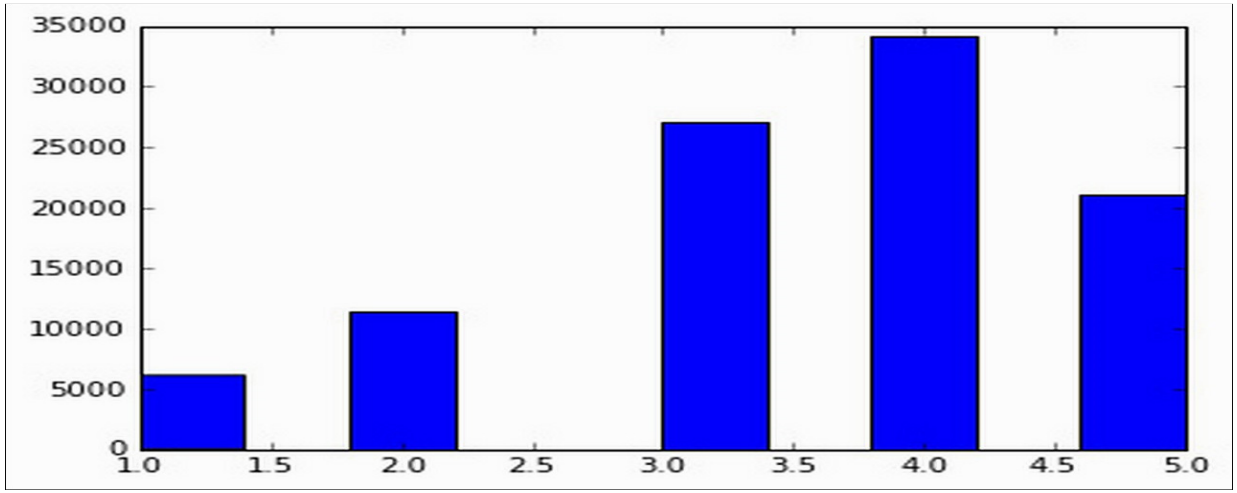
Data exploration

In this section, we will explore the MovieLens dataset and also prepare the data required for building collaborative filtering recommendation engines using python.

Let's see the distribution of ratings using the following code snippet:

```
import matplotlib.pyplot as plt
plt.hist(df['Rating'])
```

From the following image we see that we have more movies with 4 star ratings:

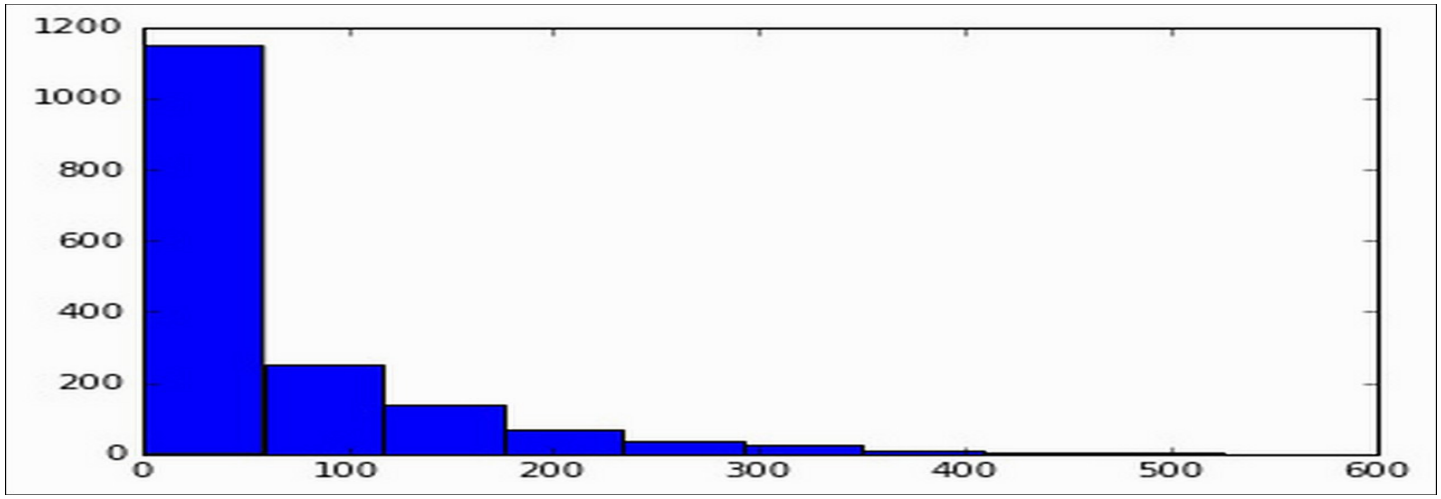


Using the following code snippet, we shall see the counts of ratings by applying the `groupby()` function and the `count()` function on DataFrame:

```
In [21]: df.groupby(['Rating'])['UserID'].count()
Out[21]:
Rating
1      6110
2     11370
3     27145
4     34174
5     21201
Name: UserID, dtype: int64
```

The following code snippet shows the distribution of movie views. In the following code we apply the `count()` function on DataFrame:

```
plt.hist(df.groupby(['ItemId'])['ItemId'].count())
```



From the previous image, we can observe that the starting ItemId has more ratings than later movies.

Rating matrix representation

Now that we have explored the data, let's represent the data in a rating matrix form so that we can get started with our original task of building a recommender engine.

For creating a rating matrix, we make use of NumPy package capabilities such as arrays and row iterations in a matrix. Run the following code to represent the data frame in a rating matrix:

NOTE

In the following code, first we extract all the unique user IDs and then we check the length using the shape parameter.

Create a variable of `n_users` to find the total number of unique users in the data:

```
n_users = df.UserID.unique().shape[0]
```

Create a variable `n_items` to find the total number of unique movies in the data:

```
n_items = df['ItemId '].unique().shape[0]
```

Print the counts of unique users and movies:

```
print(str(n_users) + ' users')
943 users


print(str(n_items) + ' movies')
1682 movies
```

Create a zero value matrix of size (*n_users x n_items*) to store the ratings in the cell of the matrix ratings:

```
ratings = np.zeros((n_users, n_items))
```

For each tuple in the DataFrame, `df` extracts the information from each column of the row and stores it in the rating matrix cell value as follows:

```
for  row in df.itertuples():
ratings[row[1]-1, row[2]-1] = row[3]
```

Run the loop and the whole DataFrame movie ratings information will be stored in the matrix ratings of the `numpy.ndarray` type as follows:

```
type(ratings)
<type 'numpy.ndarray'>
```

Now let's see the dimensions of the multidimensional array 'ratings' using the shape attribute as follows:

```
ratings.shape
(943, 1682)
```

Let's see the sample data for how a ratings multidimensional array looks by running the following code:

```
ratings
array([[ 5.,  3.,  4., ...,  0.,  0.,  0.],
       [ 4.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 5.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  5.,  0., ...,  0.,  0.,  0.]])
```

We observe that the rating matrix is sparse as we see a lot of zeros in the data. Let's determine the `sparsity` in the data, by running the following code:

```
sparsity = float(len(ratings.nonzero()[0]))
sparsity /= (ratings.shape[0] * ratings.shape[1])
sparsity *= 100
print('Sparsity: {:4.2f}%'.format(sparsity))
Sparsity: 6.30%
```

We observe that the sparsity is `6.3%` that is to say that we only have rating information for `6.3%` of the data and for the others it is just zeros. Also please note that, the `0` value we see in the rating matrix doesn't represent the rating given by the user, it just means that they are empty.

Creating training and test sets

Now that we have a ratings matrix, let's create a training set and test set to build the recommender model using a training set and evaluate the model using a test set.

To divide the data into training and test sets, we use the `sklearn`package's capabilities. Run the following code to create training and test sets:

Load the `train_test_split` module into the python environment using the following import functionality:

```
from sklearn.cross_validation import train_test_split
```

Call the `train_test_split()` method with a test size of `0.33`and random seed of `42`:

```
 ratings_train, ratings_test = train_test_split(ratings,test_size=0.33,
random_state=42)
```

Let's see the dimensions of the train set:

```
 ratings_train.shape (631, 1682)
#let's see the dimensions of the test set
```

```
ratings_test.shape (312, 1682)
```

For user-based collaborative filtering, we predict that a user's rating for an item is given by the weighted sum of all other users' ratings for that item, where the weighting is the cosine similarity between each user and the input user.

The steps for building a UBCF

The steps for building a UBCF are:

- Creating a similarity matrix between the users
- Predicting the unknown rating value of item *i* for an active user *u*by calculating the weighted sum of all the users' ratings for the item.

  ## TIP

  Here the weighting is the cosine similarity calculated in the previous step between the user and neighboring users.

- Recommending the new items to the users.

User-based similarity calculation

The next step is to create pairwise similarity calculations for each user in the rating matrix, that is to say we have to calculate the similarity of each user with all the other

users in the matrix. The similarity calculation we choose here is cosine similarity. For this, we make use of pairwise distance capabilities to calculate the cosine similarity available in the `sklearn` package as follows:

```
import numpy as np
import sklearn

#call cosine_distance() method available in sklearn metrics module
by passing the ratings_train set.
The output will be a distance matrix.
dist_out = 1-
sklearn.metrics.pairwise.cosine_distances(ratings_train)

# the type of the distance matrix will be the same type of the
rating matrix
type(dist_out)
<type 'numpy.ndarray'>
#the dimensions of the distance matrix will be a square matrix of
size equal to the number of users.
dist_out.shape
(631, 631)
```

Let's see a sample dataset of the distance matrix:

```
dist_out
```

```
array ([[ 1.        ,  0.36475764,  0.44246231, ...,  0.02010641,
         0.33107929, 0.25638518],
       [ 0.36475764, 1.        ,  0.42635255, ..., 0.06694419,
         0.27339314, 0.22337268],
       [ 0.44246231, 0.42635255, 1.        , ..., 0.06675756,
         0.25424373, 0.22320126],
       ...,
       [ 0.02010641, 0.06694419, 0.06675756, ..., 1.        ,
         0.04853428, 0.05142508],
       [ 0.33107929, 0.27339314, 0.25424373, ..., 0.04853428,
         1.        ,  0.1198022],
       [ 0.25638518, 0.22337268, 0.22320126, ..., 0.05142508,
         0.1198022, 1.    ]])
```

Predicting the unknown ratings for an active user

As previously mentioned, the unknown values can be calculated for all the users by taking the dot product between the distance matrix and the rating matrix and then normalizing the data with the number of ratings as follows:

```
user_pred = dist_out.dot(ratings_train) /
np.array([np.abs(dist_out).sum(axis=1)]).T
```

Now that we have predicted the unknown ratings for use in the training set, let's define a function to check the error or performance of the model. The following code defines a function for calculating the root mean square error (RMSE) by taking the predicted values and original values. We use `sklearn` capabilities for calculating RMSE as follows:

```
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)
```

We call the `get_mse()` method to check the model prediction error rate as follows:

```
get_mse(user_pred, ratings_train)
7.8821939915510031
```

We see that the model accuracy or RMSE is `7.8`. Now let's run the same `get_mse()` method on the test data and check the accuracy as follows:

```
get_mse(user_pred, ratings_test)
8.9224954316965484
```

User-based collaborative filtering with the k-nearest neighbors

If we observe the RMSE values in the above model, we can see that the error is a bit higher. The reason may be that we have chosen all the users' rating information while making the predictions. Instead of considering all the users, let's consider only the top-N similar users' ratings information and then make the predictions. This may result in improving the model accuracy by eliminating some biases in the data.

To explain in a more elaborate way; in the previous code we predicted the ratings of the users by taking the weighted sum of the ratings of all users, instead we first chose the top-N similar users for each user and then the ratings were calculated by considering the weighted sum of the ratings of these top-N users.

Finding the top-N nearest neighbors

Firstly, for computational easiness, we shall choose the top five similar users by setting a variable, *k*.

k=5

We use the k-nearest neighbors method to choose the top five nearest neighbors for an active user. We will see this in action shortly. We choose sklearn.knn capabilities for this task as follows:

```
from sklearn.neighbors import NearestNeighbors
```

Define the `NearestNeighbors` object by passing k and the similarity method as parameters:

```
neigh = NearestNeighbors(k,'cosine')
```

Fit the training data to the `nearestNeighbor` object:

```
neigh.fit(ratings_train)
```

Calculate the top five similar users for each user and their similarity values, that is the distance values between each pair of users:

```
top_k_distances,top_k_users = neigh.kneighbors(ratings_train,
return_distance=True)
```

We can observe below that the resultant `top_k_distances`ndarray contains similarity values and top five similar users for each users in the training set:

```
top_k_distances.shape
(631, 5)
top_k_users.shape
(631, 5)
```

Let's see the top five users that are similar to user 1 in the training set:

```
top_k_users[0]
array([  0,  82, 511, 184, 207], dtype=int64)
```

The next step would be to choose only the top five users for each user and use their rating information while predicting the ratings using the weighted sum of all of the ratings of these top five similar users.

Run the following code to predict the unknown ratings in the training data:

```
user_pred_k = np.zeros(ratings_train.shape)
for i in range(ratings_train.shape[0]):
    user_pred_k[i,:] =
top_k_distances[i].T.dot(ratings_train[top_k_users][i])
/np.array([np.abs(top_k_distances[i].T).sum(axis=0)]).T
```

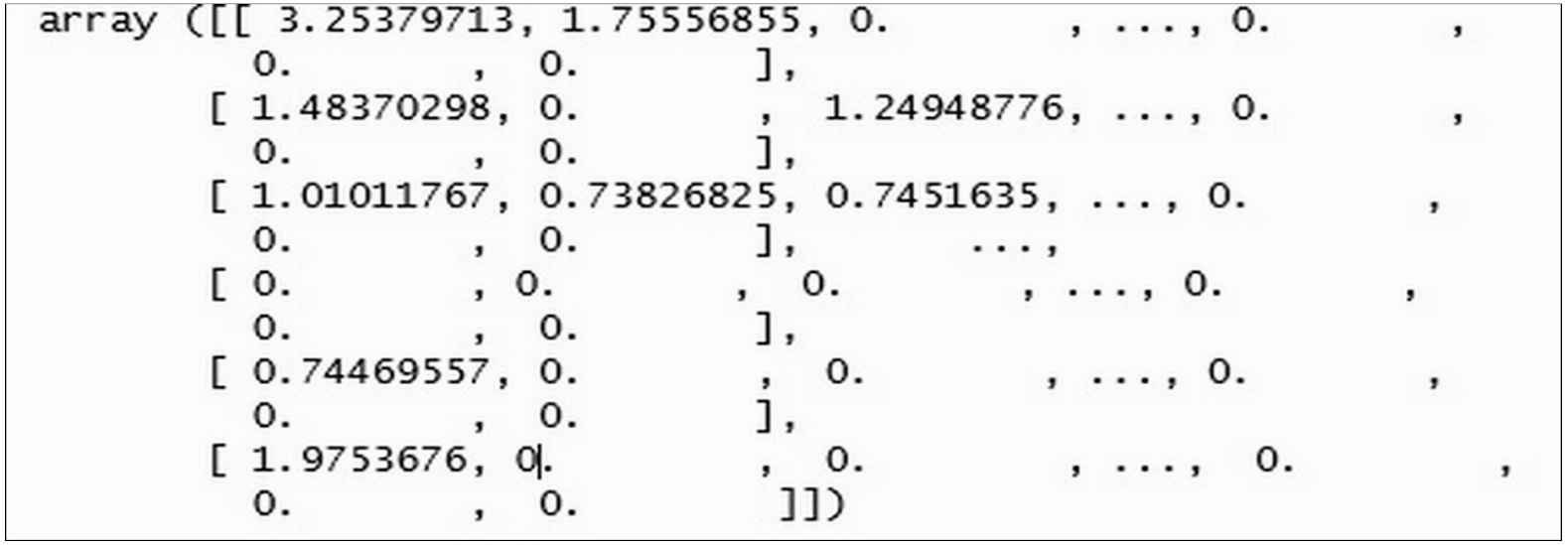
Let's see the data predicted by the model as follows:

```
user_pred_k.shape
```

```
(631, 1682)
```

```
user_pred_k
```

The following image displays the results for `user_pred_k`:



Now let's see if the model has improved or not. Run the get_mse() method defined earlier as follows:

```
get_mse(user_pred_k, ratings_train)
8.9698490022546036
get_mse(user_pred_k, ratings_test)
11.528758029255446
```

Item-based recommendations

IBCF is very similar to UBCF but with very minor changes in how we use the rating matrix.

The first step is to calculate the similarities between movies, as follows:

Since we have to calculate the similarity between movies, we use movie count as `k` instead of user count:

```
k = ratings_train.shape[1]
neigh = NearestNeighbors(k,'cosine')
```

We fit the transpose of the rating matrix to the `NearestNeighbors` object:

```
neigh.fit(ratings_train.T)
```

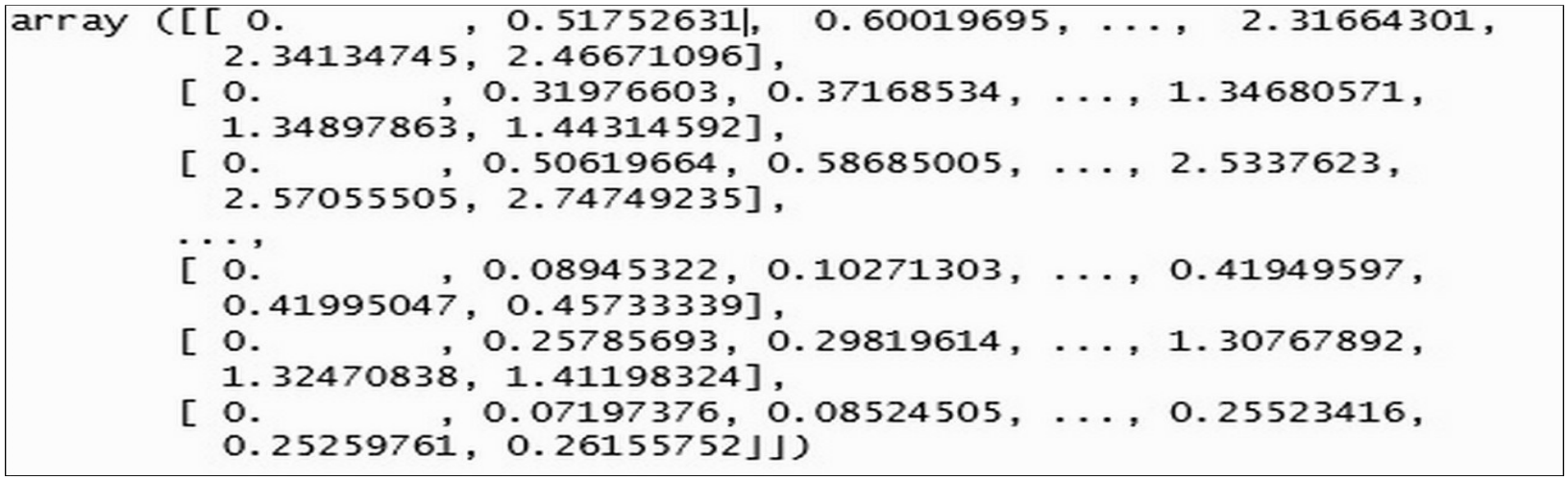
Calculate the cosine similarity distance between each movie pair:

```
top_k_distances,top_k_users = neigh.kneighbors(ratings_train.T,
return_distance=True)
top_k_distances.shape
(1682, 1682)
```

The next step is to predict the movie ratings using the following code:

```
item__pred = ratings_train.dot(top_k_distances) /
np.array([np.abs(top_k_distances).sum(axis=1)])
item__pred.shape
(631, 1682)
item__pred
```

The following image shows the result for `item_pred`:



```
array ([[ 0.        , 0.51752631,  0.60019695, ...,  2.31664301,
         2.34134745, 2.46671096],
        [ 0.        , 0.31976603, 0.37168534, ..., 1.34680571,
         1.34897863, 1.44314592],
        [ 0.        , 0.50619664, 0.58685005, ..., 2.5337623,
         2.57055505, 2.74749235],
        ...,
        [ 0.        , 0.08945322, 0.10271303, ..., 0.41949597,
         0.41995047, 0.45733339],
        [ 0.        , 0.25785693, 0.29819614, ..., 1.30767892,
         1.32470838, 1.41198324],
        [ 0.        , 0.07197376, 0.08524505, ..., 0.25523416,
         0.25259761, 0.26155752]])
```

Evaluating the model

Now let's evaluate the model using the `get_mse()` method we have defined by passing the prediction ratings and the training and test set as follows:

```
get_mse(item_pred, ratings_train)
11.130000188318895
```

```
get_mse(item_pred,ratings_test)
12.128683035513326
```

The training model for k-nearest neighbors

Run the following code to calculate the distance matrix for the top 40 nearest neighbors and then calculate the weighted sum of ratings by the top 40 users for all the movies. If we closely observe the code, it is very similar to what we have done for UBCF. Instead of passing `ratings_train` as is, we transpose the data matrix and pass to the previous code as follows:

```
k = 40
neigh2 = NearestNeighbors(k,'cosine')
neigh2.fit(ratings_train.T)
top_k_distances,top_k_movies = neigh2.kneighbors(ratings_train.T,
return_distance=True)


#rating prediction - top k user based
pred = np.zeros(ratings_train.T.shape)
for i in range(ratings_train.T.shape[0]):
    pred[i,:] =
top_k_distances[i].dot(ratings_train.T[top_k_users][i])/np.array([np.abs(t
op_k_distances[i]).sum(axis=0)]).T
```

Evaluating the model

The follow code snippet calculates the mean squared error for the training and test set. We can observe that the training error is 11.12 whereas the test error is 12.12.

```
get_mse(item_pred_k, ratings_train)
11.130000188318895
get_mse(item_pred_k,ratings_test)
12.128683035513326
```