MACHINE LEARNING LAB - TUTORIAL 10

Juan Fernando Espinosa

303158

DATA IMPORTING

The data chosen for this tutorial is **Movielens 100k movie ratings**.

```
import pandas as pd
import numpy as np
%matplotlib inline
import math
import matplotlib.pyplot as plt
from google.colab import files
from google.colab import drive
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import NMF
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import KFold
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer


#column titles for the dataset
data_cols = ['user id','movie id','rating','timestamp']
item_cols = ['movie id','movie title','release date','video_release_date','IMDb URL','unknown','Action',
'Adventure','Animation','Childrens','Comedy','Crime',
'Documentary','Drama','Fantasy','Film-Noir','Horror',
'Musical','Mystery','Romance ','Sci-Fi','Thriller',
'War' ,'Western']
user_cols = ['user id','age','gender','occupation',
'zip code']


#importing the data files onto dataframes
"""users = pd.read_csv('u.user', sep='|',
names=user_cols, encoding='latin-1')
item = pd.read_csv('u.item', sep='|',
names=item_cols, encoding='latin-1')
data = pd.read_csv('u.data', sep='\t',
names=data_cols, encoding='latin-1')"""

drive.mount('/content/drive')
!ls "/content/drive/My Drive/Colab Notebooks/LAB/tutorial 10/ml-100k/u.user"
drive.mount('/content/drive')
!ls "/content/drive/My Drive/Colab Notebooks/LAB/tutorial 10/ml-100k/u.item"
drive.mount('/content/drive')
!ls "/content/drive/My Drive/Colab Notebooks/LAB/tutorial 10/ml-100k/u.data"
drive.mount('/content/drive')
!ls "/content/drive/My Drive/Colab Notebooks/LAB/tutorial 10/ml-100k/u.genre"
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)
'/content/drive/My Drive/Colab Notebooks/LAB/tutorial 10/ml-100k/u.user'
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)
'/content/drive/My Drive/Colab Notebooks/LAB/tutorial 10/ml-100k/u.item'
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)
'/content/drive/My Drive/Colab Notebooks/LAB/tutorial 10/ml-100k/u.data'
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)
'/content/drive/My Drive/Colab Notebooks/LAB/tutorial 10/ml-100k/u.genre'
```

```
user = pd.read_csv("/content/drive/My Drive/Colab Notebooks/LAB/tutorial 10/ml-100k/u.user", sep='|', names=user_cols, encoding='lat
item = pd.read_csv("/content/drive/My Drive/Colab Notebooks/LAB/tutorial 10/ml-100k/u.item", sep='|', names=item_cols, encoding='lat
data = pd.read_csv("/content/drive/My Drive/Colab Notebooks/LAB/tutorial 10/ml-100k/u.data", sep='\t', names=data_cols, encoding='la
genre = pd.read_csv("/content/drive/My Drive/Colab Notebooks/LAB/tutorial 10/ml-100k/u.genre", sep='|', names=data_cols, encoding='l
genre = genre[['user id','movie id']]


user.head()
```

	user id	age	gender	occupation	zip code
0	1	24	M	technician	85711
1	2	53	F	other	94043
2	3	23	M	writer	32067
3	4	24	M	technician	43537
4	5	33	F	other	15213

```
# TRANSFORMING NaN VALUES INTO NUMERIC ONES - YEAR
item['video_release_date'] = item['movie title'].str.extract("\((\d{4})\)", expand=True)
item.video_release_date = pd.to_datetime(item.video_release_date, format='%Y')
item.video_release_date = item.video_release_date.dt.year
item['movie title'] = item['movie title'].str[:-7]
item.head()
```

	movie id	movie title	release date	video_release_date	IMDb URL	unknown	Action	Adventure	Animation	Childrens	Comedy	C
0	1	Toy Story	01-Jan-1995	1995.0	http://us.imdb.com/M/title-exact?Toy%20Story%2...	0	0	0	1	1	1	
1	2	GoldenEye	01-Jan-1995	1995.0	http://us.imdb.com/M/title-exact?GoldenEye%20(...	0	1	1	0	0	0	
2	3	Four Rooms	01-Jan-1995	1995.0	http://us.imdb.com/M/title-exact?Four%20Rooms%...	0	0	0	0	0	0	
3	4	Get Shorty	01-Jan-1995	1995.0	http://us.imdb.com/M/title-exact?Get%20Shorty%...	0	1	0	0	0	1	
4	5	Copycat	01-Jan-1995	1995.0	http://us.imdb.com/M/title-exact?Copycat%20(1995)	0	0	0	0	0	0	

```
# CHANGING TIMESTAMP TO NUMERIC ONE - PRINTING DATA DATASET
data.timestamp = pd.to_datetime(data.timestamp, infer_datetime_format=True)
data.timestamp = data.timestamp.dt.year
data.head()
```

	user id	movie id	rating	timestamp
0	196	242	3	1970
1	186	302	3	1970
2	22	377	1	1970
3	244	51	2	1970
4	166	346	1	1970

```
# CREATE A MERGED DATAFRAME - ALL THE INFORMATION REQUIRED

df = pd.merge(pd.merge(item, data), user)
df = df.merge(genre, on="movie id", how = 'inner')
df.head()
```

	movie id	movie title	release date	video_release_date	IMDb URL	unknown	Action	Adventure	Animation	Childrens	Comedy	Crime
0	1	Toy Story	01-Jan-1995	1995.0	http://us.imdb.com/M/title-exact?Toy%20Story%2...	0	0	0	1	1	1	
1	1	Toy Story	01-Jan-1995	1995.0	http://us.imdb.com/M/title-exact?Toy%20Story%2...	0	0	0	1	1	1	
2	1	Toy Story	01-Jan-1995	1995.0	http://us.imdb.com/M/title-exact?Toy%20Story%2...	0	0	0	1	1	1	
3	1	Toy Story	01-Jan-1995	1995.0	http://us.imdb.com/M/title-exact?Toy%20Story%2...	0	0	0	1	1	1	
4	1	Toy Story	01-Jan-1995	1995.0	http://us.imdb.com/M/title-exact?Toy%20Story%2...	0	0	0	1	1	1	

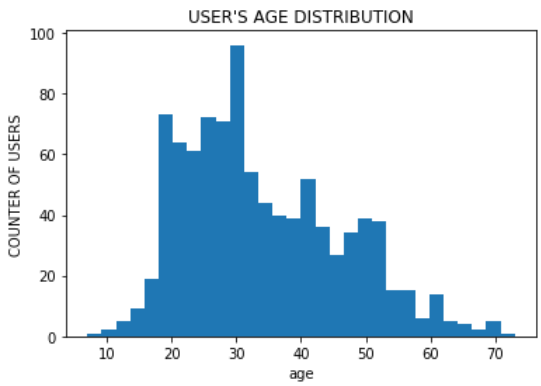
```
df_new = df.sample(frac=0.2)
```

```
#tic_tac_test = tic_tac_stratified.drop(tic_tac_train.index)
```

1. RECOMMENDER DATASET: STATISTICAL ANALYSIS

The strategy decided to analyze this dataset is to go from macro to micro trends, from demographics to interests per group of users (gender and age), region segmentation by taking advantage of the zip code.

```
# GENERAL DISTRIBUTION OF THE AGE.

user.age.plot.hist(bins=30)
plt.title("USER'S AGE DISTRIBUTION")
plt.ylabel('COUNTER OF USERS')
plt.xlabel('age');
plt.show()

# DISTRIBUTION OF GENDER IN THE DATASET.
genders = user.groupby(['gender']).count()
print('\n','\n','GENDER DISTRIBUTION INFORMATION','\n', '\n',genders)
```




```
 GENDER DISTRIBUTION INFORMATION

        user id  age  occupation  zip code
gender
F           273  273         273       273
M           670  670         670       670
```

> As it is possible to appreciate in the information above the dataset is formed 71% by Male and the 39% remaining by women.

> The user's age distribution on the contrary shows that people through the ages between 20 to 40 tend to give more reviews. Still, the information is not enough.

> Therefore, it is necessary to improve it.

AGE

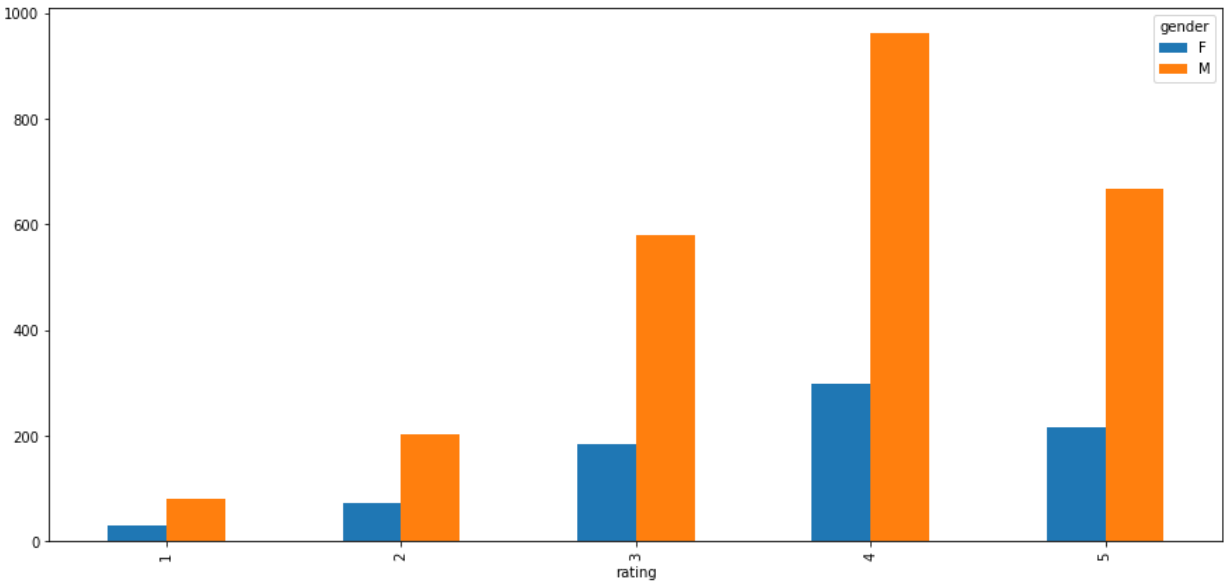
```
# BIDDING AGES INTO GROUPS

ranges = ['0-9', '10-19', '20-29', '30-39', '40-49', '50-59', '60-69', '70-79']
df['age_range'] = pd.cut(df.age, range(0, 81, 10), right=False, labels=ranges)
df[['age', 'age_range']].drop_duplicates()[:5]
df.groupby('age_range').agg({'rating': [np.size, np.mean]})
```

	rating	
	size	mean
age_range		
0-9	2	3.500000
10-19	265	3.758491
20-29	1363	3.779164
30-39	856	3.774533
40-49	473	3.699789
50-59	260	3.780769
60-69	74	3.824324
70-79	4	3.750000

> Thanks to the binding into ages' groups it is possible to derive precise conclusions: Young adults are more critical when rating movies: 65% of the population who rated are between their twenties and thirties.

```
# HOW THE RATES ARE GIVEN ACCORDING TO THE AGE OF THE POPULATION.

fig, ax = plt.subplots(figsize=(15,7))
df.groupby(['rating', 'gender']).count()['age_range'].unstack().plot.bar(ax=ax)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f33336c37b8>
```



- People between 20 and 30 years tend to rate higher, giving a majority of 4 stars to movies.
- Similar trends are experienced in groups between 30 to 39. However, the difference in number 4 ratings is of about 6.000 reviews. The distribution now is split into ratings 3 and 4. They rate in a self-positive manner.
- The older the user is the more serious the ratings are. The curve of users through ages 40 and above are almost horizontal.
- **Conclusion:** It is impotant to bear into account that younger people tend to rate higher. Therefore, it affects future predictions. One way to reduce this effect is by adding baseline estimates and reduce this effect.

GENDER

> Now, analyzing the gender distribution is necessary. A question pop up: **Does women rato more than men?**

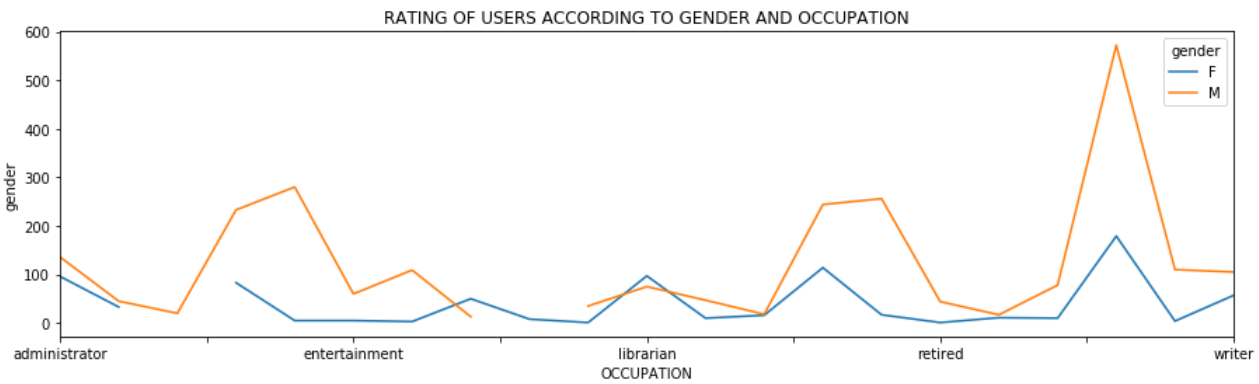
```
# SEARCH TREND BETWEEN GENDERS.

gender_dislikes = df.pivot_table(index=['movie id', 'movie title'],
                                 columns=['gender', ],
                                 values='rating',
                                 fill_value=0)
gender_dislikes[:5]
```

gender		F	M
movie id	movie title		
1	Toy Story	3.789916	3.909910
2	GoldenEye	3.368421	3.178571
3	Four Rooms	2.687500	3.108108
4	Get Shorty	3.400000	3.591463
5	Copycat	3.772727	3.140625

> It is interesting to see that there is no big difference between genders in relation to the different ratings. Therefore, there is no trend to identify if women tend to rate higher than men.

> Now, driving through the **micro analysis**. Question to solve: In what percentage the occupation and gender distribution affects the number of ratings received? Is there a trend in occupation?

```
fig, ax = plt.subplots(figsize=(15,4))
df.groupby(['occupation', 'gender']).count()['rating'].unstack().plot(ax=ax)
plt.title("RATING OF USERS ACCORDING TO GENDER AND OCCUPATION")
plt.ylabel('gender')
plt.xlabel('OCCUPATION');
plt.show()
```



> This graph is biased considering the previously mentioned distribution between men and women. Careers such as entertainment, writers and retired people are the ones who rate the most.

> **Is it possible to categorize gender considering its place?** Zip code will allow to determine trends based on regions.

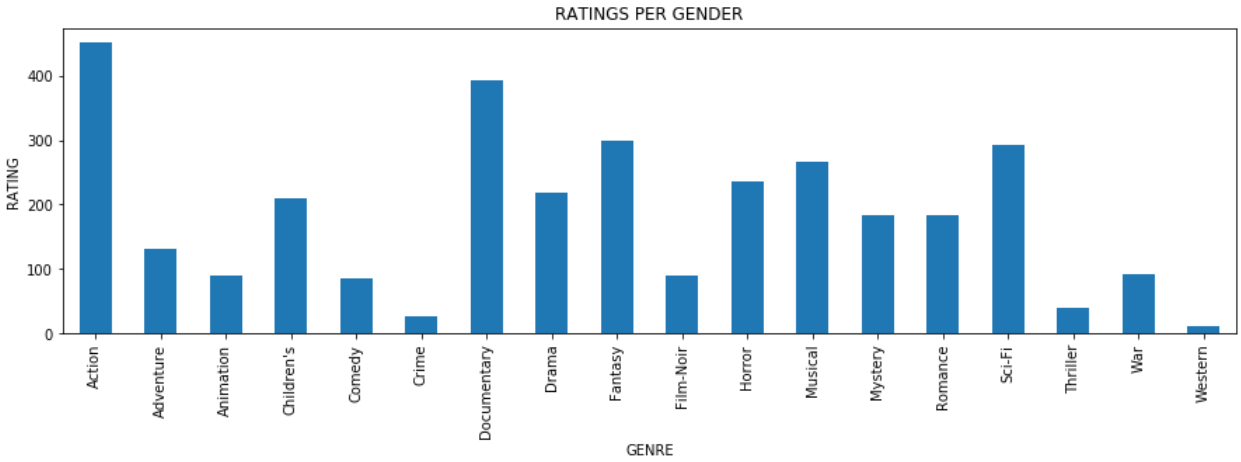
```
zip_code_influence = df.pivot_table(index=['zip code'],
                        columns=['gender'],
                        values='rating',
                        fill_value=0)
zip_code_influence[:5]
```

gender	F	M
zip code		
00000	1.0	5.0
01002	0.0	4.5
01080	0.0	3.5
01331	0.0	3.0
01375	0.0	4.4

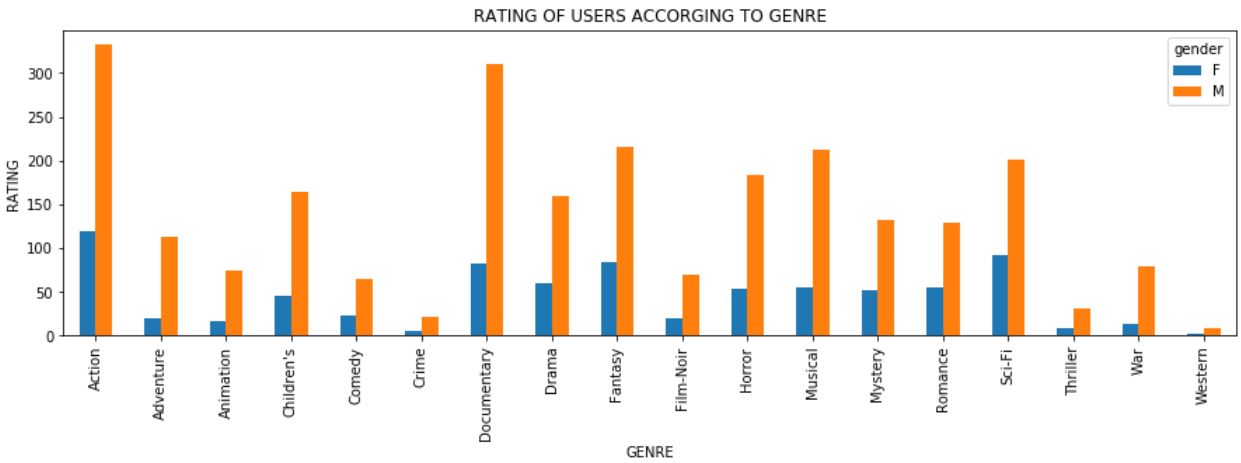
The table above could give interesting information: It is already known that men rate more than women. If detailed analyzing the data the extraction of regions where women rate the most drives the research in a way to determine movies' preference by region. Not included but possible to make. (It is impossible to cover all the different trends. Google Analytics would do the job =])

Popularity of Genres

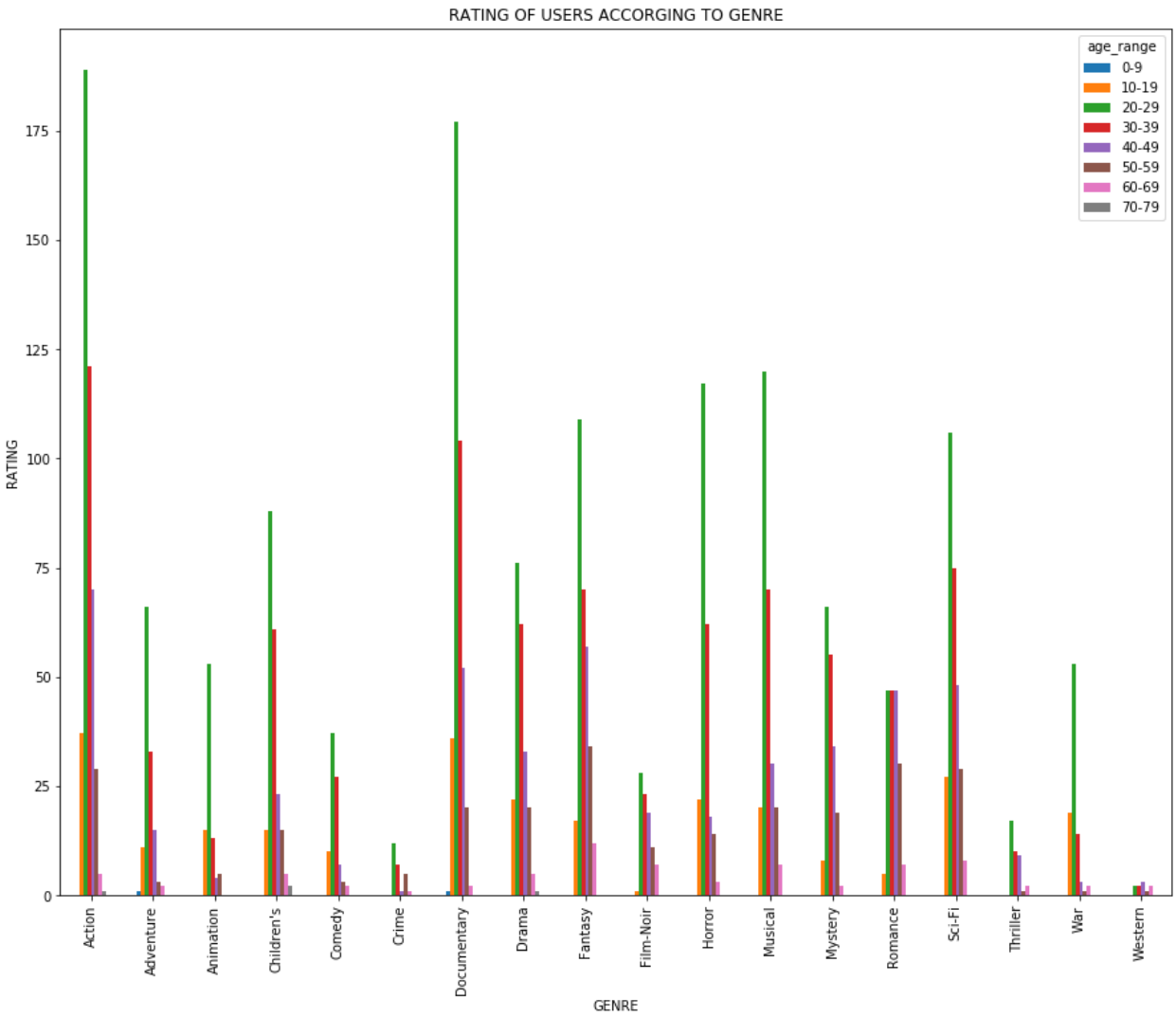
```
fig, ax = plt.subplots(figsize=(15,4))
df.groupby(['user id_y'])['rating'].count().plot.bar(ax=ax)
plt.title("RATINGS PER GENDER")
plt.ylabel('RATING')
plt.xlabel('GENRE');
plt.show()
```




```
fig, ax = plt.subplots(figsize=(15,4))
df.groupby(['user id_y', 'gender']).count()['rating'].unstack().plot.bar(ax=ax)
plt.title("RATING OF USERS ACCORGING TO GENRE")
plt.ylabel('RATING')
plt.xlabel('GENRE');
plt.show()
```




```
# DISTRIBUTION PER AGE

fig, ax = plt.subplots(figsize=(15,12))
df.groupby(['user id_y', 'age_range']).count()['rating'].unstack().plot.bar(ax=ax)
plt.title("RATING OF USERS ACCORGING TO GENRE")
plt.ylabel('RATING')
plt.xlabel('GENRE');
plt.show()
```



The above charts speak for themselves:

- Action, documentary, and Sci-fi are the most rated movies by users.
- Western, crime, and Thriller are the least rated ones.
- Women ratings are different than men: they like movies no matter the genre is.
- If we look closer the above graph, it is interesting to see how the ratings differs per group of age. For instance, users above its 40 tend to rate action movies, documentaries, fantasy movies, and they rate little comedy movies.
- On the contrary, young people like action movies, surprisingly documentaries, horror and fantasy movies, among others.

Conclusion: from this graph, a lot of insight could be gained for an ideal recommender system based on the age of the user.

2. Implement basic matrix factorization (MF)

Matrix Factorization process explained in few steps as a fast understanding of the topic:

> The information is not always complete. Therefore, for recommender systems predictions has to be made to improve the experience for users. Since a few datapoints are known the idea behind is to find the relationship between users and items. the main matrix is splitted for that purpose in two latent vectors:

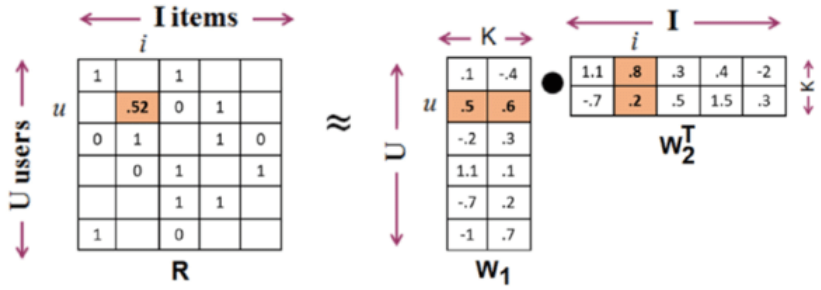
1. A vector containing **users** and the relationship between them
2. A vector containing **items** and the relationship between them

> This vectors are initialized at random.

> To get the predictions, after getting the values of the latent vectors, the dot product of them is going to output the predicted scores for the unknown items.

> The error measurement helps us to train the model and minimize the error by using SGD. The error is squared because the estimated values could be lower or greater than the real value.

> The system is able to recommend items to users.



source: [7]

Beginning of the code section

Since we want to make predictions, it is required to have the rows as users and the columns as movies. Therefore, table pivoting is required.

```
mf = df[['movie id','user id_x','rating','timestamp']]
mf = mf.pivot(index = 'user id_x', columns ='movie id', values = 'rating').fillna(0)
mf.head()
```

movie id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
user id_x																		
1	5.0	3.0	4.0	3.0	3.0	5.0	4.0	1.0	5.0	3.0	2.0	5.0	5.0	5.0	5.0	5.0	3.0	4.0
2	4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0	4.0	4.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5	4.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.0	0.0
6	4.0	0.0	0.0	0.0	0.0	0.0	2.0	4.0	4.0	0.0	0.0	4.0	2.0	5.0	3.0	0.0	0.0	0.0

```
# TRAIN AND TEST DATA SPLITTING
train = mf.sample(frac=0.7, random_state=1)
test = mf.drop(train.index)
train = train.values
test = test.values


# DATA NORMALIZATION
train =np.array(train)
test = np.array(test)
train = (train - train.min()) / (train.max() - train.min())
test = (test - test.min()) / (test.max() - test.min())
```

First, it is necessary to find the optimal values of K, alpha and beta. Therefore a Cross-validation process is executed.

```
class Matrix_Factorization():

    def __init__(self, R, TEST,K, a, b, epochs):

        self.R = R
        self.test = TEST
        self.users = R.shape[0]
        self.items = R.shape[1]
        self.K = K
        self.a = a
        self.b = b
        self.epochs = epochs

    def body(self):
        # LATENT FACTOR MATRICES HAS TO BE INITIALIZED AT RANDOM / MEASURE THE ERROR AND START TO MINIMIZE IT.
        self.U = np.random.normal(scale=1./self.K, size=(self.users, self.K))
        self.V = np.random.normal(scale=1./self.K, size=(self.items, self.K))


        # SAMPLING THE DATA FOR THE STOCHASTIC GRADIENT DESCENT PROCESS.
        self.samples = [
            (i, j, self.R[i, j])
```

```
            for i in range(self.users)
            for j in range(self.items)
            if self.R[i, j] > 0
        ]

        error_epoch1 = []
        RMSE_folds = []
        RMSE_avg = []
        hyperparameters = []
        pairs = [self.a,self.b, self.K]
        error_test = []
        RMSE_total = None
        error_epoch = []
        for i in range(self.epochs):
          np.random.shuffle(self.samples)
          self.SGD()
          mse_train, mse_test = self.MSE()
          error_epoch.append(mse_train)
          matrix = self.full_matrix
          RMSE_total = error_epoch
        RMSE_folds = sum(RMSE_total)/len(RMSE_total)
        RMSE_avg.append(RMSE_folds)
        hyperparameters.append(pairs)
        values = list(zip(RMSE_avg, hyperparameters))
        return values

    def MSE(self):
        # IMPORTANT STEP: AVOID ALL THE ZEROS VALUES.
        xs, ys = self.R.nonzero()
        xt, yt = self.test.nonzero()
        predicted = self.full_matrix()
        error_train = 0
        error_test = 0
        for x, y in zip(xs, ys):
            error_train += pow(self.R[x, y] - predicted[x, y], 2)
        for x, y in zip(xt, yt):
            error_test += pow(self.test[x, y] - predicted[x, y], 2)
        return np.sqrt(error_train), np.sqrt(error_test)


    def SGD(self):

        for i, j, r in self.samples:

            prediction = self.prediction(i, j)
            e = (r - prediction)

            # UPDATE OF THE LATENT MATRICES.
            self.U[i, :] += self.a * (e * self.V[j, :] - self.b * self.U[i,:])
            self.V[j, :] += self.a * (e * self.U[i, :] - self.b * self.V[j,:])


    def prediction(self, i, j):

        prediction = self.U[i, :].dot(self.V[j, :].T)
        return prediction

    def full_matrix(self):

        matrix = self.U.dot(self.V.T)
        return matrix

# SOURCE = [5], [6]


a = [0.1, 0.001, 0.01]
b = [0.001, 0.01, 0.1]
K = [20,100, 50]
valores = []
v= []
for alpha in a:
    for beta in b:
        for k in K:
          matrix_recommendation = Matrix_Factorization(train,test, k, alpha, beta, 50)
          values = matrix_recommendation.body()
          v.append(values)
valores.append(v)
optimum = min(valores)
print('Optimum alpha, beta and K:',optimum[1])
```

```
Optimum alpha, beta and K: [(2.4660849727463448, [0.1, 0.001, 100])]
```

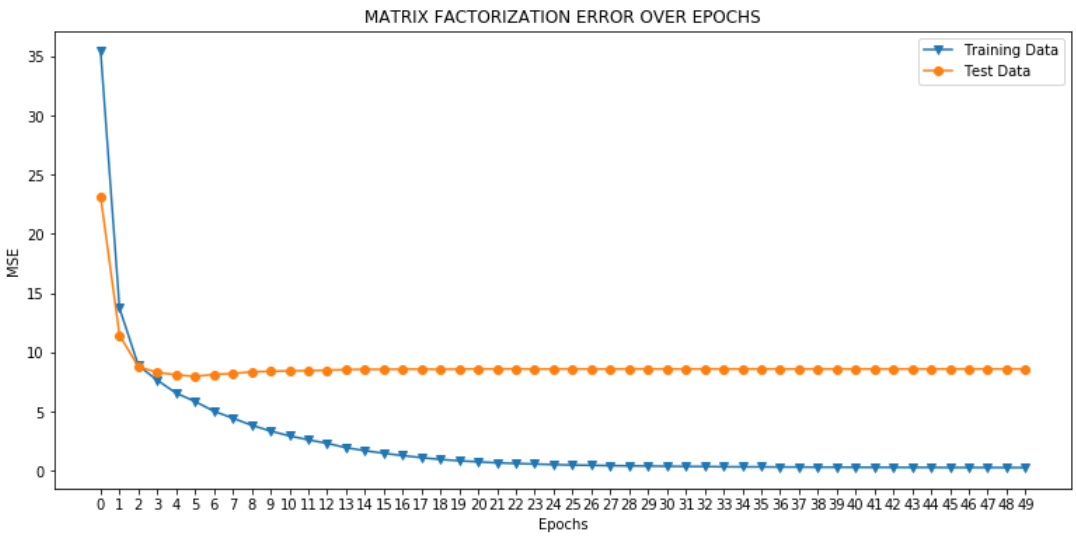
The next step is to train the model with the optimal hyperparameters and test its accuracy.

```
class Matrix_Factorization():

    def __init__(self, R, TEST, K, a, b, epochs):

        self.R = R
        self.test = TEST
        self.users = R.shape[0]
        self.items = R.shape[1]
        self.K = K
        self.a = a
        self.b = b
        self.epochs = epochs


    def body(self):
        # LATENT FACTOR MATRICES HAS TO BE INITIALIZED AT RANDOM / MEASURE THE ERROR AND START TO MINIMIZE IT.
        self.U = np.random.normal(scale=1./self.K, size=(self.users, self.K))
        self.V = np.random.normal(scale=1./self.K, size=(self.items, self.K))


        # SAMPLING THE DATA FOR THE STOCHASTIC GRADIENT DESCENT PROCESS.
        self.samples = [
            (i, j, self.R[i, j])
            for i in range(self.users)
            for j in range(self.items)
            if self.R[i, j] > 0
        ]

        error_epoch = []
        error_test = []
        for i in range(self.epochs):
            np.random.shuffle(self.samples)
            self.SGD()
            mse_train, mse_test = self.MSE()
            error_epoch.append((i, mse_train))
            error_test.append((i, mse_test))
            matrix = self.full_matrix

        return error_epoch, error_test

    def MSE(self):
        # IMPORTANT STEP: AVOID ALL THE ZEROS VALUES.
        xs, ys = self.R.nonzero()
        xt, yt = self.test.nonzero()
        predicted = self.full_matrix()
        error_train = 0
        error_test = 0
        for x, y in zip(xs, ys):
            error_train += pow(self.R[x, y] - predicted[x, y], 2)
        for x, y in zip(xt, yt):
            error_test += pow(self.test[x, y] - predicted[x, y], 2)
        return np.sqrt(error_train), np.sqrt(error_test)


    def SGD(self):

        for i, j, r in self.samples:

            prediction = self.prediction(i, j)
            e = (r - prediction)

            # UPDATE OF THE LATENT MATRICES.
            self.U[i, :] += self.a * (e * self.V[j, :] - self.b * self.U[i,:])
            self.V[j, :] += self.a * (e * self.U[i, :] - self.b * self.V[j,:])


    def prediction(self, i, j):

        prediction = self.U[i, :].dot(self.V[j, :].T)
        return prediction

    def full_matrix(self):

        matrix = self.U.dot(self.V.T)
        return matrix

# SOURCE = [5], [6]
```

```
matrix_recommendation = Matrix_Factorization(train,test, 100, 0.1, 0.001, 50)
error_train, error_test = matrix_recommendation.body()
print()
print("P x Q | MATRIX OF PREDICTIONS:","\n")
print(matrix_recommendation.full_matrix())
```

```
P x Q | MATRIX OF PREDICTIONS:

[[0.99150829 0.6161413  0.40250029 ... 0.67031611 0.79730639 0.41375444]
 [0.69177234 0.64086825 0.40493554 ... 0.72618932 0.6330671  0.49501329]
 [0.45270637 0.34401619 0.33419524 ... 0.4215859  0.33518394 0.28990053]
 ...
 [0.51662788 0.44713143 0.60013893 ... 0.5722466  0.40200073 0.49603852]
 [0.80344745 0.77514753 0.80004354 ... 0.92794464 0.7474292  0.69093359]
 [0.75081746 0.59903572 0.61982899 ... 0.7063873  0.6529432  0.49437901]]
```

```
x = [x for x, y in error_train]
y = [y for x, y in error_train]
x1 = [x1 for x1, y1 in error_test]
y1 = [y1 for x1, y1 in error_test]
plt.figure(figsize=((13,6)))
plt.plot(x, y, marker='v', label='Training Data')
plt.plot(x1, y1, marker='o', label='Test Data')
plt.xticks(x, x)
plt.xticks(x1, x1)
plt.title("MATRIX FACTORIZATION ERROR OVER EPOCHS")
plt.xlabel("Epochs")
plt.legend()
plt.ylabel("MSE")
plt.show()
```



The model implemented has a few problems:

> Even though it is converging the test set is not getting close to 0. Therefore, an overfitting is happening.

> By adding bias to the model could help to improve the accuracy of it.

> Moreover, the hyperparameters optimized are just a few. For better purposes and speed of the process just 3 of them were chosen. In action, it must use more than 5.

3. RECOMMENDER SYSTEMS USING MATRIX FACTORIZATION: libmf /sckit-learn

For this purpose I implemented a Cross-validated model in which the data is splitted in 3 folds. The process used is the following:

1. Iterate through the different values of alpha and K (model parameters)
2. Split the dataset in 3 folds in which the model is going to be trained and tested to measure the efectiveness of the parameters selected.
3. A question may pop up: why not use **gridsearchcv** directly? The problem with it is the requirement of a Y_true at the moment of *fitting* the model, therefore it is not a viable solution.
4. Next, the errors are appended and an average error per each fold is taken and is compared to the different errors output by the different hyperparameters combination.
5. Finally, the best combination is shown.

```
def RMSE_function(y_hat, y):
  xs, ys = y.nonzero()
  error_train = 0
  error_test = 0
  for x, y1 in zip(xs, ys):
    error_train += pow(y[x, y1] - y_hat[x, y1], 2)
  return np.sqrt(error_train)

def full_matrix(U, V):
  matrix = U.dot(V)
  return matrix


a = [0.1, 0.001, 0.01]
K = [20,100, 50]
valores = []
v= []
v1= []

hyperparameters = []
error_fold = []
for alpha in a:
  for k in K:
    kf = KFold(n_splits=3)
    kf.get_n_splits(train)
    error_cv = []
    v1 =[]
    for train_index, test_index in kf.split(train):
      X_train, X_test = train[train_index], train[test_index]
      pair = [alpha, k]
      model =  NMF(n_components=k, init=None, solver='cd', beta_loss='frobenius', tol=0.0001, max_iter=1, random_state=None, alpha=a
      W = model.fit_transform(X_train)
      H = model.components_
      y_hat = full_matrix(W, H)
      error = RMSE_function(y_hat,X_train)
      error_test = RMSE_function(y_hat,X_test)

      v.append(error)
      v1.append(error_test)
    error_cv.append(v1)
    avg_error = np.average(error_cv)
    error_fold.append(avg_error)
    hyperparameters.append(pair)
paired_results = list(zip(error_fold, hyperparameters))
optimum = min(paired_results)
print('Optimum alpha and K:',optimum)
```

```
Optimum alpha and K: (15.876422382628716, [0.01, 20])
```

```
# MODELING THE ALGORITHM BY USING THE OPTIMUM ALPHA AND K.

model =  NMF(alpha=0.01, init='random', l1_ratio=0.001, max_iter=100,
n_components=100, random_state=0, shuffle=False, solver='cd', tol=0.0001,
verbose=0)
model.fit(train)
result = model.inverse_transform(model.transform(train))


print(result)
```

```
[[9.99976896e-01 4.44577638e-03 3.99735930e-01 ... 5.35762312e-04
  7.98704767e-01 3.32020563e-04]
 [7.08926908e-03 1.60165326e-03 3.99687297e-01 ... 1.29123079e-04
  1.54096666e-03 3.99676114e-04]
 [1.16077579e-03 0.00000000e+00 8.64761532e-07 ... 0.00000000e+00
  0.00000000e+00 0.00000000e+00]
 ...
 [5.47259426e-03 4.84814206e-03 5.99586287e-01 ... 1.47095686e-04
  3.99374283e-01 4.68298629e-04]
 [7.99978491e-01 3.07646718e-03 7.99385773e-01 ... 5.66536323e-04
  3.17095234e-03 7.44010087e-04]
 [1.06000042e-03 5.99925009e-01 4.49679047e-05 ... 0.00000000e+00
  3.35221575e-03 5.26973617e-05]]
```

```
error_list =[]
final_Error =[]
total = []

model =  NMF(alpha=0.01, init='random', l1_ratio=0.001, max_iter=100,
```

```
n_components=20, random_state=0, shuffle=False, solver='cd', tol=0.0001,
verbose=0)
model.fit(train)
result = model.inverse_transform(model.transform(train))
print('W*H matrix uploaded:','\n', result)
W = model.fit_transform(train)
H = model.components_
y_hat = full_matrix(W, H)
error = RMSE_function(y_hat,train)
error_test = RMSE_function(y_hat,test)
print()
print('Error training', error)
print()
print('Error test', error_test)
```

```
W*H matrix uploaded:
 [[1.00067697e+00 2.62171208e-03 3.96850035e-01 ... 1.84279097e-04
  7.97265305e-01 3.55736575e-02]
 [3.25192673e-03 1.10513180e-03 3.96731304e-01 ... 0.00000000e+00
  1.41994841e-03 3.92267769e-02]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 ... 0.00000000e+00
  8.28808987e-08 1.79904007e-05]
 ...
 [2.52041954e-03 1.02165769e-03 5.95801555e-01 ... 0.00000000e+00
  3.98632477e-01 3.21727329e-02]
 [8.01055852e-01 1.32501215e-03 7.94104072e-01 ... 1.37378309e-04
  2.21840799e-03 5.56099882e-02]
 [0.00000000e+00 5.98496438e-01 0.00000000e+00 ... 0.00000000e+00
  5.89597420e-04 0.00000000e+00]]

Error training 1.3416310560085176

Error test 20.0661168174918
```

The difference between the model previously used and sklearn is that the standardized beta loss used: Frobenius. In this case this loss evaluates the distance between the value of X to the ones in WH. By using that loss function the model converges faster.

BIBLIOGRAPHY

[1] http://www.gregreda.com/2013/10/26/using-pandas-on-the-movielens-dataset/

[2] http://enhancedatascience.com/2017/04/22/building-recommender-scratch/

[3] https://scentellegher.github.io/programming/2017/07/15/pandas-groupby-multiple-columns-plot.html

[4] https://www.kaggle.com/cesarcf1977/movielens-data-analysis-beginner-s-first

[5] http://www.albertauyeung.com/post/python-matrix-factorization/

[6] https://towardsdatascience.com/music-artist-recommender-system-using-stochastic-gradient-descent-machine-learning-from-scratch-5f2f1aae972c

[7] https://www.researchgate.net/figure/An-example-of-matrix-factorization_fig1_314071424

[8] https://www.kaggle.com/gspmoreira/recommender-systems-in-python-101