Team 33 – Movie Rating Prediction using the MovieLens dataset

1. Introduction

The goal of this assignment is to predict the rating given a user and a movie, using two different methods – K-Nearest Neighbors and Matrix Factorization method on the MovieLens 100k dataset.

2. Data and its Preprocessing

The initial data set i.e. u.data, obtained from the MovieLens, a movie recommendation service, has 100,000 ratings (1-5) from 943 users on 1682 movies. Ratings are integers on a scale of 1 to 5. Feature Names were added to the Dataset. Dataset was loaded using python pandas and was transformed into a ratings matrix represented as numpy array of size users x movies (943 x 1682). We found that users have rated at least 20 movies which result in a reasonable sparsity of 6.3%. This means that 1,00,000 ratings accounts to only 6.3% of the total possible ratings.

3. Algorithms and Feature Selection

The state-of-the-art in recommender systems is based on two main approaches - neighborhood approach and latent factor models.

Both these methods rely only on past user behavior, without using any features of the users or items. These practices are best to predict user ratings for films, given only the previous ratings, without any other information about the users or movies. An alternative to these methods is content filtering, which uses user features (such as age, gender, etc.) and item features (such as movie genre, cast etc.) instead of relying on past user behavior. Neighborhood approaches and latent factor models are more accurate than content-based techniques since they are domain-free and capture subtle information (such as a user's preferences) which is hard to extract using content filtering. On the other hand, neighborhood approaches and latent factor models suffer from the cold start problem, i.e. they are not useful for new users or new items. In this work, we compare two models - KNN Algorithm and Latent Factor Model.

a) KNN Algorithm – User-based and Item-based

The user-based K nearest neighbor algorithm is the same as the item-based K-nearest neighbor algorithm, except that we focus on the similarity between two customers. We use the Euclidean metric between user u and user u'.

$$sim(u, u') = \sqrt{\sum_i (r_{u,i} - r_{u',i})^2}$$

Where $r_{u,i}$ and $r_{u',i}$ are ratings of user u and user u' to movie i. And we choose top k most similar neighbors of each user under the above defined similarity function and use the weighted means to predict the ratings. The prediction formula is as follows:

$$P_{u,m} = \frac{\sum_{j \in N_u^K(m)} sim(m,j) R_{j,u}}{\sum_{j \in N_u^K(m)} |sim(m,j)|}$$

where $N_i^K(m) = \{ j: j$ belongs to the K most similar user to user m and user u has rated j$\}$, and $R_{j,u}$ are the existent ratings (of user u on movie j) and $P_{u,m}$ is the prediction.

b) Latent Factor Model

Given a user u and a movie i, we predict the rating that the user will give to the movie as follows:

$$r_{u,i} = \mu + b_u + b_i + \gamma_u \cdot \gamma_i$$

Where μ is the global bias, and b u (b i) is the user (item) bias. γ u and γ i are latent factors for user u and movie i respectively, which will be learned during the training process. γ u and γ i are K-dimensional vectors. The error function L is defined as follows:

$$L = \sum_{u,i} (r_{u,i} - (\mu + b_u + b_i + \gamma_u \cdot \gamma_i))^2 + \lambda_{ub} \sum_u \|b_u\|^2 + \lambda_{ib} \sum_i \|b_i\|^2 + \lambda_{u\gamma} \sum_u \|\gamma_u\|^2 + \lambda_{i\gamma} \sum_i \|\gamma_i\|^2$$

Where λ_{ub} , λ_{ib} , $\lambda_{u\gamma}$, $\lambda_{i\gamma}$ are used to control the trade-off between accuracy and complexity during training, and $\sum_u b_u^2$, $\sum_i b_i^2$, $\sum_u \gamma_u^2$, $\sum_i \gamma_i^2$ penalize model complexity and reduces over-fitting.

We have the following expressions for the gradient of the error function:

$$\frac{\partial L}{\partial b_u} = 2\left(r_{u,i} - (\mu + b_u + b_i + \gamma_u \cdot \gamma_i)\right)(-1) + 2\lambda_{ub} b_u$$

$$\frac{\partial L}{\partial b_u} = 2(error_{u,i})(-1) + 2\lambda_{ub} b_u$$

$$\frac{\partial L}{\partial b_u} = -error_{u,i} + \lambda_{ub} b_u$$

$$\frac{\partial L}{\partial b_i} = -error_{u,i} + \lambda_{ib} b_i$$

$$\frac{\partial L}{\partial \gamma_u} = -error_{u,i}\gamma_i + \lambda_{u\gamma}\gamma_u$$

$$\frac{\partial L}{\partial \gamma_i} = -error_{u,i}\gamma_u + \lambda_{i\gamma}\gamma_i$$

There are two approaches to solving the above optimization problem to find all of our features-stochastic gradient descent(SGD) and alternating least squares(ALS).

We use both SGD and ALS to update these parameters end up being:

$$b_u \leftarrow b_u + \eta(error_{u,i} - \lambda_{ub} b_u)$$

$$b_i \leftarrow b_i + \eta(error_{u,i} - \lambda_{ui} b_i)$$

$$\gamma_u \leftarrow \gamma_u + \eta(error_{u,i}\gamma_i - \lambda_{ub}\gamma_u)$$

$$\gamma_i \leftarrow \gamma_i + \eta(error_{u,i}\gamma_u - \lambda_{ub}\gamma_i)$$

4. Evaluation

We would like to predict the rating for MovieLens dataset, given a user u and a movie m. This problem is closely related to the issue of recommending the best possible items to a user based on the user's previous reviews or purchases. In this work, we will focus on accurately predicting the rating itself. Accordingly, we will evaluate the performance of our models using the widely-acknowledged mean square error (MSE) defined as:

$$mse = \frac{1}{|S_{test}|} \sum_{(u,l) \in S_{test}} (r_{ul} - p_{ul})^2$$

We will split our data into training and test sets by removing ten ratings per user from the training set and placing them in the test set.

We run each of the models on the same training and test set for a fair comparison. We run SGD for 200 iterations (where each iteration is one pass through the entire training set). We used grid search to find out the best latent factor K = 80, and those four regularization terms are all equal to 0.01.

For KNN, we found that mixed user-Item based is a better approach than user based and item based approach.

- **Model Comparison**

  Latent factor models are more expressive and tend to provide more accurate results than neighborhood models because latent factor models can capture and learn the latent factors which encode users' preferences and items' characteristics. On the other hand, neighborhood models are relatively more straightforward and offer more intuitive explanations of the reasoning behind the recommendation. Both models suffer from the cold-start problem, but give good performance for user and items with enough data.

5. Results and Conclusion

Table below lists the MSE values for each of the models.

Model	MSE - Test Set
User-based KNN	2.276
Item-based KNN	2.959
Mixed User-Item based KNN	2.158
Latent Factor Model(ALS)	14.15
Latent Factor Model(SGD)	1.015

The results in above table indicate that the latent factor model with SGD optimization method performs best among the five variations. Based on the results, we conclude that latent factor model with SGD tend to perform better than the neighborhood approaches on the movie rating prediction tasks, provided there is enough data for each user and for each movie.

6. Appendix

Student ID and Name of students:
17306521 - Ashwath Kumar Salimath
17309389 - Robin K Thomas
17308212 - Ajay Kumar N

The algorithms were selected as a group and then discussed on data analysis and pre-processing. Different algorithms were suggested and worked upon and finally KNN and latent factor methods were selected. Student ID 17309389 worked on KNN and produced the results for evaluation. Student ID 17306521 and Student ID 17308212 worked on developing the code for latent factor methods and evaluated the results. All of us worked on Report Development in Collaboration.

Time Taken:

Background research – 15 Hours
Data & Pre-Processing – 5 Hours
Algorithm & Feature Selection – 15 Hours
Evaluation – 15 Hours
Conclusion & report writing – 10 Hours
Total – 60 Hours

References:

- Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. Computer, August 2009
- Latent Factor Models for Web Recommender Systems. http://www.ideal.ece.utexas.edu/seminar/LatentFactorModels.pdf.
- Y. Koren. Factor in the neighbors: Scalable and accurate collaborative filtering. ACM Transactions on Knowledge Discovery from Data (TKDD), 4, January 2010.

7. Source Code

```
#KNN
import numpy as np
import csv
import sys
import os
from os.path import join
import copy
import DropboxAPI

#Fetching DataSet from DropBox  and Unzipping the File
url ='https://www.dropbox.com/sh/euppz607r6gsen2/AAAQCu8KjT7Ii1R60W2-
Bm1Ua/MovieLens%20(Movie%20Ratings)?dl=1'
zipFileName = 'MovieLens (Movie Ratings).zip'
subzipFileName ='movielens100k/ml-100k'
userDataSet = 'u.data'
userTestDataSet = 'u1.test'
destPath = os.getcwd()
DropboxAPI.fetchData(url, zipFileName, destPath)
filePath = join(destPath, zipFileName.rsplit(".", 1)[0])
filePath = join(filePath,subzipFileName.rsplit(".", 1)[0])
fullFilePath = join(filePath,userDataSet)

#Importing the Dataset
csvfile = open(fullFilePath)
csvreader = csv.reader(csvfile, delimiter='\t')

data=[]
for row in csvreader:
   data.append(row)
csvfile.close()
testid=np.random.randint(1,100000, 10000)
testset=[]
k=1
for i in testid:
   i-=k
   k+=1
   testset.append(data[i])
print('Select test 10000 cases.')
mdata=np.array(data,dtype=int)
usernum=int(np.max(mdata[:,0]))
itemnum=int(np.max(mdata[:,1]))
print('Total user number is : '+str(usernum))
print('Total movie number is : '+str(itemnum))
```

```
fdata = np.array(np.zeros((usernum, itemnum)))
print('Formating matrix.')
for row in mdata:
    fdata[row[0]-1, row[1]-1] = row[2]
for case in testset:
    fdata[int(case[0])-1,int(case[1])-1]=0

def findKNNitem(indata, item):
    iid = int(item[1])
    uid = int(item[0])
    temp = copy.deepcopy(indata[:, iid-1])
    for j in range(itemnum):
        indata[:, j] -= temp
    indata = indata**3
    sumd=indata.sum(axis=0)
    max=sumd.max()
    nn=[]
    for l in range(5):
        while fdata[uid-1,sumd.argmin()] == 0:
            sumd[sumd.argmin()]=max
        nn.append(sumd.argmin())
    ratelist = []
    for j in range(5):
        ratelist.append(fdata[uid-1, nn[j]])
    rate = np.average(ratelist)
    error = np.absolute(int(item[2])-rate)
    return error

def findKNNuser(indata,item):
    iid = int(item[1])
    uid = int(item[0])
    temp = copy.deepcopy(indata[uid-1, :])
    for i in range(usernum):
        indata[i, :] -= temp
    indata = indata**3
    sumd=indata.sum(axis=1)
    max=sumd.max()
    nn=[]
    z=5
    for i in range(z):
        while fdata[sumd.argmin(),iid-1]==0:
            sumd[sumd.argmin()]=max
            if sumd.max()==sumd.min():
                z=i
```

```
            break

        nn.append(sumd.argmin())
    ratelist=[]
    for i in range(z):
        ratelist.append(fdata[nn[i], iid-1])
    rate = np.average(ratelist)
    error=np.absolute(int(item[2])-rate)
    return error

errorlist=[]
print('start test '+str(len(testset)))
n=1

for testcase in testset:
    if n % 100==0:
        print('Test has finished : '+str(n/100)+'%' )
    error1=findKNNuser(copy.deepcopy(fdata), testcase)
    errorlist.append((error1)**2)
    n+=1
print('User-based KNN - Test finished')
meanserror=np.average(errorlist)
print('User-based MSE: ', meanserror)

for testcase in testset:
    if n % 100==0:
        print('Test has finished : '+str(n/100)+'%' )
    error2=findKNNitem(copy.deepcopy(fdata), testcase)
    errorlist.append((error2)**2)
    n+=1
print('test finished')
meanserror=np.average(errorlist)
print('Item-based MSE: ', meanserror)

for testcase in testset:
    if n % 100==0:
        print('Test has finished : '+str(n/100)+'%' )
    error1=findKNNuser(copy.deepcopy(fdata), testcase)
    error2=findKNNitem(copy.deepcopy(fdata), testcase)
    errorlist.append((error1/2+error2/2)**2)
    n+=1
print('test finished')
meanserror=np.average(errorlist)
print('Mixed User-Item based: ',meanserror)
```

```
#Matrix Factorization

#Importing the Libraries
import numpy as np
import pandas as pd
from os.path import join
import os
import DropboxAPI


#Fetching DataSet from DropBox  and Unzipping the File
url ='https://www.dropbox.com/sh/euppz607r6gsen2/AAAQCu8KjT7Ii1R60W2-
Bm1Ua/MovieLens%20(Movie%20Ratings)?dl=1'
zipFileName = 'MovieLens (Movie Ratings).zip'
subzipFileName ='movielens100k/ml-100k'
userDataSet = 'u.data'
userTestDataSet = 'u1.test'
destPath = os.getcwd()
DropboxAPI.fetchData(url, zipFileName, destPath)
filePath = join(destPath, zipFileName.rsplit(".", 1)[0])
filePath = join(filePath,subzipFileName.rsplit(".", 1)[0])
#fullFilePath = join(filePath,userDataSet)

#Importing the Dataset
names = ['user_id', 'item_id', 'rating', 'timestamp']
df = pd.read_csv(join(filePath,userDataSet), sep='\t', names=names)


#Calculating Number of Unique Users and Unique Movies
n_users = df.user_id.unique().shape[0]
n_items = df.item_id.unique().shape[0]

# Create r_{ui}, our ratings matrix
ratings = np.zeros((n_users, n_items))
for row in df.itertuples():
    ratings[row[1] - 1, row[2] - 1] = row[3]


# Split into training and test sets.
# Remove 10 ratings for each user
# and assign them to the test set
```

```
def train_test_split(ratings):
    test = np.zeros(ratings.shape)
    train = ratings.copy()
    for user in range(ratings.shape[0]):
        test_ratings = np.random.choice(ratings[user, :].nonzero()[0],
                                        size=10,
                                        replace=False)
        train[user, test_ratings] = 0.
        test[user, test_ratings] = ratings[user, test_ratings]

    # Test and training are truly disjoint
    assert (np.all((train * test) == 0))
    return train, test


train, test = train_test_split(ratings)
print('ratings shape', ratings.shape)

from sklearn.metrics import mean_squared_error


def get_mse(pred, actual):
    # Ignore nonzero terms.
    pred = pred[actual.nonzero()].flatten()
    actual = actual[actual.nonzero()].flatten()
    return mean_squared_error(pred, actual)


def get_rmse(pred, actual):
    # Ignore nonzero terms.
    import math
    pred = pred[actual.nonzero()].flatten()
    actual = actual[actual.nonzero()].flatten()
    return math.sqrt(mean_squared_error(pred, actual))


from numpy.linalg import solve


class ExplicitMF():
    def __init__(self,
                 ratings,
                 n_factors=40,
```

```
        learning='sgd',
        item_fact_reg=0.0,
        user_fact_reg=0.0,
        item_bias_reg=0.0,
        user_bias_reg=0.0,
        verbose=False):
    """
    Train a matrix factorization model to predict empty
    entries in a matrix. The terminology assumes a
    ratings matrix which is ~ user x item
    Params
    ======
    ratings : (ndarray)
       User x Item matrix with corresponding ratings
    n_factors : (int)
       Number of latent factors to use in matrix
       factorization model
    learning : (str)
       Method of optimization. Options include
       'sgd' or 'als'.
    item_fact_reg : (float)
       Regularization term for item latent factors
    user_fact_reg : (float)
       Regularization term for user latent factors
    item_bias_reg : (float)
       Regularization term for item biases
    user_bias_reg : (float)
       Regularization term for user biases
    verbose : (bool)
       Whether or not to printout training progress
    """

    self.ratings = ratings
    self.n_users, self.n_items = ratings.shape
    self.n_factors = n_factors
    self.item_fact_reg = item_fact_reg
    self.user_fact_reg = user_fact_reg
    self.item_bias_reg = item_bias_reg
    self.user_bias_reg = user_bias_reg
    self.learning = learning
    if self.learning == 'sgd':
       self.sample_row, self.sample_col = self.ratings.nonzero()
```

```
        print('sample_row', self.sample_row)
        print('sample_col', self.sample_col)
        self.n_samples = len(self.sample_row)
    self._v = verbose

def als_step(self,
             latent_vectors,
             fixed_vecs,
             ratings,
             _lambda,
             type='user'):
    """
    One of the two ALS steps. Solve for the latent vectors
    specified by type.
    """
    if type == 'user':
        # Precompute
        YTY = fixed_vecs.T.dot(fixed_vecs)
        lambdaI = np.eye(YTY.shape[0]) * _lambda

        for u in range(latent_vectors.shape[0]):
            latent_vectors[u, :] = solve((YTY + lambdaI),
                                         ratings[u, :].dot(fixed_vecs))
    elif type == 'item':
        # Precompute
        XTX = fixed_vecs.T.dot(fixed_vecs)
        lambdaI = np.eye(XTX.shape[0]) * _lambda

        for i in range(latent_vectors.shape[0]):
            latent_vectors[i, :] = solve((XTX + lambdaI),
                                         ratings[:, i].T.dot(fixed_vecs))
    return latent_vectors

def train(self, n_iter=10, learning_rate=0.1):
    """ Train model for n_iter iterations from scratch."""
    # initialize latent vectors
    # self.user_vecs = np.random.normal(scale=1./self.n_factors,\
    #                                   size=(self.n_users, self.n_factors))
    self.user_vecs = np.zeros(shape=(self.n_users, self.n_factors))
    # self.item_vecs = np.random.normal(scale=1./self.n_factors,
    #                                   size=(self.n_items, self.n_factors))
    self.item_vecs = np.zeros(shape=(self.n_items, self.n_factors))
```

```
    if self.learning == 'als':
        self.partial_train(n_iter)
    elif self.learning == 'sgd':
        self.learning_rate = learning_rate
        self.user_bias = np.zeros(self.n_users)
        self.item_bias = np.zeros(self.n_items)
        self.global_bias = np.mean(self.ratings[np.where(self.ratings != 0)])
        self.partial_train(n_iter)

def partial_train(self, n_iter):
    """
    Train model for n_iter iterations. Can be
    called multiple times for further training.
    """
    ctr = 1
    while ctr <= n_iter:
        if ctr % 10 == 0 and self._v:
            print('\tcurrent iteration: {:d}'.format(ctr))
        if self.learning == 'als':
            self.user_vecs = self.als_step(self.user_vecs,
                                           self.item_vecs,
                                           self.ratings,
                                           self.user_fact_reg,
                                           type='user')
            self.item_vecs = self.als_step(self.item_vecs,
                                           self.user_vecs,
                                           self.ratings,
                                           self.item_fact_reg,
                                           type='item')
        elif self.learning == 'sgd':
            self.training_indices = np.arange(self.n_samples)
            np.random.shuffle(self.training_indices)
            self.sgd()
        ctr += 1

def sgd(self):
    for idx in self.training_indices:
        u = self.sample_row[idx]
        i = self.sample_col[idx]
        prediction = self.predict(u, i)
```

```
        e = (self.ratings[u, i] - prediction)  # error

        # Update biases
        self.user_bias[u] += self.learning_rate * \
                        (e - self.user_bias_reg * self.user_bias[u])
        self.item_bias[i] += self.learning_rate * \
                        (e - self.item_bias_reg * self.item_bias[i])

        # Update latent factors
        self.user_vecs[u, :] += self.learning_rate * \
                            (e * self.item_vecs[i, :] - \
                             self.user_fact_reg * self.user_vecs[u, :])
        self.item_vecs[i, :] += self.learning_rate * \
                            (e * self.user_vecs[u, :] - \
                             self.item_fact_reg * self.item_vecs[i, :])

    def predict(self, u, i):
        """ Single user and item prediction."""
        if self.learning == 'als':
            return self.user_vecs[u, :].dot(self.item_vecs[i, :].T)
        elif self.learning == 'sgd':
            prediction = self.global_bias + self.user_bias[u] + self.item_bias[i]
            # print 'global_bias_user_bias_item_bias', prediction
            prediction += self.user_vecs[u, :].dot(self.item_vecs[i, :].T)
            # print 'prediction', prediction
            return prediction

    def predict_all(self):
        """ Predict ratings for every user and item."""
        predictions = np.zeros((self.user_vecs.shape[0],
                                self.item_vecs.shape[0]))
        print('predictions_all shape', predictions.shape)
        for u in range(self.user_vecs.shape[0]):
            for i in range(self.item_vecs.shape[0]):
                predictions[u, i] = self.predict(u, i)
        # print predictions
        return predictions

    def calculate_learning_curve(self, iter_array, test, learning_rate=0.1):
        """
        Keep track of MSE as a function of training iterations.
```

```
    Params
    ======
    iter_array : (list)
        List of numbers of iterations to train for each step of
        the learning curve. e.g. [1, 5, 10, 20]
    test : (2D ndarray)
        Testing dataset (assumed to be user x item).
    The function creates two new class attributes:
    train_mse : (list)
        Training data MSE values for each value of iter_array
    test_mse : (list)
        Test data MSE values for each value of iter_array
    """
    iter_array.sort()
    self.train_mse = []
    self.test_mse = []
    self.train_rmse = []
    self.test_rmse = []
    iter_diff = 0
    for (i, n_iter) in enumerate(iter_array):
        if self._v:
            print('Iteration: {}'.format(n_iter))
        if i == 0:
            self.train(n_iter - iter_diff, learning_rate)
        else:
            self.partial_train(n_iter - iter_diff)

        predictions = self.predict_all()

        self.train_mse += [get_mse(predictions, self.ratings)]
        self.train_rmse += [get_rmse(predictions, self.ratings)]
        self.test_mse += [get_mse(predictions, test)]
        self.test_rmse += [get_rmse(predictions, test)]
        if self._v:
            print('Train mse: ' + str(self.train_mse[-1]))
            print('Train rmse: ' + str(self.train_rmse[-1]))
            print('Test mse: ' + str(self.test_mse[-1]))
            print('Test rmse: ' + str(self.test_rmse[-1]))
        iter_diff = n_iter
```

```
# MF_ALS = ExplicitMF(train, n_factors=40, \
#                 user_reg=0.0, item_reg=0.0)
# iter_array = [1, 2, 5, 10, 25, 50, 100]
# MF_ALS.calculate_learning_curve(iter_array, test)

import matplotlib.pyplot as plt
import seaborn as sns

sns.set()


def plot_learning_curve(iter_array, model):
    plt.plot(iter_array, model.train_mse, \
            label='Training', linewidth=5)
    plt.plot(iter_array, model.test_mse, \
            label='Test', linewidth=5)

    plt.xticks(fontsize=16);
    plt.yticks(fontsize=16);
    plt.xlabel('iterations', fontsize=30);
    plt.ylabel('MSE', fontsize=30);
    plt.legend(loc='best', fontsize=20);


"""
#plot_learning_curve(iter_array, MF_ALS)
iter_array = [1, 2, 5, 10, 25, 50, 100, 200]
latent_factors = [5, 10, 20, 40, 80]
regularizations = [0.001, 0.01, 0.1, 1.]
regularizations.sort()
best_params = {}
best_params['n_factors'] = latent_factors[0]
best_params['reg'] = regularizations[0]
best_params['n_iter'] = 0
best_params['train_mse'] = np.inf
best_params['test_mse'] = np.inf
best_params['model'] = None
for fact in latent_factors:
    print 'Factors: {}'.format(fact)
    for reg in regularizations:
        print 'Regularization: {}'.format(reg)
```

```
        MF_SGD = ExplicitMF(train, n_factors=fact, learning='sgd',\
                        user_fact_reg=reg, item_fact_reg=reg, \
                        user_bias_reg=reg, item_bias_reg=reg)
        MF_SGD.calculate_learning_curve(iter_array, test, learning_rate=0.001)
        min_idx = np.argmin(MF_SGD.test_mse)
        if MF_SGD.test_mse[min_idx] < best_params['test_mse']:
            best_params['n_factors'] = fact
            best_params['reg'] = reg
            best_params['n_iter'] = iter_array[min_idx]
            best_params['train_mse'] = MF_SGD.train_mse[min_idx]
            best_params['test_mse'] = MF_SGD.test_mse[min_idx]
            best_params['model'] = MF_SGD
            print 'New optimal hyperparameters'
            print pd.Series(best_params)
plot_learning_curve(iter_array, best_params['model'])
print 'Best regularization: {}'.format(best_params['reg'])
print 'Best latent factors: {}'.format(best_params['n_factors'])
print 'Best iterations: {}'.format(best_params['n_iter'])"""

best_als_model = ExplicitMF(ratings, n_factors=20, learning='als', \
                        item_fact_reg=0.01, user_fact_reg=0.01, verbose=True)

iter_array = [50]
best_als_model.calculate_learning_curve(iter_array, test)

best_sgd_model = ExplicitMF(train, n_factors=80, learning='sgd', \
                        item_fact_reg=0.01, user_fact_reg=0.01, \
                        user_bias_reg=0.01, item_bias_reg=0.01, verbose=True)
# best_sgd_model.train(200, learning_rate=0.001)
iter_array = [200]
best_sgd_model.calculate_learning_curve(iter_array, test)

"""
def cosine_similarity(model):
    sim = model.item_vecs.dot(model.item_vecs.T)
    norms = np.array([np.sqrt(np.diagonal(sim))])
    return sim / norms / norms.T
als_sim = cosine_similarity(best_als_model)
sgd_sim = cosine_similarity(best_sgd_model)
# Load in movie data
idx_to_movie = {}
```

```
with open('ml-100k/u.item', 'r') as f:
    for line in f.readlines():
        info = line.split('|')
        idx_to_movie[int(info[0])-1] = info[4]
# Build function to query themoviedb.org's API
import requests
import json
# Get base url filepath structure. w185 corresponds to size of movie poster.
api_key = '0f1e88eacf0ad51529b6557515c266fe'
headers = {'Accept': 'application/json'}
payload = {'api_key': api_key}
response = requests.get("http://api.themoviedb.org/3/configuration",\
                        params=payload,\
                        headers=headers)
response = json.loads(response.text)
base_url = response['images']['base_url'] + 'w185'
def get_poster(imdb_url, base_url, api_key):
    #get IMDB movie ID
    response = requests.get(imdb_url)
    movie_id = response.url.split('/')[-2]
    #query themoviedb.org API for movie poster path
    movie_url = 'http://api.themoviedb.org/3/movie/{:}/images'.format(movie_id)
    headers = {'Accept': 'application/json'}
    payload = {'api_key': api_key}
    response = requests.get(movie_url, params=payload, headers=headers)
    try:
        file_path = json.loads(response.text)['posters'][0]['file_path']
    except:
        # IMDB movie ID is sometimes no good. Need to get correct one.
        movie_title = imdb_url.split('?')[-1].split('(')[0]
        payload['query'] = movie_title
        response = requests.get('http://api.themoviedb.org/3/search/movie',\
                                params=payload,\
                                headers=headers)
        try:
            movie_id = json.loads(response.text)['results'][0]['id']
            payload.pop('query', None)
            movie_url = 'http://api.themoviedb.org/3/movie/{:}/images'\
                    .format(movie_id)
            response = requests.get(movie_url, params=payload, headers=headers)
            file_path = json.loads(response.text)['posters'][0]['file_path']
```

```
        except:
            # Sometimes the url just doesn't work.
            # Return '' so that it does not mess up Image()
            return ''
    return base_url + file_path
from IPython.display import HTML
from IPython.display import display
def display_top_k_movies(similarity, mapper, movie_idx, base_url, api_key, k=5):
    movie_indices = np.argsort(similarity[movie_idx, :])[::-1]
    images = ''
    k_ctr = 0
    #start i at 1 to not grab the input movie
    i = 1
    while k_ctr < k:
        movie = mapper[movie_indices[i]]
        poster = get_poster(movie, base_url, api_key)
        if poster != '':
            images += "<img style='width: 120px; margin: 0px; \
                    float: left; border: 1px solid black;' src='%s' />"\
                    % poster
            k_ctr += 1
        i += 1
    display(HTML(images))
def compare_recs(als_similarity, sgd_similarity, mapper,
                 movie_idx, base_url, api_key, k=5):
    #display input
    display(HTML('<font size=5>'+'Input'+'</font>'))
    input_poster = get_poster(mapper[movie_idx], base_url, api_key)
    input_image = "<img style='width: 120px; margin: 0px; \
                    float: left; border: 1px solid black;' src='%s' />" \
                % input_poster
    display(HTML(input_image))
    # Display ALS Recs
    display(HTML('<font size=5>'+'ALS Recs'+'</font>'))
    display_top_k_movies(als_similarity, idx_to_movie,\
                         movie_idx, base_url, api_key)
    # Display SGD Recs
    display(HTML('<font size=5>'+'SGD Recs'+'</font>'))
    display_top_k_movies(sgd_similarity, idx_to_movie,\
                         movie_idx, base_url, api_key)
idx = 0 # Toy Story
```

```
compare_recs(als_sim, sgd_sim, idx_to_movie, idx, base_url, api_key)
idx = 1 # GoldenEye
compare_recs(als_sim, sgd_sim, idx_to_movie, idx, base_url, api_key)
idx = 20 # Muppet Treasure Island
compare_recs(als_sim, sgd_sim, idx_to_movie, idx, base_url, api_key)
idx = 40 # Billy Madison
compare_recs(als_sim, sgd_sim, idx_to_movie, idx, base_url, api_key)
idx = 500 # Dumbo
compare_recs(als_sim, sgd_sim, idx_to_movie, idx, base_url, api_key)
```