UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA
GIOVANNI DEGLI ANTONI

ALGORITHM FOR MASSIVE DATASET
PROJECT REPORT

Bianchi Davide
12820A

ACADEMIC YEAR 2023-2024

Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

Indice

1 Introduction

The goal of this project is to implement a **similar movie pair detector** from scratch. Each movie will be represented by aggregating information from various files in the dataset in a meaningful and non-trivial way. The process involves several key steps:

- **Dataset Exploration** – Analyzing and investigating the dataset.
- **The Inverted Index Structure** – Building an efficient indexing structure for token retrieval.
- **TF-IDF Measure** – Computing term frequency-inverse document frequency for feature representation.
- **Cosine Distance** – Implement and measure the similarity between movie representations.
- **Locality Sensitive Hashing with Cosine Similarity** – Constructing hash table for efficiency movie comparison.
- **Similarity Detection on the Movie Dataset** – Identifying and linking similar movie pairs in the massive dataset.
- **Experiments and Evaluation** - Evaluate the performance of the algorithm on a large-scale dataset.

In this project, Python 3.11 version is used along with the Apache Spark API 3.5.4 version, a powerful framework designed for processing Big Data distributed across multiple clusters.

2 Dataset Preparation

2.1 Dataset Information

The dataset used in this project is the Letterboxd Movies Dataset, which contains various information about films. It is composed by the following `csv` files along with their respective attributes:

- `actors.csv`: Id, Name, Role
- `countries.csv`: Id, Country
- `crew.csv`: Id, Role, Name
- `genres.csv`: Id, Genre
- `languages.csv`: Id, Type, Language
- `movies.csv`: Id, Name, Date, Tagline, Description, Minute, Rating
- `posters.csv`: Id, Link
- `releases.csv`: Id, Country, Date, Type, Rating
- `studios.csv`: Id, Studio
- `themes.csv`: Id, Theme

Additionally, there is a directory containing the image of the posters of the films not used in this project.

2.2 Dataset Sampling

To reduce the overall running time, the dataset is subsampled. This process involves selecting only 1% of the records from the `movies.csv` file and filtering the other files based on the selected movie IDs. The data are then organized in two directories:

- Full dataset: *dataset/full_dataset*
- Subsampled dataset: *dataset/small_dataset*

The algorithms can be run on both the subsampled dataset and the full dataset.

2.3 Dataset Preprocessing

In this section, all `csv` files are loaded into the distributed dataset. Each record in the Resilient Distributed Dataset (RDD) is represented as tuple containing the movieID and each attributes of the record.

For each file, the preprocessing steps are as follows:

- Load the `csv` file.
- Parse each record and ensure it follows the expected data pattern.
- Remove any quotation marks from the records.
- Store the processed data in the format: `[`('movieID', 'attribute 1', 'attribute 2', ...]. If a record does not follow a specific pattern it's discarded.

After this initial preprocessing, the documents are represented using the *Bag of Words* model. This model encodes each document in terms of a set of words, or more precisely, tokens, that appear in it. The preprocessing steps for each document in the RDD are as follows:

- Retrieve the document from the RDD.
- For each record `(movieId, string)`, split the string in tokens and remove any stop words encountered.
- Store the result as a new record in the format: `(movieId, [list of tokens])`.

Once these preprocessing steps are completed, the dataset is ready for use by the algorithms described in Section 4.

3 The Inverted Index structure

An inverted index is a data structure commonly used to efficiently search and retrieve documents that contain a specific token (or word). It maps tokens (terms) to the list of documents or records that contain them, rather than mapping documents to the tokens they contain.

With this structure it is easy to quickly retrieve documents that contain a specific token without having to look through every document individually. In this project this structure is implemented by the *invert* function. This function, given a pair `(MovieId, token list)`, returns a list of pairs `(token, MovieId)`. A *swap* function is also used to swap the position of the key and its associated values. This function, given a pair `(token, (MovieId1, MovieId2))` returns a pair `((MovieId1, MovieId2), token)`. This two functions are used to create the final structure from movie comparison.

4 Algorithms

This section presents the key algorithms used to identify similar movies. The first is the Term Frequency-Inverse Document Frequency (TF-IDF) model, which assigns varying importance to words based on their occurrence within individual documents and across the entire dataset.
Next, the Cosine Distance algorithm is presented, which quantifies the similarity between two documents using their TF-IDF representations.

4.1 Term Frequency - Inverse Document Frequency (TF-IDF)

To improve the *Bag of Words* technique, it is important to assign different weights to the tokens contained in a document to reflect which terms are more relevant. A heuristic that helps identify these terms is the TF.IDF measure (Term Frequency times Inverse Document Frequency), obtained by multiplying two indicators defined below.

Term Frequency
The Term Frequency gives more weight to tokens that tend to appear multiple times in the same document. It is calculated as the relative frequency of the token in the document. In other words, if the document d contains 100 tokens and among them the token t appears five times, the Term Frequency of t in d is $TF(t, d) = \frac{5}{100} = 0.05$. The term frequency is implemented by the function *tf*.

Inverse Document Frequency
The Inverse Document Frequency assigns a high weight to tokens that rarely occur in a dataset. Given a token t and a set of documents U, the Inverse Document Frequency of t is equal to $IDF(t) = ln(\frac{N}{n(t)})$, where N is the total number of documents in U and $n(t)$ indicates the number of documents in U that contain t. The logarithm is applied in the Inverse Document Frequency (IDF) formula to scale the values and prevent extremely

large weights for rare words. The inverse document frequency is implemented by the function *idfs*.

TF-IDF
The TF-IDF measure of a token t in a document d is the product of the Term Frequency and the Inverse Document Frequency: $TF.IDF(t,d) = TF(t,d) \cdot IDF(t)$. A high TF.IDF value indicates a token that frequently occurs in a document but rarely in others. The term frequency - inverse document frequency is implemented by the function *tfidf*.

4.2 Cosine Similarity

The measure of similarity between documents used in this project is the cosine similarity, which interprets two objects as directions in a space and calculates the cosine of the angle formed by these directions. When the angle between two vectors is small the similarity is high and vice versa.

The documents are encoded as directions in space, and therefore as vectors. Each possible token in our corpus represents a dimension: a generic document will have as the component in the dimension corresponding to a token the corresponding value of the TF.IDF measure.

4.2.1 Cosine Similarity Formula

The similarity between two documents can be measured by computing the cosine of the angle between the two directions in the space. This can be easily done by recalling that $a \cdot b = \|a\|\|b\| \cos\theta$, where $a \cdot b$ denotes the dot product between two vectors a and b, θ is the angle between these vectors, and $\|a\|$ denotes the norm of a.
Therefore:

$$\text{sim}(a,b) = \cos\theta = \frac{a \cdot b}{\|a\|\|b\|} = \frac{\sum a_i b_i}{\sqrt{\sum a_i^2}\sqrt{\sum b_i^2}}$$

It should be noted that, although this approach involves considering a large number of dimensions, the vectors can be stored in a dictionary containing only the non-zero components, taking advantage of the sparsity property. More precisely, for each token t with a non-zero TF.IDF value $i \neq 0$, the key t in such a dictionary will be associated with the value i. The cosine similarity is implemented by the following functions: *dotprod*, *norm*, *cossim*, *cosine_similarity*.

4.2.2 Fast Cosine Similarity

In the project a faster cosine similarity has been implemented. This function use precomputed norms and tf-idf weight saved in a broadcast variable. This reduces communication costs between nodes and improves the performance because each worker has local access to the data.

5 Locality Sensitive Hashing for Cosine Similarity

Finding similar movies efficiently in a large dataset can be computationally expensive if every pair needs to be compared. To address this challenge, the **Locality-Sensitive Hashing (LSH)** is used. It is a technique that approximates nearest neighbor search by ensuring that similar items are mapped to the same bucket with high probability. LSH is particularly useful for **cosine similarity**, where the goal is to identify vectors (representing movies) that have small angular distances. How LSH works for Cosine Similarity:

- **Generate Random Hyperplanes**:
  - Create multiple random hyperplanes that pass through the origin in the vector space.
  - Each hyperplane acts as a separator, dividing the space into two halves.
  - The function used to create the hyperplanes is: *generate_random_hyperplanes*.
- **Hashing Movie Vectors**:
  - Each movie is represented as a high-dimensional sparse vector (e.g., TF-IDF features).
  - The vector is projected onto each hyperplane:
    - If the projection is positive, assign 1.
    - If the projection is negative, assign 0.
  - This results in a unique binary hash for each vector.
  - The function used to compute the binary hashes is: *compute_hashes*; instead the function used to create a tf.idf sparse vector is *to_sparse_vector*
- **Constructing the Hash Table**:
  - Convert the binary hash into an integer to use as a bucket index.

- Store the movie IDs in a hash table, where each bucket contains movies with the same hash values.
- The function used to convert binary hash values into integers is: *bin_to_hash*

Advantages: Instead of comparing all pairs, only movies within the same bucket are compared using the cosine similarity, significantly reducing the number of required distance computations.

Sparse Vector Structure: each movie is represented by its TF-IDF values. These values are stored in a Compressed Sparse Row (CSR) vectors for each film. This structure is efficient for high dimensional data since it store only non zero values. The CSR matrix consists of three main array:

- Data array: contains the non-zero value
- Column indices: store the column indices to the values in data
- Row counts: cumulative count of non-zero values per row

Hash table size: the size of the hash table, meaning the number of available buckets, depends on the number of hash functions (hyperplanes) used. In particular, if b hyperplanes are used, the number of possible buckets is 2^b.
A small b results in fewer buckets, causing more movies to collide and increasing the rate of false positives (movies with low similarity are hashed in the same bucket). On the other hand, a large b creates many buckets, but only a subset will be populated and fewer movies with high similarity will be grouped together in the same bucket. Choosing the appropriate hash table size depends on the dimension of the dataset and the number of features. For large datasets with a lot of features, a higher number of buckets helps minimize collisions and improve precision, while for smaller datasets, fewer buckets are needed to avoid sparsity.

6 Similarity Detection on the Movie Dataset

Similarity detection refers to the process of identifying how similar two or more items (such as objects, texts, images, or data records) are to each other. In this project, the goal is to identify pairs of similar movies by leveraging the functions defined in the previous sections. The key steps in the similarity detection process are outlined below:

- **1. Dataset preparation**

- **1.1. Load the csv Files**
  Load the `csv` files (like actors.csv, genres.csv, etc.) into the distributed system. The format of each record is the following: `('MovieId', 'Attributes')` and they are saved in a Resilient Distributed Dataset (RDD).
- **1.2. Convert the Data Format**
  Each map task convert each data entry from `('MovieId', 'Attributes')` format to `('MovieId', [token list])` format. This involves processing the relevant fields (e.g., title, description, genres, etc.) to create a tokenized list for each movie.
- **1.3. Create a Combined Dataset**
  Combine the relevant RDDs (e.g., movies, genres, actors, etc.) into a single RDD. This resulting dataset will facilitate efficient processing and comparison of movie records. The data format is the same: `('MovieId', [token list])`

- **2. Compute the TF-IDF values**
  - **2.1. Compute Inverse Document Frequency (IDF) Values**
    - Calculate the Inverse Document Frequency (IDF) values for the entire Combined Dataset. The IDF is used to weigh terms based on their frequency across documents, helping to highlight important (less common) words.
    - Store the computed IDF values as a broadcast variable. Broadcast variables in Spark are sent to the workers only once and stored locally for quick access, optimizing the computation process.
    - The data format is: `{'token1': idfsToken1, 'token2':idfsToken2, ...}`
  - **2.2. Apply TF-IDF Transformation**
    - Use the previously computed IDF values to perform a Term Frequency-Inverse Document Frequency (TF-IDF) transformation. This process generates an RDD that maps each `MovieId` to a dictionary of tokens, where each token is associated with its corresponding TF-IDF measure. The TF-IDF measure helps to evaluate the importance of each token in relation to the Combined Dataset.
    - The data format is: `[('MovieId1', {'token1':idfsToken1, 'token2':idfsToken2, ...}), ..., ('MovieId2', {'token1':idfsToken1, 'token2':idfsToken2, ...})]`

- **3. Apply the Locality-Sensitive-Hashing technique**
  - **3.1. Compute the sparse TF-IDF Matrix**: the TF-IDF dataset computed in 2.2 is converted into a Compressed Sparse Row (CSR) matrix used to

compute the hashes. More specifically, each record is transformed by the map task in a csr vector.

- **3.2. Compute the hyperplanes**: the random hyperplanes matrix is computed and broadcasted to each worked node in the system for efficient computations.
- **3.3. Compute the hash values**: for each movie vector the hash value is computed and converted from a binary number to an integer.
- **3.4. Create the Hash Table**: after computing hash values for all movie vectors, they are grouped together into buckets where each bucket correspond to a unique hash value. The structure is: bucket_number -> [movie_id_list]. Movies in the same bucket are likely to be similar since they have been hashed to the same index based on their feature vector.

- **4. Create the final structure for Movie Comparisons**
  - **4.1. Filter Movies**
    - Select a bucket from the hash table and retrieve the list of associated movieIds
    - Use this movieId values to filter from the tf-idf RDD a specific set of movies and their tf-idf data.
  - **4.2. Invert the TF-IDF Weights**
    - Create an inverted index structure, which maps each token to the list of movie IDs containing that token. This index allows for fast retrieval of movies that share common tokens.
    - The RDD computed at point 2.2 is inverted and saved in the new inverted RDD.
    - The value of the TF-IDF are not considered. The data format is: `[('token1', 'MovieId1'), ('token2', 'MovieId1'), ..., ('tokenN', 'MovieIdN')]`
  - **4.3. Identify Common Tokens Between Combined Dataset Records**
    - Create a new RDD that includes only tokens shared between movies. Each element in this RDD is a pair where the key is a token, and the value is a `(MovieId1, MovieId2)` pair.
    - Swap the elements in each pair, so that the key becomes the `(MovieId1, MovieId2)` pair, and the value is the common token.
  - **4.4. Avoid Record with the Same Key**
    - Filter and remove the records that have the same film ids: eg. ('1010', '1010')

    * Sort the same key with switched movieIds: eg. ('1111', '0000') = ('0000', '1111') and remove the duplicates with distinct().
    * `groupByKey` the elements in the dataset and the final RDD every record maps each (MovieId1, MovieId2) to a list of common tokens. The data format is: `(('MovieId1', 'MovieId2'), [token list])`

- **5. Compute the Cosine Similarity using precomputed TF-IDF and Norms**
  - **5.1. Compute Vector Norms**
    * Generate an additional RDD that maps each `MovieId` to the norm of its corresponding TF-IDF vector.
    * Convert this RDD into dictionaries and store them as broadcast variables allowing fast access to the norms during similarity calculations.
    * The data format is: `[('MovieId1', normMovieId1),('MovieId2', normMovieId2), ...]`
  - **5.2. Compute the Fast Cosine Similarity**
    * The Fast Cosine Similarity is computed for each record (pair of movies) in the Common Tokens Dataset based on their common tokens.
    * Each map task takes in input the key pairs, the list of common tokens, the norms and the tf-idf weights for each record.
    * This function is more efficient because the tf-idf weights and the norm are stored in two broadcast variables. This reduces communication costs between nodes and improves the performance because each worker has local access to the data.
    * The result is an RDD that contains the cosine similarity scores. The data format is: `(('MovieId1', 'MovieId2'), Cosine Similarity)`

7 Experiments and Evaluation

In the previous section a general method to detect similarities between movies has been presented. In this section the goal is to aggregate in different ways the informations contained in the various files in the dataset and compute their similarities. The number of bits used for the hash values is set to 5, resulting in 32 distinct bucket numbers. This choice is made to maintain a low running time for the algorithm. The test procedure follows the steps presented in the previous section. For each section, a plot is provided, displaying the average similarities between movies for each bucket. The buckets with

the highest cosine similarity are selected and their content is inspected to see: the pair of similar movies, their similarity value and which list of tokens they have in common. The result are then verified inspecting the csv files from the dataset.

7.1 Test1: similarity based on Movie Genre

In this section, the similarity between movies is done comparing the Movie Genres. After constructing the hash table, the average cosine similarity between movies in the same bucket is computed and the result is shown in the plot 1.
This plot represents the average cosine similarity scores between movies assigned to the same bucket of the hash table. The x-axis shows the bucket number and the y-axis shows the cosine similarity values.

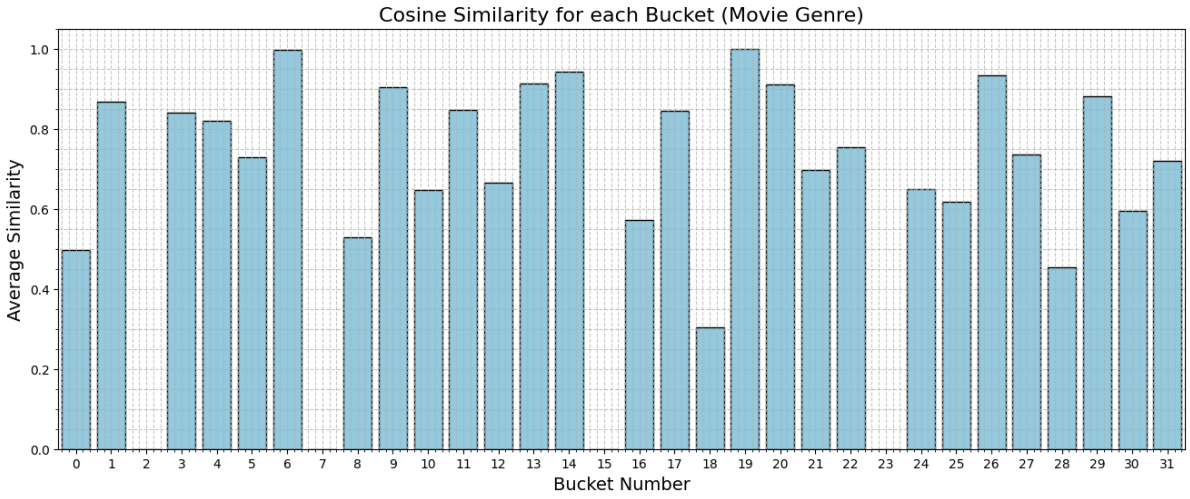

Figura 1: Comparing Movie Genre

It is shown that:

- The average similarity values of most of the movies are above 0.6 meaning that using only the genre attribute, the hashing technique effectively groups similar movies together.
- There are some buckets with lower similarity values, indicating that certain that some movies do not fit into a single group because they are identified by multiple genres.
- A few buckets reach similarity values close to 1, which suggests that movies in these buckets share very strong genre-based relationships.

In the Jupyter notebook, the movies grouped in the highest similarity bucket are retrieved and the similarity between them is computed. Specifically, the following details are printed: Movie Name 1, Movie name 2, Genre Movie 1, Genre Movie 2, Common Tokens and Similarity. In this test, the similarity among movies in the same bucket is high because most films share a significant number of tokens, and the number of distinct genres is relatively small. The inspected bucket has an average cosine similarity of 1, indicating that all movies within this group share the exact same genres. With a high cosine similarity, only a few movies are expected to be in that bucket. But since the movie genres are limited, there ends up being a lot of movies in the bucket. In the next tests, additional attributes will be incorporated into the dataset to improve differentiation between movies based on more than just their genre.

7.2 Test2: similarity based on Movie Genre and Theme

In this test, more informations are aggregated to the dataset. The similarity between movies is done comparing the Movie Genre and Theme. After constructing the hash table, the average cosine similarity between movies in the same bucket is computed and the result is shown in the plot 2. This plot represents the average cosine similarity scores between movies assigned to the same bucket of the hash table. The x-axis shows the bucket number and the y-axis shows the cosine similarity values.

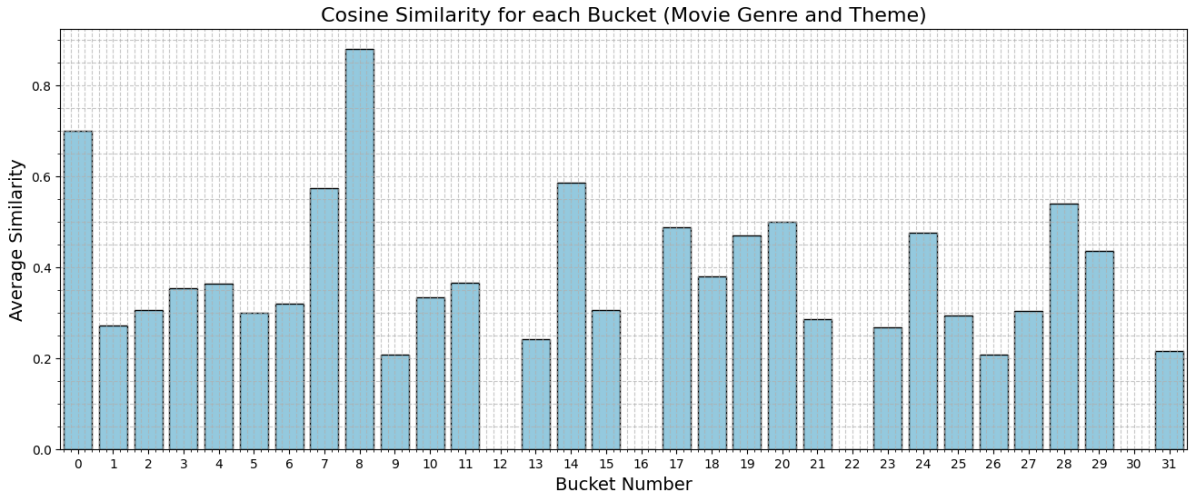

Figura 2: Comparing Movie Genre and Theme

It is shown that:

- The average similarity values of the list of movies belonging to the same buckets is decreased from the last test because the Movie Theme attribute has been added to the dataset.
- There are some buckets with lower similarity values, but two buckets have a similarity value higher than 0.65. This suggest that movies in these buckets share very strong genre-themes relationships.

In the Jupyter notebook, the movies grouped in the highest similarity bucket are retrieved and the similarity between them is computed. Specifically the following details are printed: Movie Name 1, Movie name 2, Tokens Movie 1, Tokens Movie 2, Common Tokens and Similarity. In this test, the inspected bucket has an average cosine similarity of 0.88 but contains only three movies (compared to over 50 in the first test). This indicates that these movies are strongly connected, sharing most of their theme and genre tokens, but only a few of them exhibit extremely high similarity.

7.3 Test3: similarity based on Movie Name, Genre and Theme

In this test, more informations are aggregated to the dataset. The similarity between movies is done comparing the Movie Name, Genre and Theme. After constructing the hash table, the average cosine similarity between movies in the same bucket is computed and the result is shown in the plot 3. This plot represents the average cosine similarity scores between movies assigned to the same bucket of the hash table. The x-axis shows the bucket number and the y-axis shows the cosine similarity values.

It is shown that:

- The average similarity values of the list of movies belonging to the same buckets is decreased from the last test because the Movie Name attribute has been added to the dataset.
- There are some buckets with lower similarity values, but two buckets have a similarity value higher than 0.6. This suggest that movies in these buckets share very strong relationships in terms of genre and themes.

In the Jupyter notebook, the movies grouped in the highest similarity bucket are retrieved and the similarity between them is computed. Specifically the following details are printed: Movie Name 1, Movie name 2, Tokens Movie 1, Tokens Movie 2, Common Tokens and Similarity. In this test, the inspected bucket has an average cosine similarity of 0.61 but contains only three movies. This indicates that these movies are strongly

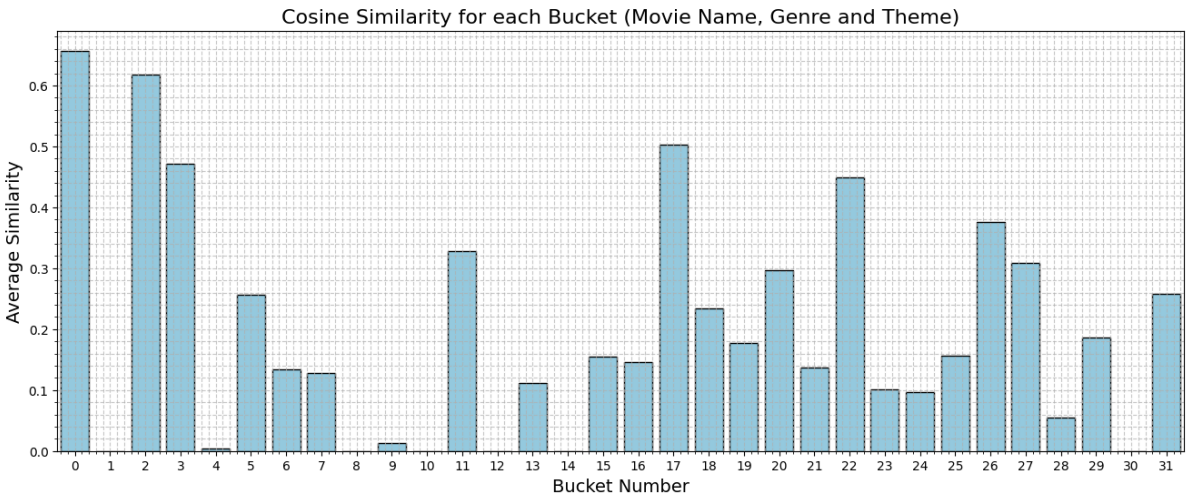

Figura 3: Comparing Movie Name, Genre and Theme

connected and share most of their theme and genre tokens. Additionally, none of the common tokens include the movie title, demonstrating that adding an attribute with a unique value for most movies does not impact the similarity score.

7.4 Test4: similarity based on Movie Name, Genre, Theme, Description

In this test, more informations are aggregated to the dataset. The similarity between movies is done comparing the Movie Name, Genre, Theme and Description. Also in this case, 5 bits are used for the hash values resulting in 32 buckets. After constructing the hash table, the average cosine similarity between movies in the same bucket is computed and the result is shown in the plot 4. This plot represents the average cosine similarity scores between movies assigned to the same bucket of the hash table. The x-axis shows the bucket number and the y-axis shows the cosine similarity values.

It is shown that:

- Most of the average similarity values for movies within the same bucket are below 0.1. This indicates that adding the movie description makes it more difficult to achieve a high average similarity for each record.
- However, one bucket has a similarity value greater than 0.25, suggesting that the movies in this bucket share very strong connections in terms of genre, themes, and

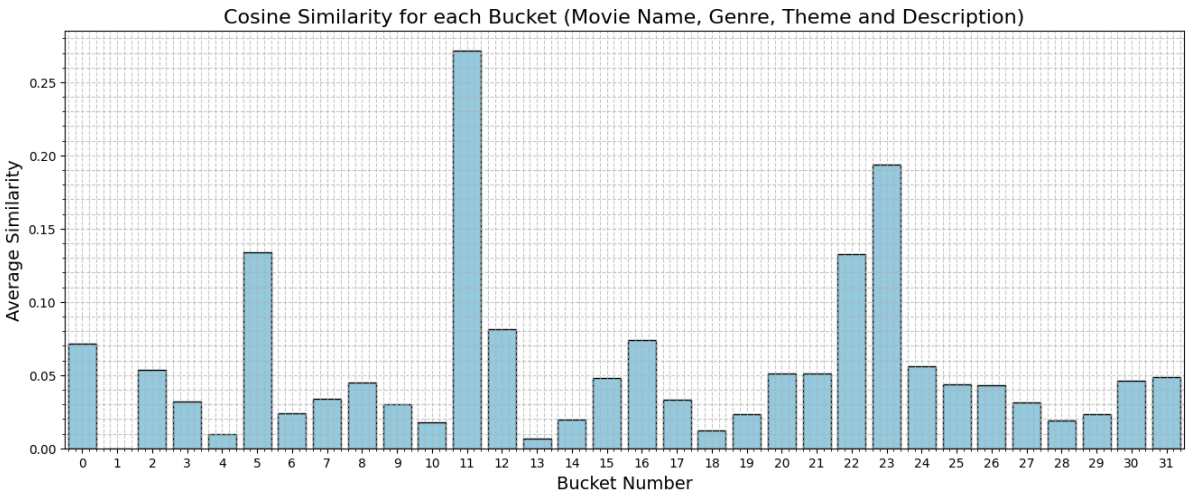

Figura 4: Comparing Movie Name, Genre, Theme and Description

descriptions.

In the Jupyter notebook, the movies grouped in the highest similarity bucket are retrieved and the similarity between them is computed. Specifically the following details are printed: Movie Name 1, Movie name 2, Tokens Movie 1, Tokens Movie 2, Common Tokens and Similarity. In this test, the inspected bucket has an average cosine similarity of 0.27 but contains only two movies. This indicates that these movies are strongly connected and share most of their theme, genre, and description tokens. The most influential tokens in determining similarity are still the genre and theme tokens, as they contain the most relevant shared information across all movies.

8 Final Considerations

In this project, various techniques for measuring the similarity between movies using their attributes have been explored. The use of Term Frequency-Inverse Document Frequency (TF-IDF) for transforming textual data into numerical representations, followed by the application of Cosine Similarity to compute the movie similarity. The Locality Sensitive Hashing (LSH) was also implemented to improve the efficiency of similarity detection minimizing the number of comparisons.

Through the series of tests, the similarity measurements based on different combinations of movie attributes (Movie Name, Genre, Theme and Description) showed that higher similarity values were often found in movies sharing only Genre attributes. Adding

new informations at the dataset for each test causes the average similarities among movies grouped in the same bucket to decrease. This was due to the fact that the vector representation of each movie became more detailed, making it harder to find exact matches. The movies that have high similarity values in the last test indicates that there is a strong semantic connection across all the attributes but lose generalization.

In conclusion, increasing the number of features in each test improves accuracy by identifying only highly similar movies. However, it may reduce generalization, as movies that are related but not extremely similar may no longer be compared.