \geq \sigma_2 \geq \cdots \geq 0$).

By *column orthonormal*, we mean that $U^T U = I$ and $V^T V = I$, where I is an identity matrix.

$$A_{[m \times n]} = U_{[m \times r]} \Sigma_{[r \times r]} \left(V_{[n \times r]} \right)^T$$

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1m} \\ u_{21} & u_{22} & \dots & u_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ u_{r1} & u_{r2} & \dots & u_{rm} \end{bmatrix} \times \begin{bmatrix} \sigma_{11} & 0 & \dots & 0 \\ 0 & \sigma_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_{rr} \end{bmatrix} \times \begin{bmatrix} v_{11} & v_{12} & \dots & v_{1n} \\ v_{21} & v_{22} & \dots & v_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{r1} & v_{r2} & \dots & v_{rn} \end{bmatrix}^T$$

Probabilistic Matrix Factorization

Probabilistic matrix factorization (**?**) is very similar to singular value decomposition. We define the probabilistic matrix factorization as follows:

$$A = Q^T P$$

where predictions $\hat{a}_{ui}$ for user u for item i, are:

$$\hat{a}_{ui} = q_i^T p_u$$

where q_i and p_u are the latent factors for items and users respectively.

Unlike the classical SVD, this matrix factorization can be applied to sparse matrices. We minimize the regularized squared error using stochastic gradient descent to estimate all of the unknowns as follows:

$$\sum_{a_{ui} \in A_{train}} (a_{ui} - \hat{a}_{ui})^2 + \lambda \left(||q_i||^2 + ||p_u||^2 \right)$$

where $a_{ui} - \hat{a}_{ui}$ is the error and λ is the regularization parameter.

Extending Probabilistic Matrix Factorization

The probabilistic matrix factorization can be extended to include user and item specific biases (b_u and b_i respectively). See (**?**), (**?**), and (**?**) for additional details.

The lower-dimensional approximation $\hat{a}_{ui}$ for a_{ui} is:

$$\hat{a}_{ui} = \mu + b_u + b_i + q_i^T p_u$$

The bias b_u and the latent factors p_u are assumed to be zero if user u is new. Similarly, if an item i is new, the latent factors p_i and bias b_i are assumed to be zero.

where q_i and p_u are again the latent factors for items and users.

The regularized squared error is minimized to find all of the unknowns as follows:

$$\sum_{a_{ui} \in A_{train}} (a_{ui} - \hat{a}_{ui})^2 + \lambda \left(b_i^2 + b_u^2 + ||q_i||^2 + ||p_u||^2 \right)$$

The minimization is performed over all the elements (a_{ui}) of the matrix A using stochastic gradient descent (SGD)[1]:

$b_u \leftarrow b_u + \gamma((a_{ui} - \hat{a}_{ui}) - \lambda b_u)$

$b_i \leftarrow b_i + \gamma((a_{ui} - \hat{a}_{ui}) - \lambda b_i)$

[1] These steps are performed `n_epochs` times. The baselines (i.e., b_u and b_i) are initialized to 0. To initialize the latent user and item factors we draw from a normal distribution.

$p_u \leftarrow p_u + \gamma((a_{ui} - \hat{a}_{ui}) \cdot q_i - \lambda p_u)$
$q_i \leftarrow q_i + \gamma((a_{ui} - \hat{a}_{ui}) \cdot p_u - \lambda q_i)$
where γ is the learning rate.

Importance of Normalization

To ensure that properties are compared in a way consistent with comparisons in the real world, the magnitudes of attribute values comprising the input data matrix should be scaled into roughly the same range. It is common to divide entries in each column of a dense matrix by the standard deviation of that column.

It may be more appropriate to normalize by keeping zero entries fixed when a matrix is sparse. The mean of non-zero entries is typically subtracted from the non-zero entries so that they become zero-centered. Only the non-zero entries are divided by the standard deviation of the column mean[2]. We must carefully consider the form of normalization because selecting the correct approach can significantly improve predictive performance.

[2] Normalization should not be used if zero values have some special significance.

Implementation

Modern libraries implementing SVD are based on extensive research sing methods more complex than the method implemented in the next section. In the following section, we use the simplest approach for illustrative purposes. There are more efficient and more numerically stable methods available.

Power Method

In the following sub-section, we implement singular value decomposition (SVD) in Python using the power method.

Briefly, the power method for singular value decomposition of a matrix $A \in \mathbb{R}^{m \times n}$ works as follows [cite]:

The algorithm starts by computing the first singular value σ_1 and the left and right singular vectors u_1 and v_1 of A, for which $\min_{i>j} \log(\sigma_i / \sigma_j) \geq \lambda$

1. Generate x_0 such that $x_0(i) \sim$ N(0,1)
2. for i in $[1, \ldots, k]$ where $k = \min(m, n)$:
3. $x_i \leftarrow A^T \leftarrow A x_{xi-1}$
4. $v_1 \leftarrow x_i / \|x_i\|$
5. $\sigma_1 \leftarrow \|Av_1\|$
6. $u_1 \leftarrow Av_1 / \sigma 1$
7. return (σ_1, u_1, v_1)

We perform this process for each

First, we define a function to create a random unit vector:

```
def randomUnitVector(n):
    unnormalized = [random.normalvariate(0, 1) for _ in range(n)]
    theNorm = sqrt(sum(x * x for x in unnormalized))
    return [x / theNorm for x in unnormalized]
```

Next, we define a function to compute the one-dimensional singular value decomposition (SVD):

```
def svdPowerMethod_1d(A, epsilon=1e-10):
    # The one-dimensional singular value decomposition (SVD)
    # determine number of rows and columns of A
    m, n = A.shape
    # compute random unit vector
    x = randomUnitVector(min(m,n))
    # initalize previous V to None
    lastV = None
    # initialize V with random unit vector
    V = x
    # if number of rows greater than number of columns
    if m > n:
        # take dot product of A transpose and A
        B = np.dot(A.T, A)
    # if number of rows is equal to or less than number
    # of columns
    else:
        # take dot product of A and A transpose
        B = np.dot(A, A.T)
    # initialize number of iterations to 0
    iterations = 0
    # while
    while True:
        # increment number of iterations
        iterations += 1
        # remember last V
        lastV = V
        # compute new V
        V = np.dot(B, lastV)
        # normalize V by norm of V
        V = V / norm(V)
        # check for convergence based on error threshold
        if abs(np.dot(V, lastV)) > 1 - epsilon:
            # print number of iterations it took to converge
```

```
            print("Converged in "+str(iterations))
            # return V
            return V
```

Now, we define a function to compute the singular value decomposition (SVD) of a matrix, A, using the power method:

```
def svdPowerMethod(A, k=None, epsilon=1e-10):
    # Compute singular value decomposition (SVD) of matrix A
    # using the power method. If k is None, compute full-rank
    # decomposition
    #
    # A = input matrix [m x n]
    # k = number of singular values to use

    # create float array
    A = np.array(A, dtype=float)
    # find n rows and m columns of A
    m, n = A.shape
    # create
    svdEstimate = []
    # if k is not provided compute full-rank decomposition
    if k is None:
        # determine full-rank k
        k = min(m, n)
    # iterate from 0 to k
    for i in range(k):
        # make copy of matrix
        matrixFor1D = A.copy()
        # extract the singular value, U, and V
        for singularValue, u, v in svdEstimate[:i]:
            #
            matrixFor1D -= singularValue * np.outer(u, v)
        # number of rows is greater than number of columns
        if m > n:
            # compute next singular vector (v)
            v = svdPowerMethod_1d(matrixFor1D,
              epsilon=epsilon)
            # comput u from v
            u_unnormalized = np.dot(A, v)
            # next singular value
            # compute norm of unnormalized u
            sigma = norm(u_unnormalized)
            # normalize u by singular value
```

```
            u = u_unnormalized / sigma
        else:
            # next singular vector
            u = svdPowerMethod_1d(matrixFor1D,
              epsilon=epsilon)
            # take dot product of A transpose and u
            v_unnormalized = np.dot(A.T, u)
            # next singular value
            # compute norm of unnormalized v
            sigma = norm(v_unnormalized)
            # normalize v by singular value
            v = v_unnormalized / sigma
        # store singular values, u, v
        svdEstimate.append((sigma, u, v))
    # extract singular values, u, v
    output = [np.array(x) for x in zip(*svdEstimate)]
    singularValues, us, vs = output
    # return SVD result
    return singularValues, us.T, vs
```

Validation

In this section, we validate the results of our implementation.

First we define a matrix A:

```
# create sample matrix
A = np.array([
  [4, 1, 1],
  [2, 5, 3],
  [1, 2, 1],
  [4, 5, 5],
  [3, 5, 2],
  [2, 4, 2],
  [5, 3, 1],
  [2, 2, 5],
  ], dtype='float64')
# determine n rows and m columns
m,n = A.shape
```

Create a random unit vector:

```
# set random seed
random.seed(randomSeed)
np.random.seed(randomSeed)
```

```
# build random unit vector
x = randomUnitVector(min(m,n))
```

Let's have a look at the output of the `randomUnitVector` function:

```
## [0.8037039719066417, -0.48587519802452295, 0.34348976329159925]
```

To compute the full-rank decomposition of the 8 by 3 matrix A we call the `svdPowerMethod(A)` function:

```
# set random seed
random.seed(randomSeed)
np.random.seed(randomSeed)
# compute SVD
singularValues,U,V = svdPowerMethod(A)
```

```
## Converged in 6
## Converged in 24
## Converged in 2
```

The 8 by 3 matrix of left singular vectors U:

```
print(U)
```

```
## [[ 0.22155208 -0.52086662 -0.39333697]
##  [ 0.39458523  0.23924155  0.35445363]
##  [ 0.15830232  0.03055162  0.15299691]
##  [ 0.53347447  0.19168664 -0.19949776]
##  [ 0.39692635 -0.08648347  0.41053093]
##  [ 0.31660463  0.06110323  0.30599382]
##  [ 0.34630265 -0.64128879 -0.07381356]
##  [ 0.32840218  0.45969551 -0.62355828]]
```

The 3 by 3 diagonal singular value matrix Σ:

```
print(np.diag(singularValues))
```

```
## [[ 15.09626916   0.           0.        ]
##  [  0.           4.30056855   0.        ]
##  [  0.           0.           3.40701739]]
```

The 3 by 3 matrix of right singular vectors V:

```
print(V)
```

```
## [[ 0.54184843  0.67070986  0.50650623]
##  [-0.75152966  0.11682294  0.6492731 ]
##  [-0.37630233  0.73246207 -0.56735868]]
```

We reconstitute the matrix A:

```
# reconstitute matrix A
Sigma = np.diag(singularValues)
# reconstitute matrix A
AA=np.dot(U, np.dot(Sigma, V))
```

```
print(AA)
```

```
## [[ 4.  1.  1.]
##  [ 2.  5.  3.]
##  [ 1.  2.  1.]
##  [ 4.  5.  5.]
##  [ 3.  5.  2.]
##  [ 2.  4.  2.]
##  [ 5.  3.  1.]
##  [ 2.  2.  5.]]
```

```
# define number of digits for rounding
nDigits=10
```

We can see that the original and reconstituted matrices are the same to 10 decimal places:

```
print(np.round(A - AA, decimals=nDigits))
```

```
## [[ 0.  0.  0.]
##  [ 0.  0.  0.]
##  [ 0.  0.  0.]
##  [ 0.  0.  0.]
##  [ 0.  0.  0.]
##  [ 0.  0.  0.]
##  [ 0.  0.  0.]
##  [ 0.  0.  0.]]
```

We check our own implement of SVD against the `numpy.linalg.svd()` implementation.

```
from numpy.linalg import svd
# set random seed
random.seed(randomSeed)
np.random.seed(randomSeed)
U_np,singularValues_np,V_np = svd(A,full_matrices=False)
```

U:

```
print(U_np)
```

```
## [[-0.22155201 -0.52086121  0.39334917]
##  [-0.39458526  0.23923575 -0.35445911]
##  [-0.15830232  0.03054913 -0.15299759]
##  [-0.53347449  0.19168874  0.19949342]
##  [-0.39692635 -0.08649009 -0.41052882]
##  [-0.31660464  0.06109826 -0.30599517]
##  [-0.34630257 -0.64128825  0.07382859]
##  [-0.32840223  0.45970413  0.62354764]]
```

Σ:

```
print(np.diag(singularValues_np))
```

```
## [[ 15.09626916   0.           0.        ]
##  [  0.           4.30056855   0.        ]
##  [  0.           0.           3.40701739]]
```

The singular values from `numpy.linalg.svd()` match those produced by our own function.

V:

```
print(V_np)
```

```
## [[-0.54184808 -0.67070995 -0.50650649]
##  [-0.75152295  0.11680911  0.64928336]
##  [ 0.37631623 -0.73246419  0.56734672]]
```

We again reconstitute the matrix A - this time using the output from `numpy.linalg.svd()` - and display the differences between the original and reconstituted matrices:

```
# reconstitute matrix A
Sigma_np = np.diag(singularValues_np)
# reconstitute matrix A
AA_np=np.dot(U_np, np.dot(Sigma_np, V_np))
# display difference
print(np.round(A - AA_np, decimals=nDigits))
```

```
## [[ 0.  0.  0.]
##  [ 0.  0.  0.]
##  [ 0.  0.  0.]
##  [ 0.  0.  0.]
##  [ 0.  0.  0.]]
```

```
##  [ 0.  0.  0.]
##  [ 0.  0.  0.]
##  [ 0.  0.  0.]]
```

```
print(np.round(U-U_np, decimals=2))
```

```
## [[ 0.44  0.   -0.79]
##  [ 0.79  0.    0.71]
##  [ 0.32  0.    0.31]
##  [ 1.07  0.   -0.4 ]
##  [ 0.79  0.    0.82]
##  [ 0.63  0.    0.61]
##  [ 0.69  0.   -0.15]
##  [ 0.66  0.   -1.25]]
```

```
print(np.round(Sigma-Sigma_np, decimals=2))
```

```
## [[ 0.  0.  0.]
##  [ 0.  0.  0.]
##  [ 0.  0.  0.]]
```

As expected, the singular values are almost identical.

```
print(np.round(V-V_np, decimals=2))
```

```
## [[ 1.08  1.34  1.01]
##  [ 0.    0.    0.  ]
##  [-0.75  1.46 -1.13]]
```

We do not expect the left and right singular vectors to be the same because for these the solution is not unique. However, both implementations can be used to reconstitute the original matrix as expected.

Thus far, we have shown results from the full-rank decomposition. In many applications of SVD - including those involving recommender systems - our objective is to map a sparse high- dimensional space to a dense low-dimensional space. For example, if we have a large user-by-item explicit ratings matrix, we may want to create a more compact representation of that ratings space using linearly independent latent factors. We can think of the latent factors as an abstraction that allows us to represent each user and item by a linear combination of other users and items. Classical SVD is one approach to solve for such latent factors, but unfortunately it can only be applied to a dense matrix.

Using independent latent factors often simplifies the interpretation of large datasets because it drastically reduces the number of interacting variables.

Using the same toy dataset as above, we can illustrate these concepts more concretely.

We perform the matrix decomposition for the largest k singular values, then reconstitute the matrix using only the associated k latent user and item factors:

```
# set random seed
random.seed(randomSeed)
np.random.seed(randomSeed)
# compute SVD
kSingularValues,kU,kV = svdPowerMethod(A,k=2)
```

```
## Converged in 6
## Converged in 24
```

We reconstitute the approximate matrix $\hat{A}$ using k factors (in this case, just two user factors and two item factors):

```
print(kU.shape)
```

```
## (8L, 2L)
```

```
print(kSingularValues.shape)
```

```
## (2L,)
```

```
print(kV.shape)
```

```
## (2L, 3L)
```

```
# reconstitute matrix A
kSigma = np.diag(kSingularValues)
# reconstitute matrix A
kAA=np.dot(kU, np.dot(kSigma, kV))
```

We can compare the original matrix A and the two-factor approximation to that matrix $\hat{A}$:

```
print(AA)
```

```
## [[ 4.  1.  1.]
##  [ 2.  5.  3.]
##  [ 1.  2.  1.]
##  [ 4.  5.  5.]
##  [ 3.  5.  2.]
##  [ 2.  4.  2.]
##  [ 5.  3.  1.]
##  [ 2.  2.  5.]]
```

```
print(kAA)
```

```
## [[ 3.49571503  1.98157675  0.23967928]
##  [ 2.45443385  4.11545707  3.68515918]
##  [ 1.19615253  1.61819452  1.29574317]
##  [ 3.74423019  5.49784886  4.61437065]
##  [ 3.5263288   3.97551553  2.79355666]
##  [ 2.39230506  3.23638904  2.59148633]
##  [ 4.90536595  3.18420256  0.85731832]
##  [ 1.20055553  3.55609656  3.79466129]]
```

Despite the use of an extremely small dataset, the two-factor approximation looks reasonable.

Limitations

Before we move on to an application of matrix factorization in a recommender system, we need to consider the limitations of our simple implementation of SVD. This approach requires that our input matrix A is dense with no missing elements. In many applications, we do not have any missing data. Unfortunately, for the application outlined in the next section, we are working with very sparse data, namely movie ratings data. In this application, we have many users that have not rated specific movies.

To address the issue with missing data, we can either impute the missing data, or we can use an approach that does not require the user-by-item ratings matrix to be dense. There are many ways to impute missing data, but this approach is not our preferred method for our application because it often leads to biased results. In fact, our objective is to infer the missing values in a way that improves the accuracy of our ratings predictions. Instead, in the next section, we generate recommendations using a third-party library implementation of an SVD-inspired matrix factorization approach that rolls Σ into U and V and solves for the latent factors using stochastic gradient descent (SGD). This approach - albeit more complex than the simpler methods for imputing missing data (e.g., filling missing values with zeros) - tends to lead to better results in practice.

Applications: Recommender Systems

Now that we have validated our simple implementation of singular value decomposition, we move to an application of on SVD-like matrix factorization to making movie recommendations based on a sparse user-by-movie ratings matrix. Although implementations of these more advanced methods are not overly complex, they are beyond the scope of this project.

Ratings

In the last project, we created our own functions to download the movielens dataset (**?**). We re-use these functions to acquire the required data, then we explore a matrix factorization inspired by SVD in the following sub-sections.

```
# import packages
import requests
import zipfile
import pandas as pd
from collections import defaultdict
# import from Surprise package
from surprise import Reader, Dataset, SVD, accuracy
from surprise.model_selection import cross_validate
from surprise.model_selection import GridSearchCV
from surprise.model_selection import train_test_split
# import python packages
import numpy as np
from numpy.linalg import norm
import random
from math import sqrt
# set random seed
randomSeed = 12345678
```

We define a function to download the movielens data:

```
# function to download movielens data
def download(download_url, download_path):
    req = requests.get(download_url, stream=True)
    with open(download_path, 'wb') as fd:
        for chunk in req.iter_content(chunk_size=2**20):
            fd.write(chunk)
```

We download the data:

```
# define downlad file name
downloadFile='ml-100k.zip'
# define download path
#download_path='/projects/ms/github/DATA_612/Project_2/ml-100k.zip'
download_path='F:/Dropbox/projects/ms/github/DATA_612/Project_3/data/ml-100k.zip'
# define download URL
download_url='http://files.grouplens.org/datasets/movielens/ml-100k.zip'
# dowload movielens data
download(download_url, download_path)
```

We extract the data from the zip file and read it into memory:

```
# define unzip path
unzip_path='F:/Dropbox/projects/ms/github/DATA_612/Project_3/data/'
# unzip movielens data
zf = zipfile.ZipFile(download_path)
# extract user by item file
user_by_item_file=zf.extract('ml-100k/u.data', unzip_path)
# extract title by item file
title_by_item_file=zf.extract('ml-100k/u.item', unzip_path)
# read user by item
df_user_by_item = pd.read_csv(user_by_item_file, sep='\t',
    header=None,names=['user_id','item_id','rating',
    'titmestamp'])
# read title by item
df_title_by_item = pd.read_csv(title_by_item_file, sep='|',
    encoding='latin-1',header=None,usecols=[0,1],
    names=['item_id','title'])
# join ratings by user and item to title
df = pd.merge(df_title_by_item,df_user_by_item,
  on='item_id')
```

We create the data object required for the modeling below as follows:

```
# load reader library
reader = Reader()
# load ratings dataset with Dataset library
data = Dataset.load_from_df(df[['user_id', 'item_id',
  'rating']], reader)
```

We set the random seed to ensure that our results are comparable across models and parameters and perform 5-fold cross validation using default parameters.

```
random.seed(randomSeed)
np.random.seed(randomSeed)
# select SVD algorithm
algo = SVD()
# compute RMSE and MAE of SVD algorithm using 5-fold cross
# validation
result=cross_validate(algo, data, measures=['RMSE', 'MAE'],
    cv=5, verbose=True,return_train_measures=True)
# extract mean test RMSE
```

```
## Evaluating RMSE, MAE of algorithm SVD on 5 split(s).
##
##                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
```

```
## MAE (testset)     0.7393  0.7408  0.7374  0.7367  0.7383  0.7385  0.0015
## RMSE (testset)    0.9372  0.9367  0.9382  0.9338  0.9358  0.9363  0.0015
## MAE (trainset)    0.5435  0.5407  0.5447  0.5425  0.5424  0.5428  0.0013
## RMSE (trainset)   0.6857  0.6826  0.6863  0.6841  0.6846  0.6847  0.0013
## Fit time          3.53    3.41    3.68    3.51    3.53    3.53    0.09
## Test time         0.12    0.13    0.12    0.16    0.12    0.13    0.02
```

```
meanTestRMSE=result[ 'test_rmse'].mean()
```

Now we define a function to return the top-N recommendations for each user from a set of predictions:

```
def get_top_n(predictions, n=10):
    # return top-N recommendations for each user from
    # a set of predictions.
    #
    # predictions(list of Prediction objects): list of
    #   predictions, as returned by test method of an
    #   algorithm.
    # n(int): number of recommendation to output for each
    #   user
    # First map predictions to each user
    top_n = defaultdict(list)
    for uid, iid, true_r, est, _ in predictions:
        top_n[uid].append((iid, est))
    # sort predictions for each user and retrieve
    # k highest ones
    for uid, user_ratings in top_n.items():
        user_ratings.sort(key=lambda x: x[1],
          reverse=True)
        top_n[uid] = user_ratings[:n]
    # return dict where keys are user (raw) ids and
    # values are lists of tuples:
    # [(raw item id, rating estimation), ...] of size n.
    return top_n
```

We perform a grid search to see if parameter tuning has a large impact on the cross-validation results.

```
random.seed(randomSeed)
np.random.seed(randomSeed)
# select best algo with grid search
print('Grid Search...')
```

```
## Grid Search...
```

```
param_grid = {'n_epochs': [10, 20, 30],
    'lr_all': [0.002, 0.005, 0.01,0.015,0.02,0.05],
    'biased' : [True,False]}
grid_search = GridSearchCV(SVD, param_grid, measures=['rmse'],
  cv=5)
random.seed(randomSeed)
np.random.seed(randomSeed)
# fit using best model
grid_search.fit(data)
# use best from grid search
algo = grid_search.best_estimator['rmse']
bestRMSE = grid_search.best_score['rmse']
```

Based on the grid search performance, our best mean RMSE is 0.9349 across all 5-folds as compared to an RMSE of 0.9363 using the default setting.

```
# create dataframe
results_df = pd.DataFrame.from_dict(grid_search.cv_results)
# extract required columns
results_table=results_df[['param_biased','param_lr_all',
    'param_n_epochs','mean_test_rmse','split0_test_rmse',
    'split1_test_rmse','split2_test_rmse','split3_test_rmse',
    'split4_test_rmse']].sort_values(by=['mean_test_rmse'])
columnMap={'param_biased' : 'biased',
    'param_lr_all' : 'lr_all',
    'param_n_epochs' : 'n_epochs',
    'mean_test_rmse' : 'Mean',
    'split0_test_rmse' : 'Fold 1',
    'split1_test_rmse' : 'Fold 2',
    'split2_test_rmse' : 'Fold 3',
    'split3_test_rmse' : 'Fold 4',
    'split4_test_rmse' : 'Fold 5'}
results_table.rename(columns=columnMap,inplace=True)
```

Table 1: Cross Validation Grid Search Results

	biased	lr_all	n_epochs	Mean	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
12	TRUE	0.010	10	0.9348750	0.9368453	0.9326648	0.9400571	0.9343072	0.9305008
7	TRUE	0.005	20	0.9355820	0.9382398	0.9349163	0.9392720	0.9336113	0.9318704
18	TRUE	0.015	10	0.9426034	0.9438945	0.9418240	0.9465407	0.9417196	0.9390383
8	TRUE	0.005	30	0.9433269	0.9434149	0.9402553	0.9494798	0.9421754	0.9413088
2	TRUE	0.002	30	0.9442025	0.9447973	0.9437270	0.9478009	0.9423920	0.9422954
6	TRUE	0.005	10	0.9475390	0.9469244	0.9455973	0.9535895	0.9460974	0.9454863

Table 1: Cross Validation Grid Search Results *(continued)*

	biased	lr_all	n_epochs	Mean	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
10	FALSE	0.005	20	0.9497297	0.9531797	0.9451862	0.9582560	0.9430730	0.9489537
1	TRUE	0.002	20	0.9499354	0.9499536	0.9466126	0.9561723	0.9484599	0.9484785
15	FALSE	0.010	10	0.9523057	0.9548259	0.9494948	0.9591440	0.9475622	0.9505014
24	TRUE	0.020	10	0.9534087	0.9555793	0.9539409	0.9583536	0.9503949	0.9487747
13	TRUE	0.010	20	0.9534627	0.9545080	0.9493247	0.9585864	0.9516861	0.9532082
11	FALSE	0.005	30	0.9540219	0.9558355	0.9467989	0.9628424	0.9539161	0.9507166
21	FALSE	0.015	10	0.9587496	0.9578953	0.9524090	0.9656512	0.9565441	0.9612485
0	TRUE	0.002	10	0.9629259	0.9626848	0.9621614	0.9684718	0.9608468	0.9604649
5	FALSE	0.002	30	0.9669861	0.9686213	0.9601490	0.9753893	0.9639826	0.9667883
16	FALSE	0.010	20	0.9687109	0.9730970	0.9653290	0.9753331	0.9644957	0.9652996
19	TRUE	0.015	20	0.9712050	0.9700949	0.9659577	0.9817724	0.9709792	0.9672207
14	TRUE	0.010	30	0.9724613	0.9739662	0.9726431	0.9772661	0.9735757	0.9648556
27	FALSE	0.020	10	0.9740866	0.9760116	0.9697793	0.9851365	0.9681277	0.9713776
25	TRUE	0.020	20	0.9786535	0.9818809	0.9750265	0.9846828	0.9755748	0.9761024
9	FALSE	0.005	10	0.9789184	0.9823198	0.9722576	0.9847271	0.9761824	0.9791053
20	TRUE	0.015	30	0.9790591	0.9840390	0.9760283	0.9813419	0.9791114	0.9747750
32	TRUE	0.050	30	0.9820579	0.9851533	0.9797947	0.9864946	0.9808701	0.9779767
26	TRUE	0.020	30	0.9824204	0.9899788	0.9807815	0.9844189	0.9818474	0.9750753
17	FALSE	0.010	30	0.9838530	0.9909788	0.9766307	0.9896107	0.9817425	0.9803023
22	FALSE	0.015	20	0.9844030	0.9868656	0.9795131	0.9917337	0.9824823	0.9814202
31	TRUE	0.050	20	0.9863426	0.9916190	0.9839269	0.9911838	0.9816248	0.9833584
30	TRUE	0.050	10	0.9900026	0.9939988	0.9843486	0.9972007	0.9830999	0.9913648
23	FALSE	0.015	30	0.9903610	0.9892473	0.9874562	0.9956821	0.9882252	0.9911940
28	FALSE	0.020	20	0.9964822	0.9976153	0.9880864	1.0047535	0.9944382	0.9975175
29	FALSE	0.020	30	0.9967369	1.0002384	0.9907719	1.0057679	0.9907449	0.9961614
4	FALSE	0.002	20	1.0018124	1.0058133	0.9948831	1.0076120	1.0000990	1.0006548
35	FALSE	0.050	30	1.0049878	1.0124879	0.9987499	1.0124524	0.9989879	1.0022609
34	FALSE	0.050	20	1.0119295	1.0165681	1.0032755	1.0219896	1.0064015	1.0114126
33	FALSE	0.050	10	1.0214832	1.0241429	1.0136486	1.0324887	1.0163014	1.0208345
3	FALSE	0.002	10	1.1734483	1.1647023	1.1574413	1.1845815	1.1796485	1.1808682

- The 'biased' column indicates whether we include biases b_i an and b_u in our model (i.e., whether or not we use probabilistic matrix factorization or the extended probabilistic matrix factorization).
- 'lr_all' is parameter is the learning rate

Now, that we have looked at the 5-fold cross validated performance for a range of parameters, we demonstrate how the model can be used to produce recommendations.

```
# define training / testing split
testTrainSplit=0.75
```

We train an extended probabilistic matrix factorization model in-sample using 75% of the data, then we generate recommendations for all of the out-of-sample users.

```
random.seed(randomSeed)
np.random.seed(randomSeed)
# sample random trainset and testset
# test set is made of 75% of the ratings.
trainset, testset = train_test_split(data, test_size=testTrainSplit)
# create predictions from
algo.fit(trainset)
predictions = algo.test(testset)
accuracy.rmse(predictions)
```

```
## RMSE: 0.9759
```

```
# define top N
topN=10
```

We now find the top 10 recommendations for the test set:

```
iid='item_id'
# map item id to title
item2Title=df_title_by_item.set_index(iid)['title'].to_dict()
# get top 10 recommendation for each user
top_n = get_top_n(predictions, n=topN)
# extract top N recommendations for each user
listTopN=list()
for uid, user_ratings in top_n.items():
    listTopN.append([uid,[item2Title[int(iid)] for (iid,
    _) in user_ratings]])
# convert to dataframe
dfTopN=pd.DataFrame(listTopN)
# rename columns
dfTopN.rename(columns={0 : 'user_id',1 : 'recommendations'},
    inplace=True)
# set index to user ID
dfTopN=dfTopN.set_index('user_id')
```

We can test the recommendations out-of-sample. We can predict the rating for an item by a user that was not part of the training set and compare that prediction with their actual rating.

```
# define raw user id (string)
user_id = str(531)
# define raw item id (string)
item_id = str(421)
trueRating=4.0
# get a prediction for specific users and items.
pred = algo.predict(user_id, item_id, r_ui=trueRating,
  verbose=True)
```

```
## user: 531        item: 421        r_ui = 4.00   est = 3.53   {u'was_impossible': False}
```

```
print(pred)
```

```
## user: 531        item: 421        r_ui = 4.00   est = 3.53   {u'was_impossible': False}
```

```
print(dfTopN.head())
```

```
##                                              recommendations
## user_id
## 1        [Good Will Hunting (1997), Blade Runner (1982)...
## 2        [Star Wars (1977), L.A. Confidential (1997), T...
## 3        [Schindler's List (1993), Good Will Hunting (1...
## 4        [Star Wars (1977), One Flew Over the Cuckoo's ...
## 5        [Close Shave, A (1995), Star Wars (1977), Raid...
```

We extract the top 10 recommendations for an example user as follows:

```
# define example user
user_id=5
# extract top N recommendations for example user
recommendationList=dfTopN['recommendations'].loc[user_id]
```

Finally, here are the recommendations:

```
sampleRecommendations=pd.DataFrame(recommendationList,
  columns=['recommendations'])
print(sampleRecommendations)
```

```
##                              recommendations
## 0                      Close Shave, A (1995)
## 1                           Star Wars (1977)
## 2             Raiders of the Lost Ark (1981)
## 3                 Princess Bride, The (1987)
## 4           Silence of the Lambs, The (1991)
## 5               To Kill a Mockingbird (1962)
```

```
## 6                                Jaws (1975)
## 7     Monty Python and the Holy Grail (1974)
## 8                  Young Frankenstein (1974)
## 9  Indiana Jones and the Last Crusade (1989)
```

Although we have not delved into measures beyond the root mean squared error to understand the utility of these recommendations to our users, there are many more techniques that can be used to help us understand our data once we have the latent factor loadings.

Software

This project was created using base R (**?**) and the R markdown (**?**), tint (**?**), kableExtra (**?**), reticulate, (**?**), and tidyverse (**?**) libraries. Anaconda Python (**?**) was used for most of the implementation code and relied considerably on the scientific python stack (particularly (**?**), (**?**),(**?**), and (**?**) and all of their dependencies)

Finally, the implementation of the power method for SVD relied heavily on lecture notes and code available on the internet ([@] and (**?**) respectively). The Surprise Python package (**?**) was used to performance probabilistic matrix factorization in the application section of the paper. This package can be installed with conda as follows:

conda install -c conda-forge scikit-surprise

Appendix A: Singular Value Decomposition (SVD) Relation to Eigen-Decomposition

The singular value decomposition (SVD) can be applied to any $m \times n$ matrix, whereas the eigen-decomposition can be applied only to *diagonalizable matrices*[3]. In this appendix, we briefly outline the relationship between SVD and the less general eigen-decomposition.

[3] A diagonalizable matrix is one where blaw blaw

Recall that for any singular value decomposition (SVD)

$$A = U\Sigma V^T$$

For SVD, A does not need to be symmetric.

The eigen-value decomposition is defined as

$$A = X\Lambda X^T$$

For eigen-decomposition, A must be symmetric (i.e. $m = n$).

U, V, X are orthonormal (i.e., $U^T U = I$, $V^T V = I$, and $X^T X = I$)

Given the singular value decomposition of A, the following two relations hold:

$$AA^T = U\Sigma V^T \left(U\Sigma V^T\right)^T = U\Sigma(V\Sigma^T U^T) = U\Sigma\Sigma^T U^T$$

$$A^T A = V\Sigma^T U^T \left(U\Sigma V^T\right) = V\Sigma\Sigma^T V^T$$

If A is an object-by-attribute input data matrix with zero means and unit standard deviations, AA^T and $A^T A$ are the correlation matrices for objects-to-objects and attributes-to-attributes respectively.

The right-hand sides of the two relations expressed above (namely equation blaw1 and blaw2), describe the eigen-decompositions of the left-hand sides. The columns of V (i.e., the right-singular vectors) are the eigenvectors of $A^T A$ (i.e., $V = X$ and $V^T = X^T$). The columns of U (i.e., the left-singular vectors) are the eigenvectors of AA^T (i.e., $U = X$). The non-zero elements of Σ (i.e., the non-zero singular values) are the square roots of the non-zero eigenvalues of $A^T A$ or AA^T (i.e., $\Sigma\Sigma^T = \Lambda$). This means that the eigenvalues are equivalent to the squared singular values (i.e., $\lambda_i = \sigma_i^2$.

Substituting these into the equation immediately above, we get $A = X\Lambda X^T$.