Machine Learning Engineer Nanodegree

Sekhar Mekala

Wednesday, May 17, 2017

Abstract

As a part of the Machine Learning Engineer Nanodegree program, I worked on developing a jokes recommendation system using the dataset available at http://eigentaste.berkeley.edu/dataset/. This report provides a complete implementation of the recommender system using Python 2.7 and Python packages such as Numpy, Pandas, Sklearn etc. The recommender system was developed using the SGD (Stochastic Gradient Descent) Algorithm, and an optimal method to identify the ideal rating threshold, which determines whether a user liked an item or not was also thoroughly discussed. The recommender's source code was developed as generic as possible, so that the same pipeline of code can be used to develop any recommendation system such as movies recommender, songs recommender, books recommender, wine recommender etc. For the jokes recommender we obtained an average AUC (Area Under the Curve) as lying between 80% to 85% for almost all the test datasets.

Keywords: Jokes recommender, SGD, Stochastic Gradient Descent Algorithm, Python Recommender System pipeline

1.0 Recommender systems

A recommendation engine is a system, which helps us to predict the user responses to options. As per Wikipedia, Recommender systems have become increasingly popular in recent years, and are utilized in a variety of areas including movies, music, news, books, research articles, search queries, social tags, and products in general.

Why do we need recommendation systems?

Imagine a bookstore that sells books in a shop, where customers visit the physical shop and purchase the books. In such a business, the books are displayed to the user based on the general popularity of the item, and it is *not* possible to arrange the display of all the available books since the shelf space is limited in a physical store. But in online shops, there is no limit to the shelf space and we can display as many items as we want for each customer. But if we just display all the available items to the users, then the customer is swamped with many items and soon s/he loses interest in buying the items from the online store. Hence we need to identify the items a customer would like based on the available information about the customer, his/her previous purchases, other customers' purchases, items features, and items ratings. We can determine if a user likes a product, by predicting the potential rating the user could give to the product. If we predict that the user might give a higher rating to a product, then we can propose the product to the user. Hence, we need to estimate the ratings a customer would give to the items, which were previously unseen by the user.

Data representation in Recommendation systems

The problem of building a recommendation system can be classified as a supervised machine learning problem. However, to provide customized recommendations to users, we cannot rely on a traditional machine learning algorithms, since traditional techniques require the typical features/outcome setting (for supervised learning), and such approach will compel us to develop a separate predictive model for each customer. Given that a user rates only a very small fraction of the available items, it is not possible to develop a separate predictive model for each user. So instead of representing the problem as a traditional supervised learning problem (table with independent and dependent variables), we represent the items recommendation problem in the form of a matrix. The users are represented as rows and the items are represented as columns. The ratings given by each user to each item represent the elements of the matrix. Such rating matrix is called Utility matrix, and this matrix is usually very sparse. Most of the recommender systems will predict the missing elements in the utility matrix, and propose the items to the user based on the values estimated for the missing ratings.

In this project, we are building a recommender system for the jokes based on the Jester dataset. Users will rate the jokes at the website: http://eigentaste.berkeley.edu. These jokes and their ratings were made publicly available at http://eigentaste.berkeley.edu/dataset/. I chose the *dataset 2* consisting of 150 items (or jokes) rated by 59132 users. This dataset has approximately 1.7 million ratings.

2.0 Data exploration

The *Jester dataset 2* is a compressed file, which is publicly available at http://eigentaste.berkeley.edu/dataset/. This dataset has been downloaded and unzipped. We have 2 files present in the compressed file. The details of these datasets are given below:

- The *jester_ratings.dat* dataset. It has the following format (tab separated):

  user-id, *joke-id*, *rating*

- The *jester_items.dat* dataset. It has the *joke-id* and the actual joke (HTML text), separated by a ":"

The following code block will read the *jester_ratings.dat* dataset to a data frame *ratings_df*. The *jester_items.dat* data will be used later in section 9.1.

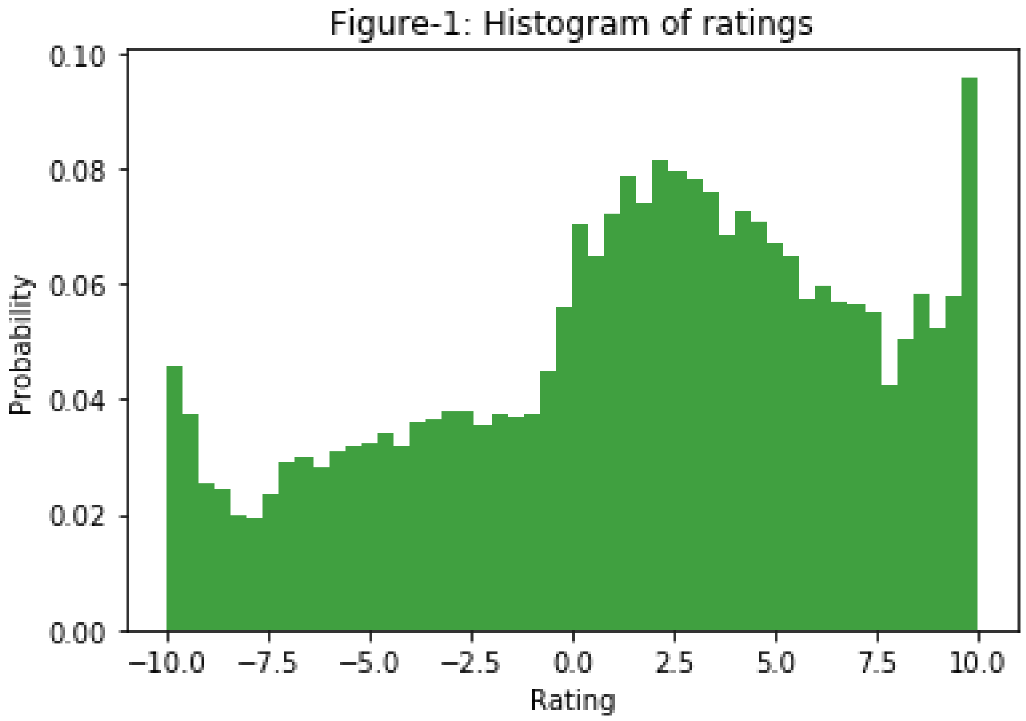
Python source code

```
Importing the required python packages:
import pandas as pd
import numpy as np
from IPython.display import display # Allows the use of display() for DataFrames
import time
import warnings
import itertools
warnings.filterwarnings('ignore')
import matplotlib.pyplot as plt
%matplotlib inline
from scipy.sparse.linalg import svds
from bs4 import BeautifulSoup #To extract text from HTML
import io #To process buffer data

##Reading the ratings dataset to a file
ratings_df = pd.read_csv("jester_ratings.dat",sep="\t\t",header=None)
ratings_df.columns = ["User_ID", "Joke_ID","Rating"]
#print "Initial rows of the ratings data:"
#display(ratings_df.head())
```

2.1 How are the ratings distributed?

Let us create a histogram of ratings, using the following python code:

```
#To create _Figure-1: Histogram_

import numpy as np
import matplotlib.mlab as mlab
import matplotlib.pyplot as plt

x = np.array(ratings_df["Rating"])
# the histogram of the data
plt.hist(x, 50, normed=1, facecolor='green', alpha=0.75)
plt.xlabel('Rating')
plt.ylabel('Probability')
plt.title("Figure-1: Histogram of ratings")
plt.show()
```

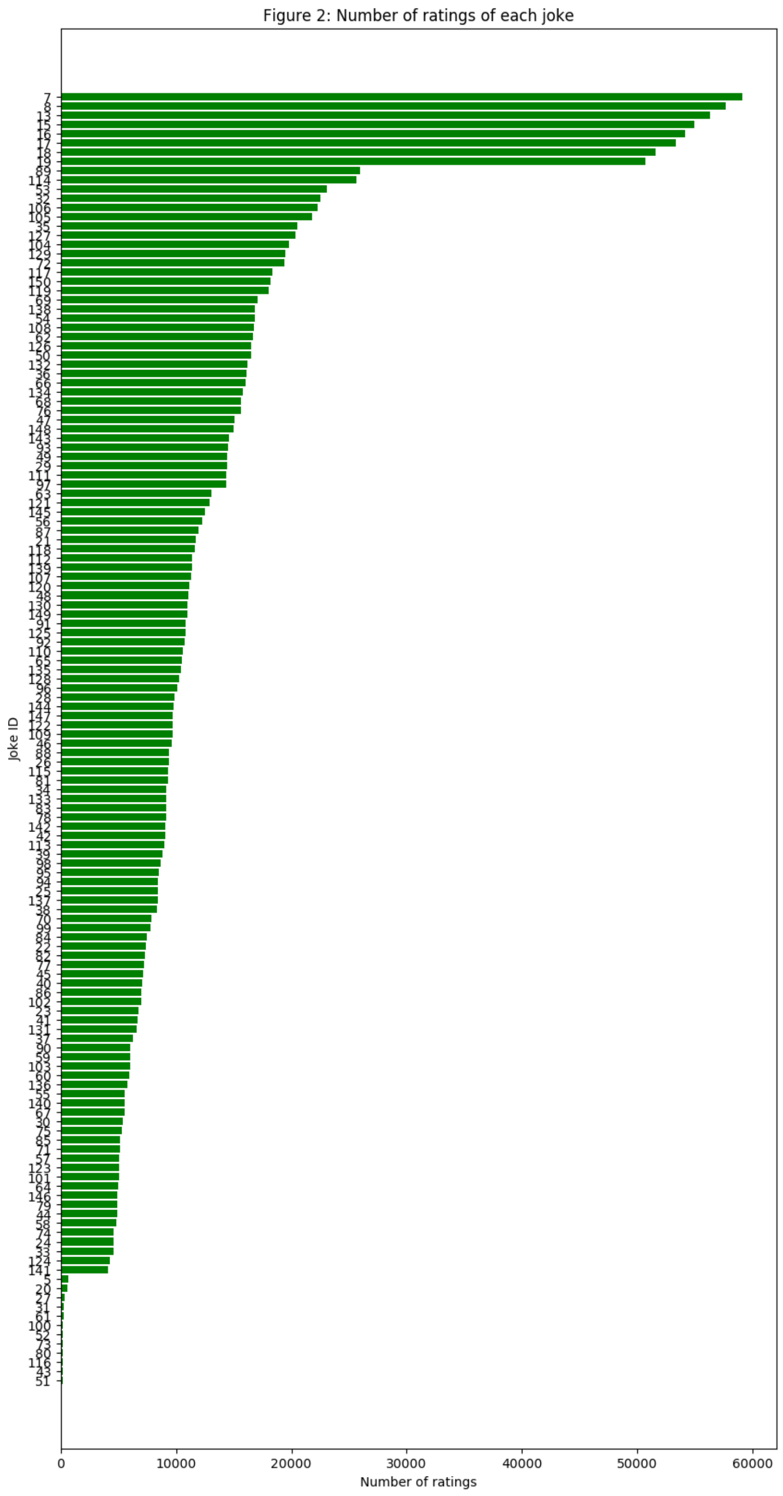


From Figure-1 (histogram displayed above), we can infer the following:

- The ratings are *not* normally distributed.
- Most of the ratings were given between 0 and 5.
- Approximately 9% of the items were given an approximate rating of 10.

2.2 Which jokes are most popular?

Let us get the list of all jokes along with the number of times each one has been rated. Jokes which were rated many times (not necessarily rated high) can be considered as the most *popular jokes* since the recommendation system presents those popular items quite frequently to most of the users.



To create *Figure 2: Number of ratings of each joke*, use the following python code:

```
import matplotlib.pyplot as plt
plt.rcdefaults()
import numpy as np
import matplotlib.pyplot as plt
```

```
plt.rcdefaults()
fig, ax = plt.subplots(figsize=(10, 20))

# Example data
Joke_ID = list(cnt_ratings["Joke_ID"])
y_pos = np.arange(len(Joke_ID))
counts = list(cnt_ratings["Log_scaled_counts"])


ax.barh(y_pos, counts, align='center',
        color='green')
ax.set_yticks(y_pos)
ax.set_yticklabels(Joke_ID)
ax.invert_yaxis()  # labels read top-to-bottom
ax.set_xlabel('Number of ratings')
ax.set_ylabel('Joke ID')
ax.set_title('Figure 2: Number of ratings of each joke')

plt.show()
```

Figure-2 shows that some of the jokes were rated many times and some of the jokes were *not* rated many times. This could be due to various reasons. The jokes with less number of ratings could be new entries into the system, or the jokes recommender system (http://eigentaste.berkeley.edu) might be presenting some jokes more frequently than the other jokes (based on the recommendation engine's logic). For example, if the recommendation engine estimates that some of the jokes would not be liked by the user for most of the times, then such jokes are not displayed to the users frequently and they remain unrated (or rated very less number of times) by the users. Such problem of not displaying some of the products can be fixed using the explore-exploit logic. All the recommendation engines estimate the chance that a user likes a product, and displays such products to the user. But in the explore-exploit method, the recommendation engines will include some of the products randomly in the display list, even though the system estimates that the randomly included products would not be liked by the user. This method will fix the problem of unrated items, and also gives a chance to get users attention on less popular items. In this project, we will not implement the explore-exploit logic since it is not possible to estimate the success of that method unless we really display the jokes to the user in real time and capture the user responses.

2.3 Other details

- Average rating: 1.617
- Std. deviation of the rating: 5.302
- Range of ratings: -10 to +10
- Higher the rating, higher the chance that the user has liked a joke

Given that the standard deviation of the ratings is 5.302, there is a very high variability in ratings. This can also be inferred from Figure-1, which shows that the data is not normally distributed.

3.0 Problem statement

In recommendation systems, we have a Utility matrix that shows the affinity of all users towards all available items. But this Utility matrix is very sparse, since most of the users might not have looked or experienced all items, and the main goal of recommender systems is to identify the potential items the user might be interested in. One way to accomplish this goal is based on matrix factorization method. We have to express the Utility matrix as a product of two matrices, to estimate the missing entries in the Utility matrix. In mathematical terms, we define our main objective as given below.

Objective: Express the matrix M as a product of U and V. Where M is an mXn matrix, U is an mXd matrix and V is a dXn matrix, where d is the number of latent factors.

Mathematically, let

$$M = \begin{bmatrix} r_{11} & r_{12} & . & . & r_{1n} \\ r_{21} & r_{22} & . & . & r_{2n} \\ r_{31} & r_{32} & . & . & r_{3n} \\ . & . & . & . & . \\ . & . & . & . & . \\ r_{m1} & r_{m2} & . & . & r_{mn} \end{bmatrix}$$

$$U = \begin{bmatrix} u_{11} & . & u_{1d} \\ u_{21} & . & u_{2d} \\ u_{31} & . & u_{3d} \\ . & . & . \\ . & . & . \\ u_{m1} & . & u_{md} \end{bmatrix}$$

$$V = \begin{bmatrix} v_{11} & v_{12} & . & . & v_{1n} \\ v_{21} & v_{22} & . & . & v_{2n} \\ . & . & . & . & . \\ v_{d1} & v_{d2} & . & . & v_{dn} \end{bmatrix}$$

Then M can be expressed as the matrix product of U and V, as shown below:

$$\begin{bmatrix} r_{11} & r_{12} & . & . & r_{1n} \\ r_{21} & r_{22} & . & . & r_{2n} \\ r_{31} & r_{32} & . & . & r_{3n} \\ . & . & . & . & . \\ . & . & . & . & . \\ r_{m1} & r_{m2} & . & . & r_{mn} \end{bmatrix} \approx \begin{bmatrix} u_{11} & . & u_{1d} \\ u_{21} & . & u_{2d} \\ u_{31} & . & u_{3d} \\ . & . & . \\ . & . & . \\ u_{m1} & . & u_{md} \end{bmatrix} \begin{bmatrix} v_{11} & v_{12} & . & . & v_{1n} \\ v_{21} & v_{22} & . & . & v_{2n} \\ . & . & . & . & . \\ v_{d1} & v_{d2} & . & . & v_{dn} \end{bmatrix}$$

In recommender systems, the matrix M is a sparse matrix with many unknown entries. An example of such a sparse matrix is shown below:

$$M = \begin{bmatrix} & r_{12} & . & . & r_{1n} \\ r_{21} & & . & . & r_{2n} \\ r_{31} & & . & . & \\ . & . & . & . & . \\ . & . & . & . & . \\ r_{m1} & r_{m2} & . & . & \end{bmatrix}$$

For such sparse matrices, we cannot use SVD (Singular Value Decomposition) method to factorize the matrix. Hence for recommendation systems our main objective is to estimate the U and V matrices considering only the available data in M. Once we obtain optimal U and V matrices (based on the available data in M), we can get the matrix product UV, and estimate an approximate value of the missing elements in M, by comparing the corresponding elements between M and UV. Another advantage of U, V factorization is to identify the hidden dimensions (also called latent factors), which map both the user and items to a common set of dimensions/coordinate system. Such mapping will help to identify users/items/user-item pairs, which are near to each other.

Two of the prominent methods to estimate the U and V matrices are:

- Alternating Least Squares (ALS)
- Gradient descent method (Stochastic Gradient Descent or SGD)

For this project, I will be using SGD algorithm, since as per my analysis, SGD has performed better than ALS. See my detailed analysis of SGD and ALS methods at https://goo.gl/5u5isI

The logic of SGD algorithm is explained below:

3.1 Solution: Stochastic Gradient Descent (SGD)

1. Initialize U and V to random values. We can assume 0s for NA values in M, use SVD method to obtain U and V, and use these values as the initial values of U and V.
2. Randomly choose U or V
3. If U is chosen, randomly choose a row i from U. To estimate the row U_i, perform the following:

   3a. Get the list of all columns in M_i row, where we have available values. Call these locations as C

   3b. Let $V_C = V[:, C]$, where $V[:, C]$ is the list of all columns in V, corresponding to the column numbers present in C

   3c. Estimate new value of U_i as:

   $$U_i^{new} = U_i - \alpha \nabla_{U_i} f_{ij}(U_i, V_i)$$

   where α is the learning rate, and $\nabla_{U_i} f_{ij}(U_i, V_j)$ is defined as follows:

   $$\nabla_{U_i} f_{ij}(U_i, V_j) = [M_i - U_i.V_C].V_C^T + \lambda_1 U_i$$

   where λ_1 is the regularization parameter
4. If V is chosen, randomly choose a column j from V. To estimate the column V_j, perform the following:

   4a. Get the list of all rows in M_j column, where we have available values. Call these locations as R

   4b. Let $U_R = U[R, :]$, where $U[R, :]$ is the list of all rows in U, corresponding to the row numbers present in R

   4c. Estimate new value of V_j as:

   $$V_j^{new} = V_j - \alpha \nabla_{V_j} f_{ij}(U_i, V_i)$$

   where α is the learning rate, and $\nabla_{V_j} f_{ij}(U_i, V_j)$ is defined as follows:

   $$\nabla_{V_j} f_{ij}(U_i, V_j) = [M_j - U_R.V_j]^T.U_R + \lambda_2 V_j$$

   where λ_2 is the regularization parameter

4.0 Metrics

In this project we will use the following metrics:

- RMSE (Root Mean Squared Error)
- AUC (Area Under the Curve) in ROC (Receiver Operating Characteristics curve)

RMSE - Root Mean Square Error will be used to implement the SGD. By means of RMSE, we will identify those U and V matrices, whose product is as close as possible to the given Utility matrix. RMSE will be used to tune the SGD parameters.

AUC - Area Under the Curve will be used to get the performance of our predictive model (which predicts if a user likes/dislikes an item). The average AUC will be used to compare the performance of our models (for various sets of random data), at different rating thresholds (the threshold determines if a user likes or dislikes an item). Hence we will use AUC to get an estimate of our algorithm's performance based on the optimal SGD parameters identified using RMSE, and average AUC will be used to get the optimal cut-off rating, which identifies if a user likes the item.

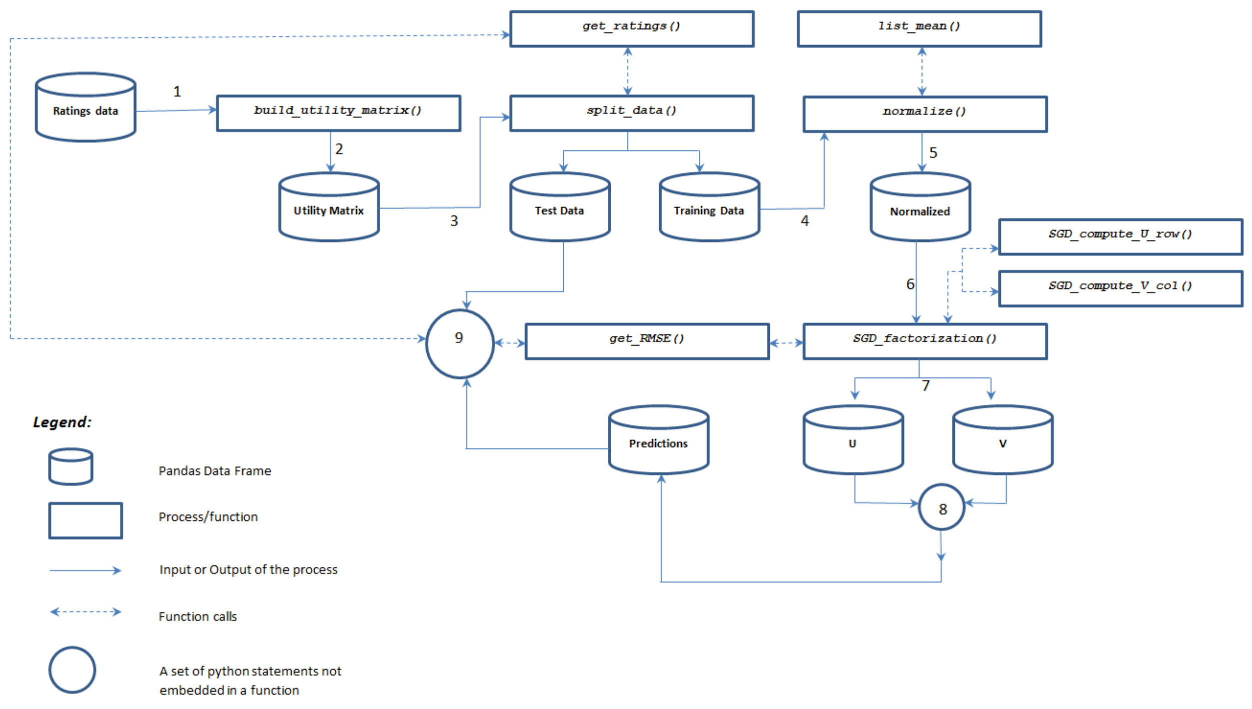
4.1 Benchmark model

The recommender system's performance is evaluated using the AUC of ROC curve, and it must be greater than 50% in order to outperform random guessing. So we will use the AUC of 50% as the basic model, and use this value as the minimal AUC value that our algorithm must outperform.

5.0 Process Design & Implementation

We will use the following process flow to implement our recommendation system using SGD algorithm in Python language.

Figure 3: SGD based recommender work-flow



Steps

1. The pandas data frame "Ratings data" (with columns: user_id, item_id, rating) is supplied as input to *build_utility_matrix()* function. The input data frame **must** have only 3 columns with the names:

user_id, item_id, rating representing the *user ID, item ID* and *rating* respectively. The *user ID* and *item ID* columns must be integers, and the *rating* column can be any real number.

2. *build_utility_matrix()* produces *Utility Matrix* as output.
3. *Utility Matrix* is supplied as input to *split_data()*, and *split_data()* produces two data frames: *Test* and *Training*, In this process, *split_data()* utilizes the function *get_ratings()*, which gets the rating from the utility matrix based on the column and row indices or column and row numbers. The splitting of the data happens randomly.
4. The training data frame is supplied as input to *normalize()* function. The *normalize()* function utilizes *list_mean()*, which computes the mean of the input list ignoring the NA values.
5. The *normalize()* function produces a normalized data frame using the training data.
6. The normalized data frame is supplied as input to *SGD_factorization()* function. The *SGD_function()* will make use of the functions *SGD_compute_U_row()*, *SGD_compute_V_col()* and *get_RMSE()*. The *SGD_compute_U_row()* will adjust the values of the U matrix's row, the *SGD_compute_V_col()* will adjust the V matrix's column, and the *get_RMSE()* will be used to compute the Root Mean Squared error (RMSE) between the normalized matrix and UV matrix product. Since the normalized input matrix is sparse, only the available elements in the input matrix will be used to compute the RMSE
7. The *SGD_factorization()* will produce U and V factors, which are optimized based on the SGD algorithm
8. The U and V matrices are multiplied to get the UV matrix. This UV matrix will have the same dimensions as the Utility matrix, but the UV matrix will have al the elements unlike the Utility matrix (which is a sparse matrix).
9. The test data, UV matrix are used to compute the test RMSE error

6.0 SGD implementation in python

We will code the following functions to compute U and V based on the SGD algorithm explained in the above section 3.1. The source code of these functions is implemented in in Python 2.7. All the functions are shown in *Figure-3* are given below:

build_utility_matrix(df)

The **build_utility_matrix(df)** function will create a Utility matrix. It accepts a pandas data frame *df* as input and the input data frame must have *"user_id","item_id","rating"* as the column names. The function will return a Utility matrix as output, with rows representing the *user_ids* and columns representing the *item_ids*.

Python code for build_utility_matrix()

```
def build_utility_matrix(df):

        ##Confine the columns to just user ID, Item ID, and rating only
        df_final = df[["user_id","item_id","rating"]]

        #Some users have rated the same item multiple times. So taking mean of such ratings
        df_final = df_final.groupby(["user_id","item_id"]).mean()
        df_final = df_final.reset_index()

        #Building the utility matrix
        Utility = df_final.pivot(index="user_id",columns="item_id",values="rating")
        Utility.columns.names=[" "]
        return Utility
```

get_ratings(Utility,row_idx,col_idx,indices=True)

The function **get_ratings(Utility,row_idx,col_idx,indices=True)** will obtain the rating present at the intersection of User-Item of the Utility matrix. The function will accept *Utility* matrix, *row_idx*, *col_idx* and a boolean variable *indices*. If *indices* is True, then *col_idx* and *row_idx* will be treated as the row name and column name. If *indices* is False, then *col_idx* and *row_idx* will be treated as row number and column number (row and column numbers begin at 0). The function will return the rating present at the intersection of *col_idx* and *row_idx*

Python code for get_ratings()

```
#Define a function that gets the ratings values at the
#intersection of row_idx and col_idx values, where
#row_idx and col_idx are lists

def get_ratings(Utility,row_idx,col_idx,indices=True):
    '''
       row_idx and col_idx are lists containing the indices of the utility matrix.
       If indices=True, then the row_idx and col_idx represent the actual index value
       and column name else, they represent the row number and column number respectively.

    '''
    ratings = list()

    if len(row_idx) == len(col_idx):
        if indices:
            for i in xrange(len(col_idx)):
                ratings.append(Utility.loc[row_idx[i],col_idx[i]])
        else:
            for i in xrange(len(col_idx)):
                ratings.append(Utility.iloc[row_idx[i],col_idx[i]])
        return ratings
    else:
        print "Error. The lengths of the row and col locations must be same"
```

split_data(Utility,test_perc=20)

The Utility matrix will be split into test and training data (20:80 for test:training respectively, by default). The **split_data(Utility,test_perc=20)** function will perform this split. If only one rating is available for an item, the function must eliminate such item's rating from test data. The split must be random. The function takes *Utility* matrix as the input, along with the desired percentage of test data, and returns the Training data (another Utility matrix with some of the ratings replaced with NA). The ratings, which were replaced by NA values will be used as test data.

Python code for split_data()

```
def split_data(Utility,test_perc=20):
        #What number makes the 20% of the ratings?
        #Find the cell locations where there is a true rating.
        rows,cols=np.where(~np.isnan(Utility))

        #What percentage of cells have the true rating?
        nan_perc = 100-100*float(len(cols))/(Utility.shape[0]*Utility.shape[1])
        non_nan_perc = 100*float(len(cols))/(Utility.shape[0]*Utility.shape[1])

        #Find the number of observations needed for the test data
        test_number = np.trunc(test_perc / 100.0 * len(cols))

        #Find the locations (row,col) in the Utility matrix
        #wherever we have a genuine value. The ratings_locations data frame (defined below)
        #will have these (row-column) details.
        ratings_locations = pd.DataFrame(zip(rows,cols),columns = ["row","column"])

        #Get the column locations which have at least 2 ratings.
        #This will make sure that we do not accidentally select a
        #rating (which is the only rating available for the item or row)
        ratings_counts = ratings_locations.groupby(["column"])['row'].count()
        #print ratings_counts

        #display(ratings_counts)
        cols_num_2_ratings = list(ratings_counts[ratings_counts>1].index)
        test_row_num = list()
        test_col_num = list()

        sample_count = 0

        train_df = Utility.copy()
        for i in cols_num_2_ratings:
            #The random choice is important. It helps us to randomly
            #select the row location so that we do not select the value from the
            #same row location.

            #Get the location of the row and columns into variables
            temp_test_row = np.random.choice(np.where(~np.isnan(Utility.iloc[:,i]))[0])
            temp_test_col = i

            #Keep a track of the sample size
            sample_count = sample_count + 1

            #If a user-item pair is selected for test, make that value as NA in training data
            #However, there is a chance that the whole row could be NA, after this operation,
            #since a column might be already selected and made as NA.
            #So save the current value in temp_value, and
            #re-assign to the same row-column value in train_df, if the operation
            #results in the whole row filled with NA.
            temp_value=train_df.iloc[temp_test_row,temp_test_col]
            train_df.iloc[temp_test_row,temp_test_col] = np.nan
            #Undo the change in training data, if the change results in all NAs in the row
            if np.isnan(train_df.iloc[temp_test_row,:]).all():
                sample_count = sample_count - 1
                train_df.iloc[temp_test_row,temp_test_col] = temp_value
                continue
            test_row_num.append(temp_test_row)
            test_col_num.append(temp_test_col)

            #Stop data selection once we obtain the desired number of samples
            if sample_count >= test_number:
                break


        ##Get the ratings at the intersection of row and column numbers from the
        ##Utility matrix
        test_ratings=get_ratings(Utility,test_row_num,test_col_num,indices=False)

        ##Prepare the test data frame
        user_id = [Utility.index[i] for i in test_row_num]
        item_id = [Utility.columns[i] for i in test_col_num]

        test_df = pd.DataFrame(zip(test_row_num,test_col_num,
                          user_id,
                          item_id,
                          test_ratings),columns=["row_number","column_number",
                                                 "user_id","item_id","rating"])

        return [train_df,test_df]
```

list_mean(l)

Will get the mean of the list of a numpy array, eliminating the NA elements.

Python code for list_mean()

```
def list_mean(l):
    if np.isnan(l).all():
        print "list_mean() message: All NA values. Check the logic."
    return np.nanmean(l)
```

normalize(M)

The function **normalize(M)** will normalize the input data (Utility Matrix M). It returns a normalized numpy array, average ratings of each item, and average rating given by each user. The items and users averages will help us to denormalize the ratings to the actual ratings. We will define a function *normalize(M)* that performs the normalization of Utility matrix M based on the following logic. Normalization helps us to eliminate user and item biases.

1. For each row, get the respective mean (ignoring the NANs).
2. Subtract the means obtained in step-1 from the respective rows

3. Get the means of the columns of the modified matrix from step-2
4. Subtract the column means from the respective columns of the modified matrix, obtained in step-2

Python code for normalize()

```
##Normalize the Utility matrix:
##Subtract the avg user rating and avg item rating from the item in M


def normalize(M):
    #Convert M to a numpy array
    #print "In normalize"
    M = np.array(M)

    #Get the column means (or items mean)
    items_mean=np.apply_along_axis(list_mean,0,M)

    #If an item is NOT rated by any user, then we will get NA for mean
    #So for such instances
    if np.isnan(np.sum(items_mean)):
        print "WARNING: Items has NAN values. which is incorrect"

    #Subtract the columns mean from the respective columns
    M_normalized = M[:,] - items_mean

    #Get the rows means (or users mean) using the partially normalized matrix
    users_mean = np.apply_along_axis(list_mean,1,M)
    if np.isnan(np.sum(users_mean)):
        print "WARNING: Users has NAN values. which is incorrect"

    #Subtract the rows means
    M_normalized = (M_normalized[:,].T- users_mean)

    #Transform back and return
    return [M_normalized.T, items_mean, users_mean]
```

SGD_compute_U_row(i,M,U,V,d,lambda1,alpha)

This function will accept 7 variables as inputs and computes the gradient for a specific row in U, while keeping all other U and V elements constant. The parameters details are given below:

i = desired row number in U that needs to be estimated (row numbers begin from 0)

M = Utility matrix, which needs to be factorized

U = U matrix or factor in the expression M = UV

V = V matrix or factor in the expression M = UV

d = desired number of latent factors or columns in U

lambda1 = regularization parameter

alpha = learning rate

Python code for SGD_compute_U_row()

```python
def SGD_compute_U_row(i,M,U,V,d,lambda1,alpha):
    '''
     i = desired row number in U, which needs to be computed
     M = Utility matrix of size mXn
     V = V component of size dXn
     lambda1 = reg factor
    '''
    num_of_rows, num_of_cols = M.shape

    #np.where will return a tuple.
    C = np.where(~np.isnan(M[i,]))[0]

    V_C = V[:,C].copy()
    M_i = M[i,:].copy()
    M_i = M_i[~np.isnan(M_i)]
    return alpha*(-1*np.dot((M_i - np.dot(U[i,],V_C)),V_C.T) + lambda1 * U[i,])
```

SGD_compute_V_col(j,M,U,V,d,lambda2,alpha)

This function will also accept 7 variables as inputs, and computes the gradient for a specific column in V, while keeping all other U and V elements constant. The parameters details are given below:

j = desired column number in V that needs to be estimated (column numbers begin from 0)

M = Utility matrix, which needs to be factorized

U = U matrix or factor in the expression M = UV

V = V matrix or factor in the expression M = UV

d = desired number of latent factors or rows in V

lambda2 = regularization parameter

alpha = learning rate

Python code for SGD_compute_V_col()

```python
def SGD_compute_V_col(j,M,U,V,d,lambda2,alpha):
    '''
     j = desired column number in V, which needs to be computed
     M = Utility matrix of size mXn
     U = U component of size mXd
     lambda2 = reg factor
    '''
    num_of_rows, num_of_cols = M.shape

    #np.where will return a tuple.
    R = np.where(~np.isnan(M[:,j]))[0]

    U_R = U[R,:].copy()
    M_j = M[:,j].copy()
    M_j = M_j[~np.isnan(M_j)]
    return alpha*(-1*np.dot((M_j - np.dot(U_R,V[:,j])).T,U_R) + lambda2 * V[:,j])
```

get_RMSE_error(M,U,V)

This function will compute the RMSE between M and UV, considering only the available elements in M. It takes 3 parameters M, U and V. M is the Utility matrix, U is the U component and V is the V component in M = UV.

Python code for get_RMSE_error()

```
def get_RMSE(M,U,V):
    return np.sqrt(np.nanmean(np.square(M - np.dot(U,V))))
```

SGD_factorization(M,d, lambda1, lambda2, n, error_diff, seed,alpha)

In SGD we will estimate the row of U or column of V by choosing randomly U or V and choosing the row from U or column from V randomly. The randomness in choosing the elements is important. The function SGD_factorization function will accept 8 parameters. The first parameter is M (utility matrix), the second parameter is *d* (the desired number of latent factors), the third parameter is *lambda1* (regularization factor for computing U's row), *lambda2* (regularization factor for computing V's column), *n* is the maximum number of iterations, and *error_diff* is the least acceptable error difference between the consecutive iterations, *seed* will accept a number which can be used to reproduce the results, and *alpha* is the learning rate. The algorithm stops updating the elements of U and V once the specified number of iterations is reached or when the error difference between consecutive iterations is less than or equal to *error_diff*

Python code for SGD_factorization()

```
def SGD_factorization(M,d, lambda1, lambda2, n, error_diff, seed,alpha):
    import numpy as np
    #Initialize U and V
    M_rows, M_cols = M.shape
    M_non_nan = np.nan_to_num(M)

    U, s, vt = svds(M_non_nan, k=d, ncv=None, tol=0, which='LM',
                    v0=None, maxiter=None, return_singular_vectors=True)
    V = np.dot(np.diag(s),vt)
    error = []
    error.append(get_RMSE(M,U,V))

    for count in xrange(n):
            #Do not use any seed here. Let it stay random.
            #In future enhancements, we will use seed parm
            pick_1 = np.random.randint(0,2)
            if pick_1 == 0:
                pick_2 = np.random.randint(0,M_rows)
                U[pick_2,] = U[pick_2,] - SGD_compute_U_row(pick_2,M,U,V,d,lambda1,alpha)
                error.append(get_RMSE(M,U,V))
                if np.absolute(error[-2] - error[-1]) <= error_diff:
                    return [U, V, error]
            else:
                pick_2 = np.random.randint(0,M_cols)
                V[:,pick_2] = V[:,pick_2] - SGD_compute_V_col(pick_2,M,U,V,d,lambda2,alpha)
                error.append(get_RMSE(M,U,V))
                if np.absolute(error[-2] - error[-1]) <= error_diff:
                    return [U, V, error]
    return [U, V, error]
```

7.0 Tuning SGD algorithm's parameters

For SGD algorithm, we have the following hyper-parameters to tune:

- λ_1: The regularization parameter while finding U matrix
- λ_2: The regularization parameter while finding V matrix
- d: Desired number of latent factors
- n: Number of users to select. This indirectly controls the volume of the training data
- In our evaluation, we will use the same values for λ_1 and λ_2. We will use the following values for these parameters:

$$\lambda_{1,2} = [0.1, 1, 2, 3, 4, 5, 10, 100]$$

- We will select n as 59132. We have a maximum of 59132 users in the dataset
- Out of the n users ratings, 80% of data will be used for training and 20% for testing. The training data and test data are randomly selected
- The value of d (latent factors) will have the following values:

$$d = [2, 5, 10, 15, 20, 25, 30, 35, 40, 50, 60, 80, 100]$$

- We will use a constant learning rate of 0.00001
- For each parameter combination, we will get the test and training error
- The experiment is repeated 5 times. For each iteration, we select 80% of the data for training and 20% of the data for testing (randomly)
- The average of test and training errors for all the parameters combinations in all the iterations are obtained. The least average test error is finally picked as the optimal set of parameters to implement the SGD algorithm for the Jokes dataset

Python code for training SGD

```
#Rename the ratings_df columns, since the build_utility() function requires these column names:
#["user_id","item_id","rating"]

ratings_df.columns = ["user_id","item_id","rating"]

#Initialize the lambda1 values
#The same values will be used for lambda2 also
lambda1 = [0.1,1,2,3,4,5,10,20,30,40,100]

#Select the ratings data based on the
#following number of users
total_users = [59132]

#lambda1 = [1,2]
#total_users = [20,30]
```

```
#Learning rate for SGD
alpha = 0.00001
#alpha = [0.01, 0.001, 0.0001, 0.00001, 0.0000001,0.000000001]

#Error tolerance
#Maximum error acceptable to terminate the iterations
error_diff = 0.0000000001

#Seed to reproduce the results
seed = 10

#Place holders for metrics
reg = []
num_of_users = []
latent_factors = []
train_error = []
test_error = []
run_time = []
algorithm = []
iterations = []
LR = []
#Maximum number of iterations
n = 1000
from time import time

df = ratings_df.copy()
Utility = build_utility_matrix(df)
train_df,test_df=split_data(Utility,test_perc=20)
epoch = list()
for episode in [1,2,3,4,5]:
    print "Training episode: {}".format(episode)
    train_normalized,train_items_mean,train_users_mean = normalize(train_df)
    train_users=train_df.shape[0]
    for a in lambda1:
        #print "evaluating lambda={} iteration".format(a)
        c = [2,5,10,15,20,25,30,35,40,50,60,80,100]
        for d in c:
            #Running SGD for the given parameters combination
            #for rate in alpha:
                #LR.append(rate)
                algorithm.append("SGD")
                reg.append(a)
                num_of_users.append(b)
                latent_factors.append(d)
                start = time() # Get start time
                U, V, error = SGD_factorization(np.array(train_normalized),d,
                                                a, a, n, error_diff, seed,alpha)
                end = time() # Get end time
                train_error.append(error[-1])
                iterations.append(len(error))
                run_time.append(end-start)

                temp_UV = np.dot(U,V) + train_items_mean
                temp_UV = temp_UV.T + train_users_mean
                temp_UV = temp_UV.T
                predicted_ratings = get_ratings(pd.DataFrame(temp_UV),
                                                list(test_df["row_number"]),
                                                list(test_df["column_number"]),indices=False)
                test_error.append(np.sqrt((np.nanmean(np.square(
                            np.array(test_df["rating"]) - np.array(predicted_ratings))))))
                epoch.append(episode)


performance_df = pd.DataFrame(zip(algorithm,reg,num_of_users,latent_factors,run_time,
                          train_error,test_error,iterations,epoch),
                         columns=['algorithm','reg','num_of_users','latent_factors',
                        'run_time','train_error','test_error','iterations','epoch'])

#Writing the performance metrics to a file, so that we do not have to
#run the above block again
performance_df.to_csv("performance_df.csv")
```

The above code will output the following:

```
Training episode: 1
Training episode: 2
Training episode: 3
Training episode: 4
Training episode: 5
```

NOTE: Do NOT execute the above code, since this code will run for a while (although you are welcome to test). It ran for 1 hour on a computer with 16GB RAM. To save time, I saved the performance metrics of various parameters combinations in a file named “performance_df.csv”. This file can be downloaded from the Github location: https://goo.gl/mEE6G6

Reading the performance metrics file and displaying some of its contents

```
performance_df = pd.read_csv("performance_df.csv")
#display(performance_df.sort(["test_error"]))

display_df =
  performance_df.groupby(
    ['algorithm','reg','num_of_users','latent_factors'])
    .mean()
    .reset_index()
    .sort("test_error")[['algorithm','reg','latent_factors',
                         'run_time','train_error','test_error',
                         'iterations']]
display_df.columns = ["Algorithm","Reg",
                      "Latent_Factors",
                      "Avg_Run_Time",
                      "Avg_Training_Error",
                      "Avg_Test_Error",
                      "Avg_Iterations"]
```

```
print "Average performance metrics obtained for various parameters combinations."
print "\nThe metrics are sorted in the increasing order of average TEST error"
print "\nDisplaying the top rows only."
display(display_df.head(10))


print "Displaying the performance metrics sorted in the increasing order of TRAINING error:"
display(display_df.sort(["Avg_Training_Error"]).head(5))
```

Output of the above code block (The runtimes are measured in seconds):

```
Average performance metrics obtained for various parameters combinations.

The metrics are sorted in the increasing order of average TEST error

Displaying the top rows only.
```

	Algorithm	Reg	Latent_Factors	Avg_Run_Time	Avg_Training_Error	Avg_Test_Error	Avg_Iterations
23	SGD	1.0	60	3.2880	1.890782	3.695142	7.0
140	SGD	100.0	60	6.5704	1.890789	3.695788	23.0
127	SGD	40.0	60	4.5240	1.890779	3.695791	13.2
114	SGD	30.0	60	4.4480	1.890782	3.695791	12.6
101	SGD	20.0	60	6.2584	1.890778	3.695793	21.2
49	SGD	3.0	60	4.4800	1.890769	3.695793	13.0
36	SGD	2.0	60	4.6008	1.890782	3.695793	13.4
10	SGD	0.1	60	3.0220	1.890780	3.695793	5.8
75	SGD	5.0	60	4.6580	1.890803	3.695793	13.2
62	SGD	4.0	60	3.1560	1.890777	3.695793	6.2

```
Displaying the performance metrics sorted in the increasing order of TRAINING error:
```

	Algorithm	Reg	Latent_Factors	Avg_Run_Time	Avg_Training_Error	Avg_Test_Error	Avg_Iterations
129	SGD	40.0	100	2.9224	1.009886	3.739950	6.4
64	SGD	4.0	100	2.9700	1.009887	3.739955	6.2
25	SGD	1.0	100	2.2340	1.009889	3.739955	3.4
77	SGD	5.0	100	3.5940	1.009889	3.739955	9.6
51	SGD	3.0	100	3.8440	1.009889	3.739955	10.4

From the above display, we can infer the following:

- The optimal number of latent factors is 60 since the average test error is the least for 60 latent factors.
- The Regularization (Reg) parameter has no effect on the average test error since the test error is almost the same for all the regularization parameters, and with 60 latent factors.
- The average training error for 100 latent factors is minimum, but the average test error is not the minimum for 100 latent factors. This suggests that overfitting is happening at 100 latent factors

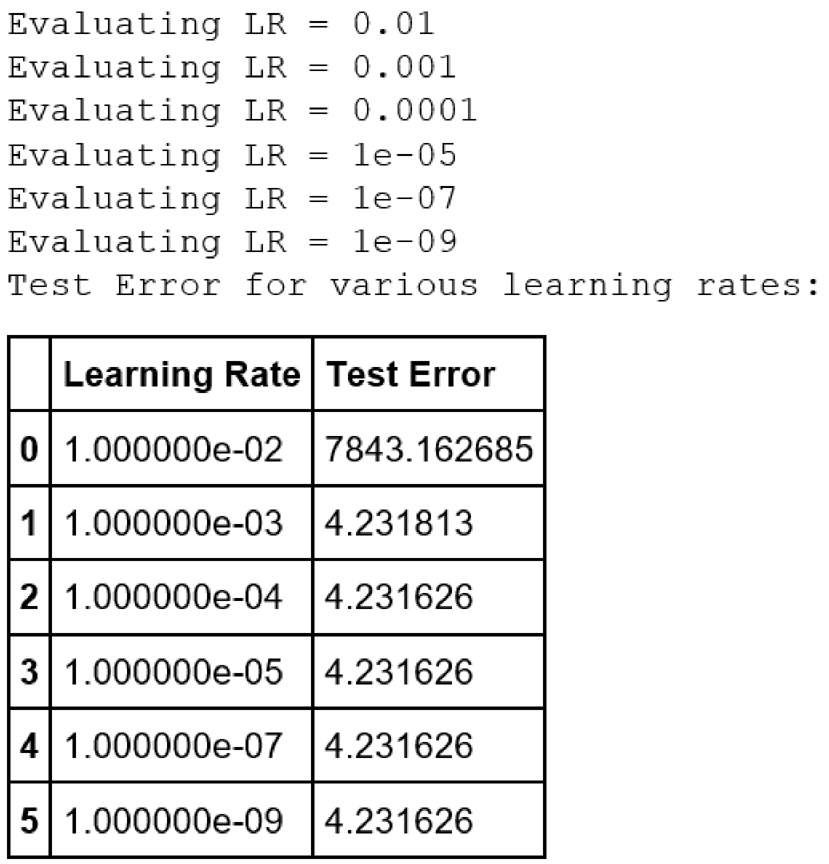
Since Reg value has no significant effect on the test error, we will choose the following parameters combination as the optimal combination:

$$Reg = 1, LatentFactors = 60, Learningrate = 0.00001$$

Note that we did **not** vary the learning rate while gathering the performance metrics. Let us check *how* the learning rate effects the test error. The following code block will use the [0.01, 0.001, 0.0001, 0.00001, 0.0000001,0.000000001] values as the learning rate, along with the above parameter values, and computes the test error for each learning rate.

```
train_error_1 = list()
test_error_1 = list()
iterations_1 = list()
ratings_df.columns = ["user_id","item_id","rating"]
df = ratings_df.copy()
Utility = build_utility_matrix(df)
train_df,test_df=split_data(Utility,test_perc=20)
#train_users_1=train_df.shape[0]
train_normalized,train_items_mean,train_users_mean = normalize(train_df)

#print train_users

#SGD_factorization(np.array(train_normalized),d, a, a, n, error_diff, seed,alpha)
alpha_1 = [0.01, 0.001, 0.0001, 0.00001, 0.0000001,0.000000001]

for rate in alpha_1:
    print "Evaluating LR = {}".format(rate)
    U, V, error = SGD_factorization(np.array(train_normalized),60,
                                    1, 1, 1000, 0.0000000001, 10,rate)
    #print error[-1]

    train_error_1.append(error[-1])
    iterations_1.append(len(error))
    temp_UV = np.dot(U,V) + train_items_mean
    temp_UV = temp_UV.T + train_users_mean
    temp_UV = temp_UV.T

    predicted_ratings = get_ratings(pd.DataFrame(temp_UV),
                                    list(test_df["row_number"]),
                                    list(test_df["column_number"]),
                                    indices=False)
    test_error_1.append(np.sqrt(np.nanmean(np.square(np.array(test_df["rating"])
                                                     - np.array(predicted_ratings)))))

print "Test Error for various learning rates:"
pd.DataFrame(zip(alpha_1,test_error_1),columns = ["Learning Rate", "Test Error"])
```

Output of the above code block:



```
Evaluating LR = 0.01
Evaluating LR = 0.001
Evaluating LR = 0.0001
Evaluating LR = 1e-05
Evaluating LR = 1e-07
Evaluating LR = 1e-09
Test Error for various learning rates:
```

	Learning Rate	Test Error
0	1.000000e-02	7843.162685
1	1.000000e-03	4.231813
2	1.000000e-04	4.231626
3	1.000000e-05	4.231626
4	1.000000e-07	4.231626
5	1.000000e-09	4.231626

The learning rate of 0.01 is not optimal. But any other learning rate listed in the above display is optimal, although we stick to the learning rate value of 0.00001, which was used to evaluate the algorithm's performance for various parameters combinations. Also, we will select the learning rate which is not too slow (since the convergence will take a long time) or too fast, since the overshooting may occur. In the above display, we obtained a huge test error of 7843, because of overshooting in the gradient descent algorithm.

8.0 Training the best model and evaluating the ROC Area

We will use the following parameters to train the SGD model and plot the ROC (Receiver Operating Characteristic) curves to identify if the SGD model's performance is acceptable. These are the parameters which were identified based on the average test error in 5 different runs of the SGD algorithm using different sets of randomly selected training data.

$$Reg = 1, LatentFactors = 60, Learningrate = 0.00001$$

If the AUC (Area Under the ROC Curve) is less than or equal to 0.5, then our model is not a good model and our model is no better than just random guessing.

Python code to build the recommendation model with optimal hyper-parameters

```
#Train the model with the optimal parameter combinations identified
train_df,test_df=split_data(Utility,test_perc=20)
train_normalized,train_items_mean,train_users_mean = normalize(train_df)


U, V, error = SGD_factorization(np.array(train_normalized),60, 1, 1,
                                1000, 0.0000000001, 10,0.00001)
#In the above call, we used 60 latent factors, regularization parms as 1, 1
#0.0000000001 is the error tolerance (stop the iteration if
#the consecutive errors difference is less than
# or equal to 0.0000000001)
#Learning rate is 0.00001

pred = np.dot(U,V) + train_items_mean
pred = pred.T + train_users_mean
pred = pred.T
# pred will have the same shape as the utility matrix.
# But pred matrix will have an estimate of the missing
# entries in the Utility matrix
```

8.1 Getting the test data predictions SGD

Whenever our model predicts a rating of more than 10 or less than -10, we change that rating to 10 and -10 respectively. Also we will normalize the predicted ratings using the following formula:

$$x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Since our ratings always belong to $[-10, 10]$, we can rewrite the above formula as:

$$x_{new} = \frac{x - (-10)}{10 - (-10)} = \frac{x + 10}{20}$$

The main advantage of this scaling is that all the new ratings will have a value between $[0, 1]$. These new values can be interpreted as the probability that a user likes the movie, and this interpretation will help us to plot the ROC curve for our model.

The following code block will get the predicted ratings for the test data using SGD method.

Python code to normalize the predictions

```
#Get the predicted ratings for the test data
SGD_predicted_ratings = get_ratings(pd.DataFrame(pred),list(test_df["row_number"]),
                                   list(test_df["column_number"]),indices=False)

SGD_predicted_ratings = np.array(SGD_predicted_ratings)

#If any predicted rating is more than 10, then make that rating as 10
```

```
SGD_predicted_ratings[SGD_predicted_ratings > 10] = 10

#If any predicted rating is less than -10, then make that rating as -10
SGD_predicted_ratings[SGD_predicted_ratings < -10] = -10

#Normalize the rating to [0,1] interval
SGD_predicted_prob = (SGD_predicted_ratings+10)/20
```

Using the below code block, we add the predicted ratings and probabilities to the test data to the test data frame. The code also displays a sample set of records from the test data frame, after adding the additional columns.

```
#Add the predicted ratings and probabilities (normalized ratings) to the test data frame
test_df["SGD_predicted_ratings"] = SGD_predicted_ratings
test_df["SGD_predicted_prob"] = SGD_predicted_prob
test_df.head()
```

Output of the above code block:

	row_number	column_number	user_id	item_id	rating	SGD_predicted_ratings	SGD_predicted_prob
0	48	0	55	5	-6.656	-6.728410	0.163579
1	39437	1	42402	7	-6.938	-7.884690	0.105765
2	2106	2	2281	8	-5.594	1.388690	0.569434
3	40590	3	43612	13	0.188	-0.638931	0.468053
4	42453	4	45579	15	-6.656	0.787004	0.539350

8.2 Change ratings prediction to a classification problem

To change the ratings prediction problem as a binary classification (like/dislike) problem, we assume that a user likes a joke if he gives a rating of more than 5, and dislikes the joke, if he rates the joke less than or equal to 5. Based on this assumption, we will add a new column "actually_liked" to the test data. This column will have a 1 if the user has given more than 5 rating to a joke, else this column will have 0.

The following code block will add another column to the test data frame, and the new column will represent if the system predicts that a user in the test data likes or dislikes an item. The code also displays initial rows of test data frame, after adding the boolean column.

```
#Make a copy of the rating
actually_liked = test_df["rating"].copy()

#Encode the probability score as 0/1 to
#represent the prediction that the user really likes or dislikes
#1 - if the user likes the joke
#0 - if the user dislikes the joke
actually_liked[actually_liked <= 5] = 0
actually_liked[actually_liked > 5] = 1
test_df["actually_liked"]=actually_liked
```

```
print "Diplaying a set of initial rows from the test data:"
test_df.head()
```

Output of the above code block:

	row_number	column_number	user_id	item_id	rating	SGD_predicted_ratings	SGD_predicted_prob	actually_liked
0	48	0	55	5	-6.656	-6.728410	0.163579	0.0
1	39437	1	42402	7	-6.938	-7.884690	0.105765	0.0
2	2106	2	2281	8	-5.594	1.388690	0.569434	0.0
3	40590	3	43612	13	0.188	-0.638931	0.468053	0.0
4	42453	4	45579	15	-6.656	0.787004	0.539350	0.0

8.3 AUC (Area Under the Curve) measure

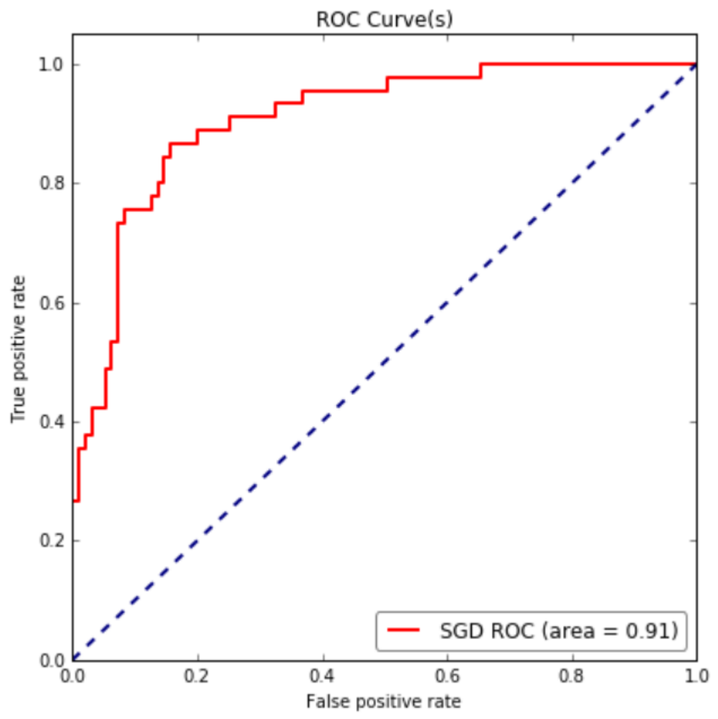
Using the predicted classification of all jokes for each user in the test data, we will plot a ROC (Receiver Operating Characteristics) curve, and estimate the AUC under the curve. The following code will plot the ROC curve for our predictive model.

```
#Let us create a function to plot the ROC curves
def plot_roc_curves(fpr,tpr,ax,models,colors=["red","green",
                                              "darkorange","green",
                                              "black","magenta","cyan"]):

    import matplotlib.pyplot as plt
    from sklearn.metrics import auc
    lw=2 #Line weight
    for key in range(len(fpr)):
        line1, = ax.plot(fpr[key], tpr[key], linewidth=2,color=colors[key],
                 label=models[key]+' ROC (area = %0.2f)' % auc(fpr[key],tpr[key]))
    line2,=ax.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
    ax.legend(loc='lower right',fancybox=True, framealpha=0.5)

    ax.set_ylabel('True positive rate')
    ax.set_xlabel('False positive rate')
    ax.set_title('ROC Curve(s)')
    ax.set_xlim([0.0, 1.0])
    ax.set_ylim([0.0, 1.05])

    return ax



#Import the required packages
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

#Dictionary to collect the False Positive Rates at various probabilities thresholds
fpr = dict()

#Dictionary to collect the True Positive Rates at various probabilities thresholds
```

```
tpr = dict()

#Dictionary to collect the probability thresholds used to compute the TPR and FPR
thresholds=dict()

#Get the probabilities that the target class=1
SGD_clf_test_scores=np.array(test_df["SGD_predicted_prob"])

#Get the FPR, TPR, thresholds used for unoptimized classifier
fpr[0], tpr[0], thresholds[0] = roc_curve(test_df["actually_liked"],
                                          y_score=SGD_clf_test_scores,
                                          pos_label=1)
print "SGD Area Under the Curve:" + str(auc(fpr[0],tpr[0]))

fig, ax = plt.subplots(figsize=(6,6))
#fig.suptitle('Categorical variables bar plots')
models=["SGD"]
plot_roc_curves(fpr,tpr,ax,models)
plt.tight_layout()
```

Output of the above code block:

```
SGD Area Under the Curve:0.907134502924
```



8.4 Finding the optimal threshold

Using the test data, we obtained an AUC of 0.91, which is a very good score. We used a rating threshold of 5, to determine if a user will like a joke. But how do we know that this threshold is the optimal threshold? The AUC of 0.91 might have resulted due to pure randomness in the selection of test data, and it is not possible to confirm that 5 is the optimal threshold with just 1 instance of AUC score.

We will find the ROC using various thresholds ranging from [4, 10] (in the increments of 0.1), for various random selections of the training data, and obtain the average ROC. We will select the threshold, for which the average ROC value is the maximum.

The following code will compute the AUC for various thresholds for 5 times and obtains the average AUC for each threshold. This code will run for approximately 1 hour. So, in order to save the time, we saved the results of this code to a file called roc_analysis.csv, which can be downloaded from https://goo.gl/mEE6G6.

```
from time import time
threshold = np.arange(4,10.0,0.1)
thres = []
roc = []
epoch = []
ratings_df.columns = ["user_id","item_id","rating"]
df = ratings_df.copy()
Utility = build_utility_matrix(df)
for episode in [1,2,3,4,5]:
#for episode in [1]:
    start = time()
    print "Starting episode: {}".format(episode)

    for t in threshold:
        train_df,test_df=split_data(Utility,test_perc=20)
        train_normalized,train_items_mean,train_users_mean = normalize(train_df)

        U, V, error = SGD_factorization(np.array(train_normalized),60, 1, 1,
                                         1000, 0.0000000001, 10,0.00001)

        pred = np.dot(U,V) + train_items_mean
        pred = pred.T + train_users_mean
        pred = pred.T
        SGD_predicted_ratings = get_ratings(pd.DataFrame(pred),list(test_df["row_number"]),
                                            list(test_df["column_number"]),indices=False)
        if (np.isnan(SGD_predicted_ratings).any()):
            print SGD_predicted_ratings

        SGD_predicted_ratings = np.array(SGD_predicted_ratings)
        SGD_predicted_ratings[SGD_predicted_ratings > 10] = 10
        SGD_predicted_ratings[SGD_predicted_ratings < -10] = -10
        SGD_predicted_prob = (SGD_predicted_ratings+10)/20
        test_df["SGD_predicted_ratings"] = SGD_predicted_ratings
        test_df["SGD_predicted_prob"] = SGD_predicted_prob
        actually_liked = test_df["rating"].copy()
        #print len(actually_liked[actually_liked > t])
        #print len(actually_liked[actually_liked <= t])
        if ((len(actually_liked[actually_liked > t]) == 0)
            | (len(actually_liked[actually_liked <= t]) == 0)):
                continue
        actually_liked[actually_liked <= t] = 0
        actually_liked[actually_liked > t] = 1
        test_df["actually_liked"]=actually_liked
        #Dictionary to collect the False Positive Rates at
        #various probabilities thresholds
        fpr = dict()
        #Dictionary to collect the True Positive Rates at
        #various probabilities thresholds
        tpr = dict()

        #Dictionary to collect the probability thresholds
        #used to compute the TPR and FPR
        thresholds=dict()
        SGD_clf_test_scores=np.array(test_df["SGD_predicted_prob"])
        if (np.isnan(SGD_clf_test_scores).any()):
            print "SGD_clf_test_scores contains NAN: {}".format(SGD_clf_test_scores)
            print test_df

        fpr[0], tpr[0], thresholds[0] = roc_curve(test_df["actually_liked"],
                                                  y_score=SGD_clf_test_scores, pos_label=1)

        thres.append(t)
        roc.append(auc(fpr[0],tpr[0]))
        epoch.append(episode)
    end = time()
    print "Completed episode: {}".format(episode)
    print "Processing time for episode: {} is {} seconds".format(episode,end-start)

#Save the results to a file
roc_analysis = pd.DataFrame(zip(thres, roc, epoch),columns=["threshold","auc","epoch"])
roc_analysis.to_csv("roc_analysis.csv")
```

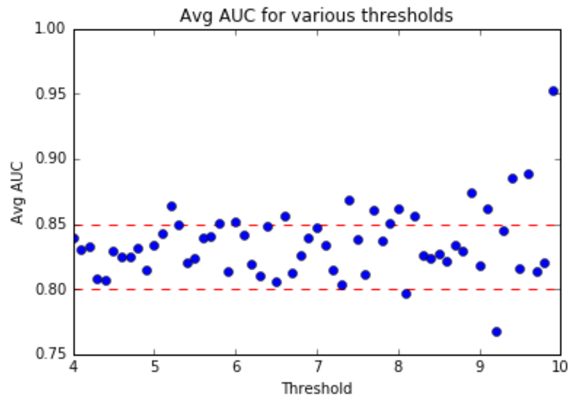
The output of the above code block is given below:

```
Starting episode: 1
Completed episode: 1
Processing time for episode: 1 is 480.348999977 seconds
Starting episode: 2
Completed episode: 2
Processing time for episode: 2 is 496.98300004 seconds
Starting episode: 3
Completed episode: 3
Processing time for episode: 3 is 494.321000099 seconds
Starting episode: 4
Completed episode: 4
Processing time for episode: 4 is 495.914000034 seconds
Starting episode: 5
Completed episode: 5
Processing time for episode: 5 is 506.942000151 seconds
```

The following code block will read the file, which was created in the above code block, and plots the average AUC values for different thresholds:

```
import pandas as pd

import matplotlib.pyplot as plt
import numpy as np

#Read the file roc_analysis.csv, which was created in the above block
roc_analysis = pd.read_csv("roc_analysis.csv")
roc_analysis = roc_analysis[["threshold","auc"]]
display_df=roc_analysis.groupby(["threshold"]).mean().reset_index().sort(["auc"],ascending=False)


x = display_df["threshold"]
y = display_df["auc"]

fig, ax = plt.subplots()
#ax.fmt_ydata = millions
plt.title("Avg AUC for various thresholds")

plt.plot((4, 10), (0.8, 0.8), '--',color="red")
plt.plot((4, 10), (0.85, 0.85), '--',color="red")
ax.set_xlabel("Threshold")
ax.set_ylabel("Avg AUC")
plt.plot(x, y, 'o')

#print "\nAvg. AUC values for various thresholds"
plt.show()
```

Output of the above code block:



From the above plot, we can infer the following:

- For most of the thresholds, the average AUC values lie between 0.8 and 0.85.
- As the threshold value increases, the variance of the average AUC values increases.

- The highest AUC value obtained is 0.95, while the least AUC value obtained is 0.77, and both occurred for a threshold value of more than 9.
- Most of the higher AUC values have occurred for a threshold of 7.5 or more.
- For all the thresholds, the average AUC values are way above 0.5, and this suggests that our recommender is doing a better job than pure random guessing that a user likes an item.

Even though most of the higher AUC values have occurred for a threshold of 7.5 or more, we do not want to use the 7.5 as the threshold, since we will end up setting the bar high, and the number of recommended items will be less. But setting the threshold to a value of 4 will qualify many items as the recommended items for a user, and may dilute the quality of the recommendations. So, let us choose the threshold value of 5 to determine if a user likes the item. If the predicted rating of an item for a user is more than 5, recommend the item to the user, else do not recommend.

9.0 Recommending jokes based on the predicted ratings by SGD method

We will create a function, that takes the user_ID as input and returns the items recommended to the user ID. We will also provide the predicted ratings matrix as input, along with the ratings data frame (containing all the ratings), and the Utility matrix produced using the ratings data frame. The ratings data frame will help the function to get the list of items which were already rated by the user.

Training the algorithm using the complete ratings data.

```
#train_df,test_df=split_data(Utility,test_perc=20)
Utility_normalized,Utility_items_mean,Utility_users_mean = normalize(Utility)
U, V, error = SGD_factorization(np.array(Utility_normalized),60, 1, 1, 1000,
                                0.0000000001, 10,0.00001)
pred = np.dot(U,V) + Utility_items_mean
pred = pred.T + Utility_users_mean
pred = pred.T
```

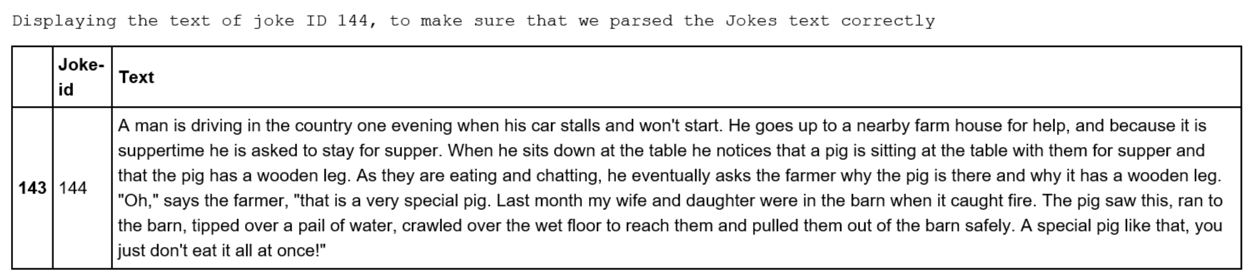
9.1 Getting the jokes text

The following code will process the *jester_items.dat*, extracts the joke IDs and the jokes text, and outputs a data frame called *items_df*. This data frame will be referenced to extract the Jokes text, based on a joke ID.

```
#Use bs4 to extract the text from HTML documents
from bs4 import BeautifulSoup

with open("jester_items.dat", 'r') as file :
    filedata = file.read()

soup = BeautifulSoup(filedata)
text = soup.get_text()

#Output the text to a file.
with open("output.txt", "wb") as outfile:
    outfile.write(text)

#Define a list "l"
l = list()
for line in open(r'output.txt'):
    #if line != " ":
        l.append(line.strip(":\n"))
#print l

#Weed out empty elements from the list
l = [i for i in l if i != '']

## Define a function that checks if a string has a number
def RepresentsInt(s):
    try:
        int(s)
        return True
    except ValueError:
        return False
#List object to accumulate the joke IDs
```

```
id = list()

##List object to accumulate the jokes text
text = list()

#Temporary string object to append jokes text, since
#A joke can be split into more than one elements in "l"
s = str()
for i in l:
    if RepresentsInt(i):
            if len(s) > 0: ##This will ignore the initial condition
                text.append(s)
                s = str()
            id.append(int(i))
    else:
        s = s + " " + str(i)
text.append(s)

pd.options.display.max_colwidth = 10000
items_df = pd.DataFrame(zip(id,text),columns=["Joke-id","Text"])

print "Displaying the text of joke ID 144, to make sure that we parsed the Jokes text correctly"
display(items_df[items_df["Joke-id"] == 144])
```

Output of the above code block:



```
Displaying the text of joke ID 144, to make sure that we parsed the Jokes text correctly
```

	Joke-id	Text
143	144	A man is driving in the country one evening when his car stalls and won't start. He goes up to a nearby farm house for help, and because it is suppertime he is asked to stay for supper. When he sits down at the table he notices that a pig is sitting at the table with them for supper and that the pig has a wooden leg. As they are eating and chatting, he eventually asks the farmer why the pig is there and why it has a wooden leg. "Oh," says the farmer, "that is a very special pig. Last month my wife and daughter were in the barn when it caught fire. The pig saw this, ran to the barn, tipped over a pail of water, crawled over the wet floor to reach them and pulled them out of the barn safely. A special pig like that, you just don't eat it all at once!"

The following function will take a user ID as input and gives the list of recommended items as the output.

```
def get_SGD_recommendations(user_id,ratings_df,pred_utility,Utility):
    #Get the list of jokes, which are already rated by the user:
    already_rated = list(ratings_df[(ratings_df["user_id"] == user_id)]["item_id"])

    top_user_ratings = ratings_df[(ratings_df["user_id"] == user_id)].sort(["rating"],
                                                               ascending=[0]).head(5)




    #Note that the joke IDs are NOT named using consecutive integers
    #Get the items not rated by the user ID
    not_rated = set(ratings_df["item_id"]) - set(already_rated)


    row_idx = [user_id] * len(not_rated)
    col_idx = list(not_rated)
    #Convert pred matrix to data frame
    pred_df = pd.DataFrame(pred)

    #Change the row and column names
    pred_df.index = Utility.index
    pred_df.columns = Utility.columns

    #Get ratings at the intersection of the rows and columns list
    ratings = get_ratings(pred_df,row_idx,col_idx,indices=True)

    ratings = np.array(ratings)

    #Change the ratings to the allowed interval
    ratings[ratings > 10] = 10
    ratings[ratings < -10] = -10
    return [ratings,row_idx,col_idx,top_user_ratings]

#Predict the ratings for the user ID 129
user_id = 129
pred_ratings,row_idx,col_idx,top_user_ratings =
  get_SGD_recommendations(user_id,ratings_df,pred,Utility)
print "The user ID {} has rated these jokes high (top 5 jokes):".format(user_id)
display(top_user_ratings)

display(items_df[items_df["Joke-id"].isin(list(top_user_ratings["item_id"]))])

display_df=pd.DataFrame(zip(row_idx,col_idx,pred_ratings),
                        columns=["user_id","item_id","predicted_ratings"])

display_df["predicted_ratings"].min()
display_df=display_df.sort(["predicted_ratings"],ascending=0)

recommended_items = list(display_df["item_id"])[0:5]
print "Top 5 recommended items for the user ID: {} are given below.".format(user_id)
items_df[items_df["Joke-id"].isin(recommended_items)]
```

Output of the above code block:

```
The user ID 129 has rated these jokes high (top 5 jokes):
```

	user_id	item_id	rating
4819	129	15	-0.781
4816	129	7	-4.906
4815	129	5	-5.438
4818	129	13	-9.562
4817	129	8	-9.719

	Joke-id	Text
4	5	Q. What's O. J. Simpson's web address? A. Slash, slash, backslash, slash, slash, escape.
6	7	How many feminists does it take to screw in a light bulb? That's not funny.
7	8	Q. Did you hear about the dyslexic devil worshiper? A. He sold his soul to Santa.
12	13	They asked the Japanese visitor if they have elections in his country. "Every morning," he answers.
14	15	Q: What did the blind person say when given some matzah? A: Who the hell wrote this?

```
Top 5 recommended items for the user ID: 129 are given below.
```

	Joke-id	Text
34	35	An explorer in the deepest Amazon suddenly finds himself surrounded by a bloodthirsty group of natives. Upon surveying the situation, he says quietly to himself, "Oh God, I'm screwed." The sky darkens and a voice booms out, "No, you are NOT screwed. Pick up that stone at your feet and bash in the head of the chief standing in front of you." So with the stone he bashes the life out of the chief. He stands above the lifeless body, breathing heavily and looking at 100 angry natives... The voice booms out again, "Okay....NOW you're screwed."
52	53	One Sunday morning William burst into the living room and said, "Dad! Mom! I have some great news for you! I am getting married to the most beautiful girl in town. She lives a block away and her name is Susan." After dinner, William's dad took him aside. "Son, I have to talk with you. Your mother and I have been married 30 years. She's a wonderful wife but she has never offered much excitement in the bedroom, so I used to fool around with women a lot. Susan is actually your half-sister, and I'm afraid you can't marry her." William was heart-broken. After eight months he eventually started dating girls again. A year later he came home and very proudly announced, "Dianne said yes! We're getting married in June." Again his father insisted on another private conversation and broke the sad news. "Dianne is your half-sister too, William. I'm awfully sorry about this." William was furious! He finally decided to go to his mother with the news. "Dad has done so much harm.. I guess I'm never going to get married," he complained. "Every time I fall in love, Dad tells me the girl is my half-sister." His mother just shook her head. "Don't pay any attention to what he says, dear. He's not really your father."
67	68	A man piloting a hot air balloon discovers he has wandered off course and is hopelessly lost. He descends to a lower altitude and locates a man down on the ground. He lowers the balloon further and shouts, "Excuse me, can you tell me where I am?" The man below says, "Yes, you're in a hot air balloon, about 30 feet above this field." "You must work in Information Technology," says the balloonist. "Yes I do," replies the man. "And how did you know that?" "Well," says the balloonist, "what you told me is technically correct, but of no use to anyone." The man below says, "You must work in management." "I do," replies the balloonist, "how did you know?" "Well," says the man, "you don't know where you are, or where you're going, but you expect my immediate help. You're in the same position you were before we met, but now it's my fault!"
71	72	On the first day of college, the Dean addressed the students, pointing out some of the rules: "The female dormitory will be out-of-bounds for all male students and the male dormitory to the female students. Anybody caught breaking this rule will be finded 20***thefirsttime." Hecontinued," Anybodycaughtbreakingthisrulethesecondtimewillbefined***60. Being caught a third time will cost you a fine of $180. Are there any questions?" At this point, a male student in the crowd inquired: "How much for a season pass?"
104	105	A couple of hunters are out in the woods in the deep south when one of them falls to the ground. He doesn't seem to be breathing, and his eyes are rolled back in his head. The other guy whips out his cell phone and calls 911. He gasps to the operator, "My friend is dead! What can I do?" The operator, in a calm and soothing voice, says, "Alright, take it easy. I can help. First, let's make sure he's dead." There is silence, and then a gun shot is heard. The hunter comes back on the line. "Okay. Now what??"

The above display shows that the user ID 129 has rated all the jokes with a negative rating. But our recommender is still able to predict the jokes he likes. But NOTE that we cannot determine if he really likes these jokes unless we present these jokes to the user and capture his response.

Let us get the recommended jokes for the user ID 17:

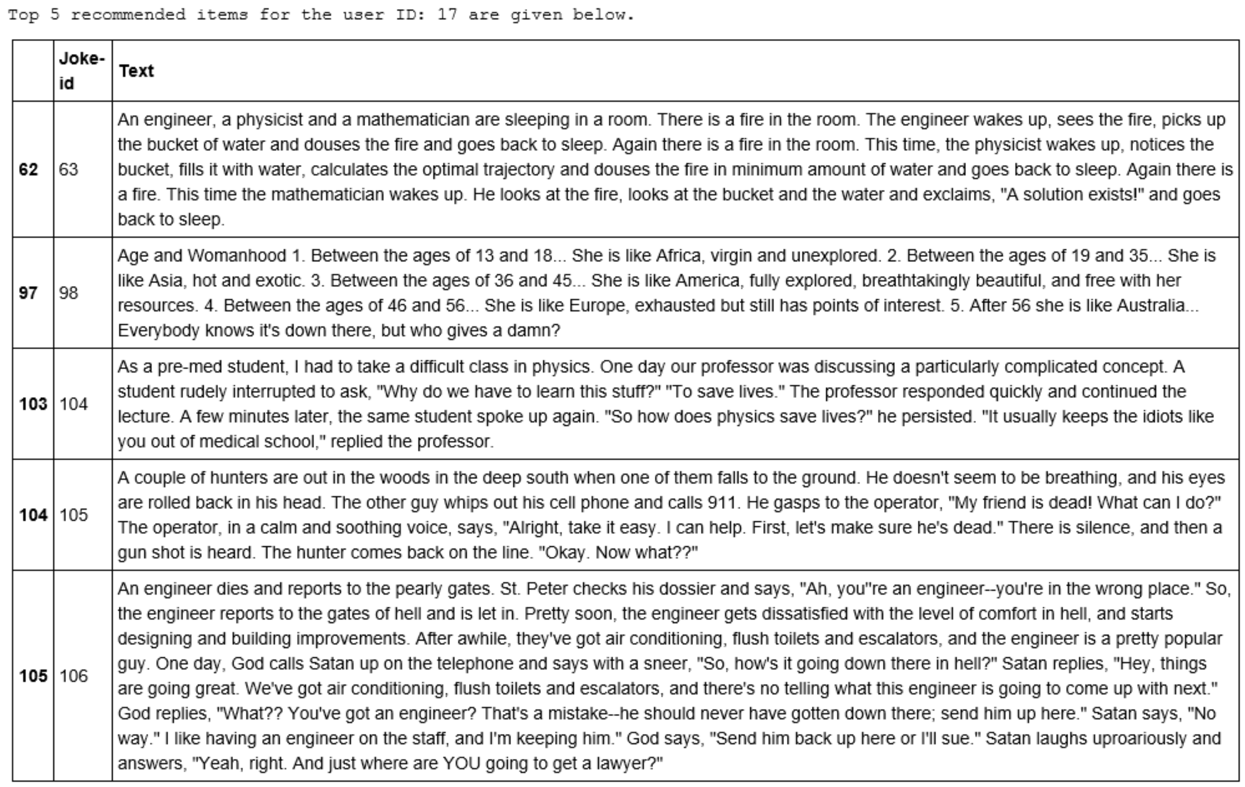
```
user_id = 17
pred_ratings,row_idx,col_idx,top_user_ratings =
  get_SGD_recommendations(user_id,ratings_df,pred,Utility)
print "The user ID {} has rated these jokes high (top 5 jokes):".format(user_id)
display(top_user_ratings)

display(items_df[items_df["Joke-id"].isin(list(top_user_ratings["item_id"]))])

display_df=pd.DataFrame(zip(row_idx,col_idx,pred_ratings),
                        columns=["user_id","item_id","predicted_ratings"])

display_df["predicted_ratings"].min()
display_df=display_df.sort(["predicted_ratings"],ascending=0)

recommended_items = list(display_df["item_id"])[0:5]
print "Top 5 recommended items for the user ID: {} are given below.".format(user_id)
items_df[items_df["Joke-id"].isin(recommended_items)]
```

Output of the above code block:

```
The user ID 17 has rated these jokes high (top 5 jokes):
```

	user_id	item_id	rating
755	17	17	10.000
766	17	35	10.000
771	17	49	10.000
758	17	20	10.000
770	17	27	9.938

	Joke-id	Text
16	17	How many men does it take to screw in a light bulb? One. Men will screw anything.
19	20	What's the difference between a Macintosh and an Etch-a-Sketch? You don't have to shake the Mac to clear the screen.
26	27	Bill Clinton returns from a vacation in Arkansas and walks down the steps of Air Force One with two pigs under his arms. At the bottom of the steps, he says to the honor guardsman, "These are genuine Arkansas Razor-Back Hogs. I got this one for Chelsea and this one for Hillary." The guardsman replies, "Nice trade, Sir."
34	35	An explorer in the deepest Amazon suddenly finds himself surrounded by a bloodthirsty group of natives. Upon surveying the situation, he says quietly to himself, "Oh God, I'm screwed." The sky darkens and a voice booms out, "No, you are NOT screwed. Pick up that stone at your feet and bash in the head of the chief standing in front of you." So with the stone he bashes the life out of the chief. He stands above the lifeless body, breathing heavily and looking at 100 angry natives... The voice booms out again, "Okay....NOW you're screwed."
48	49	Three engineering students were gathered together discussing the possible designers of the human body. One said, "It was a mechanical engineer. Just look at all the joints." Another said, "No, it was an electrical engineer. The nervous systems many thousands of electrical connections." The last said, "Actually, it was a civil engineer. Who else would run a toxic waste pipeline through a recreational area?"



```
Top 5 recommended items for the user ID: 17 are given below.
```

	Joke-id	Text
62	63	An engineer, a physicist and a mathematician are sleeping in a room. There is a fire in the room. The engineer wakes up, sees the fire, picks up the bucket of water and douses the fire and goes back to sleep. Again there is a fire in the room. This time, the physicist wakes up, notices the bucket, fills it with water, calculates the optimal trajectory and douses the fire in minimum amount of water and goes back to sleep. Again there is a fire. This time the mathematician wakes up. He looks at the fire, looks at the bucket and the water and exclaims, "A solution exists!" and goes back to sleep.
97	98	Age and Womanhood 1. Between the ages of 13 and 18... She is like Africa, virgin and unexplored. 2. Between the ages of 19 and 35... She is like Asia, hot and exotic. 3. Between the ages of 36 and 45... She is like America, fully explored, breathtakingly beautiful, and free with her resources. 4. Between the ages of 46 and 56... She is like Europe, exhausted but still has points of interest. 5. After 56 she is like Australia... Everybody knows it's down there, but who gives a damn?
103	104	As a pre-med student, I had to take a difficult class in physics. One day our professor was discussing a particularly complicated concept. A student rudely interrupted to ask, "Why do we have to learn this stuff?" "To save lives." The professor responded quickly and continued the lecture. A few minutes later, the same student spoke up again. "So how does physics save lives?" he persisted. "It usually keeps the idiots like you out of medical school," replied the professor.
104	105	A couple of hunters are out in the woods in the deep south when one of them falls to the ground. He doesn't seem to be breathing, and his eyes are rolled back in his head. The other guy whips out his cell phone and calls 911. He gasps to the operator, "My friend is dead! What can I do?" The operator, in a calm and soothing voice, says, "Alright, take it easy. I can help. First, let's make sure he's dead." There is silence, and then a gun shot is heard. The hunter comes back on the line. "Okay. Now what??"
105	106	An engineer dies and reports to the pearly gates. St. Peter checks his dossier and says, "Ah, you"re an engineer--you're in the wrong place." So, the engineer reports to the gates of hell and is let in. Pretty soon, the engineer gets dissatisfied with the level of comfort in hell, and starts designing and building improvements. After awhile, they've got air conditioning, flush toilets and escalators, and the engineer is a pretty popular guy. One day, God calls Satan up on the telephone and says with a sneer, "So, how's it going down there in hell?" Satan replies, "Hey, things are going great. We've got air conditioning, flush toilets and escalators, and there's no telling what this engineer is going to come up with next." God replies, "What?? You've got an engineer? That's a mistake--he should never have gotten down there; send him up here." Satan says, "No way." I like having an engineer on the staff, and I'm keeping him." God says, "Send him back up here or I'll sue." Satan laughs uproariously and answers, "Yeah, right. And just where are YOU going to get a lawyer?"

We can see that the user liked a joke related to engineering students' topic, and the recommended jokes 63 and 106 are about engineers and 104 is about a pre-med *student*. The joke ID 17 is about men (liked joke), and the joke ID 98 is about women (recommended joke). Jokes 35 (liked joke) and 106 are also related to some extent since both refer to God.

10.0 Conclusion

In this project, we developed a complete frame work in Python to build recommendation engines based on SGD algorithm.

In this project, I was able to achieve the following:

1. Predict the ratings for all the items for all the users.
2. We were able to identify optimal rating threshold as 5 to determine if a user likes the item.
3. We were able to run the algorithm on a laptop with 16GB of RAM (the ratings dataset has 1.7 Million ratings)
4. We were able to quantify the performance of our algorithm using AUC (Area Under the Curve) and obtained an average AUC score of 0.8, which is a good score since it is way higher than the AUC score of 0.5 (obtained by random guessing).

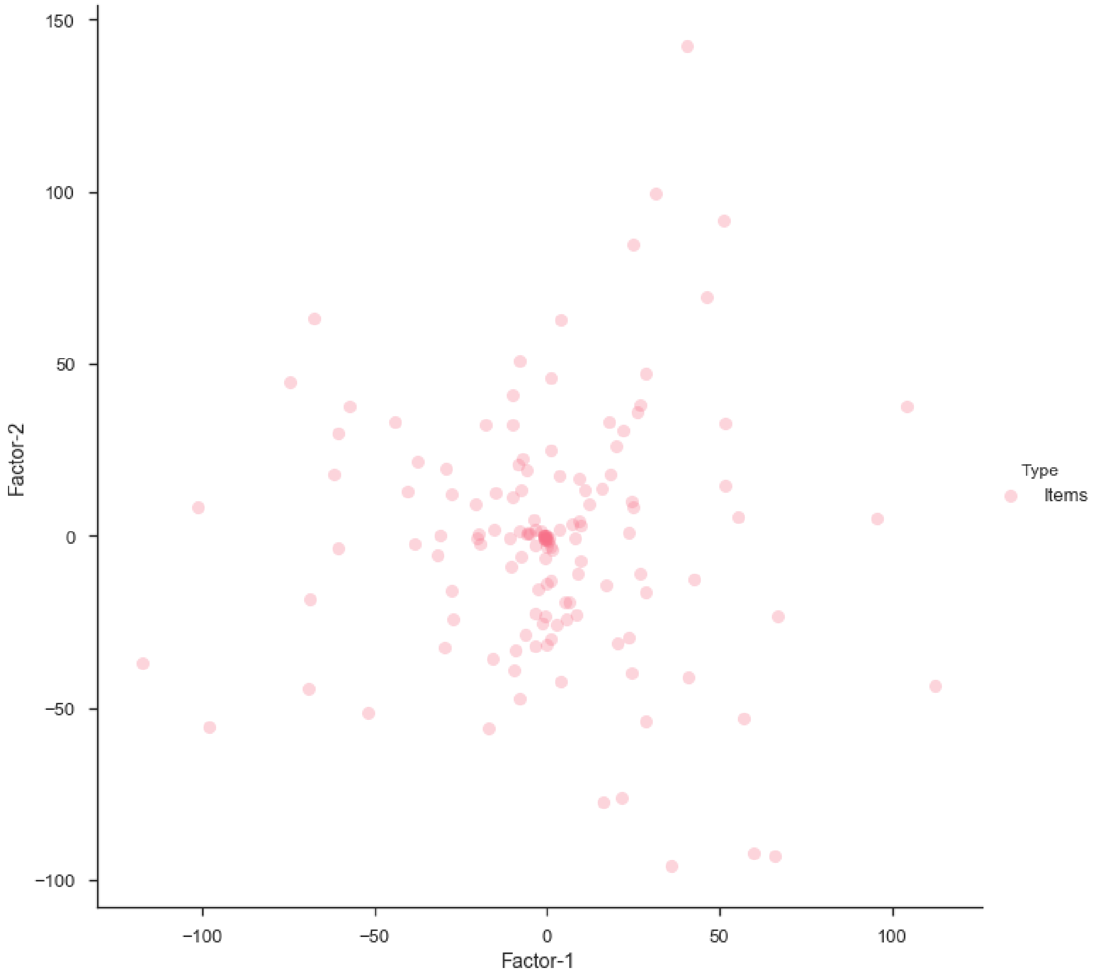
The main disadvantage of SGD is the cold start problem. Cold-start problem refers to the problem of dealing with new users and new items. For new users, we have limited knowledge since we do not know much about the user's ratings. For new items, we do not know who would like the item. Dealing with new items is relatively easier, since we can develop a separate recommender based on the item features by comparing the new item features with the existing item features, and measuring the similarity (such as cosine similarity) between the new and existing items, and recommend the new items to the users who liked similar items. To recommend items to new users, we can use the available features of the users (such as user's demographic information, if captured by the system) and obtain a similarity score with the existing users. Based on the identified similar users, we can recommend items to the new users, which were liked by similar existing users. But it is not easy to capture the user's features, and the similarity score computed between the new and existing users may not give us good results. But the features of items can be easily captured, and it is relatively easy to map existing items with new items.

SGD can indirectly address the cold start problem, by using the concept of clustering. An important application of SGD algorithm is to map the users and items to a common coordinate system. In this project we used 60 as the latent factors or optimal number of dimensions. The reduced number of dimensions can also help us to cluster the users or items, so that we can map new user or item to the relevant cluster and propose the *new items* to the **existing** users or existing items to the _new users). To demonstrate this, let us just consider the first 2 factors in the 60 latent factors, and plot the items data (present in V matrix).

Python code to plot jokes coordinates in the first 2 factors of the 60 latent factors:

```
import seaborn as sns
#Get only the first 2 factors out of 60 factors
V = V[0:2,:]

#Create a pandas data frame
items = pd.DataFrame(V.T)
items.columns = ["Factor-1","Factor-2"]
items["Type"] = "Items"

#Plot the points on the first 2 dimensions of the latent factors from V
sns.set(style="ticks", context="talk")
pal= sns.color_palette("husl", 2)
sns.lmplot(x="Factor-1",y="Factor-2",fit_reg=False,hue="Type",
           palette=pal,data=items,size=10,scatter_kws={'alpha':0.3})

plt.show()
```

Output of the above code block:



From the above figure, we can infer that the items (or jokes) can be clustered into clusters. When a new joke is added to the system, then we can compare the textual content of the new joke with the existing jokes (using NLP - Natural Language Processing techniques), and assign the new joke to the cluster to which the semantically closest joke belongs to. NOTE that I considered the first 2 factors out of the 60 factors to make the visualization simple and interpretable. But in our proposed recommendation system, we will be considering all the 60 latent factors, and cluster the data based on the 60 latent factors.

I encountered many challenges during the development of this project. The major challenges are listed below:

1. The logic to randomly split the data into test and training data was challenging. An item could be rated by only one user. If we pick such single user ratings into test data, then we will end up having all NA values for some of the rows in the training data, and our SGD algorithm outputs NA values for some of the predicted ratings. Another challenge was the selection of the same item's ratings and moving the selected ratings to test data multiple times. Assume that we have an item I, which has 2 ratings given by 2 different users. If we select one of the available ratings (due to random choice), and move that rating to the test data, and make the corresponding element as NA in the training data, then we will have only 1 available rating in the training data. But if our algorithm selects the same item I again, then the row related to item I in the training data will have all the NA values. I handled this problem by making sure that there will be at least 1 available rating for each item in the training dataset. See the *split_data()* function for more details.

2. Processing a 1.7 Million ratings dataset to select the best hyper parameters has run for almost 1 hour on a laptop with 16 GB RAM.
3. Determining optimal threshold based on the average AUC scores also ran for more than 1 hour.

10.1 Future work

I would like to address the following as a part of future work:

- Analyze methods to address the cold start problem.
- Build many recommenders, combine their outputs and analyze if the quality of the recommendations improves.

References:

1. The project report is based on the IPython notebook: https://goo.gl/Xgc9Pb. This was developed by me, and two algorithms SGD and collaborative filtering were analyzed
2. Eigentaste: A Constant Time Collaborative Filtering Algorithm. Ken Goldberg, Theresa Roeder, Dhruv Gupta, and Chris Perkins. Information Retrieval, 4(2), 133-151. July 2001
3. Jure Leskovec, Anand Rajaraman and Jeffrey D. Ullman 2014. Mining of Massive Datasets (Chapter 9)
4. Deepak K. Agarwal and Bee-Chung Chen. Statistical Methods for Recommender Systems