ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

UNIVERSITY OF PIRAEUS

Department of Computer Science
Postgraduate Studies Programme

Recognition of Standards

Coursework 2022-2023

Teaching Professor: Mr. Tsihritzis Georgios
Supervising Professor: Mr. Sotiropoulos Dionysios

Koutras Ioannis - Prodromos ΜΠΛΛ21039

Kormazos Evangelos ΜΠΛ21036 Hartsias

Marios ΜΠΛ21080

Piraeus, May 2023

Contents

1. Introduction

1.1 Objectives of the Work

The goal of the work is to generate movie recommendations for users based on their preferences and ratings, using artificial neural networks and data clustering techniques. Achieving this goal will allow us to provide more personalized movie recommendations to users, thus improving their experience and satisfaction with the movie recommendation service.

1.2 Description of the data

The problem we will solve is to develop movie recommendation algorithms using artificial neural networks and data clustering techniques and later measure accuracy in terms of prediction. The dataset we will work on can be downloaded from https://ieee-dataport.org/open-access/imdb-movie-reviews-dataset and is for a set of users $\square=\{U1,U2,\cdots,UN\}$ expressing preferences over a set of movie objects $\square=\{I1,I2,\cdots,IN\}$. The user $u\in\boldsymbol{U}$ assigns to movie $i\in\boldsymbol{I}$ the preference score $R(u,\square)\in\boldsymbol{Ro}=\{1,2,3,4,5,6,7,8,9,10\}\cup\{0\}$, where a value of 0 indicates no score rather than zero .

2. Short presentation of the data

The data downloaded from the above address is in a file named Dataset.npy and has the following format:

ur4592644	tt0120884	10	16 January 2005
ur3174947	tt0118688	3	16 January 2005
ur3780035	tt0387887	8	16 January 2005

What we did is to convert the file with the name: Dataset.csv and then to facilitate the processing of the data, we change their format to the file: dataset_updated.csv

User	Movie	Rate	Time
3174947	0118688	3	2005/01/16
3780035	0387887	8	2005/01/16

3. Pre-processing of data

3.1 Isolation of unique users and unique movies

To isolate unique users and movies we have used the Python libraries: pandas as pd. So we isolate users and movies in a dataframe using the unique() function.

```
User = df['User'].unique()
Movie= df['Movie'].unique()
print(f "Number of unique users: {len(User)}") print(f
"Number of unique items: {len(Movie)}\n")
```

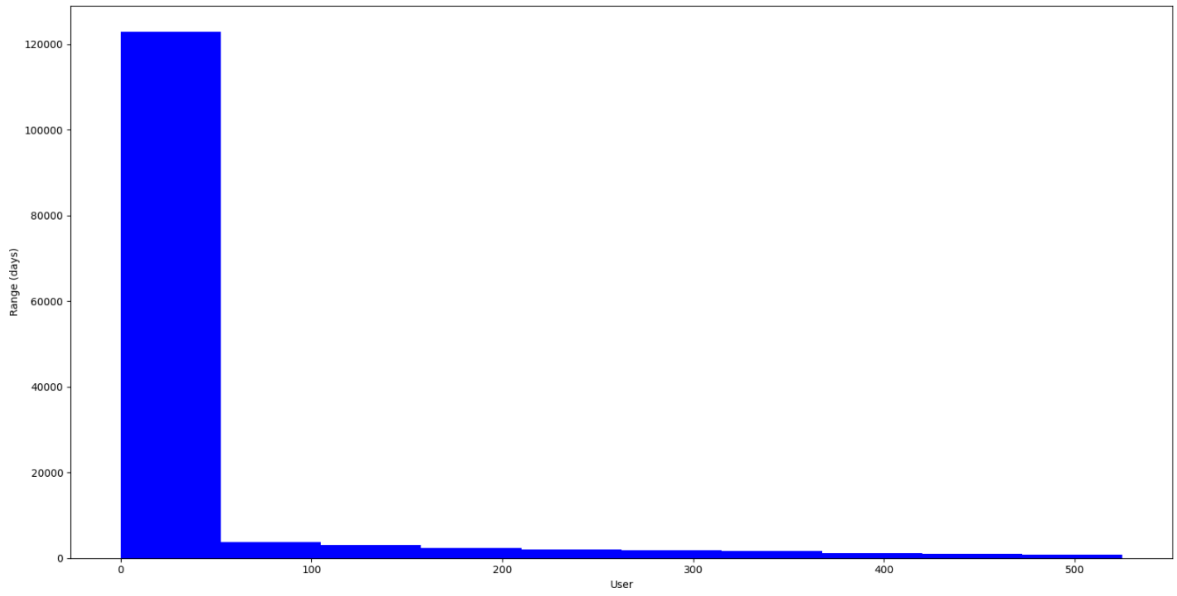
3.2 Isolation of unique users and unique movies using Rmin and Rmax

In this query we find Users with minimum Rmin and maximum Rmax ratings in order to "clip" the data into a range of ratings between Rmin and Rmax. The interpretation that could arise is for example that we are referring to users who depending on the number of ratings belong to a category. E.g. those who watch a maximum of 3 movies per week.

```
Rmin = 10
Rmax = 50
# Find the set of users with the required number of ratings
print("All User Ratings ")
user_counts = df.groupby('User').count()['Movie']
print(user_counts)
print("\n")
# Filter the user_counts array based on the condition Rmin <= count
<= Rmax
print("All User Ratings between "+ f"{Rmin}"+" and
"+f"{Rmax}")
user_counts_filtered = user_counts[user_counts.between(Rmin, Rmax)]
print(user_counts_filtered)
print("\n")
#Group the data by User and get the minimum and maximum Time for each
group
user_time = df.groupby('User')['Time'].agg(['min', 'max'])
# print(user_time)
# print("\n")

# convert the 'min' and 'max' columns to a datetime format
user_time['min'] = pd.to_datetime(user_time['min'])
user_time['max'] = pd.to_datetime(user_time['max'])
# calculate the range of time in days
user_time['Range'] = (user_time['max'] - user_time['min']).dt.days
#print the result
print(user_time)
print("\n")
```

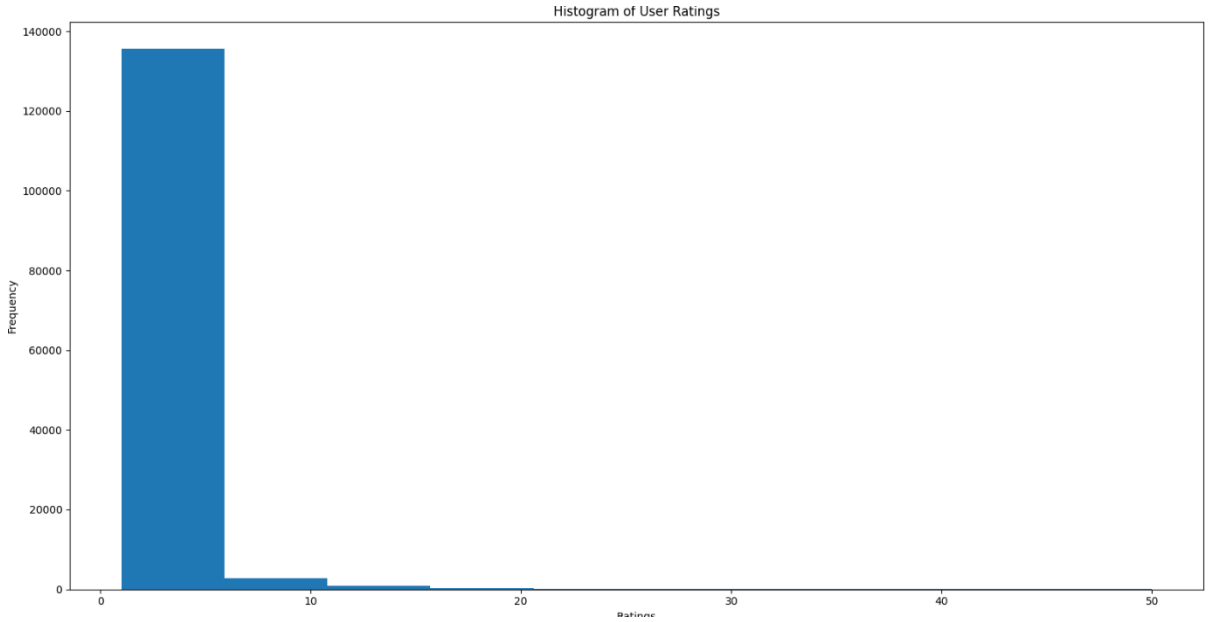
3.3 User Evaluation Range-Frequency histograms and number of user evaluations.

- Range in user evaluation days



0 to 50 users evaluate every 120000 days and so on.

- Frequency chart: number of user reviews



0 to 5 users rate many movies while the rest rate quite a few less.

Our Histograms show a result that is consistent with logic, since most users rate very rarely it makes sense that they have rated fewer movies. The opposite is also true for the minority of users who rate quite often and many movies.

```
#plotting the results
plt.hist(user_time['Range'],color='blue')

#    X-axis    name
plt.xlabel('User')
# Y-axis name
plt.ylabel('Range (days)')
plt.show()

# Plot the histogram
plt.hist(user_counts_filtered)
plt.title("Histogram of User Ratings")
plt.xlabel("Ratings")
plt.ylabel("Frequency")
plt.show()
```

3.4 Preference vector table R[user,movie].

We then created a vector table R in which users' preferences for the movies they have seen are included. Where there is a zero means no rating and not a zero review. The result looks something like this.

$$R = \begin{pmatrix} 10 & 0 & \cdots & 0 & 0 \\ 0 & 3 & & 0 & 0 \\ \vdots & & \ddots & \vdots & \\ 0 & 0 & \cdots & 8 & 0 \\ 0 & 3 & & 0 & 5 \end{pmatrix}$$

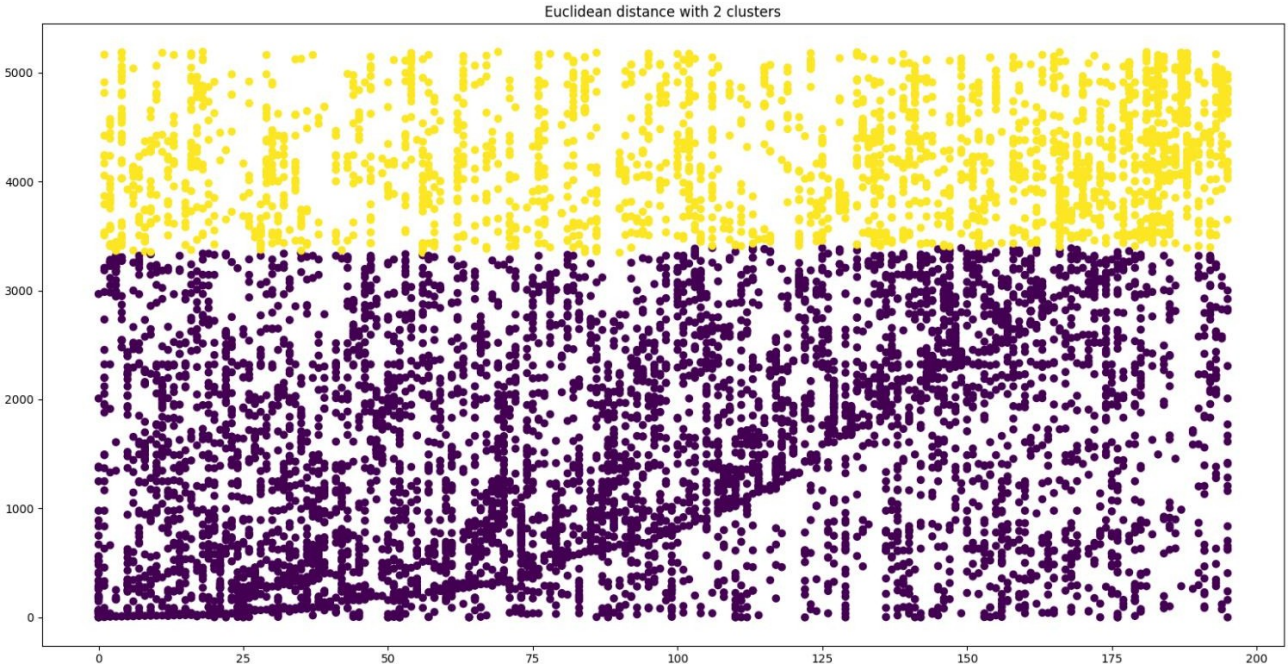
```
# Create a 2D numpy array filled with zeros
R = np.zeros((len(User), len(Movie)))

# Loop through the dataframe and populate the R array with ratings
for i, row in df.iterrows():
    user_index = np.where(User == row['User'])[0][0]
    movie_index = np.where(Movie == row['Movie'])[0][0]
    R[user_index, movie_index] = row['Rate']
print(R)
```

Data clustering algorithms

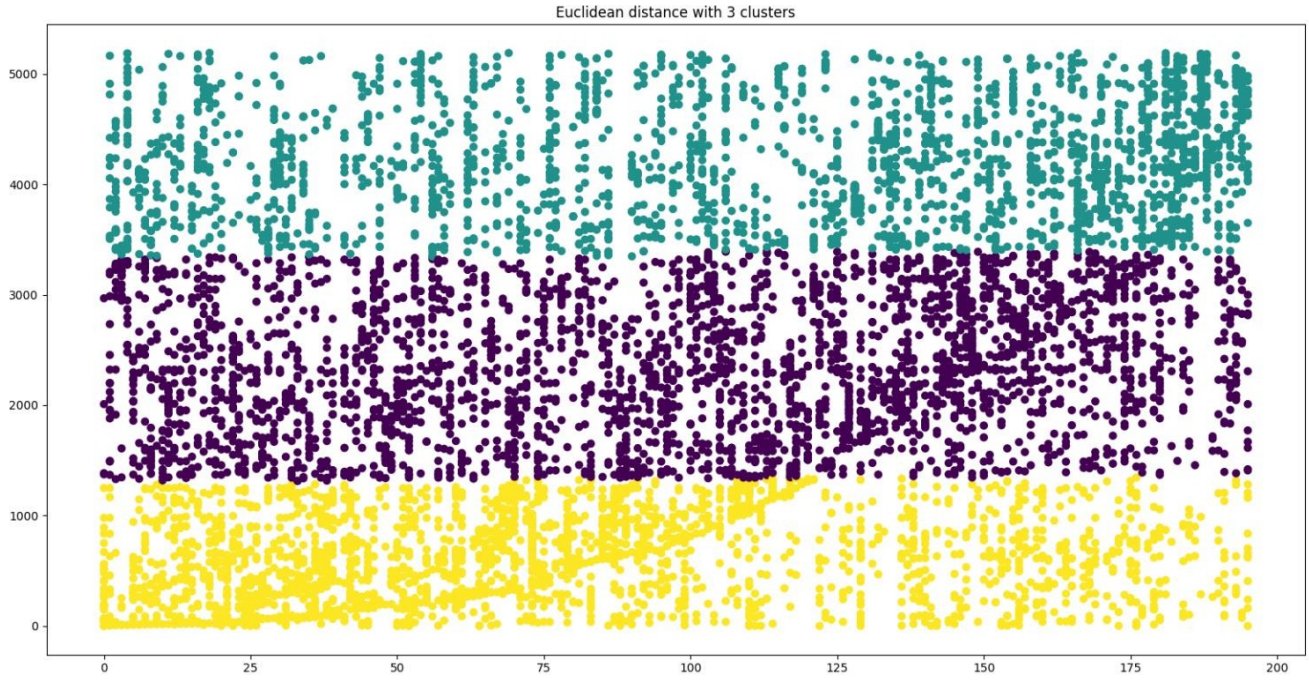
4.1 k-means, metric: Euclidean (number of data: 500,000)

T h e default norm used by k-means in our Python library is Euclidean; applying it to the R matrix gives us the following results:

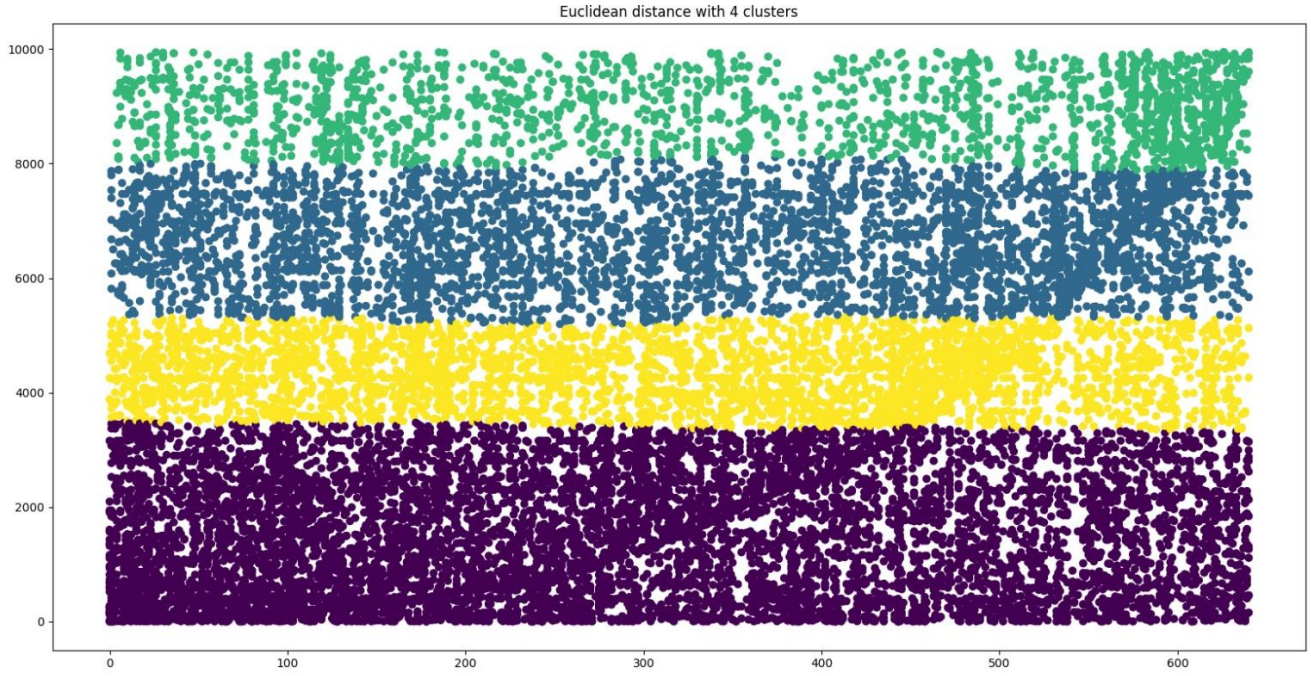
- *clusters= 2*



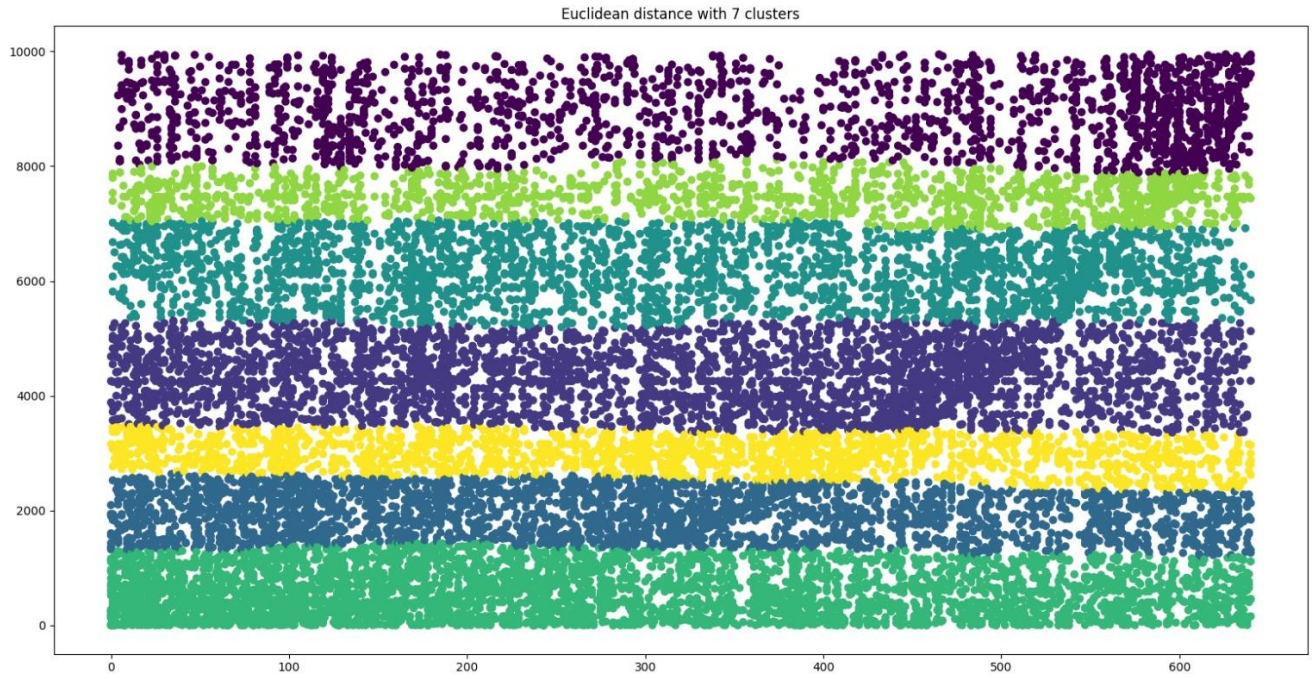
- *clusters= 3*



- *clusters= 3*



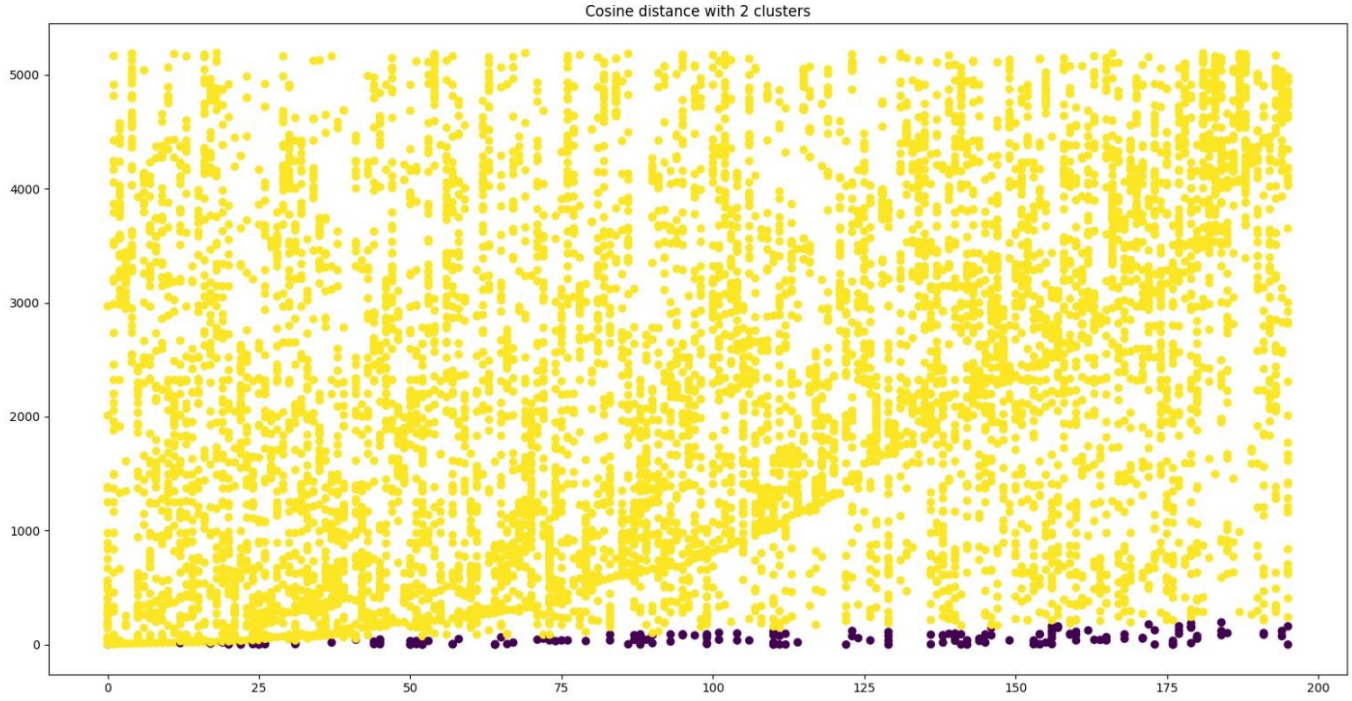
- *clusters= 7*


Euclidean distance with 7 clusters
10000
8000
6000
4000
2000
0
0
100
200
300
400
500
600

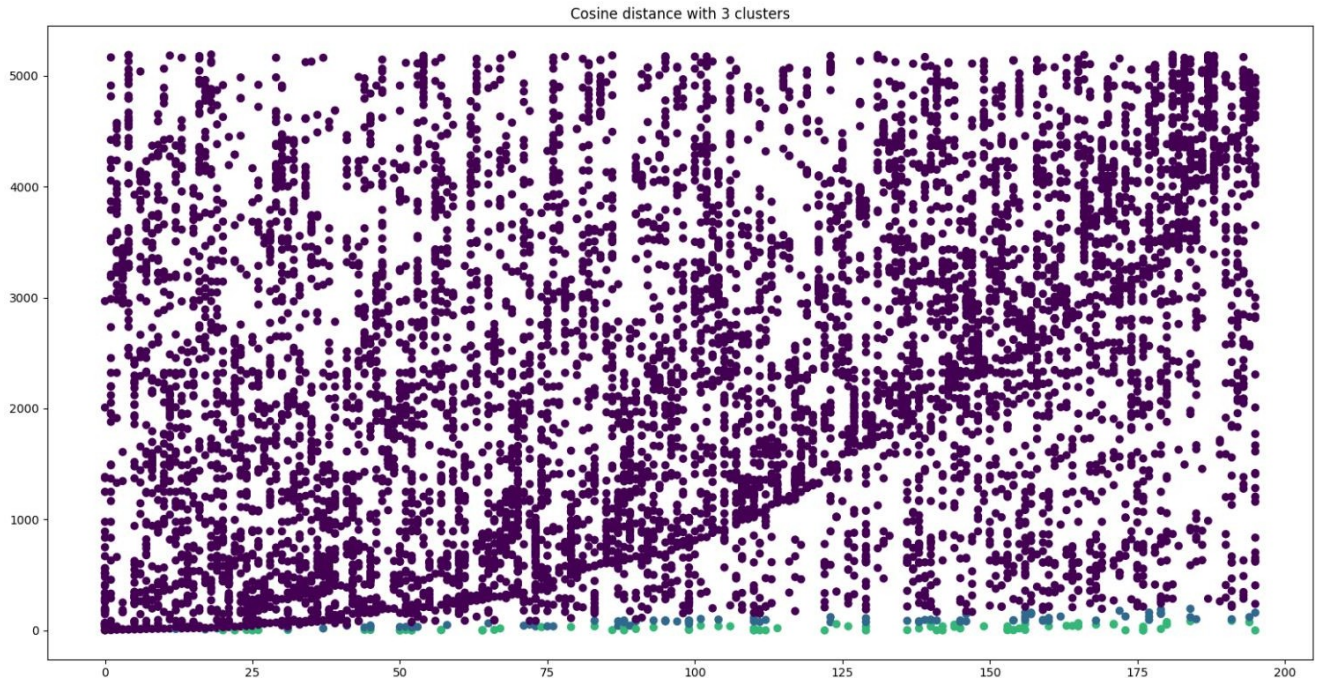
4.2 k-means, metric: Sentinel (data number: 500,000)

Changing the norm in our code will display the following data:

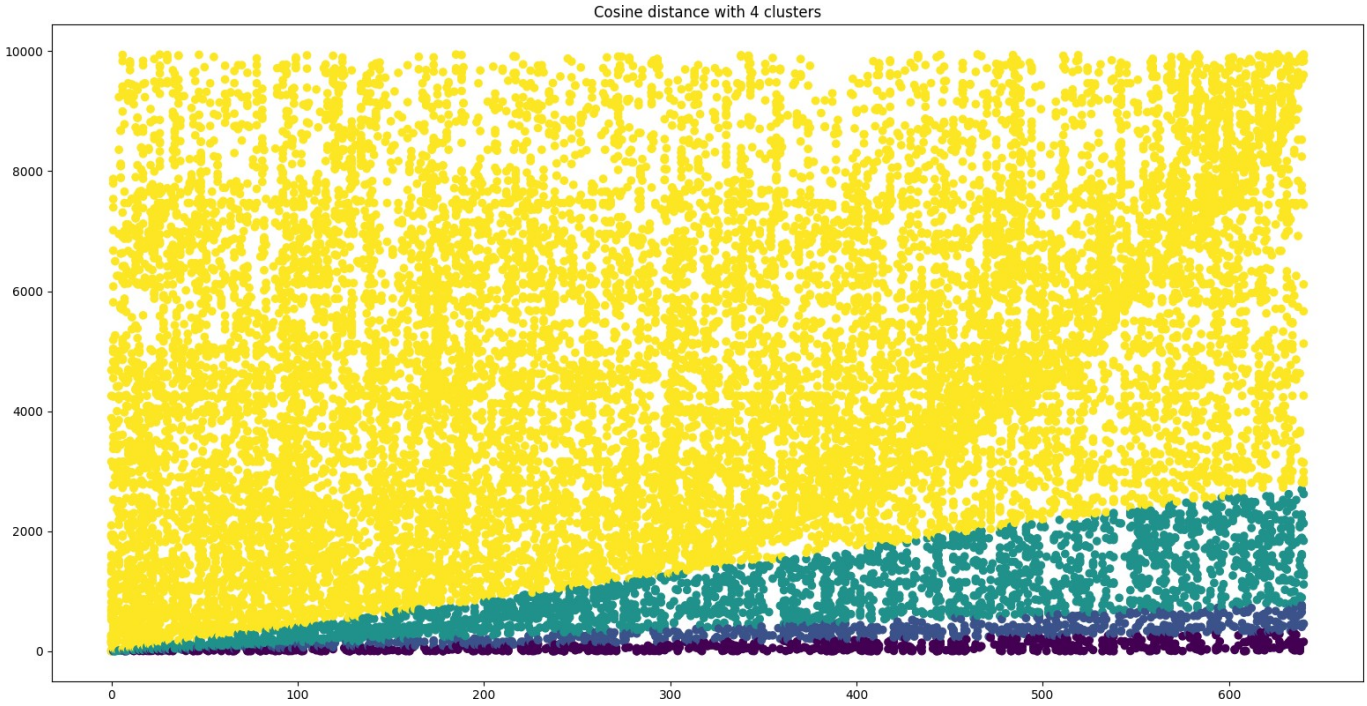
- *clusters= 2*



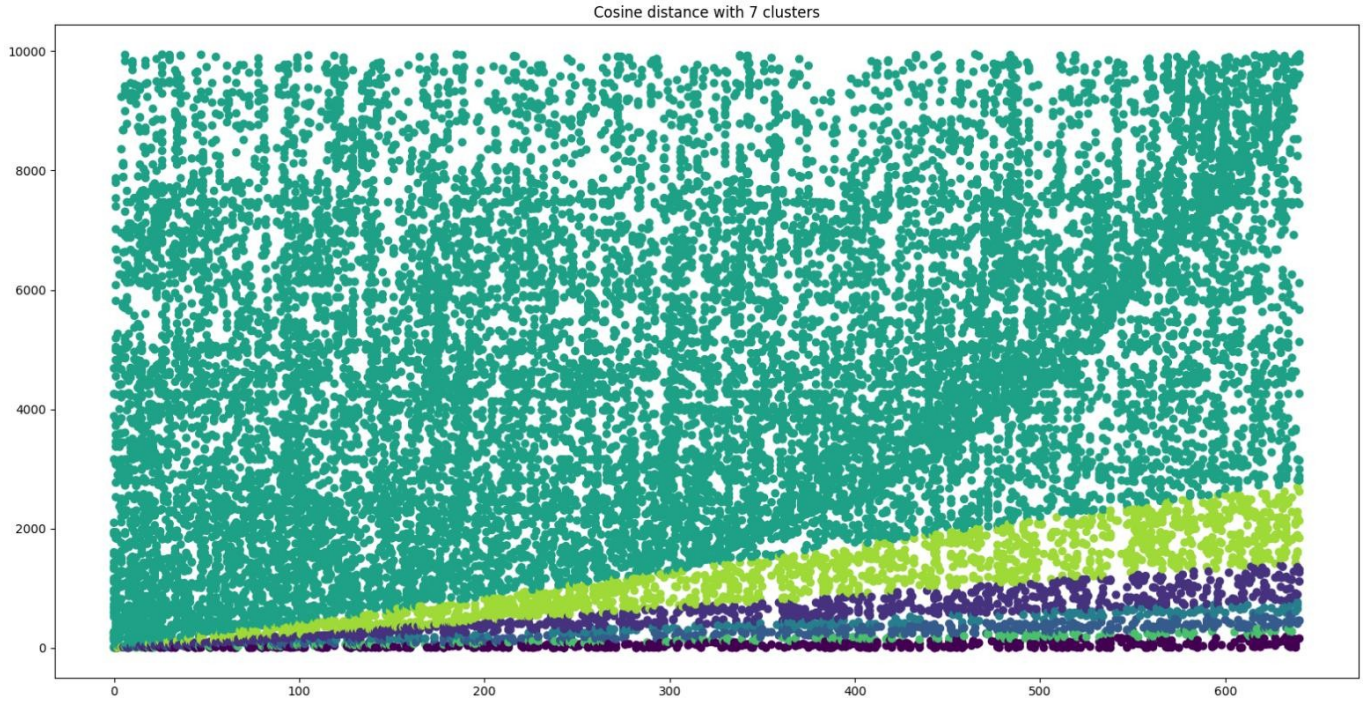
- *clusters= 3*



- *clusters= 4*



- *clusters= 7*



4.3 Commentary on results:

- Euclidean distance:

The Euclidean distance measures the straight line distance between two points in Euclidean space. In this context, it calculates the distance between two preference vectors. A shorter Euclidean distance indicates that users have similar preferences, while a longer distance indicates dissimilar preferences. The Euclidean distance can be affected by the scale of the rating system and may not be ideal when dealing with sparse data, as it does not take into account the direction of the preference vectors. However, it can be a good measurement when comparing user preferences on a fixed scale (e.g. from 1 to 10) and when the data is dense.

- Sentinel-like similarity:

Cosine similarity measures the cosine of the angle between two preference vectors. Values range between -1 (completely dissimilar) and 1 (completely similar). A value of 0 indicates that the vectors are orthogonal, meaning that they are not related. Cosine similarity is less sensitive to the magnitude of the ratings and focuses more on the direction of the preference vectors. This makes it more suitable for situations where users have different rating scales or when dealing with sparse
data.

Our code:

```
#Graph with Euclidean and cosine metric
User = df['User'].unique()
Movie= df['Movie'].unique()
#4 Question----------------------------------------------------------------------------------------
----------------------------


# Create a 2D numpy array filled with zeros
R = np.zeros((len(User), len(Movie)))

# Loop through the dataframe and populate the R array with ratings
for i, row in df.iterrows():
    user_index = np.where(User == row['User'])[0][0]
    movie_index = np.where(Movie == row['Movie'])[0][0]
    R[user_index, movie_index] = row['Rate']

try:
    answer = int(input("Enter the number of Cluster(s) - L you want
to get the Plot (Empty->Best Elbow method): "))
except ValueError:
    answer = None
if answer is None:
    answer=optimal_clusters('euclidean',R)
    print("Euclidean clusters Elbow=" + str(answer))
    make_plot('euclidean',answer,R)
```

```
    answer=optimal_clusters('cosine',R)
    print("Cosine clusters Elbow=" + str(answer))
    make_plot('cosine',answer,R)

else:
   print("Euclidean clusters Elbow=" + str(answer))
   make_plot('euclidean',answer,R)
   print("Cosine clusters Elbow=" + str(answer))
   make_plot('cosine',answer,R)
```

The functions we need to implement :

```
def make_plot(metric, cluster, array):
    # Remove zero values from the array
    array array_nonzero = array[array
    != 0]

    # Reshape the array into a 2D format
    array_reshaped = np.column_stack(np.where(array != 0))

    # Create a mapping from original indices to reshaped indices
    index_mapping = {tuple(idx): i for i, idx in
enumerate(np.column_stack(np.where(array != 0)))}

    # Calculate distances based on the specified metric
    if metric == 'cosine':
        distances = cosine_distances(array_reshaped)
        clustering = AgglomerativeClustering(n_clusters=cluster,
affinity='precomputed', linkage='average')
        clustering.fit(distances)
        plt.title(f'{metric.capitalize()} distance with {cluster}
clusters')
    elif metric == 'euclidean':
        distances = euclidean_distances(array_reshaped)
        clustering = AgglomerativeClustering(n_clusters=cluster,
affinity='precomputed', linkage='average')
        clustering.fit(distances)
        plt.title(f'{metric.capitalize()} distance with {cluster}
clusters')
    elif metric == 'jaccard':
        distances = jaccard_distances(array)
        clustering = AgglomerativeClustering(n_clusters=cluster,
affinity='precomputed', linkage='average')
        clustering.fit(distances)
        plt.title(f'Jaccard distance with {cluster} clusters')
    else:
        raise ValueError("Invalid metric specified. Supported
metrics: 'cosine', 'euclidean'")

    # Create a labels array with the same shape as array_reshaped
    reshaped_labels = np.full(array_reshaped.shape[0], -1)
    for i, label in enumerate(clustering.labels_):
        reshaped_labels[index_mapping[tuple(array_reshaped[i])]] =
label

    # Plot the points with different colors for each cluster
    plt.scatter(array_reshaped[:, 0], array_reshaped[:, 1],
c=reshaped_labels, cmap='viridis')

    # Set plot title and show the plot
    plt.show()
```

```
def jaccard_distances(array):
    n_users, n_movies = array.shape
    distances = np.zeros((n_users, n_users))

    for u in range(n_users):
        for v in range(u+1, n_users):
            u_rated = set(np.where(array[u, :] > 0)[0])
            v_rated = set(np.where(array[v, :] > 0)[0])
            intersection = len(u_rated & v_rated)
            union = len(u_rated | v_rated)
            if union == 0:
                distances[u, v] = 1
            else:
                distances[u, v] = 1 - (intersection / union)
            distances[v, u] = distances[u, v]
            #print(distances)

    return distances



def optimal_clusters(metric, array, min_clusters=2, max_clusters=10):
    array_nonzero = array[array != 0]
    array_reshaped = np.column_stack(np.where(array != 0))

    if metric == 'cosine':
        distances = cosine_distances(array_reshaped)
    elif metric == 'euclidean':
        distances = euclidean_distances(array_reshaped)
    elif metric == 'jaccard':
        distances = jaccard_distances(array)
    else:
        raise ValueError("Invalid metric specified. Supported
metrics: 'cosine', 'euclidean', 'jaccard'")

    cluster_range = range(min_clusters, max_clusters + 1)
    silhouette_scores = []

    for n_clusters in cluster_range:
        clustering = AgglomerativeClustering(n_clusters=n_clusters,
affinity='precomputed', linkage='average')
        clustering.fit(distances)
        labels = clustering.labels_
        silhouette_avg = silhouette_score(distances, labels,
metric='precomputed')
        silhouette_scores.append(silhouette_avg)

    best_cluster_num =
cluster_range[silhouette_scores.index(max(silhouette_scores))]
    return best_cluster_num
```

4. Recommendation generation algorithms using artificial neural networks

5.1 Explanation of Jaccard metric

The given metric in Python is called Jaccard - distance is a metric that measures the dissimilarity between two sets by comparing their intersection and union. In the context of user element tables, where users represent rows and elements (e.g. bands) represent columns, the Jaccard distance
compares the sets of items scored per two users.

$$dist(u, v) = 1 - \frac{|\varphi(u) \cap \varphi(v)|}{|\varphi(u) \cup \varphi(v)|}$$

A Jaccard distance of 0 indicates that the two users have rated the same set of items, while a distance of 1 means that they have no items in common in their rated sets.

Disadvantages of Jaccard distance compared to Euclidean distance and cosine similarity:

BINARY information: Jaccard distance only considers whether or not users have rated items and does not take into account actual ratings. Euclidean distance and cosine similarity, on the other hand, take into account rating values, providing more nuances about user preferences.

Sparse data: In user data tables, there are often many missing values, with
as a result the tables are sparse. Jaccard distance may be less effective in such cases, as it only takes into account the presence or absence of ratings. Euclidean distance and cosine similarity can work with sparse data by exploiting non-zero evaluations.

Sensitivity to the size of the scored sets: the Jaccard distance is sensitive to the size of the scored sets, as it takes into account their intersection and union. If two users have very large or very small sets of rated items, the Jaccard distance may not be as informative. Euclidean distance and cosine similarity are less sensitive to the size of nominal sets, as they focus on the relationship between the score values.

Unchanging scale: Cosine similarity is scale invariant, meaning that it is not affected by the magnitude of the rating values. This is useful when comparing users who may have different rating scales or trends. Jaccard distance and Euclidean distance are not scale invariant, which may affect their ability to accurately capture the similarity between users with different evaluation behaviors.

Despite these drawbacks, Jaccard distance can still be useful in some cases, such as when dealing with binary data or when the focus is on comparing data sets rather than actual evaluation values. The choice of the similarity metric depends on the specific problem and the nature of the data being analyzed.

$$dist_{euclidean}(R_u, R_v) = \sqrt{\sum_{k=1}^{m} |R_u(k) - R_v(k)|^2 \, \lambda_u(k)\lambda_v(k)} \quad (2)$$ έτσι ώστε:

$$\lambda_j(k) = \begin{cases} 1, & R(u_j, i_k) > 0; \\ 0, & R(u_j, i_k) > 0. \end{cases} \quad (3)$$

$$dist_{cosine}(R_u, R_v) = 1 - \left| \frac{\sum_{k=1}^{m} R_u(k) R_v(k) \lambda_u(k)\lambda_v(k)}{\sqrt{\sum_{k=1}^{m} R_u(k)^2 \lambda_u(k)\lambda_v(k)} \sqrt{\sum_{k=1}^{m} R_v(k)^2 \lambda_u(k)\lambda_v(k)}} \right| \quad (4)$$

Finally, in relation (2) the grouping of users is based on the favorite movies of two users, while in (4) only if they have both seen a movie, without caring about their preference for it, is taken into account. Therefore, we can say that in our case Jaccard distance divides users based on the type of movie they like (e.g., they like comedies but not Woody Allen movies)

Separate users into groups (Optional question)

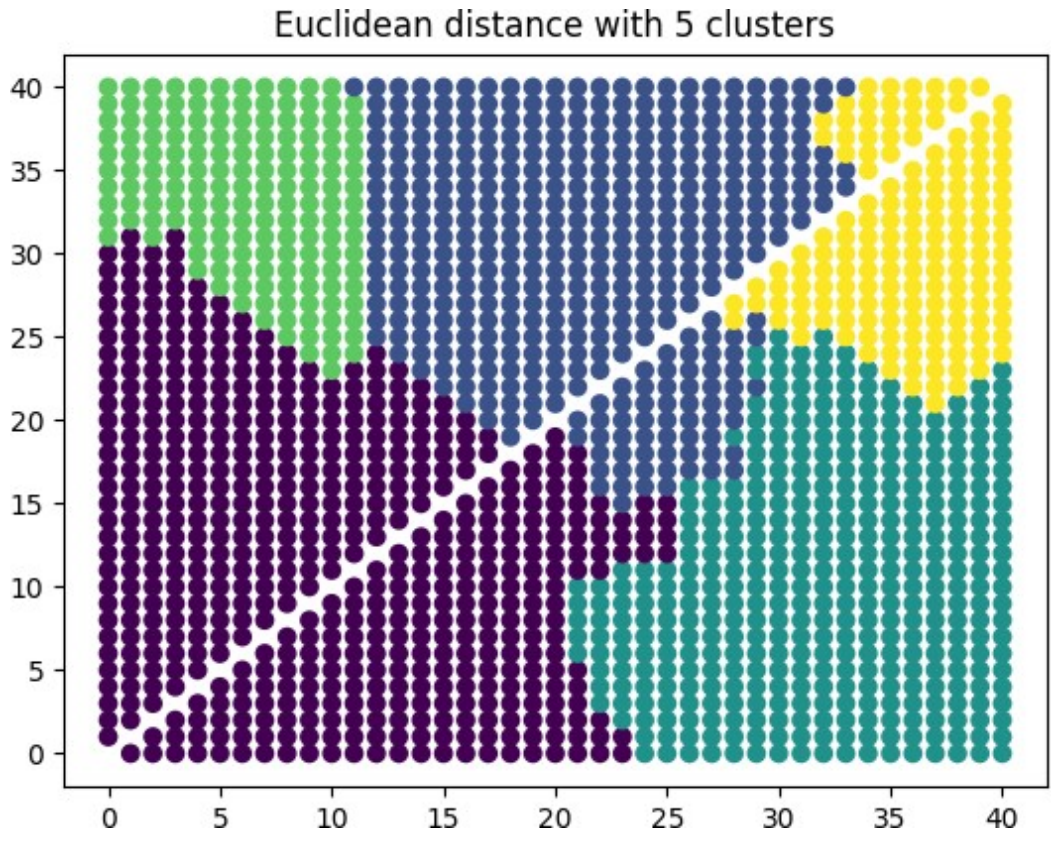
After constructing the quadratic similarity matrix (using Jaccard distances)

```
Rmin = 30
Rmax = 40
# Group by and filter dataframe based on number of ratings per user
user_counts = df.groupby('User').count()['Movie']
user_counts_filtered = user_counts[user_counts.between(Rmin, Rmax)]
df_filtered = df[df['User'].isin(user_counts_filtered.index)]

# Print number of users with ratings between Rmin and Rmax
num_users_filtered = len(user_counts_filtered)
print(f "Number of users with ratings between {Rmin} and {Rmax}:
{num_users_filtered}")
print(user_counts_filtered)

# Create a list of unique users and movies
User = df_filtered['User'].unique()
Movie = df_filtered['Movie'].unique()

# Create a 2D numpy array filled with zeros
R = np.zeros((len(User), len(Movie)))
```

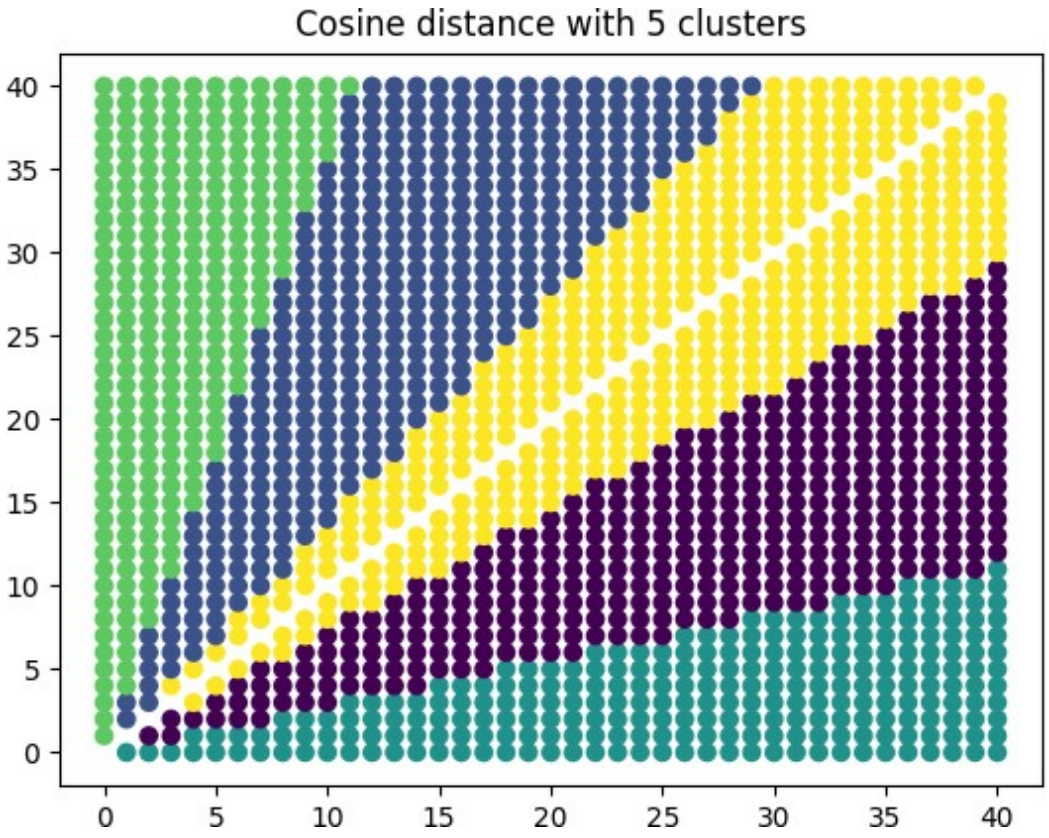
```
# Loop through the filtered dataframe and populate the R array with
ratings
for i, row in df_filtered.iterrows():
    user_index = np.where(User == row['User'])[0][0]
    movie_index = np.where(Movie == row['Movie'])[0][0]
    R[user_index, movie_index] = row['Rate']

jaccard_array = jaccard_distances(R)
```

We will divide users into 5 categories based on:

- Euclidean metric:



- Normalized metric:



5.2 Creating a function to determine neighbor preferences After we

filter the data:

```
Rmin = 40
Rmax = 50
# Group by and filter dataframe based on number of ratings per user
user_counts = df.groupby('User').count()['Movie']
user_counts_filtered = user_counts[user_counts.between(Rmin, Rmax)]
df_filtered = df[df['User'].isin(user_counts_filtered.index)]

# Print number of users with ratings between Rmin and Rmax
num_users_filtered = len(user_counts_filtered)
print(f "Number of users with ratings between {Rmin} and {Rmax}:
{num_users_filtered}")
#print(user_counts_filtered)

# Create a list of unique users and movies
User = df_filtered['User'].unique()
Movie = df_filtered['Movie'].unique()

# Create a 2D numpy array filled with zeros
R = np.zeros((len(User), len(Movie)))

# Loop through the filtered dataframe and populate the R array with
ratings
for i, row in df_filtered.iterrows():
    user_index = np.where(User == row['User'])[0][0]
    movie_index = np.where(Movie == row['Movie'])[0][0]
    R[user_index, movie_index] = row['Rate']
```

We will find the k nearest neighbours based on the metric of relation (4):

And we will store them in a dictionary-dictionary in order to be able to manage them:

```
# Define the number of nearest k-1 neighbours to
consider k = 31

# Compute the Jaccard similarity matrix for all users in the filtered
dataset
simularity_matrix = 1 - jaccard_distances(R)

# Create a list to store the top k users for each user in the
filtered dataset
neighbor_list = []

print(simularity_matrix)
print(jaccard_distances(R))
print('/n')

id_answer_list=[]
id_dictionary={user: {} for user in User}
# Loop over each user in the filtered dataset
for i, user in enumerate(User):
    # Get the row of the Jaccard similarity matrix corresponding to
the current user
    jaccard_row = simularity_matrix[i, :]
```

```
    # Sort the Jaccard similarity row in descending order and get the
indices of the top k users
    topk_indices = np.argsort(-jaccard_row)[:k]

    # Get the user IDs of the top k users (excluding the current
user)
    topk_users = [User[idx] for idx in topk_indices if User[idx] !=
user]

    # Store the top k users in the neighbor list for the current user
    neighbor_list.append(topk_users)

    # Print the top k users for the current user #print(f
    "Neighbors of user {user}: {topk_users}")

    id_answer_list.append(user)
    id_dictionary[user]={'user' : user,'neighbor_list' :
neighbor_list}
    neighbor_list=[]

# Loop over each user in the answer list
for user in id_answer_list:
    # Print the neighbor list for the current user print(f
    "Neighbor list for user {user}:
{id_dictionary[user]['neighbor_list']}")

print(id_answer_list)
```

We will do the same procedure for each user's answers so that opposite each line of the dictionary with the neighbours we have the corresponding answers.

```
ratings_answers = []
for user in id_answer_list:
    # Find the index of the user in the User array
    user_index = np.where(User == user)[0][0]

    # Access the row of the R array corresponding to the user's index
to get the ratings for all movies
    user_ratings = R[user_index, :]
    ratings_answers.append(user_ratings)

print(ratings_answers)
print(len(ratings_answers))

ratings_dictionary = {user: {} for user in User}
for user_id, user_data in id_dictionary.items():
    user_ratings = []
    for neighbor_list in user_data['neighbor_list']:
        for neighbor in neighbor_list:
            neighbor_index = np.where(User == neighbor)[0][0]
            neighbor_ratings = R[neighbor_index, :]
            ratings_dictionary[user_id][neighbor] = neighbor_ratings

for user_id, user_data in ratings_dictionary.items():
    neighbor_ratings = user_data.values() print(f
    "Ratings of neighbors for user {user_id}:
{list(neighbor_ratings)}")
```

Based on the above in our code, we will print the nearest neighbours for each neighbour as follows:

```
Neighbor list for user 470968: [[3167531, 2885836, 1790888, 3613088, 892463, 547823, 64493, 214
Neighbor list for user 2074560: [[1790888, 4670962, 1732001, 3478561, 4195782, 2854818, 892463,
```

We will then look up the ratings of these users in our original table and create a dictionary where for each neighbor we will have their ratings.

```
Ratings of neighbors for user 470968: [array([0., 7., 0., ..., 0., 0., 0.]), array([0., 0., 0., ..., 0., 0., 0.]), a
Ratings of neighbors for user 2074560: [array([1., 0., 0., ..., 0., 0., 0.]), array([0., 0., 0., ..., 0., 0., 0.]),
```

Then we need to search for a user in order to recommend a movie to them:

```
manual_input = input("Do you want to input a user manually? (y/n)")

if manual_input.lower() == "y":
    # Prompt the user to input the user ID
    #input_user = int(input("Enter the ID of the user:"))
    input_user = 6222226
else:
    # Select a random user from the dataframe
    input_user = int(df_filtered['User'].sample().iloc[0])

    # Print the selected user
    print("Randomly selected user:", input_user)

# Check if input user exists in dataset
if input_user not in User:
    print("User not found in dataset.")
else:
    # Find the index of the user in the User array
    user_index = np.where(User == input_user)[0][0]

    # Access the row of the R array corresponding to the user's index
to get the ratings for all movies
    user_ratings = R[user_index, :]

    # Print the ratings of the input user for all movies print(f
    "Ratings of user {input_user} for all movies:
{user_ratings}")

    ratings_dictionary = {user: {} for user in User}

    for user_id, user_data in id_dictionary.items():
        user_ratings = []
        for neighbor_list in user_data['neighbor_list']:
            for neighbor in neighbor_list:
                neighbor_index = np.where(User == neighbor)[0][0]
                neighbor_ratings = R[neighbor_index, :]
                ratings_dictionary[user_id][neighbor] =
neighbor_ratings
```

5.3 Configuring the data

To test the model we will build, we will first have to split our data by a ratio of 80-20. Into what we will need to train it and what we will need to test it.

```
train_ratings_dict = {}
test_ratings_dict = {}

# Loop over each user and their neighbor ratings
for user_id, user_data in ratings_dictionary.items():
    # Collect the neighbor ratings into a list
    neighbor_ratings_list = list(user_data.values())

    # Combine the neighbor ratings into a 2D numpy array
    input_features = np.array(neighbor_ratings_list)

    # Create an array of actual ratings for the input user, based on
the ratings_answers dictionary
    actual_rating = np.array([ratings_answers[np.where(User ==
user_id)[0][0]]] * len(input_features))

    # Split the input features and actual ratings into training and
testing sets
    train_features, test_features, train_labels, test_labels =
train_test_split(
        input_features, actual_rating, test_size=0.2, random_state=42
    )

    # Create a dictionary containing the training features and labels
for the current user
    train_ratings_dict[user_id] = {'input_features': train_features,
'labels': train_labels}

    # Create a dictionary containing the testing features and labels
for the current user
    test_ratings_dict[user_id] = {'input_features': test_features,
'labels': test_labels}

# Combine the training features and labels for all users into a
single dataframe and numpy array, respectively
x_train = pd.concat(
    [pd.DataFrame(train_ratings_dict[user_id]['input_features']) for
user_id in train_ratings_dict.keys()],
    ignore_index=True)
y_train = np.concatenate([train_ratings_dict[user_id]['labels'] for
user_id in train_ratings_dict.keys()])

# Combine the testing features and labels for all users into a single
dataframe and numpy array, respectively
x_test = pd.concat(
    [pd.DataFrame(test_ratings_dict[user_id]['input_features']) for
user_id in test_ratings_dict.keys()],
    ignore_index=True)
y_test = np.concatenate([test_ratings_dict[user_id]['labels'] for
user_id in test_ratings_dict.keys()])
```

5.4 Neural network construction

After testing different architectures either in terms of layers or by using different algorithms we came up with the following model:

```
model = Sequential()
model.add(Embedding(len(Movie) + 1, 16,
input_length=x_train.shape[1]))
model.add(Flatten())
model.add(Dense(len(Movie), activation='linear'))

#model.add(Dense(64, activation='relu'))
#model.add(Dense(32, activation='relu'))
#model.add(Dense(16, activation='linear'))
#model.add(Dense(32, activation='relu'))
#model.add(Dense(len(Movie), activation='relu'))
#model.add(Dense(len(Movie), activation='relu'))
model.add(Dense(len(Movie), activation='sigmoid'))
model.add(Dense(len(Movie), activation='softmax'))
#model.add(Dense(len(Movie), activation='relu'))


model.compile(loss='categorical_crossentropy', optimizer='sgd',
metrics=['accuracy', 'mean_squared_error'])
#model.compile(loss='binary_crossentropy', optimizer='sgd',
metrics=['accuracy', 'mean_squared_error'])
#model.compile(loss='mean_squared_error', optimizer='sgd',
metrics=['accuracy', 'mean_squared_error'])

model.fit(x_train, y_train, epochs=100, batch_size=32)
```

Below we will see the results in the case where the compile is done with mean_squared_error :

```
model.compile(loss='mean_squared_error', optimizer='sgd',
metrics=['accuracy','mean_squared_error'])
```

Output:

```
Epoch 49/50
19/19 [==============================] - 2s 124ms/step - loss: 1.7822 - accuracy: 0.0000e+00 - mean_squared_error: 1.7822
Epoch 50/50
19/19 [==============================] - 2s 125ms/step - loss: 1.7822 - accuracy: 0.0000e+00 - mean_squared_error: 1.7822
```

model.predict(x_test):

```
[[0.00099619 0.00218159 0.00047892 ... 0.00123741 0.00091329 0.00082378]
 [0.00099522 0.00217416 0.0004804  ... 0.00124409 0.0009137  0.00082174]
 [0.00099604 0.00217647 0.00048023 ... 0.00124023 0.00091367 0.00082427]
 ...
 [0.00099335 0.00218059 0.00047954 ... 0.00124427 0.00091225 0.00082451]
 [0.0009945  0.0021755  0.00047901 ... 0.0012412  0.00091507 0.00082457]]
```

print(y_test):

```
[[6. 3. 0. ... 0. 0. 0.]
 [6. 3. 0. ... 0. 0. 0.]
 [6. 3. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 0. 0. 0.]]
```

Next we will see the results in the compile case where the compile is done with binary_crossentropy :

```
model.compile(loss='binary_crossentropy', optimizer='sgd',
metrics=['accuracy', 'mean_squared_error'])
```

Output:

```
Epoch 49/50
19/19 [==============================] - 2s 122ms/step - loss: 0.5338 - accuracy: 0.0000e+00 - mean_squared_error: 1.7810
Epoch 50/50
19/19 [==============================] - 2s 123ms/step - loss: 0.5316 - accuracy: 0.0000e+00 - mean_squared_error: 1.7810
```

model.predict(x_test):

```
[[0.00126065 0.00267206 0.00035228 ... 0.0006449  0.00048453 0.0005348 ]
 [0.00126412 0.00266857 0.00035179 ... 0.00064343 0.00048663 0.00053267]
 [0.00126198 0.00267397 0.00035128 ... 0.00064272 0.00048798 0.00053318]
 ...
 [0.00126776 0.0026753  0.0003519  ... 0.00064053 0.00048498 0.00053381]
 [0.00126566 0.00268065 0.00035197 ... 0.00064397 0.00048687 0.00053418]]
```

Finally, we will see the results in the case where the compile is done with categorical_crossentropy:

```
model.compile(loss='categorical_crossentropy', optimizer='sgd',
metrics=['accuracy', 'mean_squared_error'])
```

Output:

```
Epoch 49/50
19/19 [==============================] - 2s 123ms/step - loss: 6589173.0000 - accuracy: 0.0068 - mean_squared_error: 1.7792
Epoch 50/50
19/19 [==============================] - 2s 125ms/step - loss: 6723368.0000 - accuracy: 0.0287 - mean_squared_error: 1.7792
```

In which they achieve greater accuracy.

5. Explanations-Bibliography

6.1 Elbow method

To identify clusters we used (optionally) the Elbow method:
The Elbow method is a technique used in cluster analysis for determining the optimal number of clusters to be used in clustering techniques such as the K-means method.

The technique works as follows: First, we perform the clustering technique for various numbers of clusters (for example, from 1 to 10) and for each number, we calculate the total sum of squares within the cluster (WSS), which is essentially the total distance between each point and the center of the cluster to which it belongs.

Then plot the WSS against the number of clusters. The resulting graph often resembles a human hand with a sharp bend or "elbow". The "elbow" is the point at which the WSS starts to decrease much more slowly, and this is usually the optimal number of clusters. The idea is that increasing the number of clusters beyond this point will not provide much better modeling of the data.

The Elbow method may not always be able to find the optimal number of clusters, especially if the data is not very well structured. However, it is a good starting point for cluster analysis.

In our code , using the function optimal_clusters we compute the number of clusters using the Elbow method:

```
def optimal_clusters(metric, array, min_clusters=2, max_clusters=10):
    array_nonzero = array[array != 0]
    array_reshaped = np.column_stack(np.where(array != 0))

    if metric == "cosine":
        distances = cosine_distances(array_reshaped) elif
    metric == "euclidean":
        distances = euclidean_distances(array_reshaped) elif
    metric == "jaccard":
        distances = jaccard_distances(array)
    else:
        raise ValueError("Invalid metric specified. Supported
metrics: 'cosine', 'euclidean', 'jaccard'")

    cluster_range = range(min_clusters, max_clusters + 1)
    silhouette_scores = []

    for n_clusters in cluster_range:
```

```
            clustering = AgglomerativeClustering(n_clusters=n_clusters,
affinity="precomputed", linkage="average")
            clustering.fit(distances) labels =
            clustering.labels_

            silhouette_avg = silhouette_score(distances, labels,
metric="precomputed")
            silhouette_scores.append(silhouette_avg)

      best_cluster_num =
cluster_range[silhouette_scores.index(max(silhouette_scores))]
      return best_cluster_num
```

6.2 Bibliography

- Books:

1. "Python Machine Learning" by Sebastian Raschka and Vahid Mirjalili (ISBN -13: 978-1789955750).
2. "Recommender Systems: The Textbook" by Charu C. Aggarwal (ISBN -13: 978-3319296579).
3. "Mining of Massive Datasets" by Jure Leskovec, Anand Rajaraman, and Jeff Ullman (ISBN-13: 978-1107077232).

- Scientific articles:

1. "Matrix Factorization Techniques for Recommender Systems" by Yehuda Koren, Robert Bell, and Chris Volinsky. This paper discusses techniques used by the team that won the Netflix Prize, a movie recommendation contest.
2. "Clustering methods in collaborative filtering based r ecommender systems" by Ewa Szymańska, Jarosław Górski, and Michał Malicki. This paper discusses the role of clustering techniques in recommender systems.
3. "Content-based movie recommendation system using the emotion and genre of the movie" by Nitin Agarwal and Dhruv Batra. This paper discusses a content-based movie recommendation system that uses the genre and the emotions of the movie.

- Online Courses and Aids:

1. Coursera's "Machine Learning" by Andrew Ng.
2. "Building Recommender Systems with Machine Learning and AI" on Udemy by Sundog Education
3. "Collaborative Filtering on MovieLens Data in Python" on DataCamp.