Collaborative Filtering Recommender Systems

steps followed by code:

- 1- Movie ratings dataset
- 2- Collaborative filtering learning algorithm
  - Collaborative filtering cost function
- 3- Learning movie recommendations
- 4- Recommendations

The goal of a collaborative filtering recommender system is to generate two vectors: For each user, a 'parameter vector' that embodies the movie tastes of a user. For each movie, a feature vector of the same size which embodies some description of the movie. The dot product of the two vectors plus the bias term should produce an estimate of the rating the user might give to that movie.

In [2]:
```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from recsys_utils import *
```

1 - Movie ratings dataset:

The data set is derived from the MovieLens "ml-latest-small" dataset.
[F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4: 19:1–19:19. https://doi.org/10.1145/2827872]

The original dataset has 9000 movies rated by 600 users. The dataset has been reduced in size to focus on movies from the years since 2000. This dataset consists of ratings on a scale of 0.5 to 5 in 0.5 step increments. The reduced dataset has $n_u = 443$ users, and $n_m = 4778$ movies.

- The matrix Y (a $n_m \times n_u$ matrix) stores the ratings $y^{(i,j)}$. The matrix R is an binary-valued indicator matrix, where $R(i,j) = 1$ if user j gave a rating to movie i, and $R(i,j) = 0$ otherwise.
- matrices $\mathbf{X}$, $\mathbf{W}$ and $\mathbf{b}$:

$$\mathbf{X} = \begin{bmatrix} --- (\mathbf{x}^{(0)})^T --- \\ --- (\mathbf{x}^{(1)})^T --- \\ \vdots \\ --- (\mathbf{x}^{(n_m-1)})^T --- \end{bmatrix}, \quad \mathbf{W} = \begin{bmatrix} --- (\mathbf{w}^{(0)})^T --- \\ --- (\mathbf{w}^{(1)})^T --- \\ \vdots \\ --- (\mathbf{w}^{(n_u-1)})^T --- \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b^{(0)} \\ b^{(1)} \\ \vdots \\ b^{(n_u-1)} \end{bmatrix}$$

- The i-th row of $\mathbf{X}$ corresponds to the feature vector $x^{(i)}$ for the i-th movie, and the j-th row of $\mathbf{W}$ corresponds to one parameter vector $\mathbf{w}^{(j)}$, for the j-th user. --> Both $x^{(i)}$ and $\mathbf{w}^{(j)}$ are n-dimensional vectors.--> In this code, $n = 10$, and therefore, $\mathbf{x}^{(i)}$ and $\mathbf{w}^{(j)}$ have 10 elements. Correspondingly, $\mathbf{X}$ is a $n_m \times 10$ matrix and $\mathbf{W}$ is a $n_u \times 10$ matrix.

Loading $\mathbf{X}$, $\mathbf{W}$, and $\mathbf{b}$ with pre-computed values:

In [3]:
```
#Load data
X, W, b, num_movies, num_features, num_users = load_precalc_params_small()
Y, R = load_ratings_small()

print("Y", Y.shape, "R", R.shape)
print("X", X.shape)
print("W", W.shape)
print("b", b.shape)
print("num_features", num_features)
print("num_movies",   num_movies)
print("num_users",    num_users)
```

```
Y (4778, 443) R (4778, 443)
X (4778, 10)
W (443, 10)
b (1, 443)
num_features 10
num_movies 4778
num_users 443
```

In [4]:
```
#compute statistics like average rating
tsmean =  np.mean(Y[0, R[0, :].astype(bool)])
print(f"Average rating for movie 1 : {tsmean:0.3f} / 5" )
```

```
Average rating for movie 1 : 3.400 / 5
```

2 - Collaborative filtering learning algorithm:

The collaborative filtering algorithm in the setting of movie recommendations considers a set of n-dimensional parameter vectors $\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(n_m-1)}$, $\mathbf{w}^{(0)}, \ldots, \mathbf{w}^{(n_u-1)}$ and $b^{(0)}, \ldots, b^{(n_u-1)}$, where the model predicts the rating for movie i by user j as $y^{(i,j)} = \mathbf{w}^{(j)} \cdot \mathbf{x}^{(i)} + b^{(j)}$. Given a dataset that consists of a set of ratings produced by some users on some movies, the goal is learning the parameter vectors $\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(n_m-1)}, \mathbf{w}^{(0)}, \ldots, \mathbf{w}^{(n_u-1)}$ and $b^{(0)}, \ldots, b^{(n_u-1)}$ that produce the best fit (minimizes the squared error).

Collaborative filtering cost function:

The collaborative filtering cost function: $$J({\mathbf{x}^{(0)},...,\mathbf{x}^{(n_m-1)},\mathbf{w}^{(0)},b^{(0)},...,\mathbf{w}^{(n_u-1)},b^{(nu-1)}})= \left[\frac{1}{2}\sum{(i,j):r(i,j)=1}(\mathbf{w}^{(j)} \cdot \mathbf{x}^{(i)} + b^{(j)} - y^{(i,j)})^2 \right]

- \underbrace{\left[\frac{\lambda}{2} \sum_{j=0}^{nu-1}\sum{k=0}^{n-1}(\mathbf{w}^{(j)}_k)^2
- \frac{\lambda}{2}\sum_{i=0}^{nm-1}\sum{k=0}^{n-1}(\mathbf{x}k^{(i)})^2 \right]}{regularization} \tag{1}$$
  The first summation in (1) is "for all i, j where $r(i, j)$ equals 1" and could be written:

$$= \left[\frac{1}{2} \sum_{j=0}^{n_u-1} \sum_{i=0}^{n_m-1} r(i,j) * (\mathbf{w}^{(j)} \cdot \mathbf{x}^{(i)} + b^{(j)} - y^{(i,j)})^2 \right] + \text{regularization}$$

The cofiCostFunc (collaborative filtering cost function) will return this cost.

Consider developing the cost function in two steps:

- First, develop the cost function without regularization. --> A test case that does not include regularization will test the implementation. --> Then, add regularization and run the tests that include regularization.

In [5]:
```
def cofi_cost_func(X, W, b, Y, R, lambda_):
    """
    Returns the cost for the content-based filtering
    Args:
      X (ndarray (num_movies,num_features)): matrix of item features
      W (ndarray (num_users,num_features)) : matrix of user parameters
      b (ndarray (1, num_users)            : vector of user parameters
      Y (ndarray (num_movies,num_users)    : matrix of user ratings of movies
      R (ndarray (num_movies,num_users)    : matrix, where R(i, j) = 1 if the i-th movie
      lambda_ (float): regularization parameter
    Returns:
      J (float) : Cost
    """
    nm, nu = Y.shape
    J = 0

    for j in range(nu):
        w = W[j,:]
        b_j = b[0,j]
        for i in range(nm):
            x = X[i,:]
            y = Y[i,j]
            r = R[i,j]
            J += np.square(r * (np.dot(w,x) + b_j - y))

    regularization_term = (lambda_/2) * (np.sum(np.square(W)) + np.sum(np.square(X)))
    J = J/2 + regularization_term

    return J
```

In [6]:
```
# Reduce the data set size so that this runs faster
num_users_r = 4
num_movies_r = 5
num_features_r = 3

X_r = X[:num_movies_r, :num_features_r]
W_r = W[:num_users_r,  :num_features_r]
b_r = b[0, :num_users_r].reshape(1,-1)
Y_r = Y[:num_movies_r, :num_users_r]
R_r = R[:num_movies_r, :num_users_r]

# Evaluate cost function
J = cofi_cost_func(X_r, W_r, b_r, Y_r, R_r, 0);
print(f"Cost: {J:0.2f}")
```

```
Cost: 13.67
```

In [7]:
```
# Evaluate cost function with regularization
J = cofi_cost_func(X_r, W_r, b_r, Y_r, R_r, 1.5);
print(f"Cost (with regularization): {J:0.2f}")
```

```
Cost (with regularization): 28.09
```

In [8]:
```
# Public tests
from public_tests_collaborative import *
test_cofi_cost_func(cofi_cost_func)
```

```
All tests passed!
```

Vectorized Implementation:

In [9]:
```
def cofi_cost_func_v(X, W, b, Y, R, lambda_):
    """
    Returns the cost for the content-based filtering
    Vectorized for speed. Uses tensorflow operations to be compatible with custom traini
    Args:
      X (ndarray (num_movies,num_features)): matrix of item features
      W (ndarray (num_users,num_features)) : matrix of user parameters
      b (ndarray (1, num_users)            : vector of user parameters
      Y (ndarray (num_movies,num_users)    : matrix of user ratings of movies
      R (ndarray (num_movies,num_users)    : matrix, where R(i, j) = 1 if the i-th movie
      lambda_ (float): regularization parameter
    Returns:
      J (float) : Cost
    """
    j = (tf.linalg.matmul(X, tf.transpose(W)) + b - Y)*R
    J = 0.5 * tf.reduce_sum(j**2) + (lambda_/2) * (tf.reduce_sum(X**2) + tf.reduce_sum(W
    return J
```

In [10]:
```
# Evaluate cost function
J = cofi_cost_func_v(X_r, W_r, b_r, Y_r, R_r, 0);
print(f"Cost: {J:0.2f}")

# Evaluate cost function with regularization
J = cofi_cost_func_v(X_r, W_r, b_r, Y_r, R_r, 1.5);
print(f"Cost (with regularization): {J:0.2f}")
```

```
Cost: 13.67
Cost (with regularization): 28.09
```

3 - Learning movie recommendations:

Training the algorithm to make movie recommendations, and a list of all movies in the dataset is in the file small_movie_list.csv

In [11]:
```
movieList, movieList_df = load_Movie_List_pd()

my_ratings = np.zeros(num_movies)          #  Initialize ratings

# the file small_movie_list.csv has id of each movie in the dataset
# For example, Toy Story 3 (2010) has ID 2700, so to rate it "5"
my_ratings[2700] = 5

#Or suppose you did not enjoy Persuasion (2007)
my_ratings[2609] = 2;

# A few movies are liked / are not liked have been selected and the ratings are gaved to
my_ratings[929]  = 5   # Lord of the Rings: The Return of the King, The
my_ratings[246]  = 5   # Shrek (2001)
my_ratings[2716] = 3   # Inception
my_ratings[1150] = 5   # Incredibles, The (2004)
my_ratings[382]  = 2   # Amelie (Fabuleux destin d'Amélie Poulain, Le)
my_ratings[366]  = 5   # Harry Potter and the Sorcerer's Stone (a.k.a. Harry Potter and
my_ratings[622]  = 5   # Harry Potter and the Chamber of Secrets (2002)
my_ratings[988]  = 3   # Eternal Sunshine of the Spotless Mind (2004)
my_ratings[2925] = 1   # Louis Theroux: Law & Disorder (2008)
my_ratings[2937] = 1   # Nothing to Declare (Rien à déclarer)
my_ratings[793]  = 5   # Pirates of the Caribbean: The Curse of the Black Pearl (2003)
my_rated = [i for i in range(len(my_ratings)) if my_ratings[i] > 0]
```

```
for i in range(len(my_ratings)):
    if my_ratings[i] > 0 :
        print(f'Rated {my_ratings[i]} for  {movieList_df.loc[i,"title"]}');
```

```
New user ratings:

Rated 5.0 for  Shrek (2001)
Rated 5.0 for  Harry Potter and the Sorcerer's Stone (a.k.a. Harry Potter and the Philos
opher's Stone) (2001)
Rated 2.0 for  Amelie (Fabuleux destin d'Amélie Poulain, Le) (2001)
Rated 5.0 for  Harry Potter and the Chamber of Secrets (2002)
Rated 5.0 for  Pirates of the Caribbean: The Curse of the Black Pearl (2003)
Rated 5.0 for  Lord of the Rings: The Return of the King, The (2003)
Rated 3.0 for  Eternal Sunshine of the Spotless Mind (2004)
Rated 5.0 for  Incredibles, The (2004)
Rated 2.0 for  Persuasion (2007)
Rated 5.0 for  Toy Story 3 (2010)
Rated 3.0 for  Inception (2010)
Rated 1.0 for  Louis Theroux: Law & Disorder (2008)
Rated 1.0 for  Nothing to Declare (Rien à déclarer) (2010)
```

Then, adding these reviews to Y and R and normalize the ratings.

In [12]:
```
# Reloading ratings
Y, R = load_ratings_small()

# Adding new user ratings to Y
Y = np.c_[my_ratings, Y]

# Adding new user indicator matrix to R
R = np.c_[(my_ratings != 0).astype(int), R]

# Normalizing the Dataset
Ynorm, Ymean = normalizeRatings(Y, R)
```

Then, training the model--> Initializing the parameters and selecting the Adam optimizer.

In [13]:
```
#  Useful Values
num_movies, num_users = Y.shape
num_features = 100

tf.random.set_seed(1234) # for consistent results
W = tf.Variable(tf.random.normal((num_users,  num_features),dtype=tf.float64),  name='W'
X = tf.Variable(tf.random.normal((num_movies, num_features),dtype=tf.float64),  name='X'
b = tf.Variable(tf.random.normal((1, num_users),   dtype=tf.float64),  name='b')

# Instantiate an optimizer.
optimizer = keras.optimizers.Adam(learning_rate=1e-1)
```

In [14]:
```
iterations = 200
lambda_ = 1
for iter in range(iterations):
    # Using TensorFlow's GradientTape
    # to record the operations used to compute the cost
    with tf.GradientTape() as tape:

        # Compute the cost (forward pass included in cost)
        cost_value = cofi_cost_func_v(X, W, b, Ynorm, R, lambda_)

    # Using the gradient tape to automatically retrieve
    # the gradients of the trainable variables with respect to the loss
    grads = tape.gradient( cost_value, [X,W,b] )
```

```
    # Run one step of gradient descent by updating
    # the value of the variables to minimize the loss.
    optimizer.apply_gradients( zip(grads, [X,W,b]) )

    # Log periodically.
    if iter % 20 == 0:
        print(f"Training loss at iteration {iter}: {cost_value:0.1f}")
```

```
Training loss at iteration 0: 2321191.3
Training loss at iteration 20: 136169.3
Training loss at iteration 40: 51863.7
Training loss at iteration 60: 24599.0
Training loss at iteration 80: 13630.6
Training loss at iteration 100: 8487.7
Training loss at iteration 120: 5807.8
Training loss at iteration 140: 4311.6
Training loss at iteration 160: 3435.3
Training loss at iteration 180: 2902.1
```

4 - Recommendations:

- Computing the ratings for all the movies and users and displaying the movies that are recommended. These are based on the movies and ratings entered as my_ratings[] above.
- To predict the rating of movie i for user j --> computing $\mathbf{w}^{(j)} \cdot \mathbf{x}^{(i)} + b^{(j)}$. This can be computed for all ratings using matrix multiplication.

In [15]:
```
# Make a prediction using trained weights and biases
p = np.matmul(X.numpy(), np.transpose(W.numpy())) + b.numpy()

#restore the mean
pm = p + Ymean

my_predictions = pm[:,0]

# sort predictions
ix = tf.argsort(my_predictions, direction='DESCENDING')

for i in range(17):
    j = ix[i]
    if j not in my_rated:
        print(f'Predicting rating {my_predictions[j]:0.2f} for movie {movieList[j]}')

print('\n\nOriginal vs Predicted ratings:\n')
for i in range(len(my_ratings)):
    if my_ratings[i] > 0:
        print(f'Original {my_ratings[i]}, Predicted {my_predictions[i]:0.2f} for {movieL
```

```
Predicting rating 4.49 for movie My Sassy Girl (Yeopgijeogin geunyeo) (2001)
Predicting rating 4.48 for movie Martin Lawrence Live: Runteldat (2002)
Predicting rating 4.48 for movie Memento (2000)
Predicting rating 4.47 for movie Delirium (2014)
Predicting rating 4.47 for movie Laggies (2014)
Predicting rating 4.47 for movie One I Love, The (2014)
Predicting rating 4.46 for movie Particle Fever (2013)
Predicting rating 4.45 for movie Eichmann (2007)
Predicting rating 4.45 for movie Battle Royale 2: Requiem (Batoru rowaiaru II: Chinkonk
a) (2003)
Predicting rating 4.45 for movie Into the Abyss (2011)


Original vs Predicted ratings:

Original 5.0, Predicted 4.90 for Shrek (2001)
Original 5.0, Predicted 4.84 for Harry Potter and the Sorcerer's Stone (a.k.a. Harry Pot
ter and the Philosopher's Stone) (2001)
Original 2.0, Predicted 2.13 for Amelie (Fabuleux destin d'Amélie Poulain, Le) (2001)
Original 5.0, Predicted 4.88 for Harry Potter and the Chamber of Secrets (2002)
Original 5.0, Predicted 4.87 for Pirates of the Caribbean: The Curse of the Black Pearl
(2003)
Original 5.0, Predicted 4.89 for Lord of the Rings: The Return of the King, The (2003)
Original 3.0, Predicted 3.00 for Eternal Sunshine of the Spotless Mind (2004)
Original 5.0, Predicted 4.90 for Incredibles, The (2004)
Original 2.0, Predicted 2.11 for Persuasion (2007)
Original 5.0, Predicted 4.80 for Toy Story 3 (2010)
Original 3.0, Predicted 3.00 for Inception (2010)
Original 1.0, Predicted 1.41 for Louis Theroux: Law & Disorder (2008)
Original 1.0, Predicted 1.26 for Nothing to Declare (Rien à déclarer) (2010)
```

Above, the predicted ratings for the first few hundred movies lie in a small range. --> Then, selecting from those top movies, movies that have high average ratings and movies with more than 20 ratings.

In [16]:
```
filter=(movieList_df["number of ratings"] > 20)
movieList_df["pred"] = my_predictions
movieList_df = movieList_df.reindex(columns=["pred", "mean rating", "number of ratings",
movieList_df.loc[ix[:300]].loc[filter].sort_values("mean rating", ascending=False)
```

Out[16]:

	pred	mean rating	number of ratings	title
1743	4.030961	4.252336	107	Departed, The (2006)
2112	3.985281	4.238255	149	Dark Knight, The (2008)
211	4.477798	4.122642	159	Memento (2000)
929	4.887054	4.118919	185	Lord of the Rings: The Return of the King, The...
2700	4.796531	4.109091	55	Toy Story 3 (2010)
653	4.357304	4.021277	188	Lord of the Rings: The Two Towers, The (2002)
1122	4.004471	4.006494	77	Shaun of the Dead (2004)
1841	3.980649	4.000000	61	Hot Fuzz (2007)
3083	4.084643	3.993421	76	Dark Knight Rises, The (2012)
2804	4.434171	3.989362	47	Harry Potter and the Deathly Hallows: Part 1 (...
773	4.289676	3.960993	141	Finding Nemo (2003)
1771	4.344999	3.944444	81	Casino Royale (2006)
2649	4.133481	3.943396	53	How to Train Your Dragon (2010)
2455	4.175743	3.887931	58	Harry Potter and the Half-Blood Prince (2009)
361	4.135287	3.871212	132	Monsters, Inc. (2001)
3014	3.967900	3.869565	69	Avengers, The (2012)
246	4.897137	3.867647	170	Shrek (2001)
151	3.971892	3.836364	110	Crouching Tiger, Hidden Dragon (Wo hu cang lon...
1150	4.898892	3.836000	125	Incredibles, The (2004)
793	4.874936	3.778523	149	Pirates of the Caribbean: The Curse of the Bla...
366	4.843375	3.761682	107	Harry Potter and the Sorcerer's Stone (a.k.a. ...
754	4.021778	3.723684	76	X2: X-Men United (2003)
79	4.242986	3.699248	133	X-Men (2000)
622	4.878342	3.598039	102	Harry Potter and the Chamber of Secrets (2002)

In [ ]:

In [ ]: