Algorithms Final Project Essay

Luca Frattegiani (1013326)

September 2022

Introduction

Recommender Systems

Recommender Systems are a type of content filtering systems. They can be defined as algorithms that are intended to suggest specific items of a certain class (which can be a product, a film, a website, ...) to a target user in a way such that the suggested article can be interesting to him. So these algorithms produces personalized recommendations and in this project we're going to focus on three different kinds of *"Collaborative Filtering Methods"*.
Collaborative Filtering Methods are based on the information provided by the user-item "Interaction Matrix" where the interactions between users (usually put as rows of the matrix) and items (usually put as columns of the matrix) are used to extract the suggestions (for instance, the interactions between users and films are represented by the ratings from "0" to "5" that the user expresses for each film he has seen). So each user (row of the matrix) correspond to a sparse vector of interactions (where the sparsity is due to the fact that the user hasn't interact with all the items).

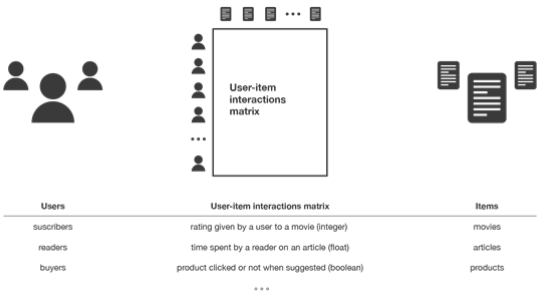

Figure 1: Interaction Matrix

They can be mainly divided in:

- **Memory-based methods:**
  These methods are concerned on using the information contained in the Intraction Matrix, indeed users are evaluated through their past interaction contained in it and suggestions are based on the research of the nearest neighbors. Furthermore, there're two types of memory-based algorithms:

  1. *User-based:* With this class of algorithms, for each target user, first of all we search its k-nearest neighbors (among the other users) in the interaction matrix, where distance is computed with respect to the interactions between users and items (so it expresses in terms of appreciations). Once we've found the k-nearest neighbors, we try to predict the rating that the target user will assign to its "unseen" items (the ones the target user hasn't interacted with) by computing the average mean of the ratings provided for those items by the k-nearest neighbors. The underlying assumption is that the suggestions for the target user can be extracted from the ratings provided by users with similar interactions.
  2. *Item-based:* These algorithms, for each target user, consider its favourite items (the ones with the highest rating) and then search the k-nearest items to target user's favourite ones. So, the suggestion the system assumes is that the user will appreciate items which are the most similar to its favourite ones, where two items are considered similar if the same users interact similarly with them (so the similarity can be computed form the Interaction Matrix).

- **Model-based methods**:
  These methods try to explain the interactions by defining an underlying model which is able to reproduce the Interaction Matrix. So there's a new representation of the users and recommendations are made basing on the information provided by the model.
  The model-based algorithm we're going to build is based on the so called *"Non Negative Matrix Factorization"* of the Interaction Matrix, that aims to explain the recorded interactions in terms of "f" main factors.
  Factors can be interpreted as unseen latent sources that connect users to items explaining why they have interacted in a certain way. In the movie example, a factor can be a certain "type" of film like "Horror" or "Commedy" which can be both an attribute of a film and a taste of the user and so it explain why the user has appreciated or not the item (for instance, if a user tends to appreciate a certain kind of film and a movie is considered to be of that kind then their interaction will be positive).
  So we're going to decompose the original matrix (of dimension $n \times p$, where "n" is the number of users and "p" the number of items) into the cross product of two sub-matrices:

  - User-Factor Matrix: It's a $n \times f$ matrix containing the relationships between users and latent factors.
  - Factor-Items Matrix: It's a $f \times p$ matrix containing the relationships between latent factors and items.

  The estimation of the two sub-matrices is done by minimizing the error term in the approximation of the original Interaction Matrix (the details of the procedure implemented in the algorithm we'll be analyzed later). To sum up, the model decomposes the Interaction Matrix in such a way:

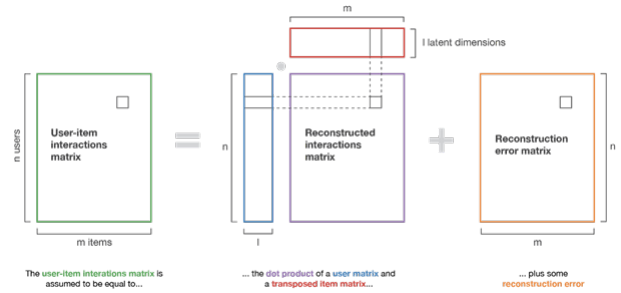

Figure 2: Interaction Matrix decomposition

Where the reconstruction of the Interaction Matrix is obtained from the cross product of the two matrices described above:

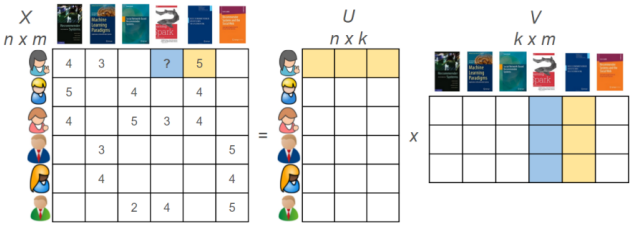

Figure 3: Sub-matrices

Support elements

Data

In order to test the performances of the algorithms we're going to use a dataset taken from Kaggle which is about movies. From that data we can extract our user Interaction Matrix to make recommendations.
Along the rows of the matrix we find the identifiers for the users while in the columns we've the movies selected as items and the entries are represented by the ratings assigned by the users to the films they've seen expressed in "number of stars" (so the measure of interaction is a numerical value between "0", which means that the user had completely disliked the item and "5", which means maximum appreciation).
The matrix appears as follow:

	Toy Story	Twelve Monkeys	Se7en	The Usual Suspects	Braveheart	Apollo 13	Star Wars	Pulp Fiction	The Shawshank Redemption	Forrest Gump	...	Independence Day	The Godfather	The Empire Strikes Back	Raiders of the Lost Ark	Return of the Jedi	Back to the Future
0	NaN	NaN	4.0	4.0	4.0	5.0	NaN	4.0	NaN	3.0	...	NaN	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN	4.0	NaN	NaN	4.5	5.0	5.0	...	NaN	NaN	NaN	NaN	3.0	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	5.0	5.0	NaN	5.0	...	NaN	5.0	5.0	5.0	5.0	5.0
3	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	4.0	...	NaN	2.5	NaN	NaN	NaN	NaN
4	3.0	NaN	NaN	NaN	5.0	NaN	5.0	NaN	5.0	3.0	...	3.0	NaN	5.0	5.0	5.0	3.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
484	NaN	4.0	NaN	NaN	NaN	NaN	4.0	4.0	5.0	4.0	...	NaN	NaN	4.0	NaN	4.0	5.0
485	NaN	NaN	NaN	NaN	2.0	4.0	NaN	4.0	NaN	5.0	...	NaN	NaN	NaN	NaN	NaN	NaN
486	NaN	5.0	NaN	NaN	4.0	4.0	NaN	5.0	NaN	4.0	...	NaN	NaN	NaN	NaN	NaN	NaN
487	4.0	3.0	5.0	5.0	3.0	3.0	NaN	NaN	5.0	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN
488	5.0	NaN	NaN	4.5	NaN	NaN	5.0	4.0	5.0	5.0	...	NaN	NaN	5.0	5.0	NaN	NaN

Figure 4: Data

The dataset contains the ratings recorded for 31 movies by 489 users and, as it is clear from the picture above, we wish to predict the possible rating assigned by each user to its "unseen" items (i. e. the entries of the matrix associated to a "NaN" value).
Such a type of data (with interactions represented by non negative numbers between 0 and 5) requires a particular kind of measure to correctly express distances between vectors, which the so called "cosine similarity" defined as:

$$sin(x,y) = cos(\theta) = \frac{x \cdot y}{||x||_e \cdot ||y||_e} = \frac{\sum_{i=0}^{n} x_i \cdot y_i}{\sqrt{\sum_{i=0}^{n} x_i^2} \cdot \sqrt{\sum_{i=0}^{n} y_i^2}}$$

where $x, y \in R^n$

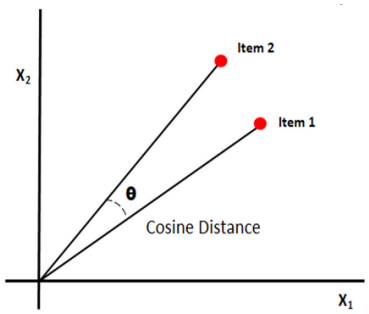

Figure 5: Cosine Similarity/Distance

It measures the cosine of the angle θ between two sparse vectors of the Interaction Matrix (made of the ratings assigned by the users to the movies) and it's a judgment of orientation rather than magnitude with respect to the origin. Since our vectors live in a positive space (the rating is always between 0 and 5), we've that $cos(\theta) \in [0, 1]$ and the cosine related to $\theta = 0$ degrees is 1 which means the data points are similar while the cosine related to $\theta = 90$ degrees is 0 which means data points are dissimilar. So, as the cosine similarity approaches 1, we'll consider the vectors to be "near" while as it approaches 0, we'll consider the data to be "distant".

Libraries and environment

The algorithms have been written in python3 language using both a script (file with extension ".py"), to import useful data structures and utility functions and Jupyter notebooks to implement and test the Recommender Systems. The libraries required for the task are (the time asymptotic complexity information have been found on online sources or have been guessed and we suppose to apply the methods to a data structure of size "n"):

1. *pandas:*
   Library to store and manage data imported from ".csv" files into pandas dataframe objects. The operations required are:
   (a) iloc ($O(1)$): Access to a row or an entry (if also the columns is specified) of the dataframe.
   (b) index $O(1)$: Returns a data structure containing the indexes of the dataframe.
   (c) drop ($O(1)$): Remove a row or a column from the dataframe.
   (d) isnull ($O(n)$): Returns a sequence of boolean values signalling where there's a null value or not.
   (e) filtering ($O(n)$): Returns a subset of the original dataframe where a certain condition is respected.
   (f) sum ($O(n)$): Sum all the elements in a row or column.
   (g) values ($O(n)$): Turns the row/column in a numpy array.
2. *numpy:*
   Library to perform computations required in the algorithms using numpy arrays. The operations required are:
   (a) isnan ($O(n)$): Returns an array of boolean values signalling where there's a null value or not.
   (b) random generation of "n" numbers ($O(n)$): Creates a numpy array filled with numbers randomly generated from a uniform distribution in certain range of values.
   (c) dot ($O(n)$): Computes the cross product between two numpy arrays of acceptable shape.
   (d) access ($O(1)$): Returns the element stored at the position specified by the given index.
   (e) sqrt ($O(n\sqrt{n})$): Computes the square root for each element inside of the numpy array.
   (f) append ($O(1)$): Insert an element at the end of the array.
   (g) reshape ($O(1)$): Change the dimensions of the array.
   (h) flatten ($O(1)$): Particular kind of reshaping where the array is turned in a 1-d one.
   (i) zeros ($O(n)$): Create a numpy array only made of "0".
3. *pyplot:*
   Library to plot some results related to the applied dataset.
4. *os:*
   Library to perform input operation that require to interact with the operative system.
5. *time:*
   Library to compute times required for the execution of the algorithms.

Data Structures

Our Recommender Systems require different type of data structures (in addiction to the numpy arrays and the pandas dataframes and series mentioned above) both because they're necessary in certain steps of the algorithms and because they provide a final nice representation of results. Some of them have been self-written (see the file "DataStructures.py" for their implementation) while other ones have been taken from the python built-in classes. Here we list them and briefly point out their features and roles in the algorithms (which will be discuss more in deep later):

- **Built-in data structures:**

  1. *lists*: Lists are a python built-in object which corresponds to a data structure that collects in an ordered sequence objects of different classes, each of them can be accessed by specifying its index position. They're a dynamic object, so we can modify their composition and size during the execution of the code (both by changing the object associated to a certain position in the list and by adding/removing some elements). We mainly use them to store the final result of the recommendations produced by an algorithm and in the implementation of other data structures. The operations required are:
     (a) append ($O(1)$): Insert an element at the end of the list.
     (b) remove ($O(n)$): Remove the element passed as input argument from the list after a research of its position.
     (c) pop ($O(1)$): Remove the element in the last position of the list.
  2. *tuples*: A tuple object, is an immutable sequence of ordered and heterogeneous elements (so we can access an object in a tuple by specifying its index position but we can't change the composition or size of the tuple during the execution of the code). We mainly it in some functions to return multiple results or in the implementation of other data structures. The class of complexity for accessing an element of a tuple is $O(1)$.
  3. *dictionaries*: Dictionaries are objects with a (key, value) structure, where the keys are used as indexes used to access the different items stores in the dictionary, so we associate an arbitrary element to each key and we can access the item by specifying the keys of the dictionary. We use them to collect the final results of the recommendations produced by the algorithms. The operations required are:
     (a) access ($O(1)$): Returns the item in the dictionary associated to the specified "key" (we consider the access time as $O(1)$ since we use only dictionaries with a small number of keys).
     (b) keys ($O(1)$): Returns an iterable object containing the keys of the dictionary.

- **Self-written data structures:**

  1. *Priority Queues*: A Priority Queue is a collection of pairs (key, item) used to store the "items" with an order determined by their associated "keys" which constitute a so called "priority" for the elements contained in the data structure. So keys define the order chosen for the objects inside the Priority Queue. They've been implemented through a list of tuples and we'll deal with the "MaxHeap" Priority Queue (where the root, or the first element of the list, is always the one associated to the highest key and its children are always pairs with a lower key). We mainly use it to find the nearest neighbors in the Interaction Matrix and highest predicted ratings. The operations required are:
     (a) insert ($O(\log_2(n))$): Inserts a new pair (key, item) into the Queue.
     (b) delete ($O(\log_2(n))$): Removes and returns the element with the highest priority from the Queue.
     (c) copy ($O(n)$): Returns a copy of the Priority Queue.
  2. *Circular Queue*: A Queue is a particular data structure to store different kind of objects and manage them with the FIFO (First In First Out) policy, so that the first element inserted in the Queue is always the first that is removed. Particularly, we implemented (also in that case by using lists and tuples) the circular version of the Queue to take advantage of its properties in terms of asymptotic complexity. We use them to store multiple "favourite items" in the algorithm related to the Item-based Recommender System. The operations required are:
     (a) enqueue ($O(1)$): Add a new element at the back of the Queue.
     (b) dequeue ($O(1)$): Remove the element at the front of the Queue which was the "oldest" element inserted in the Queue.
     (c) empty check ($O(1)$): Returns a boolean value which signals if the Queue is empty or not.

Built-in Functions

We make use of some built-in functions useful to some steps of the algorithms such as computing lengths or generate sequences. These are:

1. len(object): Returns the length of the object passed as input parameter.
2. range(n): Function that generates a sequence from "0" to "$n-1$".
3. round(numerical, digits): Function that rounds all the numerical elements (floats) inside the object passed as input argument keeping the specified number of digits.

Utility Functions

This are functions required to compute the distances between vectors on our data, they've been implemented using the libraries "pandas" and "numpy" (we decide to produce another version of these function even if there were built-in prototypes due to some adjustments needed cause of the nature of our data). We're going to show their code and comment.

- **Euclidean distance**:
  The function computes the usual euclidean distance $\left(||x-y||_e = \sqrt{\sum_{i=0}^{n} |x_i - y_i|^2}, x \in R^n \right)$ between vectors through the operations between numpy arrays, taking care only of the available values for the vector of interactions (i. e. the entries which aren't "NaN"). The function returns "inf" if the vector hasn't any valuable interaction. Here we report the code for the implementation of the function:

```
def euclidean(x, y):
    distance = np.nan
    for xi, yi in zip(x, y):
        if xi and yi:
            distance = np.nansum(distance) + np.sqrt((xi - yi)**2)

    if np.isnan(distance):
        return np.inf
    else:
        return distance
```

  Firstly, the algorithm sets the distance between vectors equal to "Nan" (in order to manage the case in which the vectors can't be compared for any of their entries, so in that situation we'll assign an infinite distance).
  Then we loop over all the elements in the two arrays to compute the euclidean distance (the use of the "np.nansum()" function is necessary due to the inizialization chosen for the distance variable).
  Finally, we check if the distance is "NaN" or not and we return its adequate value. Below we evaluate the function's asymptotic complexities:

  1. *Time:*
     For what concerns the time asymptotic complexity of this function, we assume that x and y are two numpy arrays of size "n". The algorithm performs the following operations:

Code	Costs	Times
def euclidean(x, y):		
distance = np.nan	1	1
for xi, yi in zip(x, y):	2	$n+1$
if xi and yi: distance = np.nansum(distance) + np.sqrt((xi - yi)**2)	8	n
if np.isnan(distance): return np.inf	2	1
else: return distance	1	1

So the running time of the algorithm is:

$$T(n) = 1 + 2 + 2n + 8n + 2 + 1 = 10n + 6$$

Which means it has asymptotic complexity of class $O(n)$.

2. *Space:*
   Instead, as regards the space asymptotic complexity, since the function doesn't create any additional copy of the input arrays given but just creates a new single variable to store the value of the euclidean distance, it's of class $O(1)$.

- **Cosine Similarity**:
  The function computes the cosine of the angle formed between the two vectors of interactions passed as input arguments (always using numpy arrays and methods). The first part of the algorithm aims to determine which are the indexes of the two arrays related to the positions where both of them have valid values of interactions, in order to determine which are the elements available to compute the cosine similarity. Then the calculation are performed only on the subset of indexes previously determined.

```
def cosine(x, y):
    valid = list()
    valid_x = (np.isnan(x) == False)
    valid_y = (np.isnan(y) == False)
    length = len(x)
    for bool1, bool2, index in zip(valid_x, valid_y, range(length)):
        if (bool1 and bool2):
            valid.append(index)

    x_ = x[valid].reshape((1, len(x[valid])))
    y_ = y[valid].reshape((1, len(y[valid])))

    cross_product = np.dot(x_, y_.T).flatten()[0]
    x_norm = euclidean(x_.flatten(), np.zeros(x_.shape[0]))
    y_norm = euclidean(y_.flatten(), np.zeros(y_.shape[0]))
    cosine = cross_product/(x_norm * y_norm)
    return cosine
```

Firstly, the algorithm select which are the valid entries for both the vectors in order to select the element useful to compute the distance.
Secondly, it extracts the correct sub-vectors corresponding to the indexes derived from the previous part of the code and reshape them correctly.
Finally, it computes the measure of the cosine similarity. Assuming that n is the size of the two input vectors and that $j \in [0, n]$ is the number of "valid entries" to compute the measure (we assume on average case $j = \frac{n}{2}$), we analyze its asymptotic complexities:

1. *Time:*

Code	Costs	Times
def cosine(x, y):		
valid = list()	2	1
validX = (np.isnan(x) == False)	n	1
validY = (np.isnan(y) == False)	n	1
lenght = len(x)	2	1
for bool1, bool2, index in zip(validX, validY, range(length)):	3	$j+1$
if (bool1 and bool2): valid.append(index)	3	j
xV = x[valid].reshape((1, len(x[valid])))	$2 \cdot (j+1)$	1
yV = y[valid].reshape((1, len(y[valid])))	$2 \cdot (j+1)$	1
crossProduct = np.dot(xV, yV.T).flatten()[0]	$j+1$	1
xV = xV.flatten()	1	1
yV = yV.flatten()	1	1
zeros = np.zeros(xV.shape[0])	$j+2$	1
xN = euclidean(xV, zeros)	$j+1$	1
yN = euclidean(yV, zeros)	$j+1$	1
cosine = crossProduct/(xN * yN)	3	1
return cosine	1	1

So the running time of the algorithm is:

$$T(n) = 2+n+n+2+3\cdot\frac{n}{2}+3+3\cdot\frac{n}{2}+2\cdot\frac{n}{2}+2+2\cdot\frac{n}{2}+2+\frac{n}{2}+1+1+1+\frac{n}{2}+2+\frac{n}{2}+1+\frac{n}{2}+1+3+1 = 22+10n$$

Which means it has asymptotic complexity of class $O(n)$.

2. *Space:*
The data structures created by this function during its execution are:
   - "validX": Which is a numpy array of length equal to the number of valid elements for vector "x" ($j = \frac{n}{2}$).
   - "validY": Which is a numpy array of length equal to the number of valid elements for vector "y".
   - "valid": Which is a numpy array of length equal to the number of elements which are simultaneously valid both for vector "x" and "y". According to our data, we found this length be on average equal to a quart of the input size, $\frac{n}{4}$.
   - "zeros": Which is a numpy array of length equal to the number of elements which are simultaneously valid both for vector "x" and "y".

   So, assuming that all the operations have a unitary cost for creating a new data structure, we obtain that the space asymptotic complexity is of class $O(n)$.

Algorithms

In this section, we report the code for the algorithms of recommendation describing their functionality and their asymptotic complexity. All of them produce a dictionary of recommended items (divided in 3 categories of appreciation) for a chosen "target user" as final output. For the asymptotic complexity analysis we'll consider:

1. p: Number of items.
2. n: Number of users.
3. k: Parameter for the number of neighbors in the Memory-based algorithms.
4. f: Parameter for the number of factors in the Model-based algorithm.
5. s: Parameter for the maximum number of iterations in the Model-based algorithm.

In the analysis of the time asymptotic complexity, firstly we'll determine the Running Time of the algorithm using two different values for each line of code (the elementary operations are assumed to be performed in an unitary time):

1. *Cost:* Which contains the number of elementary operations required by the considered line. In some cases, we make use of built-in or self-written functions in order to make the code more modular and readable. So the "Costs" column for these rows corresponds to the time asymptotic complexity estimated for that functions.
2. *Times:* Which contains the number of times that the line has to be repeated during the execution.

Since the algorithms usually depends on more parameters (model parameters and dimensionality's parameters), firstly we'll find the running time as a function of n and p, and then we'll see the classes of asymptotic complexities obtained by fixing alternatively one of the dimensions (so, we'll study the asymptotic complexities of the functions $T(n, \bar{p})$ and $T(\bar{n}, p)$, were the dimension with the upper bar sign is the fixed one). This evaluation is useful to see how the performances change if we need to vary just one of the dimensions.

As regards the other parameters, they don't regard the input's dimensions but anyway they produce a remarkable influence on the running times of the algorithms, so we'll see the rates of growth of the algorithms fixing all the parameters except for one of them. This will help to understand how the time/pace requirements grow if we need to change one of the parameter in order to improve the model performance.

Once we've found the asymptotic complexity's functions $T(n, p)$, we'll set $p = \alpha n$ (according to our data, we've $\alpha \simeq 15$) so we'll obtain a function that depends only on one dimension, so we can have a clearer idea of the algorithm's efficiency depending on the data dimension.

For each different Recommender System, we'll evaluate the asymptotic complexities of its main functions, considering only their upper-bounds (so, we'll focus on the big-O notation). The running times of these functions are usually composed expressions ($T(n) = f_1(n) + f_2(n) + ...$) and we'll consider only the leading terms thanks to the "sum" property (which states that if $f_1(n) = O(g_1(n))$ and $f_2 = O(g_2(n))$, then $(f_1 + f_2)(n) = O(max(g_1(n), g_2(n)))$).

At the end of each section, we'll summarize the results obtained both for the space and time asymptotic complexity.

User-based Collaborative Filtering

Description

The algorithm is made of 4 main steps performed by different functions (the "select()" function actually is contained in "nearest()" but we can consider it as a different preparatory step):

```
def recommend(data, target, k = 10, threshold = 0.33):
    selected = select(data, target, threshold)
    neighbors = nearest(data, target, threshold = threshold)
    ratings = compute_ratings(data, neighbors, target, k = k)
    suggestions = suggest(target, ratings)
    return suggestions
```

We comment each step of the algorithm:

- **select**:
  The aim of the function is to select a subset of users in the Interaction Matrix on which compute recommendations (it returns also the list of items rated by the user since it's an information we'll need in the following steps to avoid recompute it). Indeed, both to reduce the number of iterations and to focus only on the "real" nearest neighbors, we select the rows that have at least an acceptable percentage of interactions in common with the target user. The percentage of interactions required is set through the input parameter "threshold" and it represents a portion of the items rated by the target users (the ones suitable to compute distances). The code is the following:

  ```
  def select(data, target, threshold):
      valid = data.iloc[target][np.isnan(data.iloc[target]) == False].index
      r = len(valid)

      unmatches = data[valid].drop(target).isnull().sum(axis = 1)

      acceptable = unmatches[unmatches <= round((1 - threshold)*r, 0)].index

      if len(acceptable) == 0:
          return data.drop(target).index

      return (acceptable, valid)
  ```

  First of all, the function consider the items rated by the target user. Then it computes the interactions with these items for each row in the Interaction Matrix. Finally, it extracts the users that has an acceptable number of rated items to compute the cosine similarity.

- **nearest:**
  This function computes distances between the selected users and the target one, inserting the measures into a MaxHeap to rapidly find the nearest ones. The variable "indexes" stores the result of the "select" (the same for the variable "valid" that we need to transmit to the other functions) function discussed above:

  ```
  def nearest(data, target, threshold, valid):
      distances = MaxHeap()
      for user in indexes:
          distances.insert(cosine(data.iloc[target].values, data.iloc[user].values), user)
      return (distances, valid)
  ```

  This code creates a MaxHeap object and progressively fills it inserting cosine similarities between the target and all the users considered as acceptable (according to the definition of acceptable provided by the "select" step). In that way, we've that the k-nearest neighbors of the target user will be the elements related to the k highest priorities in the MaxHeap (so they'll be the first k elements to be removed).

- **compute ratings:**
  This function estimates which can be the interaction that the target user can have with its unrated items basing on the information provided by its k-nearest neighbors:

```
def compute_ratings(data, neighbors, valid, target, k = 10):
    predictions = list()
    unseen = data.columns.drop(valid)

    for item in unseen:
        rating = False
        n_comparison = 0
        new = neighbors.copy()
        for iteration in range(k):
            neighbor = new.delete()[1]
            if item in data.iloc[neighbor][np.isnan(data.iloc[neighbor]) == False].index:
                rating += data.iloc[neighbor][item]
                n_comparison += 1

        if rating:
            rating = rating/n_comparison
            predictions.append((rating, item))

    return predictions
```

  The predicted rating are stored in a list created at the beginning of the code and then we extract the items not rated by the target user. Then we loop over the "unseen" items and for each of them we estimate the predicted rating for the target user by extracting its k-nearest neighbors from the MaxHeap generated at the previous step (so we'll need another nested loop) and computing the mean value of the ratings assigned from the neighbors to the unseen items (checking if they've rated them or not). Finally, we add the ratings to the list containing the predictions (if we've obtained a prediction for that item).

- **suggest:**
  Up to now, the algorithm has finished to compute ratings for the unseen items basing on the interactions with the nearest neighbors and this last function is necessary just to obtain a clearer representation of the recommendations. We decide to divide the items in three categories of suggestions: "Strongly Recommended" if the predicted rating is greater or equal than "4", "Recommended" if the estimated rating is between "3" (included) and "4" (excluded), "Not Recommended" otherwise. So this function produces the final output of the Recommender System which is a dictionary of suggestions (the keys are the 3 different levels described above).

```
def suggest(target, ratings):
    recommandations = {
        "Strongly Recommended" : list(),
        "Recommended" : list(),
        "Not Recommended" : list()
    }

    while len(ratings) > 0:
        rating, item = ratings.pop()

        if rating >= 4:
            recommandations["Strongly Recommended"].append(item)
        elif rating >= 3:
            recommandations["Recommended"].append(item)
        else:
            recommandations["Not Recommended"].append(item)

    return recommandations
```

The function simply creates the dictionary and then populates it with a for loop that scan all the target user's unseen items for which we've been able to determine an estimate of the rating.

Asymptotic Complexity

We analyze both the Running Time and the space occupied by each step of the algorithm and finally we derive the overall asymptotic complexity:

- **select:**

Code	Costs	Times
def select(data, target):		
seen = (np.isnan(data.iloc[target]) == False)	$2+2p$	1
valid = data.iloc[target][seen].index	$r+3$	1
r = len(valid)	2	1
users = data[valid].drop(target)	$r+2$	1
booleans = users.isnull()	$(n-1)\cdot r+1$	1
unmatches = booleans.sum(axis = 1)	$(n-1)\cdot r+1$	1
acceptable = unmatches[unmatches $\leq$ round((1 - threshold)*r, 0)].index	$n+3$	1
if len(acceptable) == 0: return data.drop(target).index	4	1
return (acceptable, valid)	1	1

Where $r \in [0,p]$ is the cardinality of the subset of items with an interaction for the target user (in the average case we assume $r=\frac{p}{2}$, hypothesis also confirmed by our testing data where we've an average number of rated movies equal to 14 out of 21). The running time of the algorithm is:

$$T(n,p) = 2p+2+r+3+2+r+2+(n-1)\cdot r+1+(n-1)\cdot r+1+n+3+4+1 =$$

$$= 2p+2+\frac{p}{2}+3+2+\frac{p}{2}+2+(n-1)\cdot\frac{p}{2}+1+(n-1)\cdot\frac{p}{2}+1+n+3+4+1 = 19+2p+pn+n$$

Which means it has asymptotic complexities of class:

1. $T(n,\bar{p}) = O(n)$.
2. $T(\bar{n},p) = O(p)$.

Because the selection procedure takes care both of the users and of the film so logically the function has to loop over rows and columns of the Interaction Matrix.

Setting $p=\alpha n$, we obtain:

$$T(n) = 19+2\alpha n+2\alpha\cdot n^2+n = n^2\cdot 2\alpha+n\cdot(2\alpha+1)+19$$

So we can conclude that the "select" function runs in $O(n^2)$.

As regards space complexity, we name the cardinality of the subset of acceptable users as $j \in [0,n-1]$, in the average case we suppose $j=\frac{2}{3}n$, on the real data we found that on average the acceptable users are the 70%. The data structures created by the function are:

1. "seen": Numpy array of length $r=\frac{p}{2}$.
2. "valid": Numpy array of length $r=\frac{p}{2}$.
3. "users": Pandas Dataframe of size $(n-1)\cdot\frac{p}{2}$.
4. "boolenas": Pandas Dataframe of size $(n-1)\cdot\frac{p}{2}$.
5. "unmatches": Pandas Dataframe of size $n-1$
6. "acceptable": Index list of length $j=\frac{2}{3}n$.

So, assuming a unitary cost for each new memory space occupied we obtain:

$$\frac{p}{2}+\frac{p}{2}+(n-1)\cdot\frac{p}{2}+(n-1)\cdot\frac{p}{2}+n-1+\frac{2}{3}n = n\cdot p+\frac{5}{3}n-1 = n\cdot\left(p+\frac{5}{3}\right)-1$$

Which means that the asymptotic complexities are of class:

1. $O(n)$, fixing p.
2. $O(p)$, fixing n.

Setting $p = \alpha n$, we obtain:

$$n \cdot \left(\alpha n + \frac{5}{3}\right) - 1 = n^2 \cdot \alpha + \frac{5}{3}n - 1$$

So, also the rate of growth of the space required is of class $O(n^2)$.

- **nearest:**

Code	Costs	Times
def nearest(data, target, valid):		
distances = MaxHeap()	1	1
for user in indexes:	1	$j+1$
x = data.iloc[target].values	3	j
y = data.iloc[user].values	3	j
distance = cosine(x, y)	$p+1$	j
distances.insert(distance)	$\log_2(i)$	j
return (distances, valid)	2	1

We "simplify" the complexity of the algorithm by assuming that to turn the rows of the pandas dataframe into numpy arrays (the operations performed with the method "values" at lines 3^{rd} and 4^{th}) have complexity $O(1)$ even if actually their complexity is $O(p)$. We make this huge simplification just because we need to change the object type due to the numpy implementation of our cosine similarity function but we could've also avoided this step.
As regards the operation of insertion into the Priority Queue, we've that it has a cost which depends logarithmically on the size of the MaxHeap, which is populated progressively adding an element at each iteration of the "for" loop. So the total cost of this part of the code is:

$$\sum_{i=0}^{j-1} \log_2(i) = 2 + \sum_{i=2}^{j-1} \log_2(i) = 2 + \log_2\left(\prod_{i=2}^{j-1} i\right) = 2 + \log_2((j-1)!)$$

Assuming that for the two first insertions, the running time is equal to "1". The running time of the algorithm is:

$$T(n,p) = 1 + j + 1 + 3j + 3j + j \cdot (p+1) + (2 + \log_2((j-1)!)) + 2 =$$
$$= 1 + \frac{2}{3}n + 1 + 2n + 2n + \frac{2}{3}np + \frac{2}{3}n + 2 + \log_2\left(\left(\frac{2}{3}n - 1\right)!\right) + 2 =$$
$$= 6 + \frac{16}{3}n + \log_2\left(\left(\frac{2}{3}n - 1\right)!\right) + \frac{2}{3}np$$

Which means it has asymptotic complexities of class:

1. $T(\bar{n}, p) = O(p)$.
2. If we fix the parameter p, we've that the leading term in $T(n, \bar{p})$ is $\log_2\left(\left(\frac{2}{3}n - 1\right)!\right)$, and we can prove that it's upper bounded by the function $g(n) = \log_2(n!)$, so we look for a constant $c > 0 \mid c \cdot g(n) \geq T(n, \bar{p}), \forall n \in \mathbb{N}$:

$$c \cdot \log_2(n!) \geq \log_2\left(\left(\frac{2}{3}n - 1\right)!\right)$$
$$c \geq \frac{\log_2\left(\left(\frac{2}{3}n - 1\right)!\right)}{\log_2(n!)}, \forall n \in \mathbb{N}$$

   Since both the $\log_2(\cdot)$ and the factorial are monotonically increasing functions, and we've that $n > \frac{2}{3}n - 1, \forall n \in \mathbb{N}$, it follows that the ratio on the right side of the inequality is always lower than 1, so we can take $c = 1$. We've obtained that $T(n, \bar{p}) = O(\log_2(n!))$.

Indeed, for each of the selected users we need to compute the cosine similarity and make an insertion in the Priority Queue.

Setting $p = \alpha n$, we obtain:

$$T(n) = 6 + \frac{16}{3}n + \log_2\left(\left(\frac{2}{3}n - 1\right)!\right) + \frac{2\alpha}{3}n^2$$

Now we prove that $T(n) = O(n^2)$ by showing that the function $g(n) = n^2$ is an upper bound for $f(n) = \log_2\left(\left(\frac{2}{3}n - 1\right)!\right)$:
From the monotony of the $\log_2(\cdot)$ and the factorial function, we obtain

$$\log_2(n!) \geq \log_2\left(\left(\frac{2}{3}n - 1\right)!\right)$$

Then, since $n! = n \cdot (n-1) \cdot (n-2) \cdot ... \cdot 1$, we've that $n^n \geq n!$, so

$$n \cdot \log_2(n) \geq \log_2(n!)$$

Finally we show that exists a constant $c > 0$ such that:

$$c \cdot n^2 \geq n \cdot \log_2(n), \forall n \in \mathbb{N}$$

$$c \geq \frac{\log_2(n)}{n}, \forall n \in \mathbb{N}$$

And setting $c = 1$ we obtain that $T(n) = O(n^2)$.

As regards space complexity, the data structures created by the function are:

1. "distances": MaxHeap of size $\frac{2}{3}n$ (but since each element on our MaxHeap is indeed a couple of elements the real size is $\frac{4}{3}n$).
2. "x" and "y": Two numpy arrays of length p (the are extracted from the Interaction Matrix for each of the j iterations, but since we always overwrite the same variables we don't need to occupy more space).
3. "cosine": We saw that this function has a space complexity of class $O(p)$ (we invoke it for j-times, but since at the end of each execution the space allocated for the variables generated by the function is removed, we have that space complexity remains of class $O(p)$).

So, assuming a unitary cost for each new memory space occupied we obtain:

$$\frac{4}{3}n + 3p$$

Which means that the asymptotic complexity is of class:

1. $O(n)$, fixing p.
2. $O(p)$, fixing n.

Setting $p = \alpha n$, we obtain:

$$\frac{4}{3}n + 3\alpha n = n \cdot \left(3\alpha + \frac{4}{3}\right)$$

So that the asymptotic space complexity is $O(n)$.

- **compute ratings:**

Code	Costs	Times
def computeRatings(data, neighbors, valid, target, k = 10):		
predictions = list()	2	1
unseen = data.columns.drop(valid)	$r+2$	1
nearest = list()	2	1
for iteration in range(k):	1	$k+1$
nearest.append(neighbors.delete()[1])	$\log_2(i)+2$	k
for item in unseen:	1	$p-r+1$
rating = False	1	$p-r$
nComparison = 0	1	$p-r$
for iteration in range(k):	1	$(p-r)\cdot(k+1)$
neighbor = nearest[iteration]	2	$(p-r)\cdot k$
rated = (np.isnan(data.iloc[neighbor]) == False)	$2p+2$	$(p-r)\cdot k$
items = data.iloc[neighbor][rated].index	$t+3$	$(p-r)\cdot k$
if item in items: rating += data.iloc[neighbor][item] nComparison += 1	$t+3$	$(p-r)\cdot k$
if rating: rating = rating/nComparison predictions.append((rating, item))	5	$(p-r)$
return predictions	1	1

Where $t \in [0,p]$ is the number of items rated by the neighbor of our target user and in the average case we assume it is equal to $\frac{p}{2}$ (for the same considerations we made for the parameter r).
As regards the running time for the determination of the k-nearest neighbors (which requires "k" remove operations from the Priority Queue that progressively reduces its size), since the value of "k" is negligible (cause usually $n >> k$), we can consider the size of the Priority Queue as constant and so we've:

$$\log_2(n) + ... + \log_2(n) = \sum_{i=1}^{k} \log_2(n) = k \cdot \log_2(n)$$

The running time of the algorithm is:

$$T(n,p) = 2+r+2+2+k+1+k\cdot\log_2(n)+2k+p-r+1+p-r+p-r+(p-r)\cdot(k+1)+2k\cdot(p-r)+2k\cdot(p+1)\cdot(p-r)+$$
$$+k\cdot(t+3)\cdot(p-r)+k\cdot(p-r)\cdot(t+3)+5p-5r+1 =$$
$$= 9+\frac{p}{2}+k+k\cdot\log_2(n)+2k+\frac{p}{2}+p+\frac{k}{2}\cdot p+\frac{p}{2}+kp+kp^2+kp+3kp+k\frac{p^2}{2}+\frac{5}{2}p =$$
$$= \frac{3}{2}kp^2+\left(5+\frac{11}{2}k\right)p+k\cdot\log_2(n)+3k$$

Which means it has asymptotic complexity of class:

1. $T(\bar{n},p) = O(p^2)$.
2. $T(n,\bar{p}) = O(\log_2(n))$.

As regards the parameter k, if we fix p and n, it follows that the running time is of class $O(k)$.

Indeed, the number of neighbors linearly affects the first "for" loop, the extraction of the k-nearest users (computationally represented by the logarithmic term) and the determination of the predicted rating for the items unseen by the target user. This last depends both on the parameter "k" and on "p" since we've to loop through the unseen items and make logical comparisons for all the other user's rows (so we have "p" appears with degree 2 in the computation of the ratings). Finally, the last "p" term corresponds to the insertion of the final estimated rating in the list "predictions" for each unseen item.

Setting $p = \alpha n$, we obtain:

$$T(n) = \frac{3}{2}k\alpha^2\cdot n^2+\left(5\alpha+\frac{11\alpha}{2}k\right)n+k\cdot\log_2(\alpha n)+3k$$

So, also for this function we can conclude that the overall class of complexity is $O(n^2)$.

As regards space complexity, the data structures created by the function are:

1. "predictions": list of tuples length $p - r = \frac{p}{2}$, so the real number of elements stored in it is p.
2. "unseen": list of items of length $r = \frac{p}{2}$.
3. "nearest": list of user's codes of length k.
4. "neighbor": Row of the Interaction Matrix of length p.
5. "rated": numpy array of length $t = \frac{p}{2}$.
6. "items": list of items of length $t = \frac{p}{2}$.

So, assuming a unitary cost for each new memory space occupied we obtain:

$$k + \frac{7}{2}p$$

Which means that the asymptotic complexity is of class:

1. $O(p)$, fixing k.
2. $O(k)$, fixing p.

Setting $p = \alpha n$, we obtain:

$$k + \frac{7}{2}p = k + \frac{7\alpha}{2}n$$

Which leads to a rate of growth that is $O(n)$.

- **suggest:**

Code	Costs	Times
def suggest(target, ratings):		
recommandations = dict("Strongly Recommended" = list(), "Recommended" = list(), "Not Recommended" = list())	4	1
while len(ratings) > 0:	2	$p - r + 1$
rating, item = ratings.pop()	2	$p - r$
if rating $\geq$ 4: recommandations["Strongly Recommended"].append(item)	3	$p - r$
elif rating $\geq$ 3: recommandations["Recommended"].append(item)	3	$p - r$
else: recommandations["Not Recommended"].append(item)	3	$p - r$
return recommandations	1	1

The running time of the algorithm is:

$$T(p) = 4 + 2p - 2r + 2 + 2p - 2r + 3p - 3r + 3p - 3r + 3p - 3r + 1 =$$

$$= 4 + 2p - p + 2 + 2p - p + 3p - \frac{3}{2}p + 3p - \frac{3}{2}p + 3p - \frac{3}{2}p + 1 = \frac{13}{2}p + 7$$

Which means it has asymptotic complexity of class $O(p)$.

As regards space complexity, the function simply creates a dictionary to assign to their correct class of suggestion the $(p - r) = \frac{p}{2}$ items not already rated by the target user, so the space complexity simply depends on the number of items in the Interaction Matrix, which means it's $O(p)$.

So, by setting $p = \alpha n$ we obtain the same classes of complexity both for the space and time analysis.

Conclusions

1. *Time:*
   Since 3 out of 4 functions analyzed have a quadratic rate of growth (except for the last one, which runs in $O(n)$), we can conclude that the overall class of complexity of the algorithm is $T(n) = O(n^2)$ (by the sum property).

   Looking at the different steps and considering separately the effects of the two dimensions (n and p) and the model parameter, we see that the initial selection procedure depends both on the number of users and the number of items. In particular, fixing one of the parameters between p and n and considering the other as variable, the running time of the function grows linearly.
   The function for computing distances and making insertions in the Priority Queue has an asymptotic complexity which is linear in p (if we fix n). Instead, if we consider n to be the variable dimension, the class of complexity get worse, since $T(n, \bar{p}) = O(\log_2(n!))$.
   Differently from the two previous functions, the estimation step ("compute ratings") depends also on the value chosen for the parameter k which affects both the cost of the determination of the nearest neighbors (that requires k-removals from the Priority Queue) and the cost of the computation of the ratings (indeed, we need to check which are the rated elements of each neighbor and then we need to check if they contains the item for which we want to find an estimate). To sum up, fixing n and p, the running time of the function grows linearly in k, fixing k and p the running time rate of growth is $O(\log_2(n))$ and fixing n and k it grows quadratic in p.
   The last function's asymptotic time complexity is simply linear in p.
   We can conclude that the entire algorithm running time:

   - Considering only the row dimension n, the highest rate of growth we find for the algorithm is the one of the function that computes distances ("nearest"), and the asymptotic complexity is of class $O(\log_2(n!))$.
   - Considering only the column dimension p, here the highest class of complexity we've found is the one related to the function that computes the predicted ratings and it's $O(p^2)$.
   - Looking at the influence of the model parameter k, the running time of the algorithm grows linearly as k grows (due to the third function "compute ratings").

2. *Space:*
   Instead, for the space complexity we can briefly say that the memory required by the algorithm is of class $O(n^2)$ due to the results pointed out for the first step of the algorithm ("selection" procedure).
   In particular, if we want to see the effect produced by the variation of only one parameter per time, the rate of growth is linear with respect to both the dimensions p and n, and also with respect to the model parameter k.

Item-based Collaborative Filtering

Description

The algorithm is made of 3 steps performed by three different functions:

```
def recommend(data, target, k = 3):
    unseen = data.iloc[target][np.isnan(data.iloc[target])].index
    favourites = search_favourite(data, target)
    neighbors = nearest(data, target, favourites, unseen)
    return suggest(target, nearest, unseen, k = k)
```

Where "unseen" represents also in that case the items that haven't been rated by the target user (we recall that this operation runs in $O(p)$ since we need to check a condition on all the entries of the p-dimensional vector related to the interactions of the target user). We comment each step of the algorithm:

- **search favourite:**
  The function extracts the items with the highest rating for the target user, inserting them into a Circular Queue.

  ```
  def search_favourite(data, target, unseen):
      valid_i = data.iloc[target].drop(unseen).index
      valid_r = data.iloc[target][valid_i]

      favourites = Queue_c(len(valid_i) + 1)

      favourites.enqueue(valid_r[0], valid_i[0])
      best = valid_r[0]

      for (rating, item) in zip(valid_r, valid_i):

          if rating >= best:
              favourites.enqueue(rating, item)
              best = rating

      while favourites.first_in()[0] < best:
          favourites.dequeue()

      return favourites
  ```

  Firstly, the function finds the item rated by the target user and the ratings assigned to them. Then, it creates a Circular Queue to store the items related to the highest ratings through a "for" loop which determines the so called "favourite items".

- **nearest:**
  This function creates a Priority Queue containing all the items unrated by the target user ordered from the "most similar" to the "less similar".

  ```
  def nearest(data, target, favourites, unseen):
      nearest = {}

      while not favourites.is_empty():
          favourite = favourites.dequeue()[1]

          neighbors = MaxHeap()
          for item in unseen:
              neighbors.insert(cosine(data[item].values, data[favourite].values), item)

          nearest[favourite] = neighbors

      return nearest
  ```

The function creates a dictionary where the keys are represented by all the items that have the highest rating assigned by the target user (they can be more than 1) and the elements associated to the keys are Priority Queues to store the items which are more similar to the favourite ones.
For each favourite item (key of the dictionary), the function progressively populates the Priority Queue by looping on the items non rated by the target user, to compute their cosine similarities.

- **suggest:**
The function performs the same task seen for the user-based Recommender System, i. e. it produces a dictionary with a classification of the unseen items in three categories of suggestion. Differently from before, since in that case we don't estimate any rating, the classification is done following this idea: If an item is one of the k-nearest neighbors for more than one of the favourite items of the target user, it'll be "Strongly Recommended", if it appears as nearest neighbor just one time it'll be "recommended" and if it doesn't appear at all as nearest neighbor it'll be classified as "Not Recommended".

```
def suggest(target, nearest, unseen, k = 3):
    suggestions = {}

    for favourite in nearest.keys():
        iteration = k
        neighbors = nearest[favourite]

        while (iteration > 0) and (neighbors.size > 0):
            suggested = neighbors.delete()[1]

            if suggested not in suggestions.keys():
                suggestions[suggested] = "Recommended"
                unseen = unseen.drop(suggested)
            else:
                suggestions[suggested] = "Strongly Recommended"
            iteration -= 1

    for item in unseen:
        suggestions[item] = "Not Recommended"

    return suggestions
```

The function creates a dictionary that will contain all the unseen items as keys and the assigned class as related element.
Then the algorithm progressively populates the dictionary looping for "k" times through the Priority Queues of each target user's favourite item, assigning each of the unseen items to its proper class basing on the rule defined above.

Asymptotic Complexity

We analyze the Running Time of each step of the algorithm and finally we derive the overall asymptotic complexity:

- **search favourite:**

Code	Costs	Times
def searchFavourite(data, target, unseen):		
validI = data.iloc[target].drop(unseen).index	$r+2$	1
validR = data.iloc[target][validI]	$p-r+2$	1
favourites = QueueC(len(validI) + 1)	$p-r+1$	1
favourites.enqueue(validR[0], validI[0])	3	1
best = validR[0]	2	1
for (rating, item) in zip(validR, validI):	2	$p-r+1$
if rating $\geq$ best: favourites.enqueue(rating, item) best = rating	3	$p-r$
while favourites.firstIn()[0] $<$ best:	3	s+1
favourites.dequeue()	1	s
return favourites	1	1

Where s is the number of swaps from the Circular Queue we need to do in order to have all the films with the highest rating in it (according to our data, we found the value of s to be around the 10% of the number of item p, so we assume $s = \frac{1}{10}p$), while r is the same parameter defined for the previous algorithm of recommendations.
The running time of the algorithm is:

$$T(p) = r + 2p - r + 2 + p - r + 1 + 3 + 2 + 2p - 2r + 2 + 3p - 3r + 3s + 3 + s + 1 =$$

$$= 14 + 8p - 3p + \frac{2}{5}p = \frac{27}{5}p + 14$$

Which means it has asymptotic complexity of class $O(p)$.

As regards space complexity, the data structures created by the function are:

1. "validI": list of indexes of length $r = \frac{p}{2}$.
2. "validR": list of indexes of length $(p - r) = \frac{p}{2}$.
3. "favourites": Circular Queue of length $(p - r + 1) = \frac{p}{2} + 1$ (but since each element in the Queue is composed by a pair, the real length is $p + 2$).

So, assuming a unitary cost for each new memory space occupied we obtain:

$$\frac{p}{2} + \frac{p}{2} + p + 2 = 3p + 2$$

Which means that the asymptotic complexity is of class $O(p)$.

Also here, setting $p = \alpha n$, the classes of complexities remains the same as before for both of them.

- **nearest:**

Code	Costs	Times
def nearest(data, target, favourites, unseen):		
nearest = dict()	2	1
while not favourites.isEmpty():	2	$u+1$
favourite = favourites.dequeue()[1]	3	u
neighbors = MaxHeap()	1	u
for item in unseen:	1	$(r \cdot u) + 1$
cosine = cosine(data[item].values, data[favourite].values)	$n+3$	$r \cdot u$
neighbors.insert(cosine, item)	$\log_2(i)$	$r \cdot u$
nearest[favourite] = neighbors	2	u
return nearest	1	1

Where u is the number of "favourite" items (i. e., movies with the highest ratings) for the target user (according to our data, we assume that $u = 0.15p$).
As before, we assume that the conversion from pandas to numpy arrays has an $O(1)$ complexity, for the same reasons pointed out for the nearest function in the user-based Recommender System.
As regards the total running time of the insertions into the Priority Queue (which has a size that grows progressively), we've that the overall cost is (assuming that the first two insertions cost "1"):

$$u\cdot(3+\log_2(3)+...+\log_2(r-1)) = u\cdot\left(3+\sum_{j=3}^{r-1}\log_2(j)\right) = u\cdot\left(3+\log_2\left(\prod_{j=3}^{r-1}j\right)\right) = 3u+u\cdot\log_2\left(\frac{(r-1)!}{2}\right) =$$

$$= 2u + u\cdot\log_2((r-1)!)$$

The running time of the algorithm is:

$$T(n,p) = 2+2u+2+3u+u+r+u+1+nru+3ru+2u+u\cdot\log_2((r-1)!)+2u+1 =$$

$$= 4+1.65p+\frac{p}{2}+0.075np^2+0.225p^2+0.15p\cdot\log_2\left(\left(\frac{p}{2}-1\right)!\right)$$

Which means it has asymptotic complexity of class:

1. $T(n,\bar{p}) = O(n)$.
2. If we fix the parameter n, we've that the leading term of the running time function is $p\cdot\log_2\left(\left(\frac{p}{2}-1\right)!\right)$, indeed we can prove that:

$$p\cdot\log_2\left(\left(\frac{p}{2}-1\right)!\right) \geq p^2$$

$$\log_2\left(\left(\frac{p}{2}-1\right)!\right) \geq p$$

$$\log_2\left(\left(\frac{p}{2}-1\right)!\right) \geq \log_2(2^p)$$

$$\left(\frac{p}{2}-1\right)! \geq 2^p$$

   And since we know that the factorial grows faster than 2^p, it follows that $\exists\, n_0 \in \mathbb{N} \mid p\cdot\log_2\left(\left(\frac{p}{2}-1\right)!\right) \geq p^2\ \forall p \geq n_0$. Once we've proved which is the leading the leading term, we can show its class of complexity:

$$c\cdot p\cdot\log_2(p!) \geq p\cdot\log_2\left(\left(\frac{p}{2}-1\right)!\right)$$

$$c \geq \frac{\log_2\left(\left(\frac{p}{2}-1\right)!\right)}{\log_2(p!)}, \forall p \in \mathbb{N}$$

   Again, since both the $\log_2()$ and the factorial are monotonically increasing functions, and we've that $p > \frac{p}{2}-1, \forall p \in \mathbb{N}$, it follows that the ratio on the right side of the inequality is always lower than 1, so we can take $c = 1$. We've obtained that $T(\bar{n},p) = O(p\cdot\log_2(p!))$.

The linear term in p comes out due to the while loop that proceeds until the Circular Queue is empty and the insertion in the dictionary, while the logarithmic one is related to the insertions into the Priority Queues. Finally, the quadratic terms are referred respectively to the "for" loop and the computation of the cosine similarities (which are functions applied on n-dimensional vectors).

Setting $p = \alpha n$, we obtain:

$$T(n) = 4+1.65\alpha\cdot n+\frac{\alpha}{2}n+0.075\alpha^2\cdot n^3+0.225\alpha^2\cdot n^2+0.15\alpha\cdot n\log_2\left(\left(\frac{\alpha}{2}n-1\right)!\right) =$$

$$= 0.075\alpha^2\cdot n^3+0.15\alpha\cdot n\log_2\left(\left(\frac{\alpha}{2}n-1\right)!\right)+0.225\alpha^2\cdot n^2+n\cdot 2.15\alpha+4$$

Now we show that the class of complexity is $T(n) = O(n^3)$ since $g(n) = n^3$ is an upper bound for $f(n) = n\log_2(n!)$ using the fact that $n\log_2(n) \geq \log_2(n!), \forall n \in \mathbb{N}$:

$$c\cdot n^3 \geq n^2\log_2(n) \geq n\log_2(n!), \forall n \in \mathbb{N}$$

$$c \geq \frac{\log_2(n)}{n} \geq \frac{\log_2(n!)}{n^2}$$

And we can obtain the result for $c = 1$.

As regards space complexity, the data structures created by the function are:

1. "nearest": dictionary containing a MaxHeap for each target user's favourite item, so the total space occupied is $u \cdot 2 \cdot \frac{p}{2} = u \cdot p = 0.15p^2$.
2. "neighbors": MaxHeap generated for each target user's favourite item (but since at each iteration we overwrite the variable, we just need to consider the space required for one Priority Queue), so occupies $2 \cdot r = 2 \cdot \frac{p}{2} = p$.
3. The invocation of the "cosine" function requires has $O(n)$ space complexity.

So, assuming a unitary cost for each new memory space occupied we obtain:

$$0.15p^2 + p + n = 0.15p^2 + p + n$$

Which means that the asymptotic complexity is of class:

1. $O(n)$, fixing p.
2. $O(p^2)$, fixing n.

Setting $p = \alpha n$, we obtain:

$$0.15p^2 + p + n = 0.15\alpha^2 n^2 + \alpha n + n = n^2 \cdot 0.15\alpha^2 + n \cdot (\alpha + 1)$$

And we conclude that the class of complexity is $O(n^2)$.

- **suggest:**

Code	Costs	Times
def suggest(target, nearest, unseen, k = 3):		
suggestions = dict()	2	1
for favourite in nearest.keys():	2	$u + 1$
iteration = k	1	u
neighbors = nearest[favourite]	2	u
while (iteration > 0) and (neighbors.size > 0):	3	$u \cdot (k + 1)$
suggested = neighbors.delete()[1]	$\log_2(r) + 2$	$u \cdot k$
if suggested not in suggestions.keys(): suggestions[suggested] = "Recommended" unseen = unseen.drop(suggested)	$r + 5$	$u \cdot k$
else: suggestions[suggested] = "Strongly Recommended"	2	$u \cdot k$
iteration -= 1	1	$u \cdot k$
for item in unseen:	1	$r + 1$
suggestions[item] = "Not Recommended"	2	r
return suggestions	1	1

As regards the total cost of the deletion from the Priority Queues, we assume that the size of the MaxHeap is constant since as the dimensionality grows we've that $p >> k$. To check if the item "suggested" is contained in the dictionary's keys, we need to scan (on the worst case) all the keys that are progressively inserted into the dictionary, so the length of the array to scan progressively increases. The length reached by the keys depends on how many items are found one or more time as neighbors of the favourite items and obviously this length depends on "r" and on "k". We assume that everytime the algorithm has to scan all the "r" unseen elements. The running time of the algorithm is:

$$T(p) = 2 + 2u + 2 + u + 2u + 3u \cdot (k + 1) + uk\log_2(r) + 2uk + ruk + 5uk + 2uk + uk + r + 1 + 2r + 1 =$$

$$= 6 + 8u + 16uk + uk\log_2(r) + ruk + 3r = 6 + 1.2p + 2.4pk + 0.15pk \cdot \log_2\left(\frac{p}{2}\right) + 0.075p^2k + \frac{3}{2}p =$$

$$= 0.075kp^2 + 2.7p + 0.15pk \cdot \log_2\left(\frac{p}{2}\right) + 6$$

Which means it has asymptotic complexity of class $T(p) = O(p^2)$.
About the influence expressed by the parameter k, for a fixed dimension p, the running time grows in $O(k)$ as k grows.

As regards space complexity, the data structures created by the function are:

1. "suggestions": dictionary on which we've all the target user's unseen items with their related suggestion, so the dimension is $r = \frac{p}{2}$.
2. "neighbors": At each iteration, this variable is overwritten and contains a MaxHeap of length p.
3. "unseen": list of items that has an initial size of $r = \frac{p}{2}$.

So, assuming a unitary cost for each new memory space occupied we obtain:

$$\frac{p}{2} + \frac{p}{2} + p = 2p$$

Which means that the asymptotic complexity is of class $O(p)$.

So, by setting $p = \alpha n$ we obtain the same classes of complexity for both the analyses.

Conclusions

1. *Time:*
The 3 functions analyzed have shown 3 different behaviors for their time asymptotic complexity ($O(n)$, $O(n^2)$ and $O(n^3)$), but again applying the sum property, we conclude that the general asymptotic time complexity is $T(n) = O(n^3)$.

   As regards the evaluation of the impact produced by the single parameters on the algorithm's steps, we saw that the research of the target user's favourite items is an algorithm that simply grows linearly in p.
   The determination of the k-nearest neighbors to the favourite items depends mainly on the parameter p, the leading term of the running time function is related to the insertions into the Priority Queue, and the class of complexity of this operation is $O(p \cdot \log_2(p!))$ (since, for each insertion, we need to compute the cosine similarity). Instead, the rate of growth is linear if we fix p and consider only n as variable.
   Finally, the "suggest" function depends both on the values chosen for the parameter k and the number of items considered. The class of complexity related to p is $O(p^2)$, while the variation of k makes the running time grow linearly.
   We can conclude that the entire algorithm running time:

   - Considering only the row dimension n, the rate of growth is linear.
   - Considering only the row dimension p, the highest class of complexity obtained is $O(p \cdot \log_2(p!))$.
   - As before, the influence expressed by the model parameter k on the running time is linear ($O(k)$).

2. *Space:*
   The analysis performed leads to an asymptotic space complexity of class $O(n^2)$.
   To point out the single parameter's effects, we can briefly say that:

   - The space required is not affected by k.
   - Fixing p, the space required grows linearly in n.
   - Fixing n, the space required grows quadratic in p.

Matrix Factorization Collaborative Filtering

Description

The algorithm is made of two steps performed by two different functions:

```
def recommend(data, target, n_factors, steps = 200, alpha = 0.0002, beta = 0.02, to_train = True):
    if to_train:
        U, I, E = factorization(data, n_factors, steps = steps, alpha = alpha, beta = beta)
        temporary = np.dot(U, I)
        factorized = pd.DataFrame(temporary, columns = data.columns)

    return suggest(data, factorized, target)
```

Where in the intermediate steps, we compute the final approximation of the Interaction Matrix and we turn it into a pandas dataframe (operation that we could have avoided and that has to be done just one time, so we assume it runs in $O(1)$) run in $O(n \cdot p)$ We comment each step of the algorithm:

- **factorization:**
  The function performs a standard Non-Negative factorization of the Interaction Matrix using the method of the Gradient descend with a regularization term.

  ```
  def factorization(data, n_factors, steps = 200, alpha = 0.0002, beta = 0.02):
      n = data.shape[0]
      m = data.shape[1]

      data = np.array(data)

      U = np.random.rand(n, n_factors)
      I = np.random.rand(n_factors, p)

      for iteration in range(steps):

          for user in range(n):
              for item in range(p):

                  if data[user][item] > 0:

                      eij = data[user][item] - np.dot(U[user, :], I[:, item])

                      for factor in range(n_factors):
                          U[user][factor] = U[user][factor] + 2*alpha*(eij * I[factor][item]
                          - beta * U[user][factor])

                          I[factor][item] = I[factor][item] + 2*alpha*(eij * U[user][factor]
                          - beta * I[factor][item])

          E = 0
          for user in range(n):
              for item in range(p):
                  if data[user][item] > 0:
                      E = E + (data[user][item] - np.dot(U[user, :], I[:, item]))**2

                      for factor in range(n_factors):
                          E = E + (beta/2) * ((U[user][factor])**2 + (I[factor][item])**2)

          if E < 0.001:
              break

      return (U, I, E)
  ```

Initially, the function converts our data structures into numpy objects (just to use the numpy's function to perform the mathematics behind the factorization). Then, we inizialize randomly the two matrices "U" and "I" that we'll be used to reproduce the original Interaction Matrix (in particular, the numpy random function used, generates random value in the range $[0, 1)$ according to the uniform distribution).
Then, the proper algorithm starts: our aim is to reproduce as better as possible the original Interaction Matrix by the cross-product of the matrices U and I. The best approximation of the Interaction Matrix is the one that minimizes the value:

$$E^2 = \sum_{i=1}^{n} \sum_{j=1}^{p} e_{ij}^2$$

where each term e_{ij}^2 is the squared error in the approximation of the (i, j)-th entry of the Interaction Matrix and is defined as:

$$e_{ij}^2 = (r_{ij} - \hat{r}_{ij})^2 = \left(r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj} \right)^2$$

where K is the number of factor chosen for the factorization and p_{ik}, q_{kj} are respectively the entries of the matrices U and I.
To minimize the error, we have to know in which direction we have to modify the values of p_{ik} and q_{kj}. In other words, we need to know the gradient at the current values, and therefore we differentiate the above equation with respect to these two variables:

$$\frac{\partial e_{ij}^2}{\partial p_{ik}} = -2 e_{ij} q_{kj}$$

$$\frac{\partial e_{ij}^2}{\partial q_{ik}} = -2 e_{ij} p_{ik}$$

So we iteratively update the values of the entries of U and I following the formulas obtained above by the gradient. Every update is made by a step parameter we define as α and we set with a small value:

$$p'_{ik} = p_{ik} + 2\alpha e_{ij} q_{kj}$$

$$q'_{kj} = q_{kj} + 2\alpha e_{ij} p_{ik}$$

So at each step, the algorithm produce a better estimation of the Interaction Matrix cause we update the previous values of the entries of the two matrices. Obviously, the comparison is made only with respect to the "valid" entries of the Interaction Matrix.
We also compute the squared error E_{ij}^2 since the iterative algorithm stops when we reach a sufficiently good approximation (i. e. a small value of E_{ij}^2) or when we reach a maximum number of iteration.
Additionally, we implement a regularize version of the Gradient descend algorithm which modifies the previous formulas as follow:

$$e_{ij}^2 = \left(r_{ij} - \sum_{k=1}^{K} p_{ik} q_{kj} \right)^2 + \frac{\beta}{2} \sum_{k=1}^{K} (||P||^2 + ||Q||^2)$$

$$p'_{ik} = p_{ik} + \alpha(2 e_{ij} q_{kj} - \beta p_{ik})$$

$$q'_{kj} = q_{kj} + \alpha(2 e_{ij} p_{ik} - \beta q_{kj})$$

So, first we compute the values of the approximations for each element in the Interaction Matrix.
Then, through another "for" loop, we compute the updates of the matrices "U" and "I".
Then, the algorithm computes the total squared error in the approximation (E^2) with three nested "for" loops, and finally it checks if the approximation is good enough or if we need more iterations.

- **suggest:**
  The function computes the predicted ratings for the target users through the cross product between the matrices U and I determined by the Non-Negative Factorization done at the previous step. Then it produces the usual dictionary of suggestions:

```
def suggest(data, factorized, target):
    unseen = data.iloc[target][np.isnan(data.iloc[target])].index

    recommandations = {
    "Strongly Recommended" : list(),
    "Recommended" : list(),
    "Not Recommended" : list()
    }

    for item in unseen:

        rating = factorized.iloc[target][item]

        if rating >= 4:
            recommandations["Strongly Recommended"].append(item)
        elif rating >= 3:
            recommandations["Recommended"].append(item)
        else:
            recommandations["Not Recommended"].append(item)

    return recommandations
```

  The function starts by extracting the unrated items of the target user and then creates a dictionary to contain the suggestions divided as usual.
  Then, it loops over the unseen items, for each of them extract the estimated rating from the replication of the Interaction Matrix obtained by the Non-Negative factorization (i. e. the result of $\underset{n\times f}{U} \times \underset{f\times p}{I}$).
  Finally, it assign the unseen items to the correct key of the dictionary according to the usual rule applied on the estimated ratings.

Asymptotic Complexity

We analyze the Running Time of each step of the algorithm and finally we derive the overall asymptotic complexity:

- **factorization:**

Code	Costs	Times
def factorization(data, nFactors, steps = 200, alpha = 0.0002, beta = 0.02):		
n = data.shape[0]	3	1
p = data.shape[1]	3	1
data = np.array(data)	2	1
U = np.random.rand(n, nFactors)	$n \cdot f + 1$	1
I = np.random.rand(nFactors, p)	$f \cdot p + 1$	1
for iteration in range(steps):	1	s+1
for user in range(n):	1	$s \cdot (n+1)$
for item in range(p):	1	$s \cdot n \cdot (p+1)$
if data[user][item] > 0: eij = data[user][item] - np.dot(U[user, :], I[:, item])	$f + 5$	$s \cdot n \cdot p$
for factor in range(nFactors):	1	$s \cdot n \cdot p \cdot (f+1)$
regU = 2*alpha*(eij * I[factor][item] - beta * U[user][factor]) U[user][factor] = U[user][factor] + regU	12	$s \cdot n \cdot p \cdot f$
regI = 2*alpha*(eij * U[user][factor] - beta * I[factor][item]) I[factor][item] = I[factor][item] + regI	12	$s \cdot n \cdot p \cdot f$
E = 0	1	s
for user in range(n):	1	$s \cdot (n+1)$
for item in range(p):	1	$s \cdot n \cdot (p+1)$
if data[user][item] > 0:	2	$s \cdot n \cdot p$
E = E + (data[user][item] - np.dot(U[user, :], I[:, item]))**2	$f + 5$	$s \cdot n \cdot p$
for factor in range(nFactors):	1	$s \cdot n \cdot p \cdot (f+1)$
E = E + (beta/2) * ((U[user][factor])**2 + (I[factor][item])**2)	8	$s \cdot n \cdot p \cdot f$
if E < 0.001: break	1	s
return (U, I, E)	1	1

Where f is the number of factors chosen for the Interaction Matrix's decomposition and s is the maximum number of iteration admitted.
Additionally, we don't take care about the running time of turning a pandas dataframe into a numpy array since this operation is needed only due to the fact that we use numpy functions in the rest of the algorithm but we could have avoided to do so. The running time of the algorithm is:

$$T(n,p) = 3+3+2+nf+1+fp+1+s+1+sn+s+snp+sn+snpf+5snp+snpf+snp+$$

$$+12snpf+12snpf+s+sn+s+snp+sn+2snp+snpf+snp+8snpf+s+1 =$$

$$= 12 + n \cdot (f+4s) + pf + 5s + np \cdot (11s + 35sf)$$

Which means it has asymptotic complexity of class :

1. $T(n, \bar{p}) = O(n)$.
2. $T(\bar{n}, p) = O(p)$.

Looking at the other model parameters, we can easily see that for a fixed dimensionality (n, p), and varying one parameter only, the running time grows in $O(f)$ as f grows and in $O(s)$ as s grows.

Clearly the algorithm's running time depends strongly on the value chosen for the maximum number of iteration chosen during the implementation because it affects each operation performed except for the initial random setting of the matrices "U" and "I" (that requires respectively $n \cdot f$ and $f \cdot p$, i. e. the dimensions of the matrices). The computation of the squared errors e_{ij}^2 requires $n \cdot p \cdot f$ to be performed (for each of the $i \leq s$ iterations) like the determination of the updates (p'_{ik} and q'_{kj}) and of E (due to the "regularized version" adopted).

Setting $p = \alpha n$, we obtain:

$$T(n) = 12 + n \cdot (f + 4s) + f\alpha n + 5s + \alpha n^2 \cdot (11s + 35sf) = n^2 \cdot \alpha(11s + 35sf) + n \cdot (\alpha f + f + 4s) + (5s + 12)$$

And we obtain that $T(n) = O(n^2)$.

As regards space complexity, the data structures created by the function are:

1. "data": numpy array of size $n \times p$ containing the Interaction Matrix.
2. "U": numpy array of size $n \times f$.
3. "I": numpy array of size $f \times p$.

So, assuming a unitary cost for each new memory space occupied we obtain:

$$n \cdot p + n \cdot f + p \cdot f = n \cdot (p + f) + p \cdot f$$

Which means that the asymptotic complexity is of class:

1. $O(n)$, fixing p and f.
2. $O(p)$, fixing n and f.
3. $O(f)$, fixing n and p.

Setting $p = \alpha n$, we obtain:

$$n \cdot (p + f) + p \cdot f = n^2 \cdot \alpha + n \cdot f(\alpha + 1)$$

And again we conclude that the asymptotic complexity is $O(n^2)$.

- **suggest:**

Code	Costs	Times
def suggest(data, factorized,target):		
indexes = np.isnan(data.iloc[target])	$p + 2$	1
unseen = data.iloc[target][indexes].index	$p - r + 2$	1
recommandations = dict("Strongly Recommended" = list(), "Recommended" = list(), "Not Recommended" = list())	4	1
for item in unseen:	1	$(p - r) + 1$
rating = factorized.iloc[target][item]	2	$(p - r)$
if rating $\geq$ 4: recommandations["Strongly Recommended"].append(item)	3	$(p - r)$
elif rating $\geq$ 3: recommandations["Recommended"].append(item)	3	$(p - r)$
else: recommandations["Not Recommended"].append(item)	2	$(p - r)$
return recommandations	1	1

The running time of the algorithm is:

$$T(p) = p + 2 + p - r + 2 + 4 + p - r + 1 + 2p - 2r + 3p - 3r + 3p - 3r + 2p - 2r + 1 = 10 + 7p$$

Which means it has asymptotic complexity of class $O(p)$.

As regards space complexity, the data structures created by the function are:

1. "indexes": numpy array of length p.
2. "unseen": list of items of length $p - r = \frac{p}{2}$.
3. "recommendations": dictionary of size $r = \frac{p}{2}$.

So, assuming a unitary cost for each new memory space occupied we obtain:

$$\frac{p}{2} + \frac{p}{2} + p = 2p$$

Which means that the asymptotic complexity is of class $O(p)$.

So, by setting $p = \alpha n$ we obtain the same classes of complexity for both the analysis.

Conclusions

1. *Time:*
   As we saw for the User-based case, the class of asymptotic time complexity of this algorithm is $T(n) = O(n^2)$ (the one found for the factorization function, which represents the main step of the entire algorithm).

   Like we did before in the previous sections, now we analyze the influence explained separately by each parameter of the different functions. The algorithm of the gradient descend for the Non-negative factorization of the Interaction Matrix requires some nested loops due to the mathematical computations behind the factorization. Specifically, fixing all the parameters except for one the running time grows linearly both in the two dimensions n and p and in the model parameters f and s.
   The function that produces the suggestions has a running time that simply grows linearly in p.

2. *Space:*
   Additionally to what we've pointed out in the study of the asymptotic complexity for the two main functions, we need to recall that the step of producing the numpy array containing the cross-product between "U" and "I" (operation performed outside of the two functions described above) requires an extra $n \cdot p$ space. This doesn't produce any change in the class of asymptotic space complexity of the algorithm which is $T(n) = O(n^2)$ (that is logical cause we need to produce a new $n \times p$ factorized matrix).
   Considering the separate effects induced by all the parameters, we summarize below the previous considerations:

   - The parameter s doesn't affect the dimension of space required by the algorithm.
   - Fixing p and f, the space required grows linearly in n.
   - Fixing n and f, the space required grows linearly in p.
   - Fixing n and p, the space required grows linearly in f.

Test on real data

After having discussed about the features and implementations of the algorithms, now we show some results obtained by applying them to the data described in the second section of the essay (the test on the entire dataset can be found in the Juppyter notebooks).

General Output

The final output of each Recommender System for a given target user is a dictionary in which the items already unseen by the target user are assigned to a specific class of recommendation ("Strongly Recommended", "Recommended", "Not Recommended") basing the assignments on the different paradigms used by the three methods (indeed, we introduce this classification just to be able to compare the algorithms in terms of final suggestions obtained). The dictionary of suggestions appears like this:

```
Suggestions for user: 3

{'Strongly Recommended': ['The Lord of the Rings: The Fellowship of the Ring',
  'Fight Club',
  'The Matrix',
  'Back to the Future',
  'Return of the Jedi',
  'Raiders of the Lost Ark',
  'The Empire Strikes Back',
  'Fargo',
  'The Silence of the Lambs',
  'Dances with Wolves',
  'Terminator 2: Judgment Day',
  "Schindler's List",
  'The Shawshank Redemption',
  'Pulp Fiction',
  'Star Wars',
  'The Usual Suspects',
  'Twelve Monkeys',
  'Toy Story'],
 'Recommended': ['American Beauty',
  'Batman',
  'Jurassic Park',
  'The Fugitive',
  'True Lies',
  'Braveheart',
  'Se7en'],
 'Not Recommended': ['Independence Day']}
```

Figure 6: Final dictionary of suggestions

For the Matrix Factorization and the Item-based algorithms, we're sure that all the unseen items of the target user are assigned to a specific class of suggestion while in the User-based one some unrated movies can remain without a predicted rating (this is linked to the nature of this algorithm, indeed, if none of the k-nearest neighbors has seen one/more of the items, we aren't able to estimate the rating for the target user referred to that item).

Parameter's determination

As regards the choice of the parameters of the models, we made different attempts and we set:

- *User-based:*
  $k = 10$, since it seems to be a good compromise between the time required for the execution of the algorithm and the final result (i. e. the algorithm is able to estimate a rating for a good percentage of the target user's unseen items).

- *Item-based:*
  $k = 3$, since according to our data we found that on average number of "favourite items" for a single user is 3 or 4 while the number of unseen items for which we need to determine a suggestion is $\simeq 14$. So, with $k = 3$ usually we evaluate around 10-11 neighbors among the "unseen" items and we consider this a good

proportion in order to establish which movies appear frequently as nearest neighbors (and so they are "Strongly Recommended") and which does not appear (so they are "Not Recommended"). Indeed, the risk of setting an high value of k, is that all the unseen items tends to appear as nearest neighbors.

- *Matrix Factorization:*
  For this algorithm, we needed to make more evaluations due to the higher number of parameters it has:

  1. The value of α and β (respectively, the "update step" and the parameter for the regularization) have been decided by looking at their usual values on different online sources.
  2. Then, we discuss the value of f that has been decided after some attempts we summarize graphically below:

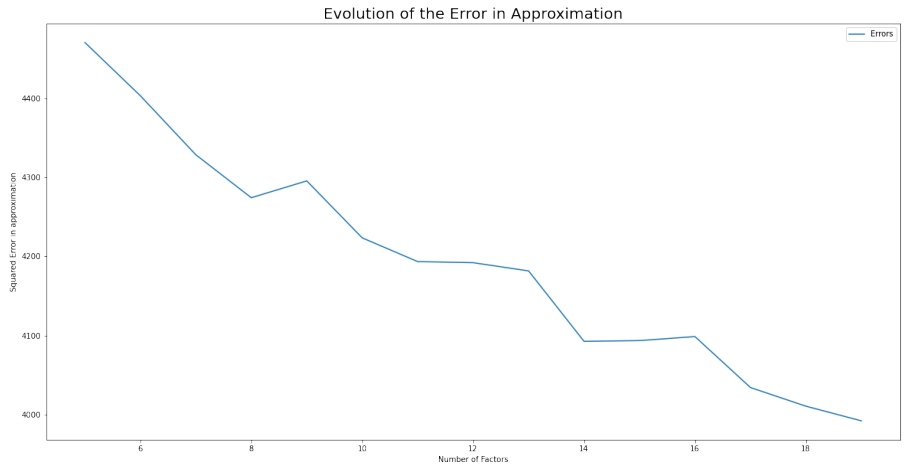

Figure 7: Decision on the parameter f

So we set $f = 14$ cause it seems to be a good compromise between the approximation of the Interaction Matrix and the numerosity of the factors with respect to the dimensionality of items $p = 31$. Obviously, a deeper statistical analysis of the data would have provided a better choice of f.

  3. Finally, for fixed α, β and f, we searched for the best value of s (maximum number of iteration allowed), in order to obtain an adequate approximation of the Interaction Matrix in a not excessive computational time required (the time required is expressed in seconds):

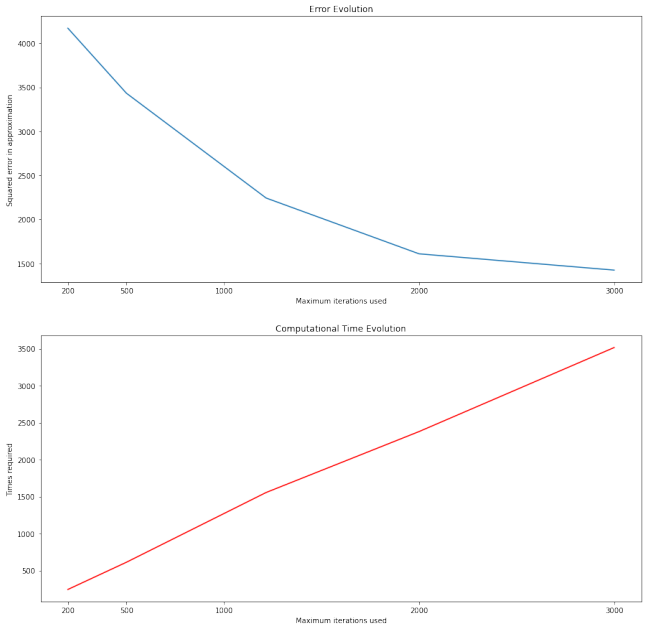

Figure 8: Decision on the parameter s

The above graphs lead us to choose $s = 2000$ since it seems to be a good trade off cause it produces the least relevant reduction in the value of E (indeed, with 3000 iteration we achieve a reduction of 200 which is low if compared with the previous ones) and the least restricted increase in time required (shifting from 2000 to 3000 produce an increase around 1900s)

Comparison

In order to compare how the different algorithms perform on the real data provided, we introduce a measure of comparison that evaluates how much two different methods agree in assigning the target user's unseen items to their class of suggestion ("Strongly Recommended", "Recommended" and "Not Recommended").
In particular, given two Recommender Systems, we evaluate how much they agree in suggestions for each user of the Interaction Matrix. The value of "accordance" for a generic user j is given by:

$$s_{1,2,j} = \frac{\sum_{i=1}^{k} a_i}{k}$$

Where k is the number of items for which we can compare the assignments provided by the different Recommender Systems (called "1" and "2") with respect to the j-th user, and a_i is the score obtained for each of the k items. The value a_i is determined as follow:

- Score "1": If the two Recommender System assign the i-th item to the both class (for instance, is the movie is assigned to the class "Not Recommended" in both the cases).
- Score "0.5": If the i-th item is assigned to similar classes (this is the case when the same item is assigned to "Strongly Recommended" by one and to "Recommended" by the other), since in that case the accordance is only partial.
- Score "0": If the i-th item is assigned to totally different classes (i. e. when one of the methods assign the item to the "Not Recommended" class and the other does not).

So, we've that $0 \leq s_{1,2,j} \leq 1$ and it approaches 1 as the two systems tends to provide similar suggestions, instead it approaches 0 as the two systems tends to disagree on the j-th user's recommendations.
Finally, we sum the value of the $s_{1,2,j}$'s for all the users in the Interaction Matrix and we obtain the final score of agreement between the two Recommender Systems:

$$S_{1,2} = \sum_{j=1}^{n} s_{1,2,j}$$

We obtained the following results:

- *User-based and Matrix Factorization*: We found a good agreement between the two methods, indeed our index is around 0.7 ($\simeq 0.68$ actually).
- *Item-based and Matrix Factorization*: Considering the Item-based algorithm, the matching between different methods of suggestion decreases significantly, indeed in that case we obtain a value of $\simeq 0.26$.
- *Item-based and User-based*: These two methods are the ones that agrees the less in determining suggestions for a certain user in the Interaction Matrix, indeed our index has a value of $\simeq 0.19$.

Final Comparison

In conclusion of the essay, we compare the three Recommender Systems and we discuss on which can be the best one, evaluating different points of view:

- **Time Asymptotic Complexity:**
  Starting from the overall evaluation of the three algorithms, clearly the one that achieves the worst performance is the Item-based, since it has asymptotic complexity of class $O(n^3)$, compared with the $T(n) = O(n^2)$ found for the other.

  Then we comment the behaviors of the running times depending on only one parameter per time. We saw that the Item-based and the Matrix Factorization solution are linear in n while the User-based has a worse class of complexity ($O(\log_2(n!))$), so this algorithm is the one that suffers more the increases in the number of users inside of the Interaction Matrix. Since n is usually the biggest dimension of the Interaction Matrix (because obviously we expect that the number of users is really bigger than the number of movies) and since on our real data we had $p = \frac{n}{15}$, that's the reason why we found this algorithm to be slower than the Item-based one.
  Looking at the parameter p, we saw that the User-based system grows quadratic in p, while the Item-based has the worst class of complexity measured with respect to this dimension ($O(p \cdot \log_2(p!))$), which is related to the operation performed on the Priority Queue needed to find the nearest neighbors.Instead, The Matrix Factorization algorithm is of class $T(\bar{n}, p) = O(p)$. So we can say that the Item-based and the User-based algorithms have a similar behaviour of their running times related to the column dimension (even if the Item based one has a worse rate of growth), while the Matrix-Factorization algorithm is the most efficient under this perspective.
  Instead, for what concerns the parameters that characterize the three models, we've seen that all the different algorithms have a running time which grows linearly with respect to their parameters (which is the number of k nearest neighbors to find for the User-based and the Item-based method and s and f in the Matrix factorization). So their running times behave in a similar way if we want to change the value of the parameters to improve the model performances. Anyway, we need to point out that the parameter s in the Matrix factorization algorithm hires a very big value (for instance, we set it equal to 2000 in our real data test) and this produces a huge increase in the time required to factorize the Interaction Matrix since the algorithm needs to loop a bunch of times to achieve a satisfactory approximation of the original matrix.

  To sum up, we can say that the Matrix Factorization algorithm seems to be the most efficient in terms of temporal asymptotic complexity since it has an overall running time of class $O(n^2)$ and the rates of growth computed for the variation of a single parameter are the bests (but the problems comes out if we're not able to reach a satisfactory approximation of the Interaction Matrix with a moderate value of s). Instead, the Item-based algorithm achieves the worst performances with respect both to the overall complexity ($O(n^3)$) and also looking at the rate of growth for the variation of p only (the reason why it has been found to be faster on our data is related to the small value of p).
  Anyway, the choice of the "best" algorithm in terms of temporal efficiency strictly depends on the nature of the data, that can have or not (like in our case) a balanced proportion between the two dimensions and can require different numbers of iterations to correctly factorize the Interaction Matrix.

- **Space Asymptotic complexity:**
  As regards the overall asymptotic space complexity of the algorithms, we immediately point out that each of them is of class $O(n^2)$, so there aren't no differences.

  Considering the separate variations of only one parameter per time, the User-based and the Matrix Factorization methods behave in a similar way, indeed, the space required grows linearly for each of the dimension involved and also with respect to their parameters.
  Instead the Item-based Recommender System has a space complexity that doesn't depend on the parameter set for the method but it has a quadratic term with respect to the parameter p.

  All considered, the worst class of complexity belongs to the Item-based Recommender System (due to the quadratic term in p that usually is way too big than the model parameters k and f).

- **Results on data:**
  Looking at the results obtained on our data, we can immediately notice that since the Item-based algorithm is the only one that doesn't ensure a predicted rating (and so a class of suggestion) for all the target user's unseen items, maybe we can consider it as the less powerful.
  Additionally, the "coefficient of similarities" between the suggestions proposed by the different methods provided in the "Comparison" Juppyter notebook, shows a remarkable accordance between the User-based and the Matrix Factorization algorithms, while the item-based one produces different results. This can mean that the Item-based recommendation is more differentiated than the other two, but maybe this stronger diversity in the recommendations proposed can also indicate that the Item-based approach is less adequate than the other two. This because the solid accordance between User-based and Matrix Factorization algorithms evidence a certain robustness in the results achieved (indeed these two methods estimate a rating while the Item-based is just a distance comparison). However, we need to say that the Item-based algorithms works mainly on the dimension related to the number of items in the matrix (p) and that probably our data had an excessive little value of p.
  Finally, we recall that the Matrix Factorization algorithm makes statistical considerations and investigation on data while the other two are based only distances between users and items expressed in terms of interactions. So, if a deeper statistical analysis of the dataset confirms that the factorization of the Interaction Matrix is correctly applicable, that method is clearly more logical and reasonable than the other two.