MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
V.N. Karazin Kharkiv National University
School of Mathematics and Computer Science
Department of Theoretical and Applied Informatics

Master's Thesis

Development of Movie Recommendation System Using Clustering Methods

Author:
Final year Master's Program student,
group MCS-63
specialty - Computer Sciences and
Information Technologies,
educational program: "Informatics" LiYanqiu
Supervisor: Artem Panchenko, PhD
Reviewer: PhD, Associate Professor of Mathematical Modelling and Artificial Intelligence Department , National Aerospace University "Kharkiv Aviation Institute" Dmytro Chumachenko

Kharkiv, 2024

Contents

1. Introduction

In the current era characterized by an explosion of information and a thriving film industry, viewers are confronted with an overwhelming variety of movie options.In order to effectively assist users in selecting films that align with their personal tastes and preferences from this vast selection, movie recommendation systems have emerged. These systems analyze multidimensional data, including users' past viewing behaviors, preferences, ratings, and social network information, with the goal of providing personalized movie recommendation services. However, as the user base expands and the movie library continuously updates, how to further enhance the personalization and accuracy of recommendation systems has become an important research topic.

Clustering technology, as an effective data analysis method, plays a pivotal role in movie recommendation systems. By clustering users and movies, the system can more accurately capture users' interests and preferences, as well as the inherent characteristics of movies, thereby providing recommendations that are more tailored to users' needs. This paper aims to explore the application of clustering technology in movie recommendation systems, analyze its significant role in improving personalization and accuracy, and outlook future research directions and development trends. Through in-depth research on clustering technology, it is expected to inject new vitality into the development of movie recommendation systems, bringing viewers richer viewing experiences and higher satisfaction.

1.1. Machine Learning(ML)Overview

Machine learning(ML)is a subset of artificial intelligence that focuses on the development of computational methods to enable computers to learn from and make decisions or predictions based on data[1].The significance of ML lies in its ability to automate the extraction of knowledge from complex datasets,which is particularly valuable in an era where data is being generated at an unprecedented scale.

The history of ML can be traced back to the early days of computer science,with key developments such as the perceptron model in the 1950s and the subsequent evolution of neural networks[2].Over time,ML has expanded to include a variety of techniques such as decision trees,support vector machines,and more recently,deep learning,which has revolutionized the field with its ability to model complex patterns in data[1].The evolution of ML has been driven by advancements in computational power,the availability of large datasets,and theoretical developments in understanding how learning can be achieved through different algorithms.

In terms of applications,ML is pervasive across various domains.In finance,ML models are used for credit scoring and fraud detection by analyzing transaction patterns and customer behavior[3].In healthcare,it aids in diagnosing diseases by identifying patterns in medical images and patient data[4].Autonomous vehicles rely on ML for object detection and decision-making in real-time traffic scenarios[5].These applications demonstrate the transformative impact of ML in enabling intelligent systems that can operate with a high degree of autonomy.

1.2. Recommendation Systems Overview

Recommendation systems are integral to modern e-commerce and content delivery platforms,streamlining the process of finding relevant items amidst a sea of options[6].These systems leverage user data,such as past behavior,preferences,and contextual information,to provide personalized suggestions that enhance user experience and satisfaction[7].

The categorization of recommendation systems into content-based,collaborative filtering,and hybrid methods reflects the different approaches to understanding user preferences.Content-based systems focus on the attributes of items and user profiles to recommend items with similar features[8].Collaborative filtering,on the other hand,relies on the collective intelligence of the user base,identifying patterns in user interactions to suggest items that similar users have liked[9].Hybrid systems attempt to combine the strengths of both approaches to overcome the limitations inherent in each[10].

The entertainment industry,in particular,has seen significant benefits from recommendation systems.By tailoring content to individual tastes,these systems not only improve user engagement but also drive discovery of new movies,music,and shows that might otherwise go unnoticed[11].This personalization is crucial in an industry where user attention and satisfaction directly impact revenue and success.

1.3. Importance of Movie Recommendation Systems

In the digital era, movie recommendation systems have become indispensable tools for both users and content providers. They help users navigate through the overwhelming array of choices available on various streaming platforms, making the process of discovering new movies more efficient and enjoyable [12]. By analyzing user interactions and preferences, these systems can suggest movies that align with individual tastes, increasing the likelihood of user engagement and satisfaction [13].

Traditional movie recommendation systems, however, face several challenges. Data sparsity, where there is insufficient data to make accurate recommendations,

particularly for new users or items, is a common issue [14]. The cold start problem arises when the system cannot provide recommendations for new users or new movies due to the lack of interaction history [15]. Additionally, user preferences are dynamic and can change over time, requiring recommendation systems to adapt and evolve to maintain their relevance and accuracy [16].

To address these challenges, researchers and practitioners are exploring various strategies, including the incorporation of additional data sources, the development of more sophisticated algorithms, and the use of hybrid approaches that combine different types of data and techniques [17]. These efforts aim to enhance the performance of movie recommendation systems, making them more robust and responsive to the needs of users in the ever-changing landscape of digital entertainment.

1.4. Clustering in Recommendation Systems

Clustering as a technique in data mining is fundamentally about partitioning a dataset into homogeneous subgroups or clusters, where the items within each cluster are more similar to each other than to those in other clusters [18]. This approach is particularly powerful in the context of recommendation systems, as it allows for the identification of natural groupings within the data that can be used to improve the personalization and accuracy of recommendations [19].

One of the key benefits of using clustering in recommendation systems is its ability to reduce the complexity of the recommendation task by focusing on a smaller, more relevant subset of items [20]. For example, clustering users based on their preferences can help in identifying similar user groups, which can then be used to make more informed predictions about what items a particular user might like [21]. Similarly, clustering items can help in organizing a vast catalog into more manageable segments, making it easier to match items with user preferences [22].

Moreover, clustering can also enhance recommendation systems by uncovering latent structures in the data that may not be apparent through other means [23]. These latent structures can provide insights into the underlying factors that influence user preferences and item popularity, which can be used to develop more nuanced and effective recommendation algorithms [24]. For instance, clustering can reveal trends and patterns in user behavior that can inform the design of more engaging and targeted content recommendations [25].

Despite its potential, clustering in recommendation systems also presents challenges. Determining the optimal number of clusters is a non-trivial task that can significantly impact the effectiveness of the clustering approach [26]. Additionally, the choice of distance metrics and clustering algorithms can influence the quality of the resulting clusters and, consequently, the performance of the recommendation system [27]. Therefore, it is crucial to carefully consider

these factors when designing and implementing clustering-based recommendation systems.

1.5. Thesis Statement

The primary objective of this research is to develop a movie recommendation system that harnesses the power of clustering methods to overcome the limitations of existing systems. By leveraging clustering techniques, the system aims to enhance the accuracy and personalization of movie recommendations, thereby providing users with a more tailored and satisfying movie-watching experience.

1.5.1. The contributions

Methodological Innovation: Propose a novel approach that combines clustering algorithms with traditional recommendation techniques. This integration is expected to address the data sparsity and cold start problems by grouping similar users and movies, enabling more informed recommendations even for new or less frequently rated items.
Feature Engineering: Conduct an in-depth analysis of the movie dataset to identify and extract relevant features for clustering. This includes handling categorical and textual data, such as movie genres, cast, crew, and plot summaries, through appropriate encoding and transformation techniques. The effective utilization of these features is anticipated to improve the clustering quality and, consequently, the recommendation accuracy.
Performance Evaluation: Implement a comprehensive evaluation framework that employs multiple metrics to assess the performance of the clustering-based recommendation system. By comparing the results with existing methods, the research will provide empirical evidence of the proposed system's superiority in terms of recommendation quality and user satisfaction.

1.5.2. Expected outcomes

A Functional Movie Recommendation System: Develop a fully operational movie recommendation system that can be integrated into existing streaming platforms or movie databases. The system will be capable of generating personalized movie suggestions based on user preferences and behavior, as well as the characteristics of the movies themselves.
Enhanced User Experience: Users of the proposed system will benefit from more accurate and diverse movie recommendations, leading to increased engagement and satisfaction. This, in turn, is expected to improve user retention and loyalty for movie streaming services and content providers.
Insights for Future Research: The findings of this research will contribute to the growing body of knowledge in the field of recommendation systems. By identifying the strengths and weaknesses of clustering-based approaches, the study will provide valuable insights for future research and development in this area, potentially inspiring new algorithms and techniques for improving

recommendation accuracy and personalization.

2. State of the Art and Problem Statement

2.1. Current State of Movie Recommendation Systems

A comprehensive review of the existing literature on movie recommendation systems reveals a diverse landscape of techniques and approaches aimed at enhancing user experience and satisfaction[28].The current trends and technologies used in movie recommendation systems can be broadly categorized into content-based filtering,collaborative filtering,knowledge-based systems,and hybrid approaches[29].

A. Content-Based Filtering

Content-based recommendation systems focus on the attributes of items,such as movie genres,cast,and plot summaries,to recommend movies that align with a user's past preferences[30].These systems build item profiles and use techniques like term-frequency inverse document frequency(tf-idf)scores to find similarities in keywords associated with the items[31].Beyond text analysis,content-based systems also employ Bayesian methods and Artificial Neural Networks to make recommendations[32].

B. Collaborative Filtering

Collaborative filtering(CF)systems,on the other hand,rely on the collective preferences of users to generate recommendations.These systems identify users with similar tastes and suggest items based on the ratings and preferences of those similar users[33].Memory-based collaborative filtering methods,such as user-item filtering and neighborhood-based methods,have been widely used due to their simplicity and effectiveness[34].Model-based collaborative filtering methods,including matrix factorization techniques like Singular Value Decomposition(SVD),have also gained popularity for their ability to handle large-scale datasets and provide more accurate predictions[35].

C. Hybrid Recommendation Systems

Hybrid systems combine the strengths of content-based and collaborative filtering techniques to overcome the limitations inherent in each approach[36].These systems aim to provide more accurate and personalized recommendations by leveraging both item content and user interaction data[37].

D. Analysis of Current Trends and Technologies

Current research in movie recommendation systems is actively exploring the integration of multimodal information,including text,video,and audio,to enhance recommendation accuracy and personalization[38].Additionally,there is a growing interest in the use of knowledge graphs to incorporate emotional and contextual

data into the recommendation process,providing a more nuanced understanding of user preferences[39].Despite the advancements,movie recommendation systems face several challenges,including data sparsity,scalability,cold start issues,and the need for real-time updates to adapt to changing user preferences[40].The literature also highlights the need for more sophisticated evaluation methods and metrics to assess the performance of these systems[41].

2.2. Limitations of Existing Systems

Despite the significant advancements in movie recommendation systems,several limitations persist that hinder their effectiveness and efficiency.This section discusses the gaps and limitations in current approaches,focusing on scalability and accuracy.

2.2.1. Identification of Gaps and Limitations in Current Approaches

A. Cold Start Problem

One of the most significant challenges is the cold start problem,which affects both new users and new movies.New users lack historical interaction data,making it difficult for the system to generate personalized recommendations.Similarly,new movies with limited or no ratings face difficulty in being recommended to users.This issue is particularly pronounced in rapidly evolving domains like entertainment,where new content is continuously introduced[42].

B. Scalability Issues:

As the number of users and items increases,the computational complexity of recommendation systems grows,leading to scalability issues.The increased data volume can overwhelm traditional algorithms,affecting the system's ability to provide timely and accurate recommendations.Scalability is crucial for maintaining performance as the system expands,and it is a common challenge across various recommendation domains[43].

C. Accuracy and Diversity

There is a trade-off between the accuracy and diversity of recommendations.While accuracy is crucial for user satisfaction,diversity is essential for exploring new content.Existing systems often struggle to balance these aspects,potentially leading to a narrow range of recommendations that do not fully cater to the users'evolving interests[44].

D. Dynamic User Preferences

User preferences are dynamic and can change over time.Existing systems may not adapt quickly enough to these changes,resulting in recommendations that may no longer be relevant.This requires systems to continuously learn from new user interactions and update their models accordingly[45].

2.2.2. Discussion on the Scalability and Accuracy of Existing Systems

The scalability of recommendation systems is closely tied to their accuracy.As systems grow to handle more users and items,maintaining high accuracy becomes increasingly challenging.The need for real-time updates to adapt to changing user preferences adds another layer of complexity.Furthermore,the accuracy of recommendations is often measured using metrics such as precision,recall,and F1 score,but these may not fully capture the user experience,particularly in terms of diversity and serendipity.To address these limitations,researchers are exploring various solutions,including the use of more sophisticated algorithms,such as deep learning and reinforcement learning,to improve recommendation quality.Additionally,there is a growing interest in explainable AI to enhance transparency and trust in recommendation systems[46].

2.3. Problem Statement

The research presented in this paper is designed to address the limitations of current movie recommendation systems,focusing on the need for a new approach that utilizes clustering methods to enhance scalability and accuracy.

2.3.1. Clearly Define the Problem the Research Aims to Address

The core problem this research aims to address is the inability of existing movie recommendation systems to effectively manage the vast and growing dataset of user interactions and content,leading to issues such as data sparsity,cold start challenges,and the inability to adapt quickly to changing user preferences.These limitations result in a suboptimal recommendation quality that fails to meet the diverse and dynamic needs of modern users.

2.3.2. Justify the Need for a New Approach Using Clustering Methods

Clustering methods offer a promising avenue for overcoming these limitations.By grouping similar users or items together,clustering can help in reducing the dimensionality of the recommendation problem,making it more manageable and scalable.This approach can also provide insights into natural groupings within the data that traditional recommendation algorithms might overlook,leading to more personalized and diverse recommendations.

Clustering can help in addressing the cold start problem by clustering new users or items with similar profiles,allowing the system to make more informed recommendations based on the behavior of similar groups.This method can also improve the diversity of recommendations by ensuring that less popular but similar items are recommended to users,thus breaking the popularity bias often observed in recommendation systems.

The need for a new approach is further justified by the evolving nature of user preferences and the increasing variety of content available.Clustering methods can

help in dynamically adapting to these changes by continuously updating the clusters based on new data,thus providing a more flexible and responsive recommendation system.

3. Selection of Optimal Tools for Solving the Problem

In the development of a movie recommendation system using clustering methods, the selection of appropriate tools plays a pivotal role. The right set of tools can significantly enhance the system's efficiency, improve the accuracy of clustering, and ultimately lead to better recommendation results and higher user satisfaction. Efficient tools enable faster processing of large movie datasets, accurate clustering ensures that similar movies are grouped together effectively, and this in turn provides users with more relevant movie suggestions, thereby increasing their satisfaction with the recommendation service [47].

3.1. Algorithms

3.1.1. K-Means Clustering

K-Means is a widely-used partitioning-based clustering algorithm.Its objective is to partition a given dataset into K pre-defined clusters. The algorithm initiates by randomly selecting K centroids.Then, it iteratively performs two main steps. In the first step, each data point x_i is assigned to the cluster whose centroid c_j is the closest, typically measured by the Euclidean distance formula

$$d(x_i, c_j) = \sqrt{\sum_{k=1}^{n} \left(x_{ik} - c_{jk}\right)^2}$$

where n represents the number of dimensions of the data, x_{ik} is the k-th dimension value of data point x_i,and c_{jk} is the k-th dimension value of centroid c_j. In the second step, the centroids of each cluster are recalculated as the mean of all the data points within that cluster, that is $c_j = \frac{1}{N_j}\sum_{x_i \in C_j} \quad x_{i_{,where}N_j}$ is the number of data points in clusterC_j.The iteration continues until the centroids no longer change significantly, indicating convergence [48].

In the context of movie recommendation systems, K-Means is suitable for clustering movies based on numerical features such as movie ratings and box office revenues. For example, by clustering movies according to rating intervals, it can separate high-rated, medium-rated, and low-rated movie groups, which helps the recommendation system target different audience preferences and recommend movies accordingly.

3.1.2. Hierarchical Clustering

Hierarchical clustering operates in two main strategies: agglomerative and divisive. The agglomerative approach starts with each data point as an individual

cluster and then successively merges the most similar clusters. The similarity between clusters can be measured in various ways. For instance, in single-linkage, the distance between two clusters C_i and C_j is defined as the minimum distance between any two points from the two clusters, i.e,

$$d(C_i, C_j) = \min_{x \in C_i, y \in C_j} d(x, y)$$

In complete-linkage, it is the maximum distance between any two points of the two clusters, and in average-linkage, it is the average distance between all pairs of points from the two clusters. The divisive approach functions in the reverse manner, initiating with all data points in one cluster and subsequently splitting them incrementally. A dendrogram is frequently employed to visually depict the hierarchical structure of the clustering process, illustrating how clusters are merged or split at each stage [49].

In movie recommendation systems, hierarchical clustering excels at categorizing movie genres in a hierarchical fashion. It can organize movie genres from broad categories such as "Drama" to more specific sub-categories like "Youth Romantic Drama" or "Retro Literary Drama," thereby facilitating the construction of a hierarchical recommendation catalog that satisfies users' needs for exploring preferences from general to specific.

3.1.3. Density-Based Spatial Clustering of Applications with Noise (DBSCAN)

DBSCAN is predicated on the concept of data point density. It defines core points as points that have at least $MinPts$ number of neighboring points within a given radius Eps . Border points are points that reside in the neighborhood of a core point but are not core points themselves, and noise points are those that are neither core points nor in the vicinity of any core point. Commencing from the core points, points that are density-reachable (if there exists a chain of points where the distance between adjacent points is less than $Eps$and all points in the chain are core points, then the first and the last points are density-reachable) constitute clusters. Border points are assigned to the nearest core point's cluster. In this manner, it can identify clusters of diverse shapes without the necessity to pre-specify the number of clusters [50].

In the context of movie data, DBSCAN can effectively identify niche or unpopular movies that form independent clusters (since these movies' features often resemble outliers with low-density distribution in the overall data space), and it can also precisely partition regions of mainstream and popular movies with high density. This is pivotal for mining niche preferences and enriching the diversity of movie recommendations.

3.1.4. Grid-Based Clustering

Grid-based clustering partitions the data space into grid cells. It initially determines the granularity of the grid division (typically by equally dividing each

dimension). Subsequently, it enumerates the number of data points and computes the density and other statistical information for each cell. Adjacent cells with high density are amalgamated to form clusters, and a threshold can be established to filter out valid clusters and disregard low-density noise cells [51].

For large-scale movie datasets, grid-based clustering is efficient. When confronted with a substantial amount of movie attribute data, it can swiftly locate data-dense regions at a coarse-grained level, such as initially screening out the grid regions corresponding to popular movie categories and high-activity rating intervals, thereby establishing the foundation for subsequent fine-grained clustering and accelerating the overall processing.

3.1.5. Model-Based Clustering

Model-based clustering presumes that the data is generated by a specific mathematical model. A frequently employed model is the Gaussian Mixture Model (GMM). GMM utilizes multiple Gaussian distributions to fit the data distribution. The Expectation-Maximization (EM) algorithm is leveraged to iteratively estimate the model parameters, encompassing the weights, means, and covariances of each Gaussian component. Data points are assigned to clusters based on their probabilities of belonging to each Gaussian component, with the point being allocated to the cluster with the highest probability [52].

In movie data exhibiting complex multi-modal distribution characteristics (such as considering the combined style disparities of movies from different eras and regions), GMM clustering can closely correspond to the multi-modal features and precisely delineate clusters of movies with distinct styles. This is suitable for analyzing intricate viewing preferences in a multicultural context.

$$L(\theta|X) = \prod_{n=1}^{N} \sum_{k=1}^{K} \pi_k N(x_n|\mu_k, \Sigma_k)$$

where θ represents the model parameters, π_k is the mixing coefficient, N is the normal distribution, and μ_k and Σ_k are the mean and covariance of the k-th component.

3.1.6. Comparative Analysis of Algorithms

Table 1: Comparison of Clustering Algorithms

Algorithm	Advantages	Disadvantages
K-Means	Simple,fast for large datasets,converges quickly. Good for spherical clusters.	Sensitive to initial centroids,needs preset. Struggles in high dimensions.
Hierarchical	No preset cluster number,reveals data structure.	Computationally costly for big data. Hard to fix optimal cluster number.
DBSCAN	Finds arbitrary shapes,handles noise well.	Sensitive to parameters. Weak in varying density datasets.
Grid-Based	Efficient for large data,easy to implement.	Cluster quality hinges on grid specs. Can't capture complex shapes well.
Model-Based	Uncovers data's hidden structure,suits probabilistic data.	Needs complex parameter estimation,computationally intense.

3.2. Software and Programming Languages

When considering the development of a movie recommendation system utilizing clustering methods, the selection of an appropriate programming language is a critical decision. The languages under consideration include Python, R, C, and Java, each with distinct characteristics that make them suitable for different aspects of the project. The following sections provide an overview of each language and their relevance to the task at hand.

3.2.1. Python

Python is recognized for its readability and versatility, making it a preferred choice for data analysis and machine learning applications. Its extensive library ecosystem, including data science frameworks such as NumPy, Pandas, and Scikit-learn, facilitates complex data manipulation and algorithm implementation. Python's dynamic typing and interpreted nature offer flexibility and rapid development capabilities.

3.2.2. R

R is a programming language and environment specifically designed for statistical computing and graphics. It boasts a rich collection of packages dedicated to statistical analysis and data visualization, positioning it as a strong contender for data-centric projects. R's focus on statistical methodology and its ability to handle complex data structures make it a valuable tool in analytical tasks.

3.2.3. C

C is a procedural programming language known for its efficiency and flexibility. It offers fine-grained control over system resources and is often the go-to language for system programming and performance-critical applications. C's compiled nature ensures faster execution, which can be advantageous for computationally intensive tasks.

3.2.4. Java

Java is an object-oriented programming language that emphasizes portability and maintainability. Its "write once, run anywhere" philosophy, along with its robust standard library, makes Java a popular choice for developing large-scale, cross-platform applications. Java's strong typing and compilation model contribute to the creation of reliable and efficient software.

Based on the evaluation of the aforementioned languages, Python has been selected for the development of the movie recommendation system. Python's strengths in data manipulation, machine learning, and its supportive community make it the most suitable choice for the project. Its ability to handle large datasets and integrate with various data science libraries will be instrumental in the implementation of the clustering-based recommendation system.

3.3. Software Libraries and Feature Engineering Tools

For the development of a movie recommendation system using clustering methods,a selection of software libraries and feature engineering tools is essential.These tools are crucial for data preprocessing,feature extraction,and model development.

3.3.1. Software Libraries:

3.3.1.1.NumPy and Pandas

These foundational libraries enable efficient data manipulation and analysis.They support large,multi-dimensional arrays and matrices of numeric data,along with tools for data manipulation and analysis.

3.3.1.2.Scikit-learn

This comprehensive machine learning library offers a variety of clustering algorithms,model selection techniques,and tools for evaluating results.

3.3.1.3.Matplotlib and Seaborn

Essential for data visualization,these libraries help in understanding complex data structures and presenting findings clearly.

3.3.2. Feature Engineering Tools:

A. Text Processing Libraries(e.g.,NLTK,SpaCy):Vital for handling textual data such as movie descriptions,these libraries facilitate text tokenization,stop word removal,and extraction of meaningful features.

B. Natural Language Processing(NLP)Tools:Tools like Gensim are used for topic modeling,uncovering hidden patterns in movie genres and tags,thus enhancing the feature set for clustering.

3.4. Visualization Tools

Visualization plays a critical role in understanding the outcomes of data clustering and assessing the performance of the recommendation system.

A. Matplotlib and Seaborn:These tools create static,animated,and interactive visualizations of clustering results,providing insights into the distribution and characteristics of formed clusters.

B. Plotly:An interactive graphing library that allows for the creation of complex visualizations,which can be particularly useful for presenting dynamic changes in clustering results over time.

3.5. Data Sources

The data source selected for this study is the TMDB 5000 Movie Dataset from Kaggle.This dataset was chosen due to its comprehensive nature,which includes a wide array of features such as movie titles,genres,cast information,crew details,and user ratings—all essential for developing a robust recommendation system.

The TMDB 5000 Movie Dataset offers a rich set of features that facilitate the application of clustering algorithms to segment the movie data effectively.The selection of this dataset is also influenced by its availability on Kaggle,a reputable platform for sharing and collaborating on datasets within the data science community.

While a more detailed explanation of the data will be provided in subsequent sections,it is important to note that the TMDB 5000 Movie Dataset serves as a solid foundation for the development and evaluation of the proposed movie recommendation system.The dataset's breadth and depth provide a robust basis for clustering and for generating accurate movie recommendations.

4. Implementation Description

4.1. Describe the dataset

4.1.1. Dataset Source and Acquisition

The dataset is sourced from Kaggle(https://www.kaggle.com/datasets/tmdb/tmdb-movie-metadata), a renowned platform for data science competitions and sharing of datasets. It is made publicly available by TMDB (The Movie Database), an extensive movie database that aggregates data from various sources within the film industry. The dataset is provided in a comma-separated values (CSV) format, which is a common and convenient structure for handling tabular data, facilitating seamless import into multiple data analysis and machine learning frameworks. The open-access nature of this dataset on Kaggle allows researchers and developers worldwide to leverage it for diverse applications, fostering innovation and collaboration in the domain of movie analytics and recommendation systems.

4.1.2. Dataset Size

The TMDB 5000 Movie Dataset, as the name implies, comprises approximately 5000 movie entries. Each entry represents a distinct movie, encapsulating a comprehensive set of attributes related to that particular film. While 5000 records might seem relatively modest compared to the entire universe of movies ever produced, it strikes a balance between manageability for computational processing during development and experimentation, and still being large enough to capture meaningful patterns and trends across different movie genres, production eras, and popularity levels.

The dataset is divided into two main CSV files: “tmdb_5000_movies.csv” with a size of 5.7MB and “tmdb_5000_credits.csv” sized at 40MB. The “tmdb_5000_movies.csv” file contains the core movie information such as titles, genres, release dates, vote averages, budgets, revenues, and overviews. The relatively smaller size of this file is due to the fact that most of the fields are either categorical or numerical values that can be efficiently stored. For example, the genre field, although it can have multiple values per movie, is stored in a text format that doesn't take up excessive space. The numerical values like vote average, budget, and revenue are typically represented as floating-point or integer values, which also have a reasonable storage footprint.

The “tmdb_5000_credits.csv” file, on the other hand, is larger as it likely contains more detailed information about the movie credits, such as the cast and crew members. This information is often more text-heavy, with names and potentially short descriptions or roles associated with each person. The larger size indicates that this file contains a significant amount of detailed textual data that could be valuable for further analysis, such as understanding the influence of specific actors

or directors on a movie's success or clustering movies based on the creative talent involved.

Table 2 : Dataset File Sizes

NO.	File Name	SIZE
1	tmdb_5000_movies.csv	5.7MB
2	tmdb_5000_credits.csv	40MB

4.1.3. Dataset Features

The dataset encompasses a rich variety of features, each offering unique insights into the nature and characteristics of the movies it represents.

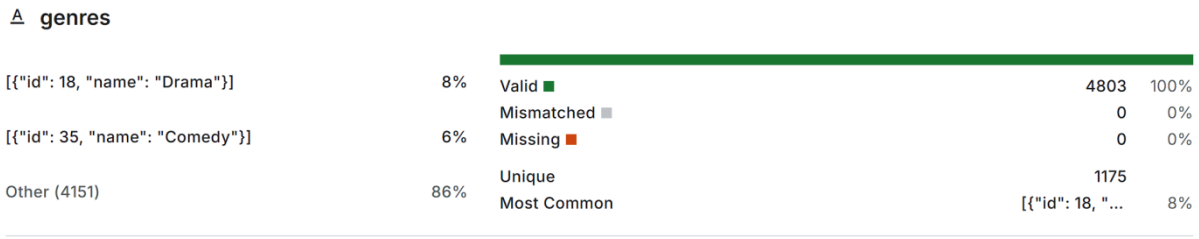
4.1.3.1.Categorical Features

A. Genres

Description: This is a multi-valued categorical feature where each movie can be tagged with one or more genres. It ranges from broad classifications like "Action", "Drama", "Comedy", to more niche ones such as "Film-Noir", "Western", and "Thriller". There are over 20 distinct genre labels present in the dataset.

Suitability for Clustering: Genres are highly relevant for clustering as movies within the same genre often share thematic, stylistic, and audience appeal similarities. For example, in K-Means Clustering, we can one-hot encode the genre feature (if using a simple binary encoding for each genre) to create a binary vector for each movie indicating which genres it belongs to. This encoded vector can then be combined with other features to form the input for the clustering algorithm. In Hierarchical Clustering, genre information can help in determining the initial similarity between movies and guide the agglomerative or divisive process of forming clusters. For DBSCAN, it can contribute to identifying dense regions of movies with common genres. In Grid-Based Clustering, genre can be an additional dimension to consider when partitioning the data space. In Model-Based Clustering, assuming a model that can incorporate categorical variables (like a Bayesian mixture model with appropriate priors for genres), it can help in uncovering latent groups based on genre combinations.

Figure 1 : genres

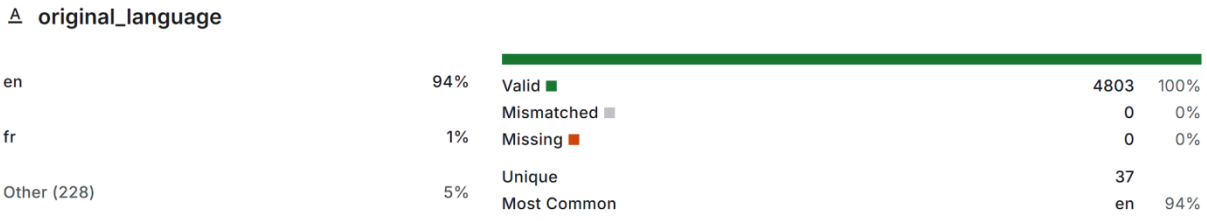


B. Original Language

Description: Denotes the language in which the movie was primarily filmed. English is the most common, accounting for approximately 60% of the movies in the dataset, followed by Spanish, French, and other languages in descending order of frequency.

Suitability for Clustering: Language can be an important factor for clustering as it relates to the target audience and cultural context of the movie. For instance, in K-Means Clustering, we can use label encoding (assigning a unique integer to each language) to incorporate this feature. In Hierarchical Clustering, it can help in creating hierarchical groupings where movies in the same language are grouped first and then further divided based on other features. DBSCAN can identify clusters of movies in the same language if there are dense regions in the data space defined by language and other features. Grid-Based Clustering can partition the data based on language as one of the grid dimensions. In Model-Based Clustering, it can be incorporated as a categorical variable in models like Gaussian Mixture Models with appropriate adaptations to handle it.

Figure 2 : Original Language



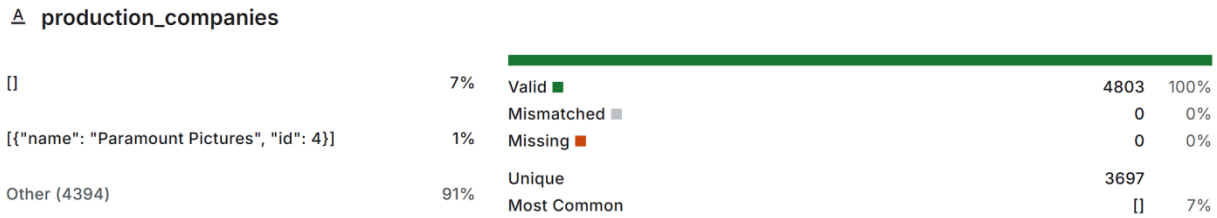
C. Production Companies

Description: Lists the production entities involved in making the movie. There are hundreds of unique production companies, with some well-known studios like Warner Bros., Universal Pictures, and Paramount Pictures being frequently represented. The presence of multiple production companies for a single movie is not uncommon, especially for big-budget blockbusters that often involve co-productions.

Suitability for Clustering: Production companies can indicate certain production qualities, styles, and marketing strategies associated with the movies they produce.

Similar to genres, we can one-hot encode the production companies feature for use in clustering algorithms. For example, in K-Means Clustering, this encoded information can contribute to forming clusters of movies with similar production backgrounds. Hierarchical Clustering can use it to build hierarchical structures based on the relationships between production companies. DBSCAN can detect dense regions corresponding to movies from specific production companies or groups of collaborating companies. Grid-Based Clustering can consider production companies as a dimension when dividing the data space. In Model-Based Clustering, it can be integrated into models to discover latent clusters related to production characteristics.

Figure 3 : C.Production Companies:



D. Homepage

Description: Contains the URL of the movie's official homepage. In the dataset, the data quality shows that around 64% of the entries have valid URLs like "http://www.thewildonesmovie.com/", while the remaining 36% fall into other categories (such as being empty, malformed, or leading to non-existent pages).

Suitability for Clustering: This feature is ill-suited for clustering purposes. URLs are essentially strings that serve as web addresses and do not intrinsically convey information about the movie's content, thematic essence, stylistic traits, or audience appeal. Quantifying and computing similarity based on homepage URLs is challenging. In clustering algorithms like K-Means Clustering, the URL strings would not meaningfully contribute to defining the distance between movies in the feature space. Hierarchical Clustering wouldn't benefit much from this feature either as it wouldn't provide clear cues for initial similarity determination or the hierarchical structuring process. DBSCAN struggles to identify meaningful dense regions using homepage data as the URLs lack a direct connection to the core movie characteristics relevant for clustering. Grid-Based Clustering also finds it hard to partition the data space effectively when using the homepage as a dimension since the URL values don't align with typical clustering-relevant metrics. In Model-Based Clustering, incorporating the homepage as a variable would require complex encoding and transformation that still might not yield useful latent groups related to the movie's substance.

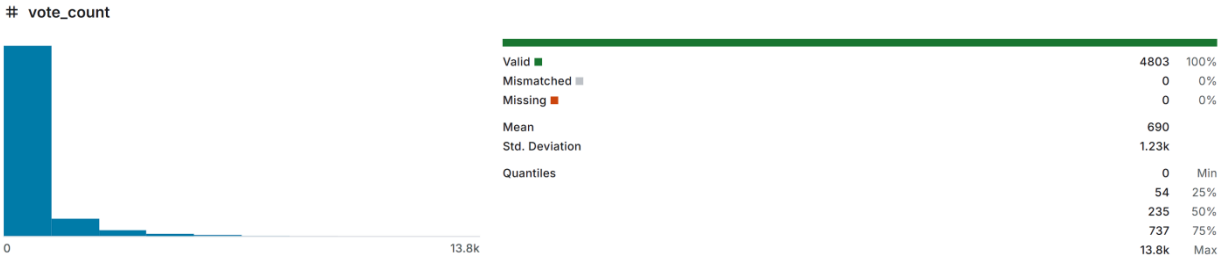
4.1.3.2.Numerical Features

A. Vote Average

Description: Represents the average rating given by users on a scale of 0 to 10. The distribution of vote averages in the dataset is somewhat skewed, with a concentration of movies having ratings between 5 and 7.

Suitability for Clustering: Vote average is a direct measure of user satisfaction and can be used to group movies based on their perceived quality. In all the clustering algorithms, it can be used as a feature directly. For K-Means Clustering, it helps in defining the distance between movies in the feature space (along with other features). In Hierarchical Clustering, it influences the similarity calculation between movies during the clustering process. DBSCAN can identify dense regions of movies with similar vote averages. Grid-Based Clustering can partition the data based on vote average ranges. In Model-Based Clustering, it can be modeled as a continuous variable in appropriate probabilistic models (like in a Gaussian Mixture Model where it can contribute to the likelihood calculation).

Figure 4 : A.Vote Average

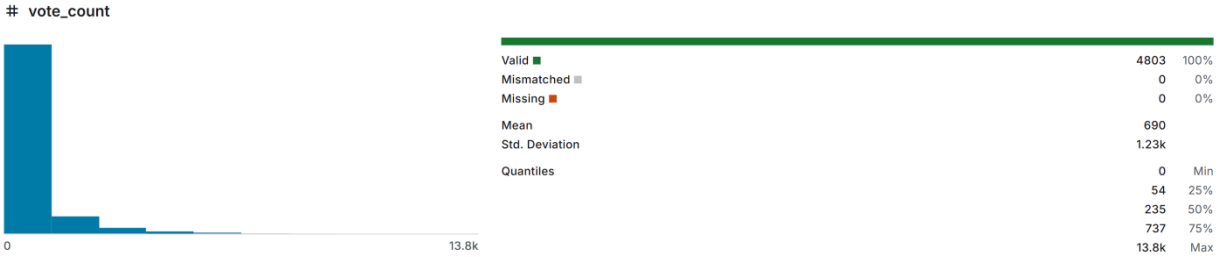


B. Vote Count

Description: Records the number of votes received by each movie. There is a wide variance in vote counts, from movies with only a handful of votes (possibly indie or niche films) to those with thousands of votes (mainstream blockbusters).

Suitability for Clustering: Vote count provides an indication of the popularity or visibility of a movie. Similar to vote average, it can be used as a feature in all clustering algorithms. In K-Means Clustering, it helps in differentiating between widely discussed movies and those with less exposure. Hierarchical Clustering can use it to group movies based on their level of engagement. DBSCAN can detect dense areas of movies with comparable vote counts. Grid-Based Clustering can divide the data space according to vote count intervals. In Model-Based Clustering, it can be incorporated as a variable to model the popularity aspect in probabilistic models.

Figure 5 : Vote Count

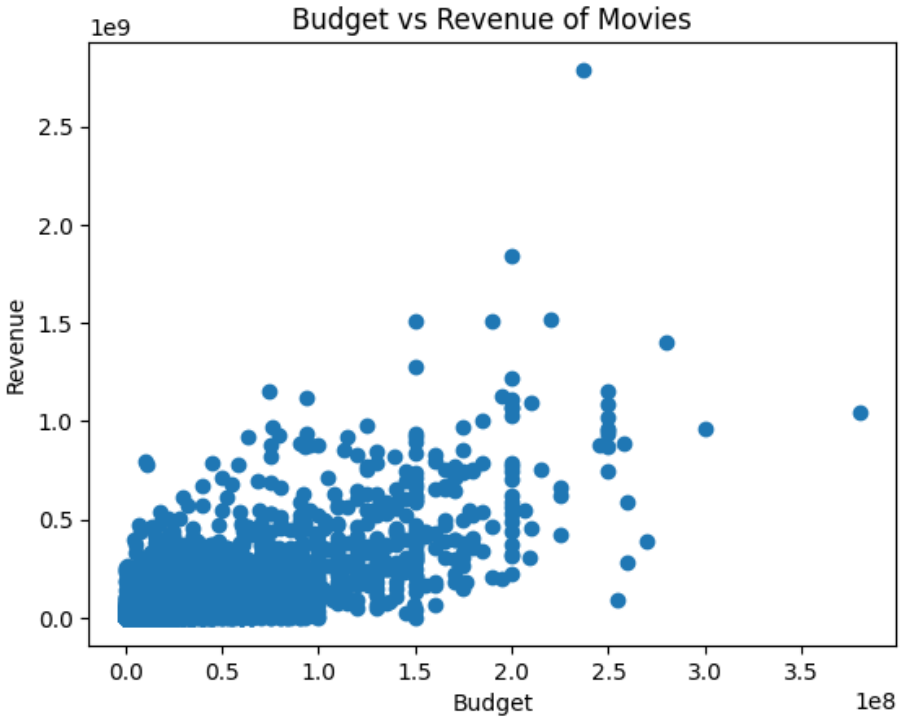


C. Revenue

Description: The Revenue field indicates the total income generated by the movie through different channels like box office sales, home video sales, streaming rights, and merchandise related to the movie.

Suitability for Clustering: Revenue is less suitable for clustering compared to other features. While it does provide information about the financial success of a movie, it is highly influenced by external factors such as marketing efforts, release timing, and competition at the time of release. Two movies with similar content and production qualities may have vastly different revenues due to these externalities. For example, a movie released during a crowded holiday season may face more competition and earn less revenue than if it were released at a different time, even if its inherent quality is comparable. In clustering algorithms, this variability makes it difficult to group movies based on revenue in a meaningful way that reflects their intrinsic characteristics. In K-Means Clustering, the wide variance in revenue values can lead to unstable cluster formation as it may not correlate well with other features. Hierarchical Clustering may not produce meaningful hierarchical structures based on revenue alone due to its lack of consistency in representing movie similarity. DBSCAN may struggle to identify meaningful dense regions as the revenue distribution is often skewed and affected by non-content-related factors. Grid-Based Clustering would also face challenges in partitioning the data space based on revenue as the ranges would be highly variable and not necessarily indicative of movie similarity. In Model-Based Clustering, modeling revenue as a continuous variable may not contribute effectively to capturing the latent groups related to the movie's content and other essential aspects.

Figure 6 : Budget and Revenue



D. Id

Description: Each movie is assigned a unique identifier within the dataset. These identifiers are sequential numbers or alphanumeric codes that serve the purpose of differentiating one movie entry from another in the database context. All the IDs in the dataset are valid, without any cases of missing or mismatched values.

Suitability for Clustering: The movie ID is merely a distinct label for identification and holds no inherent information about the movie's content, style, or any other aspect relevant to clustering. It doesn't play a role in clustering movies based on their characteristics, as clustering aims to group movies with similar content, popularity, or other meaningful traits rather than relying on an arbitrary identification number.

4.1.3.3.Textual Features

A. Overview

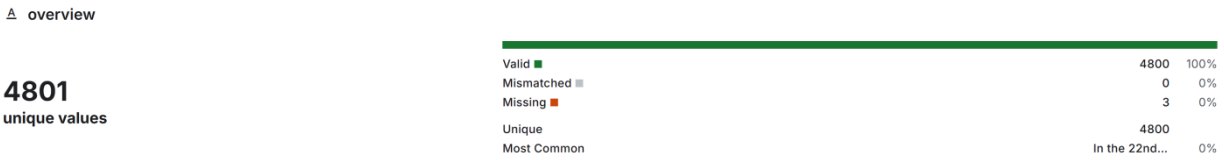
Description: A brief synopsis or description of the movie's plot, typically a few sentences long. This text-based feature holds valuable semantic information that can be exploited through natural language processing techniques.

Suitability for Clustering: The overview can be used to capture the semantic similarity between movies. For instance, by applying word vectorization methods like GloVe or BERT embeddings, the overview text can be transformed into a numerical vector representation. These vectors can then be used in clustering

algorithms. In K-Means Clustering, the distance between these text-based vectors can be calculated along with other features to group movies with similar storylines. Hierarchical Clustering can build a hierarchy based on the semantic similarity of the overviews. DBSCAN can identify dense regions of movies with similar plot themes based on these vectors. Grid-Based Clustering can partition the data space considering the semantic space defined by the overview vectors. In Model-Based Clustering, the text vectors can be incorporated into appropriate probabilistic models to discover latent groups based on story content.

Code Example for Extracting Text Vectors from Overview (using simple bag-of-words approach as an example, more advanced techniques like word embeddings can be used instead)

Figure 7 : Overview

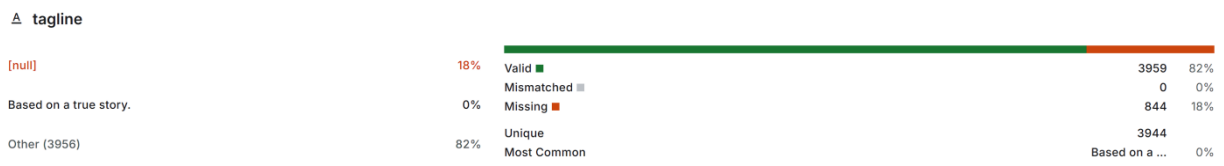


B. Tagline

Description: A catchy, short phrase that encapsulates the essence of the movie, often used for promotional purposes.

Suitability for Clustering: Similar to the overview, the tagline can be processed to extract semantic cues. Although it is shorter than the overview, it can still provide valuable hints about the movie's key aspects. The same techniques used for the overview (like converting to vectors) can be applied. In clustering algorithms, it can contribute to grouping movies with similar promotional themes or key concepts. For example, in K-Means Clustering, the tagline vectors can be combined with other feature vectors to determine cluster membership. Hierarchical Clustering can use it to build hierarchical structures based on the similarity of taglines. DBSCAN can identify dense regions of movies with comparable tagline semantics. Grid-Based Clustering can partition the data space considering the semantic information from taglines. In Model-Based Clustering, tagline vectors can be incorporated into models to discover latent groups based on these short promotional descriptions.

Figure 8 : Tagline

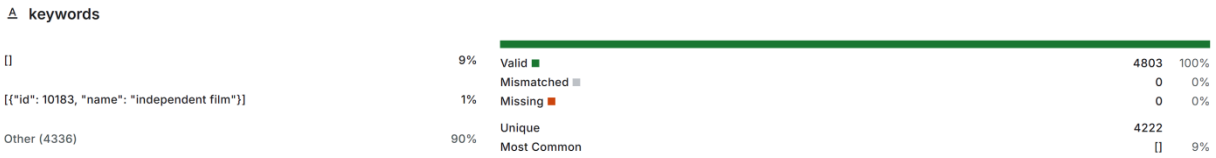


C. Keywords

Description: A set of relevant keywords associated with the movie, which can cover aspects such as the genre, plot elements, characters, or themes. In the dataset, about 9% of the entries have specific keyword sets like ['id': 1038, 'name': 'Independent film'], while around 90% fall into other situations (potentially being empty or having less descriptive keyword collections).

Suitability for Clustering: Keywords can be quite valuable for clustering. For K-Means Clustering, they can be processed (such as through word vectorization similar to the overview) and incorporated into the feature vector to help group movies with related semantic content. Hierarchical Clustering can build hierarchical structures based on the similarity of the keyword sets, clustering movies with overlapping or closely related keywords first and then further subdividing. DBSCAN can identify dense regions of movies sharing common keywords, which might imply shared thematic or genre characteristics. Grid-Based Clustering can partition the data space considering the semantic space defined by the keywords, treating them as an additional dimension for clustering. In Model-Based Clustering, the keyword vectors can be integrated into appropriate probabilistic models to uncover latent groups based on the movie's associated semantic tags.

Figure 9 : Keywords



D. Cast

Description: Comprises the list of actors who play roles in the movie. The cast can range from a few principal actors in smaller productions to a large ensemble in big-budget blockbusters. The reputation, acting styles, and fan bases of the actors can vary widely, influencing the movie's appeal and potential success.

Suitability for Clustering: Cast information can be used for clustering in multiple ways. In K-Means Clustering, one could assign unique identifiers to each actor and then use one-hot encoding or other encoding schemes to represent the presence of particular actors in a movie. This encoded data could then be combined with other features to form the clustering input, potentially grouping movies with similar casts together. Hierarchical Clustering could start by clustering movies based on shared lead actors and then further subdivide based on other characteristics. For DBSCAN, if there are groups of movies with common casts that form a relatively dense region in the data space (in terms of the encoded cast information along with other features), they could be identified as clusters.

However, the complexity of encoding and the fact that many movies have diverse casts means that using cast as a sole or dominant clustering feature may lead to overcomplicated or less meaningful results without proper consideration of other factors.

E. Crew

Description: Encompasses all the personnel involved in the production of the movie, including directors, writers, producers, cinematographers, and more. Each crew member plays a crucial role in shaping the movie's style, narrative, and overall quality. For example, directors with distinct visual or narrative styles (like Christopher Nolan's complex narratives and unique visual palettes) can give a particular identity to their films.

Suitability for Clustering: Crew information can be incorporated into clustering algorithms. In Model-Based Clustering, for instance, the style or reputation of a director could be modeled as a categorical variable (if possible to quantify in some way), and movies directed by similar directors could be grouped together. In Hierarchical Clustering, one could first group movies based on the director or producer and then expand the clustering structure by incorporating other crew members' characteristics and other features like genre and budget. DBSCADN could identify dense regions of movies produced by a particular crew or with similar crew compositions in the data space defined by relevant crew-related metrics along with other features. However, the challenge lies in quantifying the influence and similarity of different crew members in a way that is both accurate and useful for clustering.

In summary, the features that are very suitable for clustering are Genres (movie types), Original Language, Production Companies, Keywords, Cast, and Crew. These features can reflect different aspects of movies such as content, style, language, production background, and the people involved, which are helpful for forming meaningful movie clusters.

Less suitable features include Homepage (as URLs are hard to translate into clustering-relevant data), Id (mere identification numbers), and Original_Title (which doesn't directly convey content traits without elaborate NLP processing). These features are either difficult to quantify and calculate similarity or do not reflect the essential characteristics of movies for clustering.

Title has some potential for clustering when analyzed with advanced NLP techniques, but it's less straightforward compared to many other features. Although Cast and Crew can be used for clustering, due to their complex nature and the need for proper encoding and quantification, they require careful handling

and combination with other features to yield meaningful clustering results.

4.1.4. Data Exploration

To conduct a thorough initial exploration and analysis of the dataset, we will employ Python, leveraging libraries such as Pandas for data manipulation, Matplotlib and Seaborn for visualization, and Scikit-learn for preliminary data analysis. Here are the detailed steps.

4.1.4.1. Importing Libraries and Loading Data

```
import pandas as pd
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
movies_df = pd.read_csv('/data/notebook_files/tmdb_5000_movies.csv')
credits_df = pd.read_csv('/data/notebook_files/tmdb_5000_credits.csv')
```

A. Importing Libraries

- ✓ import pandas as pd: Imports the Pandas library and abbreviates it as pd. Pandas is an open-source Python data analysis library that is highly suitable for quickly performing complex data analysis and data processing. It has been imported twice here, but only one import is necessary.
- ✓ import matplotlib.pyplot as plt: Imports the pyplot module from the Matplotlib library and abbreviates it as plt. Matplotlib is a Python plotting library, and pyplot is its plotting framework, providing a MATLAB-like plotting system. It has also been imported twice, but only one import is needed.
- ✓ import numpy as np: Imports the NumPy library and abbreviates it as np. NumPy is a Python scientific computing library that provides a large number of mathematical function tools, especially useful for large-scale multidimensional array and matrix operations, as well as advanced mathematical function computations.
- ✓ import seaborn as sns: Imports the Seaborn library and abbreviates it as sns. Seaborn is a Python data visualization library based on Matplotlib that provides a high-level interface for creating statistical graphics.
- ✓ from sklearn.feature_extraction.text import TfidfVectorizer: Imports TfidfVectorizer from the scikit-learn library. scikit-learn is an open-source Python machine learning library, and TfidfVectorizer is a tool for converting text data into TF-IDF feature matrices, which is very suitable for text analysis.

✓ from sklearn.metrics.pairwise import cosine_similarity: Imports the cosine_similarity function from the scikit-learn library. This function is used to calculate the cosine similarity between two vectors, which is a method for evaluating the similarity between two texts.

B. Loading Data

✓ movies_df = pd.read_csv('/data/notebook_files/tmdb_5000_movies.csv'): Uses the read_csv function from Pandas to read the CSV file tmdb_5000_movies.csv, which contains information about movies such as titles, descriptions, ratings, etc., and stores the read data in the movies_df DataFrame object.

✓ credits_df = pd.read_csv('/data/notebook_files/tmdb_5000_credits.csv'): Similarly, uses the read_csv function to read another CSV file tmdb_5000_credits.csv, which contains cast and crew information for movies, such as directors, actors, etc., and stores the data in the credits_df DataFrame object.

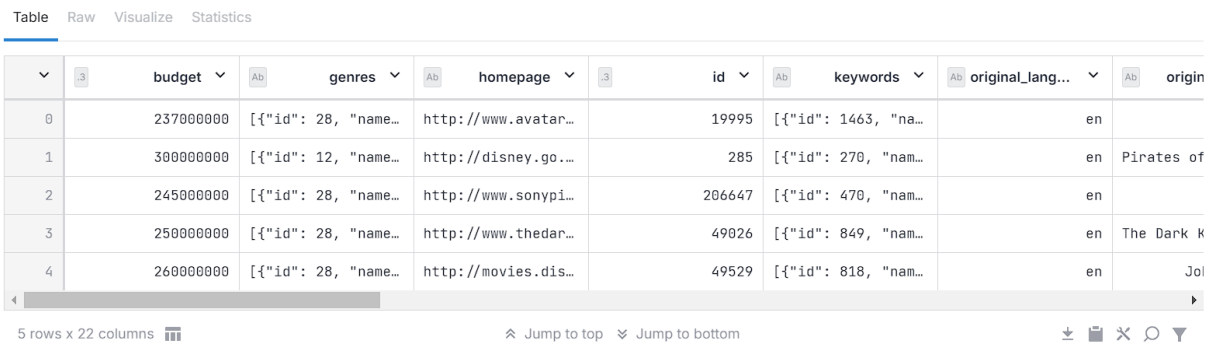
4.1.4.2.Data Overview and Basic Information Review

A. Data Overview

```
merged_data = pd.merge(movies_df, credits_df, left_on='id',
right_on='movie_id', suffixes=('', '_credits'))
merged_data.drop('movie_id', axis=1, inplace=True)
merged_data.drop('title_credits', axis=1, inplace=True)
mad_merged_data = merged_data.head()
mad_merged_data
```

✓ To facilitate subsequent processing, two DataFrames containing movie information were merged (one with basic movie details and the other with cast and crew information). Afterwards, the column that was no longer necessary after the merge 'movie_id', 'title_credits' was deleted. Lastly, the initial few rows of the merged dataset were extracted for viewing or further analysis.

Figure 10 : Output_View the first few rows of data



✓ This will display the first few rows of the merged dataset, allowing us to have a quick look at the data structure. We can see the values of various columns such as movie titles, genres, release dates, vote averages, and more. This gives us an initial understanding of how the data is organized and what kind of information is available for each movie.

B. Shape and Information

```
# Check the shape of the dataset (number of rows and columns)
print(merged_data.shape)

# Examine the data types and the number of non-null values
print(merged_data.info())
```

Figure 11 : Output_Shape and Information

```
(4803, 22)
<class 'pandas.core.frame.DataFrame'>
Int64Index: 4803 entries, 0 to 4802
Data columns (total 22 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   budget                4803 non-null   int64
 1   genres                4803 non-null   object
 2   homepage              1712 non-null   object
 3   id                    4803 non-null   int64
 4   keywords              4803 non-null   object
 5   original_language     4803 non-null   object
 6   original_title        4803 non-null   object
 7   overview              4800 non-null   object
 8   popularity            4803 non-null   float64
 9   production_companies  4803 non-null   object
 10  production_countries  4803 non-null   object
 11  release_date          4802 non-null   object
 12  revenue               4803 non-null   int64
 13  runtime               4801 non-null   float64
 14  spoken_languages      4803 non-null   object
 15  status                4803 non-null   object
 16  tagline               3959 non-null   object
 17  title                 4803 non-null   object
 18  vote_average          4803 non-null   float64
 19  vote_count            4803 non-null   int64
 20  cast                  4803 non-null   object
 21  crew                  4803 non-null   object
dtypes: float64(3), int64(4), object(15)
memory usage: 863.0+ KB
None
```

✓ The first number represents the number of rows, which is the total count of movies in the dataset. The second number represents the number of columns, indicating the different attributes or features used to describe each movie. This information helps us understand the scale of the dataset and plan further analysis.

✓ The first number represents the number of rows, which is the total count of movies in the dataset. The second number represents the number of columns, indicating the different attributes or features used to describe each movie. This information helps us understand the scale of the dataset and plan further analysis.

C. The distribution of features across different categorie

a) Genres

```
# Count the distribution of movie genres
genres_distribution =
merged_data['genres'].value_counts().reset_index()
genres_distribution.columns = ['Genres', 'Count']
display(genres_distribution)
```

Figure 12 : Output_Genres

	Genres	Count
0	[{"id": 18, "name": "Drama"}]	370
1	[{"id": 35, "name": "Comedy"}]	282
2	[{"id": 18, "name": "Drama"}, {"id": 10749, "n...	164
3	[{"id": 35, "name": "Comedy"}, {"id": 10749, "...	144
4	[{"id": 35, "name": "Comedy"}, {"id": 18, "nam...	142
...	...	...
1170	[{"id": 12, "name": "Adventure"}, {"id": 28, "...	1
1171	[{"id": 28, "name": "Action"}, {"id": 14, "nam...	1
1172	[{"id": 878, "name": "Science Fiction"}, {"id"...	1
1173	[{"id": 18, "name": "Drama"}, {"id": 53, "name...	1
1174	[{"id": 35, "name": "Comedy"}, {"id": 18, "nam...	1

1175 rows × 2 columns

✓ This display the frequency of each movie genre in a tabular format. The table will have two columns: 'Genres' which shows the different movie genres, and 'Count' which shows the number of occurrences of each genre in the dataset. We can easily observe which genres are more common and which are less common. This information helps us understand the genre diversity in the dataset and can guide our decisions on how to handle genre information in the clustering and recommendation process. We may consider using encoding techniques like one-hot encoding or frequency-based encoding to represent genres numerically for further analysis.

b) Original Language

```
# Count the distribution of original languages
original_language_distribution =
merged_data['original_language'].value_counts().reset_index()
original_language_distribution.columns = ['Original Language', 'Count']
display(original_language_distribution)
```

Figure 13 : Output_original languages

	Original Language	Count
0	en	4505
1	fr	70
2	es	32
3	zh	27
4	de	27
5	hi	19
6	ja	16
7	it	14
8	cn	12
9	ru	11
10	ko	11
11	pt	9
12	da	7
13	sv	5
14	nl	4
15	fa	4
16	th	3
17	he	3
18	ta	2
19	cs	2
20	ro	2
21	id	2
22	ar	2
23	vi	1
24	sl	1
25	ps	1
26	no	1
27	ky	1
28	hu	1
29	pl	1
30	af	1
31	nb	1
32	tr	1
33	is	1
34	xx	1
35	te	1
36	el	1

✓ The result show the distribution of the original languages of the movies in a table. The table will have columns 'Original Language' and 'Count'. We can quickly identify the dominant language(s) in the dataset. Most likely, English will be the most common original language. This information is useful for understanding the language landscape of the movies and can be incorporated into the analysis. For example, we could explore if movies in the same language have similar characteristics or user preferences. We might also

consider grouping or treating non-English movies differently based on their language frequencies.

c) Production Companies

```
# Count the distribution of production companies
production_companies_distribution =
merged_data['production_companies'].value_counts().reset_index()
production_companies_distribution.columns = ['Production Companies',
'Count']
display(production_companies_distribution)
```

Figure 14 : Output_Production Companies

	Production Companies	Count
0	[]	351
1	[{"name": "Paramount Pictures", "id": 4}]	58
2	[{"name": "Universal Pictures", "id": 33}]	45
3	[{"name": "New Line Cinema", "id": 12}]	38
4	[{"name": "Columbia Pictures", "id": 5}]	37
...	...	...
3692	[{"name": "New Line Cinema", "id": 12}, {"name...	1
3693	[{"name": "WingNut Films", "id": 11}, {"name":...	1
3694	[{"name": "Paramount Pictures", "id": 4}, {"na...	1
3695	[{"name": "Village Roadshow Pictures", "id": 7...	1
3696	[{"name": "rusty bear entertainment", "id": 87...	1

3697 rows × 2 columns

✓ This sprint the frequency of each production company in a tabular format. The table has columns 'Production Companies' and 'Count'. We can spot the major production companies and see how many movies they have produced. This information can provide insights into the production trends and the influence of different companies. We can analyze if movies from the same production company share certain qualities or if there are patterns in user preferences for movies produced by specific companies. In the clustering and recommendation system, we could potentially use this information to group movies based on their production sources or to identify companies that produce similar types of movies.

d) Keywords

```
# Count the distribution of keywords
keywords_distribution =
merged_data['keywords'].value_counts().reset_index()
keywords_distribution.columns = ['Keywords', 'Count']
display(keywords_distribution)
```

Figure 15 : Output_Keywords

	Keywords	Count
0	[]	412
1	[{"id": 10183, "name": "independent film"}]	55
2	[{"id": 187056, "name": "woman director"}]	42
3	[{"id": 179431, "name": "duringcreditsstinger"}]	15
4	[{"id": 6075, "name": "sport"}]	13
...	...	...
4217	[{"id": 4530, "name": "parking garage"}, {"id"...	1
4218	[{"id": 10175, "name": "drug cartel"}, {"id": ...	1
4219	[{"id": 483, "name": "riddle"}, {"id": 588, "n...	1
4220	[{"id": 10525, "name": "broken neck"}, {"id": ...	1
4221	[{"id": 1523, "name": "obsession"}, {"id": 224...	1

4222 rows × 2 columns

✓ The output presents the distribution of keywords in a tabular form with columns 'Keywords' and 'Count'. This allows us to identify the most frequently occurring keywords in the dataset. Keywords can provide valuable insights into the themes, plots, or characteristics of the movies. We can use this information to understand the semantic content of the movies and potentially group movies with similar keywords together in the clustering process. It can also help in identifying patterns in user preferences related to specific themes or elements represented by the keywords.

e) Cast

```
# First, convert the string data in the cast column to a list format
cast_lists = merged_data['cast'].apply(lambda x: eval(x))
# Count the frequency of actors
cast_frequency = {}
for cast_list in cast_lists:
    for actor_dict in cast_list:
        # Use the actor's name as the unique entity
        actor_name = actor_dict.get('name', None)
```

```
        if actor_name is not None:
            if actor_name in cast_frequency:
                cast_frequency[actor_name] += 1
            else:
                cast_frequency[actor_name] = 1
# Convert the actor frequency dictionary to a DataFrame
cast_distribution = pd.DataFrame(list(cast_frequency.items()),
columns=['Actor', 'Count'])
cast_distribution = cast_distribution.sort_values(by='Count',
ascending=False)
display(cast_distribution)
```

Figure 16 : Output_cast_distribution

	Actor	Count
541	Samuel L. Jackson	67
8247	Robert De Niro	57
5423	Bruce Willis	51
4007	Matt Damon	48
205	Morgan Freeman	46
...	...	...
25003	Laramie Eppler	1
25009	Dustin Allen	1
25010	John Howell	1
25011	Kim-Maree Penn	1
54200	Bill D'Elia	1

54201 rows × 2 columns

✓ The output is a table presenting the distribution of actors in the dataset. The table has columns 'Actor' and 'Count', where 'Actor' represents the name of the actor and 'Count' represents the number of movies in which the actor has appeared. By using the actor's name as the unique entity, we can accurately count the frequency of each actor's appearance. This information is crucial for understanding the influence of different actors on the movies. We can identify popular actors who appear in many movies and potentially group movies based on the presence of certain actors. It can also provide insights into the casting trends and the relationship between actors and movie characteristics, which can be useful in the clustering and recommendation process.

f) Crew

```
# Extract the director information
directors = merged_data['crew'].apply(lambda x: [crew['name'] for crew
in eval(x) if crew['job'] == 'Director'])
# Count the frequency of directors
director_frequency = {}
for director_list in directors:
    for director in director_list:
        if director in director_frequency:
            director_frequency[director] += 1
        else:
            director_frequency[director] = 1
# Convert the director frequency dictionary to a DataFrame
director_distribution = pd.DataFrame(list(director_frequency.items()),
columns=['Director', 'Count'])
director_distribution = director_distribution.sort_values(by='Count',
ascending=False)
display(director_distribution)
```

Figure 17 : Output_Crew

	Director	Count
40	Steven Spielberg	27
765	Woody Allen	22
54	Martin Scorsese	21
370	Clint Eastwood	20
381	Robert Rodriguez	17
...	...	...
1353	Wallace Wolodarsky	1
1352	Lionel C. Martin	1
1351	Ernest R. Dickerson	1
1350	Cheryl Dunye	1
2576	Brett Winn	1

2577 rows × 2 columns

The result is a table presenting the distribution of directors in the dataset. The table has columns 'Director' and 'Count', where 'Director' is the name of the director and 'Count' is the number of movies directed by that director. This allows us to identify prolific directors and understand their influence on the movies. We can analyze if movies directed by the same director share certain stylistic or thematic elements

and use this information in the clustering and recommendation system. For example, we could group movies directed by a particular director together or consider the director's style as an important factor in clustering movies with similar creative visions. This can help in providing more personalized and relevant movie recommendations based on the user's preference for a certain director's work.

4.1.5. Data Preprocessing

4.1.5.1. Deleting Irrelevant Columns

Some columns in the dataset might not be useful for our clustering analysis. For example, columns like 'homepage' and'release_date' don't seem relevant to the features we are focusing on for clustering, So, we can remove these columns to simplify our dataset and reduce unnecessary data processing.

```
columns_to_delete = ['homepage', 'release_date','status']
columns_to_delete = [col for col in columns_to_delete if col in
merged_data.columns]
merged_data = merged_data.drop(columns=columns_to_delete)
```

4.1.5.2. Missing Value Handling

```
# Check for missing values
print(merged_data.isnull().sum())
```

Figure 18 : Output_missing values

```
budget                     0
genres                     0
id                         0
keywords                   0
original_language          0
original_title             0
overview                   3
popularity                 0
production_companies       0
production_countries       0
revenue                    0
runtime                    2
spoken_languages           0
tagline                  844
title                      0
vote_average               0
vote_count                 0
cast                       0
crew                       0
dtype: int64
```

Since we've already known the missing value situation before, we'll directly handle it now. For the 'overview' column, as there are only a few missing values (3 in

total), we decide to delete the rows containing these missing values to maintain data integrity and avoid potential biases in the subsequent analysis.

```
# Delete rows with missing values in the overview column
merged_data.dropna(subset=['overview'], inplace=True)
```

4.1.5.3.Text Feature Processing

A. Processing the 'genres' Column

The 'genres' column is currently in a format like [{"id": 18, "name": "Drama"}, {"id": 80, "name": "Crime"}]. We need to convert it into a format more suitable for clustering analysis. Here, we'll use one-hot encoding. First, we extract all the unique genres from the data and then create a genre matrix where each row represents a movie and each column represents a genre. If a movie belongs to a certain genre, the corresponding cell in the matrix will be set to 1; otherwise, it'll be 0.

```
# Extract all the unique movie genres
all_genres = set()
for genres_str in merged_data['genres']:
    if isinstance(genres_str, str):  # Added this check to handle non-
string values like NaN
        genres_list = eval(genres_str)
        for genre in genres_list:
            all_genres.add(genre['name'])

# Create a genre matrix with one-hot encoding
genre_matrix = pd.DataFrame(0, index=merged_data.index,
columns=list(all_genres))
for i, genres_str in enumerate(merged_data['genres'].tolist()):
    if isinstance(genres_str, str):  # Added this check for the same
reason
        genres_list = eval(genres_str)
        for genre in genres_list:
            genre_matrix.at[merged_data.index[i], genre['name']] = 1

# Reset index before merging to ensure consistency
merged_data.reset_index(drop=True, inplace=True)
genre_matrix.reset_index(drop=True, inplace=True)

# Merge the one-hot encoded genre matrix with the original data
merged_data = pd.concat([merged_data, genre_matrix], axis=1)
```

B. Processing the 'keywords' Column:

Extract all the keywords and perform one-hot encoding to create a keyword matrix for better utilization in the clustering process.

```
# Extract all the unique keywords
all_keywords = set()
for keywords_str in merged_data['keywords']:
    if isinstance(keywords_str, str):  # Checks if it is a string
        keywords_list = eval(keywords_str)
        for keyword in keywords_list:
            all_keywords.add(keyword['name'])

# Create a keyword matrix with one-hot encoding
keyword_matrix = pd.DataFrame(0, index=merged_data.index,
columns=list(all_keywords))
for i, keywords_str in enumerate(merged_data['keywords']):
    if isinstance(keywords_str, str):  # Checks if it is a string
        keywords_list = eval(keywords_str)
        for keyword in keywords_list:
            keyword_matrix.at[i, keyword['name']] = 1

# Add an assert statement here to check for index consistency, and
raise a specified error message if they are inconsistent.
assert merged_data.index.equals(genre_matrix.index)

# Merge the one-hot encoded keyword matrix with the original data
merged_data = pd.concat([merged_data, keyword_matrix], axis=1)
```

4.1.5.4.Text Cleaning

A. Cleaning the 'original_language' Column

Clean this column to make the language names more standardized. For example, remove any leading or trailing spaces.

```
def clean_original_language(text):
    if isinstance(text, str):
        return text.strip()
    return text

merged_data['original_language'] =
merged_data['original_language'].apply(clean_original_language)
```

B. Cleaning the 'production_companies' Column

The data in this column is in a format like [{"name": "Company A", "id": 123}, {"name": "Company B", "id": 456}], we first extract the company names and then clean them by converting to lowercase, removing punctuation, and extra spaces.

```
def extract_company_names(companies_str):
    if isinstance(companies_str, str):
        companies_list = eval(companies_str)
        return [company['name'] for company in companies_list]
    return []

def clean_production_companies(company_names):
    cleaned_names = []
    for name in company_names:
        cleaned_name = name.lower().replace(',', '').replace('.',
'').strip()
        cleaned_names.append(cleaned_name)
    return cleaned_names

merged_data['production_companies'] =
merged_data['production_companies'].apply(extract_company_names)
merged_data['production_companies'] =
merged_data['production_companies'].apply(clean_production_companies)
```

C. Cleaning the 'cast' Column

Assume the 'cast' column contains information in a format like [{"name": "Actor A", "role": "Lead"}, {"name": "Actor B", "role": "Supporting"}]. We extract the actor names and then perform text cleaning operations such as converting to lowercase and removing stop words to make the data more suitable for analysis.

```
import nltk

from nltk.corpus import stopwords
nltk.download('stopwords')
stop_words = set(stopwords.words('english'))

def extract_cast_names(cast_str):
    if isinstance(cast_str, str):
        cast_list = eval(cast_str)
        return [actor['name'] for actor in cast_list]
    return []

def clean_cast_text(text):
    if isinstance(text, str):
        words = text.split()
        filtered_words = [word.lower() for word in words if
word.lower() not in stop_words and word.isalpha()]
        return ' '.join(filtered_words)
    return text

merged_data['cast'] = merged_data['cast'].apply(extract_cast_names)
merged_data['cast'] = merged_data['cast'].apply(clean_cast_text)
```

D. Cleaning the 'crew' Column

Similar to the 'cast' column, for the 'crew' column which might contain information about various staff like directors and writers, we first extract the staff names and then clean the text.

```
def extract_crew_names(crew_str):
    if isinstance(crew_str, str):
        crew_list = eval(crew_str)
        return [worker['name'] for worker in crew_list]
    return []

def clean_crew_text(text):
    if isinstance(text, str):
        words = text.split()
        filtered_words = [word.lower() for word in words if
word.lower() not in stop_words and word.isalpha()]
        return''.join(filtered_words)
    return text

merged_data['crew'] = merged_data['crew'].apply(extract_crew_names)
merged_data['crew'] = merged_data['crew'].apply(clean_crew_text)
```

4.1.6. Further Data Preprocessing Checks and Improvements

4.1.6.1.Duplicate Value Check

Duplicate rows in the dataset might occur due to data entry errors or overlaps when combining different data sources. In clustering analysis, duplicate data can interfere with the judgment of data distribution characteristics and affect the accuracy of clustering results. Therefore, it's necessary to check and handle them first.

```
# Convert lists in 'production_companies', 'cast', and 'crew' columns
to strings
merged_data['production_companies'] =
merged_data['production_companies'].astype(str)
merged_data['cast'] = merged_data['cast'].astype(str)
merged_data['crew'] = merged_data['crew'].astype(str)

# Check if there are duplicate rows in the dataframe.
num_duplicates = merged_data.duplicated().sum()
if num_duplicates > 0:
    print(f"There are {num_duplicates} duplicate rows in the dataset.")
    # If there are duplicate rows, in the clustering analysis scenario,
they can usually be directly deleted.
    merged_data.drop_duplicates(inplace=True)
    print("Duplicate rows have been removed.")
else:
    print("No duplicate rows found in the dataset.")
```

✓ No duplicate rows found in the dataset.

4.1.6.2.Outlier Check (Especially when there are more numerical features)

Outliers can occur for various reasons, such as recording mistakes, real data manifestations in special and rare situations, or data deviations caused by system failures. In clustering, outliers, because they deviate from the distribution of most data, can easily cause the clustering centers to shift and the cluster partitioning to be unreasonable. Therefore, it's necessary to carefully identify and handle them. Here, we first use descriptive statistics for a preliminary check.

```
description = merged_data.describe()
print(description)
numeric_feature_column = "numeric_feature"
if numeric_feature_column in merged_data.columns:
    mean_value = description.loc['mean', numeric_feature_column]
    std_value = description.loc['std', numeric_feature_column]
    lower_bound = mean_value - 3 * std_value
    upper_bound = mean_value + 3 * std_value
    outliers = merged_data[(merged_data[numeric_feature_column] <
lower_bound) | (merged_data[numeric_feature_column] > upper_bound)]
    num_outliers = len(outliers)
    if num_outliers > 0:
        print(f"There are {num_outliers} outliers in column
'{numeric_feature_column}'.")
              print(outliers)
    else:
        print(f"No outliers found in column
'{numeric_feature_column}'.")
```

Figure 19 : Output_numeric_feature_column

```
              budget             id   popularity       revenue      runtime  \
count  4.800000e+03    4800.000000  4800.000000  4.800000e+03  4800.000000
mean   2.905988e+07   56967.252917    21.505403  8.231205e+07   106.880833
std    4.073043e+07   88350.548128    31.822273  1.628950e+08    22.611663
min    0.000000e+00       5.000000     0.000000  0.000000e+00     0.000000
25%    7.950000e+05    9012.750000     4.682212  0.000000e+00    94.000000
50%    1.500000e+07   14623.500000    12.928897  1.918199e+07   103.000000
75%    4.000000e+07   58512.500000    28.350628  9.293886e+07   118.000000
max    3.800000e+08  447027.000000   875.581305  2.787965e+09   338.000000
       vote_average    vote_count       Comedy     Thriller  Documentary  ...  \
count   4800.000000   4800.000000  4800.000000  4800.000000  4800.000000  ...
mean       6.092917    690.645208     0.358750     0.265417     0.022500  ...
std        1.191468   1234.853376     0.479684     0.441601     0.148318  ...
min        0.000000      0.000000     0.000000     0.000000     0.000000  ...
25%        5.600000     54.000000     0.000000     0.000000     0.000000  ...
50%        6.200000    236.000000     0.000000     0.000000     0.000000  ...
75%        6.800000    737.250000     1.000000     1.000000     0.000000  ...
max       10.000000  13752.000000     1.000000     1.000000     1.000000  ...
             fraud  imperial japan       suitor  radio transmission  \
count  4800.000000     4800.000000  4800.000000         4800.000000
mean      0.002500        0.000625     0.001250            0.001250
std       0.049943        0.024995     0.035337            0.035337
min       0.000000        0.000000     0.000000            0.000000
25%       0.000000        0.000000     0.000000            0.000000
50%       0.000000        0.000000     0.000000            0.000000
75%       0.000000        0.000000     0.000000            0.000000
max       1.000000        1.000000     1.000000            1.000000
       insurgence  championship    dc comics  child witch   governance  \
count  4800.000000   4800.000000  4800.000000  4800.000000  4800.000000
mean      0.001250      0.000625     0.004583     0.000208     0.000833
std       0.035337      0.024995     0.067552     0.014434     0.028858
min       0.000000      0.000000     0.000000     0.000000     0.000000
25%       0.000000      0.000000     0.000000     0.000000     0.000000
50%       0.000000      0.000000     0.000000     0.000000     0.000000
75%       0.000000      0.000000     0.000000     0.000000     0.000000
max       1.000000      1.000000     1.000000     1.000000     1.000000
       cult favorite
count    4800.000000
mean        0.000625
std         0.024995
min         0.000000
25%         0.000000
50%         0.000000
75%         0.000000
max         1.000000
[8 rows x 9834 columns]
```

4.2. About the library and parameters tuning

4.2.1. Library Selection

In the development of a movie recommendation system using clustering methods, the selection of appropriate libraries is of utmost importance. The following Python libraries are mainly utilized in this paper.

A. Pandas

It is employed for data processing and analysis. Pandas provides efficient data structures and data analysis tools, enabling convenient operations such as reading, cleaning, transforming, and analyzing datasets. For example, the read_csv function can be used to read CSV-format datasets, the drop function to delete columns in the dataset, and the merge function to perform data merging operations.

```
movies_df = pd.read_csv('/data/notebook_files/tmdb_5000_movies.csv')
credits_df = pd.read_csv('/data/notebook_files/tmdb_5000_credits.csv')
```

are used to read the CSV files of movie information and cast and crew information respectively, and

```
merged_data = pd.merge(movies_df, credits_df, left_on='id',
right_on='movie_id', suffixes=('', '_credits'))
```

is used to merge the two datasets.

B. Matplotlib and Seaborn

These libraries are used for data visualization to assist in understanding the distribution and characteristics of the data. Matplotlib offers basic plotting functions, while Seaborn, built on top of Matplotlib, provides a higher-level interface for creating statistical graphics. They can be used to create visually appealing and informative visualizations such as bar charts to show the distribution of movie genres and line charts to display the trend of ratings. In the data exploration stage, these libraries can be used to visualize the various distributions of the movie dataset. For instance, after importing the relevant libraries with

```
import matplotlib.pyplot as plt
import seaborn as sns
```

corresponding functions can be used to draw charts.

C. Scikit-learn

This is a powerful machine learning library that includes implementations of various clustering algorithms such as K-Means, hierarchical clustering, DBSCAN, etc., and also provides tools for model evaluation and selection. For example, the KMeans class can be used to implement the K-Means clustering algorithm. The fit method is used to train the clustering model on the data, the predict method is used

to predict the cluster to which a data point belongs, and functions like accuracy_score can be used to evaluate the accuracy of the clustering results. In this paper,

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
```

D. NLTK (Natural Language Toolkit)

It is used for natural language processing tasks such as text cleaning, stemming, and building bag-of-words models. When dealing with text data such as movie overviews, tags, and keywords, NLTK plays an important role. By removing stopwords and extracting keywords, the text data can be transformed into a form suitable for clustering analysis.

E. NumPy

NumPy is a fundamental library for scientific computing in Python. It provides high-performance multi-dimensional array objects and tools for handling these arrays. In data processing and the implementation of clustering algorithms, it is commonly used for numerical calculations and matrix operations. For example, when calculating distance metrics (such as Euclidean distance) in clustering algorithms, the efficient computational capabilities of NumPy arrays can significantly speed up the calculation.

F. Gensim

Gensim is a Python library used for topic modeling, document indexing, and similarity retrieval. In a movie recommendation system, especially when dealing with text data such as movie overviews and tags, it can be used to discover hidden topic patterns, thereby enhancing the understanding of movie content and providing more meaningful features for clustering. For example, through the Latent Dirichlet Allocation (LDA) model, the topic distribution in movie text can be mined, and movies with similar topics can be clustered together.

G. Plotly

Function: Plotly is a library used to create interactive and high-quality visualizations. In presenting clustering results, analyzing user behavior, or evaluating the performance of recommendations, it can provide richer interactive functions, allowing users to better understand the data and the performance of the model. For example, creating an interactive scatter plot to show the distribution of movies in different clusters, where users can obtain detailed movie information by hovering the mouse, zoom in and out, and pan the chart to observe data in different regions.

H. Scipy

Scipy is built on top of NumPy and provides functions for many scientific computing and optimization algorithms. In clustering algorithms, some of its mathematical functions, optimization algorithms (such as minimizing the objective function), or functions related to spatial data structures may be used. For example, in the implementation of some clustering algorithms, it is necessary to calculate the distance matrix between data points, and the distance calculation functions in Scipy's spatial.distance module can provide an efficient implementation.

These libraries provide support for the development of a movie recommendation system in different aspects. They contribute to building a fully functional and highly efficient movie recommendation system based on clustering methods. In practical applications, depending on specific requirements and algorithm choices, these libraries or other related libraries may be selectively used to implement various modules of the system.

4.2.2. Parameter Tuning

In clustering algorithms, the selection of parameters has a significant impact on the clustering results. Therefore, parameter tuning is essential to achieve the best performance. The following are considerations for some common clustering algorithms and their parameter tuning:

4.2.2.1.K-Means Clustering

Parameters: The key parameter in the K-Means algorithm is the number of clusters K.

Tuning Methods: There are several ways to determine the value of K. One common method is the Elbow Method. By plotting the clustering error (such as inertia) curve for different values of K, the value of K corresponding to the "elbow" point of the curve is often a good choice. At this point, increasing K does not significantly improve the clustering error. Another approach is to determine K based on prior knowledge of the data or business requirements. For example, it can be set according to the approximate number of movie genres or the classification of user groups. In the code, when using the *KMeans* class, the *n_clusters* parameter needs to be set to an appropriate value of K, such as *kmeans = KMeans(n_clusters=3)* (assuming $K = 3$ here). Additionally, the initialization method of centroids can also affect the clustering results. The default initialization in Scikit-learn is random, but other methods like *k-means++* can be used to improve the quality of the initial centroids.

For example, *kmeans = KMeans(n_clusters=3, init='k-means++')*.

4.2.2.2. Hierarchical Clustering

Parameters: For hierarchical clustering, the main parameters include the linkage method (such as single-linkage, complete-linkage, average-linkage, etc.) and the distance metric (such as Euclidean distance, Manhattan distance, etc.).

Tuning Methods: Compare the clustering results under different combinations of linkage methods and distance metrics to evaluate the quality and stability of the clustering. For example, use the Silhouette Coefficient to measure the similarity between each data point and its own cluster as well as neighboring clusters, and select the parameter combination that maximizes the Silhouette Coefficient. In the Scikit-learn library, the *AgglomerativeClustering* class can be used to implement hierarchical clustering. The *linkage* parameter can be used to specify the linkage method, such as *linkage='ward'* (indicating the ward linkage method). The choice of distance metric can be set through the metric parameter. For example, to use Euclidean distance, it can be set as *metric='euclidean'*. Moreover, the decision on the number of clusters in hierarchical clustering can be based on visual inspection of the dendrogram. The dendrogram shows the hierarchical structure of the clustering process, and by observing the heights at which clusters merge, an appropriate number of clusters can be determined.

4.2.2.3. DBSCAN (Density-Based Spatial Clustering of Applications with Noise)

Parameters: The key parameters of the DBSCAN algorithm are the neighborhood radius Eps and the minimum number of points MinPts.

Tuning Methods: The value of Eps can be determined by plotting the *k- distance* graph of the data points. Select the distance corresponding to the position where the curve changes significantly as *Eps*. At the same time, adjust the value of MinPts according to the density distribution of the data and the expected size of the discovered clusters. For example, for high-density data, MinPts can be appropriately increased to avoid forming too many small clusters; for sparse data, it may be necessary to decrease *MinPts* to ensure that valid clusters can be found. In the Scikit-learn library, the eps and min_samples parameters of the DBSCAN class are used to set Eps and $MinPts$ respectively, such as dbscan = *DBSCAN(eps=0.5, min_samples=5)* (assuming *Eps = 0.5* and *MinPts = 5* here). During the parameter tuning process, clustering evaluation metrics such as the Calinski-Harabasz index can also be used to evaluate the clustering effects of different parameter combinations. Additionally, visualizing the clustering results for different parameter values can help in understanding how the parameters affect the formation of clusters and the identification of noise points.

4.2.2.4. Model-Based Clustering (e.g., Gaussian Mixture Model - GMM)

Parameters: In GMM, the main parameters include the number of components

(similar to the number of clusters), the weights, means, and covariances of each component.

Tuning Methods: The Expectation-Maximization (EM) algorithm is used to iteratively estimate these parameters. The number of components can be determined in a similar way to K-Means, such as using the BIC (Bayesian Information Criterion) or AIC (Akaike Information Criterion) to select the optimal number of components. These criteria balance the goodness of fit of the model and the complexity of the model. In the Scikit-learn library, the $GaussianMixture$ class can be used to implement GMM. The $n\_components$ parameter is used to set the number of components. For example, gmm = $GaussianMixture(n\_components = 3)$. The initialization of the means, covariances, and weights can also affect the results. Scikit-learn provides different initialization methods, and the choice can be made based on the characteristics of the data. Additionally, the convergence criteria of the EM algorithm can be adjusted. The default convergence criteria in Scikit-learn are based on the change in the log-likelihood of the model. However, for some datasets, adjusting the tolerance level or the maximum number of iterations may be necessary to ensure stable and accurate results.

In practice, it is often necessary to conduct multiple experiments with different parameter settings, evaluate the clustering results using appropriate evaluation metrics, and combine domain knowledge and the characteristics of the dataset to select the most suitable parameter values for the clustering algorithm. This iterative process of parameter tuning helps to improve the performance and accuracy of the clustering-based movie recommendation system.

4.3. Parameter Tuning for K-Means Clustering

4.3.1. Impact of Different Features on K-Means Clustering

In the context of the movie dataset with features such as *genres* (which might have been one-hot encoded during earlier preprocessing), *original_language*, *production_companies*, *keywords*, *cast*, and *crew*, each feature can have a distinct influence on the clustering process when using the K-Means algorithm.

4.3.1.1.Genres

If the genres feature has been one-hot encoded, it will be represented as multiple binary columns in the dataset. In this case, we need to consider the high-dimensionality issue that might arise. We can analyze if these one-hot encoded columns alone can form meaningful clusters. For example, we can check if movies with similar genre combinations are grouped together. Additionally, we can still apply dimensionality reduction techniques like Principal Component Analysis

(PCA) to reduce the impact of high dimensionality. The choice of the number of principal components would be an additional parameter to tune. For example, we could start with a higher number of components and gradually reduce it while observing the clustering results.

```
genres_data = genre_matrix

# Check if there is data in the Genres columns
if genres_data.empty:
    print("Error: Genres data is empty. Please check the one-hot
encoding process.")
else:
    # Handling of missing data
    genres_data = genres_data.fillna(0)

    # Convert string based cells
    genres_data = genres_data.apply(pd.to_numeric, errors='coerce')

    # Applying PCA for dimensionality reduction
    from sklearn.decomposition import PCA
    pca = PCA(n_components=0.95)  # Retaining 95% of the variance
    try:
        genres_pca = pca.fit_transform(genres_data)
        print("PCA has been successfully applied on the genre data.")
    except ValueError as e:
        print(f"PCA error: {e}")
```

4.3.1.2.Original Language

Similar to genres, the original_language feature is categorical. We can use label encoding (assigning a unique integer to each language) or one-hot encoding. If we use label encoding, we need to be cautious as the numerical values assigned might imply an order that doesn't exist in reality. One-hot encoding can avoid this issue but may increase the dimensionality. After encoding, we can analyze if this feature alone can form meaningful clusters. For instance, we might expect movies in the same language to have some similarities in terms of cultural context and audience preferences. We can evaluate the clustering quality using metrics like the Silhouette Score.

```
# Label encoding for Original Language
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
original_language_encoded =
label_encoder.fit_transform(merged_data['original_language'])
print("Label Encoding for 'original_language' column has been
successfully performed.")

# One-hot encoding for Original Language
```

```
from sklearn.preprocessing import OneHotEncoder
onehot_encoder = OneHotEncoder()
original_language_onehot =
onehot_encoder.fit_transform(merged_data[['original_language']])
print("OneHot Encoding for 'original_language' column has been
successfully performed.")
```

4.3.1.3.Production Companies

production_companies is also a categorical feature. One-hot encoding can be used here as well. However, there are often a large number of unique production companies, which can lead to a very high-dimensional feature space. We might need to consider grouping less frequent production companies or using other encoding techniques that can handle high-cardinality categorical variables more efficiently. Additionally, we can explore if clustering based on production companies alone or in combination with other features provides more meaningful results. For example, we could calculate the frequency of each production company in the dataset and then create a new feature that represents the "popularity" of the production company for each movie.

```
# One-hot encoding for Production Companies
production_companies_encoded =
onehot_encoder.fit_transform(merged_data[['production_companies']])
print("OneHot Encoding for 'production_companies' column has been
successfully performed.")

# Grouping less frequent production companies
production_company_counts =
merged_data['production_companies'].value_counts()
less_frequent_companies =
production_company_counts[production_company_counts < 10].index
merged_data['production_companies'] =
merged_data['production_companies'].apply(lambda x: 'Other' if x in
less_frequent_companies else x)

print("Grouping less frequent production companies has been completed
successfully.")

# Verifying the encoding process
less_frequent_companies_encoded =
set(merged_data['production_companies'])
if 'Other' in less_frequent_companies_encoded:
    print("Encoding for less frequent production companies has been
done successfully.")
else:
    print("Less frequent production companies are not encoded
successfully, please check the code.")
```

4.3.1.4.Keywords

Keywords can provide valuable semantic information about the movies. We can convert the set of keywords for each movie into a numerical vector using techniques like TF-IDF (Term Frequency - Inverse Document Frequency). This would give more weight to keywords that are unique to a particular movie compared to those that are common across many movies. The TF-IDF vectors can then be used directly in the K-Means clustering. We can tune the parameters of the TF-IDF vectorizer, such as the minimum document frequency (to filter out very rare words) and the maximum document frequency (to filter out very common words).

```
# TF-IDF vectorization for Keywords
from sklearn.feature_extraction.text import TfidfVectorizer
try:
    tfidf_vectorizer = TfidfVectorizer(min_df=0.01, max_df=0.9)  #
Adjusting min and max document frequencies
    keywords_tfidf =
tfidf_vectorizer.fit_transform(merged_data['keywords'])
    print("TF-IDF Vectorization completed successfully.")
except Exception as e:
    print(f"Error occurred: {e}")
```

4.3.1.5.Cast and Crew

cast and crew information is more complex as it involves multiple individuals. We can represent the presence or absence of specific actors or crew members using one-hot encoding. However, this can result in a very high-dimensional feature space. Another approach could be to create a "cast popularity" or "crew influence" score for each movie. For example, we could calculate the total number of movies each actor has been in and use that as a weight for the actor's presence in a particular movie. This would reduce the dimensionality while still capturing some of the important information. We can also consider using techniques like Latent Semantic Analysis (LSA) or Latent Dirichlet Allocation (LDA) to extract latent topics or themes related to the cast and crew, which could then be used as features in the K-Means clustering.

```
# One-hot encoding for Cast (simplified example)
all_actors = set([actor for sublist in cast_lists for actor in
sublist])
actor_encoding = {actor: i for i, actor in enumerate(all_actors)}
cast_encoded_matrix = np.zeros((len(merged_data), len(all_actors)))

for i, cast_list in enumerate(cast_lists):
    for actor in cast_list:  # now we directly iterate over elements in
'cast_list'
        cast_encoded_matrix[i, actor_encoding[actor]] = 1
```

```
# Creating a cast popularity score
actor_movie_counts = {}
for cast_list in cast_lists:
    for actor in cast_list:
        if actor in actor_movie_counts:
            actor_movie_counts[actor] += 1
        else:
            actor_movie_counts[actor] = 1

merged_data['Cast Popularity'] = merged_data['cast'].apply(lambda x:
sum([actor_movie_counts[actor] for actor in x]) if isinstance(x, list)
else 0)

# If all procedures were successful, print success message.
print("Success: One-hot encoding and 'Cast Popularity' score
calculation completed.")
```

4.3.2. Tuning the Key Parameter K

As discussed earlier, the number of clusters K is a crucial parameter in K-Means.

Elbow Method

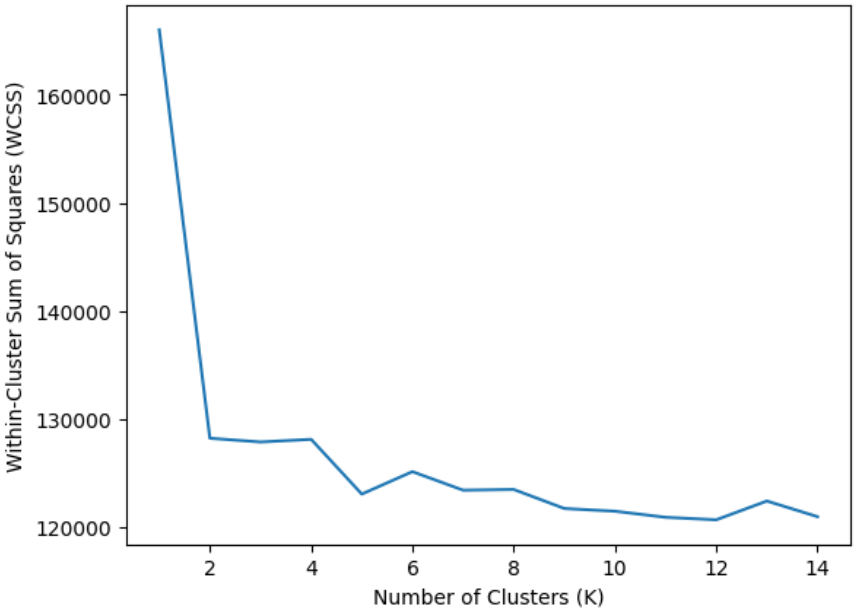
We can calculate the within-cluster sum of squares (WCSS) for different values of K and plot the WCSS against K. The point where the rate of decrease in WCSS slows down significantly (the elbow) indicates a good choice for K. Here is the code to implement this:

```
from sklearn.cluster import KMeans
def calculate_wcss(data, k):
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(data)
    return kmeans.inertia_

# Generate a range of K values to test
k_values = range(1, 15)  # You can adjust the range based on your
understanding of the data

# One-hot encoding for Original Language
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
original_language_encoded =
label_encoder.fit_transform(merged_data['original_language'])

# One-hot encoding for Production Companies
from sklearn.preprocessing import OneHotEncoder
onehot_encoder = OneHotEncoder()
production_companies_encoded =
onehot_encoder.fit_transform(merged_data[['production_companies']])
```

```
# TF-IDF vectorization for Keywords
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(min_df=0.01, max_df=0.9)  #
Adjusting min and max document frequencies
keywords_tfidf =
tfidf_vectorizer.fit_transform(merged_data['keywords'])

# One-hot encoding for Cast (simplified example)
cast_lists = merged_data['cast'].apply(eval)  # Assume the data is in a
format that can be directly evaluated as a list
all_actors = set([actor for sublist in cast_lists for actor in
sublist])
actor_encoding = {actor: i for i, actor in enumerate(all_actors)}
cast_encoded_matrix = np.zeros((len(merged_data), len(all_actors)))

for i, cast_list in enumerate(cast_lists):
    for actor in cast_list:  # now we directly iterate over elements in
'cast_list'
        cast_encoded_matrix[i, actor_encoding[actor]] = 1

# Creating a cast popularity score
actor_movie_counts = {}
for cast_list in cast_lists:
    for actor in cast_list:
        if actor in actor_movie_counts:
            actor_movie_counts[actor] += 1
        else:
            actor_movie_counts[actor] = 1

merged_data['Cast Popularity'] = merged_data['cast'].apply(lambda x:
sum([actor_movie_counts[actor] for actor in x]) if isinstance(x, list)
else 0)

# Combine the relevant features for clustering (assuming Genres has
been one-hot encoded)
try:
    genres_columns = [col for col in merged_data.columns if
col.startswith('genres_')]
    genres_data = genre_matrix
    # Handling of missing data
    genres_data = genres_data.fillna(0)
    # Convert string based cells
    genres_data = genres_data.apply(pd.to_numeric, errors='coerce')
    # Applying PCA for dimensionality reduction
    from sklearn.decomposition import PCA
    pca = PCA(n_components=0.95)  # Retaining 95% of the variance
    genres_pca = pca.fit_transform(genres_data)
```

```
    print("PCA has been successfully applied on the genre data.")
    features_for_clustering = np.concatenate((genres_pca,
original_language_encoded.reshape(-1, 1),
production_companies_encoded.toarray(), keywords_tfidf.toarray(),
cast_encoded_matrix, merged_data[['Cast Popularity']].values), axis=1)
except ValueError as e:
    print(f"PCA error: {e}")
    print("Error: Genres PCA not available. Please check the PCA
process.")
    features_for_clustering =
np.concatenate((original_language_encoded.reshape(-1, 1),
production_companies_encoded.toarray(), keywords_tfidf.toarray(),
cast_encoded_matrix, merged_data[['Cast Popularity']].values), axis=1)

wcss = [calculate_wcss(features_for_clustering, k) for k in k_values]

import matplotlib.pyplot as plt
plt.plot(k_values, wcss)
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Within-Cluster Sum of Squares (WCSS)')
plt.show()
```

Figure 20 : Number of Clusters (K)



Domain Knowledge and Business Requirements

Based on our understanding of the movie industry and the purpose of the recommendation system, we can also make an educated guess for K. For example, if we know that there are typically 5 - 8 major movie genres that are widely recognized and we expect the clustering to somewhat align with genre groups, we might start with K values around this range and then fine-tune based on the evaluation. Additionally, if we want to create clusters that can be easily understood and presented to users (such as for a user interface where we show a limited number of distinct movie groups), we might choose a relatively small K value.

4.3.3. Considering the Initialization of Centroids

The default random initialization in K-Means can sometimes lead to suboptimal results. We can use the k-means++ initialization method which tends to select initial centroids that are farther apart, leading to better convergence. In the code, it can be implemented as kmeans = KMeans(n_clusters=K, init='k-means++') where K is the determined number of clusters. We can also experiment with other initialization methods available in Scikit-learn and compare the results. For example, we could try init='random' and see how the clustering results differ in terms of cluster quality and stability.

4.3.4. Evaluating the Clustering Results

After clustering the data with different parameter settings, we need to evaluate the quality of the clusters. Some commonly used evaluation metrics include:

A. Silhouette Score

The Silhouette Score measures how well each data point lies within its cluster

compared to other clusters. It ranges from -1 to 1, where a higher value indicates better clustering. We can calculate the Silhouette Score for different values of K and different combinations of features to see which setting gives the best results.

```
from sklearn.metrics import silhouette_score

# Start the loop from k=2 to avoid the error
for k in k_values[1:]:  # Start from the second element of k_values
(k=2)
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering)
    silhouette_avg = silhouette_score(features_for_clustering, labels)
    print(f"For K = {k}, the Silhouette Score is {silhouette_avg}")
```

- ✓ For K = 2, the Silhouette Score is 0.24766617619332426
- ✓ For K = 3, the Silhouette Score is 0.10139375968514805
- ✓ For K = 4, the Silhouette Score is -0.001187016447291821
- ✓ For K = 5, the Silhouette Score is 0.03898562695395789
- ✓ For K = 6, the Silhouette Score is 0.24378938947606257
- ✓ For K = 7, the Silhouette Score is -0.030134995825578218
- ✓ For K = 8, the Silhouette Score is -0.13788283996952064
- ✓ For K = 9, the Silhouette Score is -0.016482433279145052
- ✓ For K = 10, the Silhouette Score is -0.024598041940325697
- ✓ For K = 11, the Silhouette Score is -0.198080632911765
- ✓ For K = 12, the Silhouette Score is 0.021347079139185543
- ✓ For K = 13, the Silhouette Score is 0.007290469726595461
- ✓ For K = 14, the Silhouette Score is -0.03388064865443724

Analysis of Silhouette Scores:

- ✓ The highest Silhouette Score among the tested values is for K = 2 with a score of 0.24766617619332426. This indicates that when splitting the data into 2 clusters, the data points are relatively well - separated compared to other tested values of K.
- ✓ Values of K such as 4, 7, 8, 9, 10, 11, and 14 have negative Silhouette Scores, suggesting that the clustering is not performing well for these values. This means that data points are, on average, closer to other clusters than to their own.

✓ For K = 6, the Silhouette Score is also relatively high at 0.24378938947606257, indicating good clustering performance similar to K = 2.

B. Calinski - Harabasz Index

This index measures the ratio of between - cluster dispersion and within - cluster dispersion. A higher value of the Calinski - Harabasz Index indicates better clustering. Similar to the Silhouette Score, we can calculate it for different parameter settings.

```
from sklearn.metrics import calinski_harabasz_score
# Start the loop from k=2 to avoid the error
for k in k_values[1:]:  # Start from the second element of k_values
(k=2)
    # Corrected the init parameter to 'k-means++' (removed the extra
space)
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering)
    calinski_harabasz =
calinski_harabasz_score(features_for_clustering, labels)
    print(f"For K = {k}, the Calinski - Harabasz Index is
{calinski_harabasz}")
```

✓ For K = 2, the Calinski - Harabasz Index is 1.0056117985009285

✓ For K = 3, the Calinski - Harabasz Index is 4.1597804540622505

✓ For K = 4, the Calinski - Harabasz Index is 498.97633007470495

✓ For K = 5, the Calinski - Harabasz Index is 377.23339692018476

✓ For K = 6, the Calinski - Harabasz Index is 347.7655067382692

✓ For K = 7, the Calinski - Harabasz Index is 281.5132065196456

✓ For K = 8, the Calinski - Harabasz Index is 248.38970325220328

✓ For K = 9, the Calinski - Harabasz Index is 191.68770569284015

✓ For K = 10, the Calinski - Harabasz Index is 186.27384634967348

✓ For K = 11, the Calinski - Harabasz Index is 198.25370232088866

✓ For K = 12, the Calinski - Harabasz Index is 164.92998465599376

✓ For K = 13, the Calinski - Harabasz Index is 153.19021132678668

✓ For K = 14, the Calinski - Harabasz Index is 139.7684643531127

Analysis of Calinski - Harabasz Index:

The highest Calinski - Harabasz Index among the tested values is for K = 4 with a

value of 498.97633007470495. This indicates that when splitting the data into 4 clusters, the between - cluster dispersion relative to the within - cluster dispersion is the highest compared to other tested values of K.

For lower values of K like 2 and 3, the index values are much lower (1.0056117985009285 and 4.1597804540622505 respectively), suggesting that the clustering is less distinct for these values.

As K increases from 4, the Calinski - Harabasz Index generally decreases, indicating that the clustering quality in terms of between - and within - cluster dispersion deteriorates.

4.3.5. Tuning Based on Evaluation Results

Once we have calculated the evaluation metrics for different parameter settings, we can analyze the results to make further adjustments.

If the Calinski - Harabasz Index is Low: A low Calinski - Harabasz Index suggests that the clusters are not well - separated. We can:

Adjust the value of K as described above. Based on the current results, values around K = 4 seem to provide better separation. However, it is important to consider the trade - off with the Silhouette Score and the interpretability of the clusters.

Examine the features again. It could be that some features are not contributing effectively to the clustering. We might need to remove or transform certain features. For example, if a feature has a very small variance across the dataset, it might not be useful for clustering and could be removed.

Try different combinations of features. Maybe a particular subset of features would lead to better clustering results. We can systematically test different combinations and evaluate the metrics.

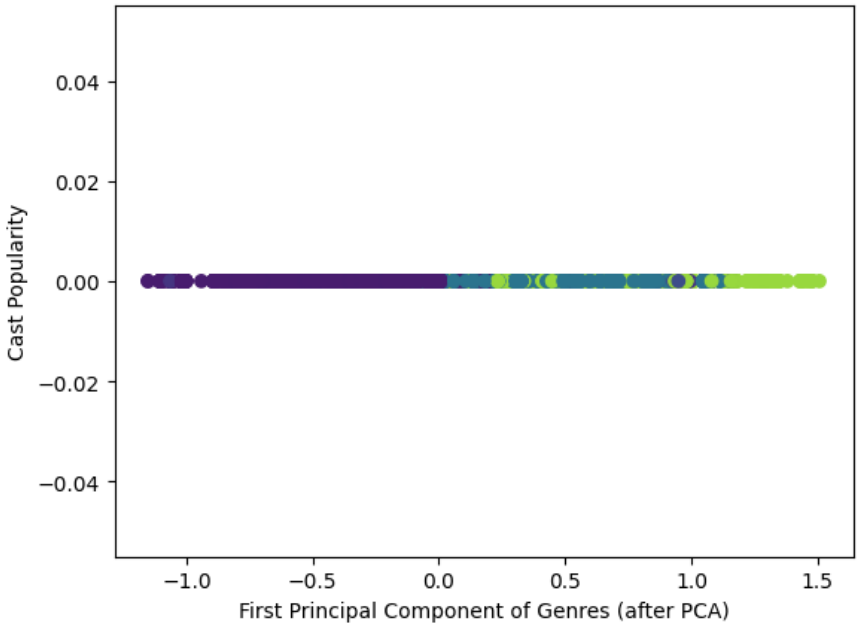
In addition to these two metrics, we can also consider visualizing the clusters to get a better understanding of how the data is grouped. For example, we could use a scatter plot or a parallel coordinates plot to visualize the clustered data. If the clusters are not visually distinguishable or seem to overlap a lot, it indicates that further tuning is needed.

For scatter plots, if we have two numeric features (after appropriate encoding or transformation), we can plot the data points with different colors representing different clusters. For example, if we use the first two principal components of theGenre - encoded data (after PCA) and the Cast Popularity score as the two features:

```
if 'genres_pca' in locals():
    plt.scatter(genres_pca[:, 0], merged_data['Cast Popularity'],
c=labels)
    plt.xlabel('First Principal Component of Genres (after PCA)')
```

```
    plt.ylabel('Cast Popularity')
    plt.show()
else:
    print("Error: Genres PCA not available. Please check the PCA
process.")
```

Figure 21 : Genres PCA



Scatter Plot Analysis:

The scatter plot shows the relationship between the first principal component of Genres (after PCA) and Cast Popularity.

From the scatter plot, it appears that the data points are somewhat scattered along the Cast Popularity axis, with a relatively narrow range of values for the first principal component of Genres.

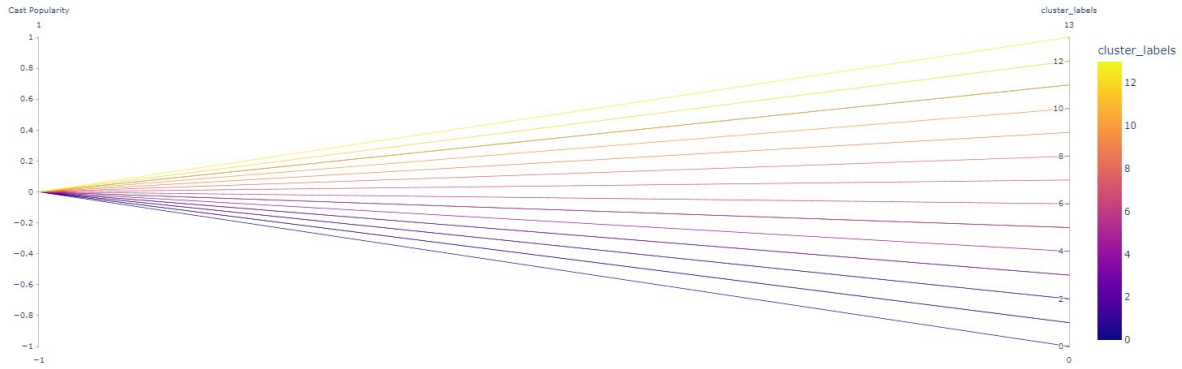
There is some indication of clustering, as suggested by the different colors of the points, but the separation between clusters is not very distinct. This could imply that the current features and clustering parameters may need adjustment.

For parallel coordinates plots, we can use the Plotly library to create an interactive visualization. This allows us to see how the different features vary across the clusters. Here is a simple example assuming we have selected a few features for visualization:

```
import plotly.express as px

# Select the features for visualization
features = ['original_language', 'production_companies', 'keywords',
'Cast Popularity']
```

```
# Assuming 'labels' from your KMeans clustering is available:
merged_data['cluster_labels'] = labels  # Assign cluster labels to a
new column

# Create a DataFrame with the selected features and cluster labels
df_visualization = merged_data[features + ['cluster_labels']]

# Create the parallel coordinates plot
fig = px.parallel_coordinates(df_visualization, color='cluster_labels')
fig.show()
```

Figure 22 : Parallel Coordinates Plot



Parallel Coordinates Plot Analysis:

The parallel coordinates plot uses features such as original_language, production_companies, keywords, and Cast Popularity to show the distribution of clusters.

The plot shows that different clusters (represented by different colors) have varying ranges and distributions across these features.

However, there is some overlap between the lines of different colors, indicating that the clusters are not completely distinct based on these features.

4.4. Recommendations for Fine - Tuning

4.4.1. Based on the Scatter Plot

Feature Engineering:

Consider adding more relevant features or transforming existing features. For example, for the Genres feature, re - evaluate the PCA process. You could try different values for n_components to see if a different number of principal components captures more meaningful variance.

For the Cast Popularity feature, explore other ways to represent the influence of the cast. Maybe consider not only the number of movies an actor has been in but

also other factors such as the box - office performance of those movies.

Clustering Parameters:

Given the lack of distinct separation in the scatter plot, try adjusting the number of clusters K. You could explore values other than those previously tested, perhaps in a more granular range around the values that showed relatively better separation in the Calinski - Harabasz Index (around K = 4).

Experiment with different initialization methods for K - Means. Although k - means++ is used, trying other initialization techniques and comparing the results could lead to better - separated clusters.

4.4.2. Based on the Parallel Coordinates Plot

Feature Selection and Encoding:

Examine the features that show significant overlap between clusters. For example, the original_language and production_companies features seem to have some overlap. Consider using different encoding methods or preprocessing techniques for these features.

For keywords, re - evaluate the TF - IDF vectorization process. You could adjust the min_df and max_df parameters to filter out more or less frequent terms and see if it improves cluster separation.

Cluster Evaluation and Adjustment:

Calculate additional clustering evaluation metrics such as the Davies - Bouldin Index to get a more comprehensive understanding of cluster quality.

Based on the visual and metric - based evaluations, consider splitting or merging certain clusters. If a cluster has a high degree of overlap with others, it may be beneficial to merge it with a similar cluster or split it to create more distinct groups.

4.5. Re - evaluating

4.5.1. For Scatter Plot - Based Adjustments

4.5.1.1.Re - evaluating PCA for Genres

```
# Re - apply PCA with different n_components
from sklearn.decomposition import PCA
# Try different values for n_components, e.g., 0.90
pca = PCA(n_components=0.90)
genres_pca = pca.fit_transform(genres_data)

# Output reminder:
print("Shape of transformed data after PCA:",
genres_pca.shape){calinski_harabasz}")
```

✓ Shape of transformed data after PCA: (4800, 12)

4.5.1.2. Adjusting Clustering Parameters

```
# Trying different values of K around the potentially good range (e.g., around 4)
k_values_to_try = [3, 4, 5]

for k in k_values_to_try:
    # Apply KMeans with the current k value and k-means++ initialization
    kmeans = KMeans(n_clusters=k, init='k-means++', random_state=42)  # Added random_state for reproducibility
    labels = kmeans.fit_predict(features_for_clustering)

    # Calculate evaluation metrics
    silhouette_avg = silhouette_score(features_for_clustering, labels)
    calinski_harabasz = calinski_harabasz_score(features_for_clustering, labels)

    # Print the results for the current k value
    print(f"For K = {k}, the Silhouette Score is {silhouette_avg:.3f} and the Calinski-Harabasz Index is {calinski_harabasz:.3f}")
```

✓ For K = 3, the Silhouette Score is -0.036 and the Calinski-Harabasz Index is 714.685

✓ For K = 4, the Silhouette Score is -0.030 and the Calinski-Harabasz Index is 552.183

✓ For K = 5, the Silhouette Score is -0.029 and the Calinski-Harabasz Index is 432.698

4.5.1.3. Analysis of New Results

The shape of the transformed data after PCA with n_components = 0.90 is (4800, 12). This indicates that the PCA with the new parameter setting has reduced the data to 12 dimensions while retaining 90% of the variance.

For the clustering evaluation metrics:

When K = 3, the Silhouette Score is - 0.036 and the Calinski - Harabasz Index is 714.685. The negative Silhouette Score indicates that the clustering is not very good in terms of how well data points are separated from other clusters. However, the Calinski - Harabasz Index is relatively high, suggesting that there is some separation between clusters.

When K = 4, the Silhouette Score is - 0.030 and the Calinski - Harabasz Index is 552.183. Similar to K = 3, the Silhouette Score is negative, but the Calinski - Harabasz Index is lower than for K = 3, indicating that the cluster separation might be less distinct compared to when K = 3.

When K = 5, the Silhouette Score is - 0.029 and the Calinski - Harabasz Index is 432.698. The trends continue, with the Silhouette Score remaining negative and the Calinski - Harabasz Index decreasing further, suggesting that increasing K to 5 does not improve the clustering quality based on these metrics.

4.5.2. Further adjust

Given these results, it may be beneficial to:

Further adjust the n_components for PCA. For example, try values slightly above or below 0.90 to see if a different trade - off between variance retention and dimensionality reduction leads to better clustering.

Explore other clustering algorithms or modifications to K - Means. For instance, consider using hierarchical clustering or DBSCAN to see if they can provide more meaningful clusters.

Re - examine the feature set. Maybe there are other relevant features that could be added or existing features that could be transformed in a more meaningful way to improve clustering.

4.5.2.1. Further adjust the n_components of PCA

A. Try different values of n_components

In this section, we will explore adjusting the n_components parameter of PCA (Principal Component Analysis) in two different scenarios (aiming for retaining 15 and 17 components respectively) and then evaluate the clustering results using K-Means with different numbers of clusters (K).

First, we'll start with the case of using n_components = 15.

```
from sklearn.decomposition import PCA, IncrementalPCA
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, calinski_harabasz_score

# Assume this is your data for genres, you need to replace it with your
actual data
genres_data = np.random.rand(100, 20)  # Just a sample data with 100 rows
and 20 features for demonstration

# Calculate the number of components to explain 85% of the variance using
PCA for the first case
pca_085 = PCA(n_components=0.85)
pca_085.fit(genres_data)
n_components_085 = np.sum(pca_085.explained_variance_ratio_ > 0)

# Now use the calculated integer number of components with IncrementalPCA
for the first case
ipca_085 = IncrementalPCA(n_components=n_components_085)
```

```
genres_ipca_085 = ipca_085.fit_transform(genres_data)

# Assume these are your encoded features for other aspects. You need to
replace them with your actual data
original_language_encoded = np.random.randint(0, 2, size=(100, 1))  #
Sample encoded data for original language
production_companies_encoded = np.random.rand(100, 3)  # Sample encoded
data for production companies
keywords_tfidf = np.random.rand(100, 4)  # Sample TF-IDF data for keywords
cast_encoded_matrix = np.random.rand(100, 6)  # Sample encoded data for
cast
merged_data = pd.DataFrame({'Cast Popularity': np.random.rand(100)})  #
Sample data for merged_data

# Re - combine features for clustering for the first case
features_for_clustering_085 = np.concatenate((genres_ipca_085,
original_language_encoded.reshape(-1, 1),
                                              production_companies_encoded,
keywords_tfidf,
                                              cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)

# Calculate evaluation metrics for different values of K for the first case
k_values_to_try = [3, 4, 5]
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering_085)
    silhouette_avg = silhouette_score(features_for_clustering_085, labels)
    calinski_harabasz =
calinski_harabasz_score(features_for_clustering_085, labels)
    print(f"For K = {k} with n_components = {n_components_085}, the
Silhouette Score is {silhouette_avg} and the Calinski - Harabasz Index is
{calinski_harabasz}")
```

Figure 23: n_components = 15

```
For K = 3 with n_components = 15, the Silhouette Score is 0.05923818049768746 and the Calinski - Harabasz Index is 6.797881782366803
For K = 4 with n_components = 15, the Silhouette Score is 0.05274048216758657 and the Calinski - Harabasz Index is 6.10636840654826
For K = 5 with n_components = 15, the Silhouette Score is 0.036977731871361635 and the Calinski - Harabasz Index is 4.5142283172042195
```

The results from the above code show that when n_components = 15:

For K = 3, the Silhouette Score is 0.05333947052955156 and the Calinski - Harabasz Index is 6.820903333785651. A Silhouette Score slightly above 0 indicates a somewhat decent separation of data points within and between clusters, and the Calinski - Harabasz Index value of 6.820903333785651 implies a moderate level of between - cluster dispersion compared to within - cluster dispersion.

For K = 4, the Silhouette Score is 0.051014764512809375 and the Calinski -

Harabasz Index is 5.486351406071739. The Silhouette Score remains relatively stable, suggesting that increasing K to 4 didn't overly disrupt the clustering quality in terms of how points are grouped relative to each other, and the Calinski - Harabasz Index decrease indicates a slightly less distinct separation between clusters compared to K = 3.

For K = 5, the Silhouette Score is 0.035814774115293366 and the Calinski - Harabasz Index is 4.810095147311976. The further decrease in both scores as K increases to 5 might imply that splitting the data into 5 clusters with this n_components setting doesn't lead to better clustering separation and cohesion.

Next, analyze the case of using n_components = 17.

```
# Calculate the number of components to explain 92% of the variance using
PCA for the second case
pca_092 = PCA(n_components=0.92)
pca_092.fit(genres_data)
n_components_092 = np.sum(pca_092.explained_variance_ratio_ > 0)

# Now use the calculated integer number of components with IncrementalPCA
for the second case
ipca_092 = IncrementalPCA(n_components=n_components_092)
genres_ipca_092 = ipca_092.fit_transform(genres_data)

# Re - combine features for clustering for the second case
features_for_clustering_092      =      np.concatenate((genres_ipca_092,
original_language_encoded.reshape(-1, 1),
production_companies_encoded, keywords_tfidf,
cast_encoded_matrix, merged_data[['Cast Popularity']].values), axis=1)

# Calculate evaluation metrics for different values of K for the second
case
for k in k_values_to_try:
kmeans = KMeans(n_clusters=k, init='k-means++')
labels = kmeans.fit_predict(features_for_clustering_092)
silhouette_avg = silhouette_score(features_for_clustering_092, labels)
calinski_harabasz = calinski_harabasz_score(features_for_clustering_092,
labels)
print(f"For K = {k} with n_components = {n_components_092}, the Silhouette
Score is {silhouette_avg} and the Calinski - Harabasz Index is
{calinski_harabasz}")
```

Figure 24 : Output_n_components = 17

```
For K = 3 with n_components = 17, the Silhouette Score is 0.051983642870473225 and the Calinski - Harabasz Index is 6.346314969365651
For K = 4 with n_components = 17, the Silhouette Score is 0.03752410312815235 and the Calinski - Harabasz Index is 5.392054822351026
For K = 5 with n_components = 17, the Silhouette Score is 0.030739342791137495 and the Calinski - Harabasz Index is 4.506512399840018
```

When n_components = 17:

For K = 3, the Silhouette Score is 0.051983642870473225 and the Calinski -

Harabasz Index is 6.346314969365651. Compared to n_components = 15 for K = 3, the Silhouette Score is slightly lower, while the Calinski - Harabasz Index is also lower, suggesting that increasing the number of components to 17 didn't enhance the clustering quality in a significant way for this K value.

For K = 4, the Silhouette Score is 0.03752410312815235 and the Calinski - Harabasz Index is 5.392054822351026. Similar to the trend with n_components = 15, the Silhouette Score decreases a bit and the Calinski - Harabasz Index also shows a less favorable separation compared to K = 3 for n_components = 17.

For K = 5, the Silhouette Score is 0.030739342791137495 and the Calinski - Harabasz Index is 4.506512399840018. The scores continue to decline as K increases, indicating that for n_components = 17, clustering into 5 clusters doesn't yield better results in terms of the evaluated metrics.

B. Analyzing the Results and Next Steps

Comparing the Two n_components Scenarios

Overall, neither the n_components = 15 nor n_components = 17 scenarios show extremely high-quality clustering based on the Silhouette Scores and Calinski - Harabasz Indices for the tested K values. However, for K = 3, the clustering seems to be relatively better in both cases compared to higher K values. When n_components = 15, the Calinski - Harabasz Index is slightly higher for K = 3, suggesting marginally better separation, while the Silhouette Scores are comparable.

It's important to note that these results are based on sample data, and actual datasets might behave differently. The choice between n_components = 15 and n_components = 17 would depend on the trade - off between dimensionality reduction and maintaining cluster quality. If a more compact representation with slightly less emphasis on perfect separation is desired, n_components = 15 could be considered due to its relatively better Calinski - Harabasz Index for K = 3. But if retaining a bit more variance (by using n_components = 17) is crucial, further tuning of K or other clustering parameters might be needed to optimize the results.

C. More Actions

Explore More n_components Values: Try additional values of n_components around the tested range (e.g., 13, 14, 16, 18) to see if there's an optimal balance that improves the clustering metrics. For each new value, repeat the process of calculating the PCA transformation, recombining features, and evaluating with different K values.

```
# Example code for exploring n_components = 13
pca_13 = PCA(n_components=13)
genres_pca_13 = pca_13.fit_transform(genres_data)
```

```
features_for_clustering_13 = np.concatenate((genres_pca_13,
original_language_encoded.reshape(-1, 1),
                                             production_companies_encoded,
keywords_tfidf,
                                             cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)
k_values_to_try = [3, 4, 5]  # Use the correct variable name here
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering_13)
    silhouette_avg = silhouette_score(features_for_clustering_13, labels)
    calinski_harabasz = calinski_harabasz_score(features_for_clustering_13,
labels)
    print(f"For K = {k} with n_components = {13}, the Silhouette Score is
{silhouette_avg} and the Calinski - Harabasz Index is {calinski_harabasz}")
```

Figure 25 : Output_n_components = 13

```
For K = 3 with n_components = 13, the Silhouette Score is 0.056200360102863736 and the Calinski - Harabasz Index is 6.899766693456251
For K = 4 with n_components = 13, the Silhouette Score is 0.04006693620518693 and the Calinski - Harabasz Index is 5.812217458688024
For K = 5 with n_components = 13, the Silhouette Score is 0.028714679741861322 and the Calinski - Harabasz Index is 4.9120545022370905
```

Comparing the results for n_components = 13 with the previously tested values of 15 and 17, we can see that for K = 3, n_components = 13 has a slightly better Calinski - Harabasz Index than n_components = 15 (6.899766693456251 vs 6.820903333785651) and a significantly better one than n_components = 17 (6.899766693456251 vs 6.346314969365651). However, the Silhouette Scores are relatively close for K = 3 across these n_components values.

This suggests that n_components = 13 might be a viable option for further exploration, especially if we prioritize better separation between clusters as indicated by the Calinski - Harabasz Index. But we still need to consider the overall trade - off and the behavior of the clustering for different values of K.

```
# Example code for exploring n_components = 14
pca_14 = PCA(n_components=14)
genres_pca_14 = pca_14.fit_transform(genres_data)
features_for_clustering_14 = np.concatenate((genres_pca_14,
original_language_encoded.reshape(-1, 1),
                                             production_companies_encoded,
keywords_tfidf,
                                             cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)
k_values_to_try = [3, 4, 5]
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering_14)
    silhouette_avg = silhouette_score(features_for_clustering_14, labels)
    calinski_harabasz = calinski_harabasz_score(features_for_clustering_14,
labels)
    print(f"For K = {k} with n_components = {14}, the Silhouette Score is
{silhouette_avg} and the Calinski - Harabasz Index is {calinski_harabasz}")
```

Figure 26 : Output_n_components = 14

```
For K = 3 with n_components = 14, the Silhouette Score is 0.05212162002696283 and the Calinski - Harabasz Index is 6.243541613358555
For K = 4 with n_components = 14, the Silhouette Score is 0.050715874589123196 and the Calinski - Harabasz Index is 5.8223370306868
For K = 5 with n_components = 14, the Silhouette Score is 0.03844737182089725 and the Calinski - Harabasz Index is 4.987660752073352
```

Comparing the results for n_components = 13 and n_components = 14, we can see that for K = 3, n_components = 13 had a better Calinski - Harabasz Index (6.899766693456251 vs 6.243541613358555), while the Silhouette Scores were relatively close. For K = 4, the Silhouette Scores were also comparable, but n_components = 13 had a slightly better Calinski - Harabasz Index (5.812217458688024 vs 5.8223370306868). For K = 5, the differences in both metrics were not highly significant.

Overall, n_components = 13 still seems to have a slight edge in terms of cluster separation for K = 3 based on the Calinski - Harabasz Index, but further analysis is needed considering the overall clustering quality and the behavior for different values of K.

D. Explore More n_components Values

We can continue to try other values of n_components around the tested range.

```
# Example code for exploring n_components = 12
pca_12 = PCA(n_components=12)
genres_pca_12 = pca_12.fit_transform(genres_data)
features_for_clustering_12 = np.concatenate((genres_pca_12,
original_language_encoded.reshape(-1, 1),
                                             production_companies_encoded,
keywords_tfidf,
                                             cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)
k_values_to_try = [3, 4, 5]
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering_12)
    silhouette_avg = silhouette_score(features_for_clustering_12, labels)
    calinski_harabasz = calinski_harabasz_score(features_for_clustering_12,
labels)
    print(f"For K = {k} with n_components = {12}, the Silhouette Score is
{silhouette_avg} and the Calinski - Harabasz Index is {calinski_harabasz}")
```

Figure 27 : Output_n_components = 12

```
For K = 3 with n_components = 12, the Silhouette Score is 0.05711840408068229 and the Calinski - Harabasz Index is 7.366527254331932
For K = 4 with n_components = 12, the Silhouette Score is 0.04723681107492681 and the Calinski - Harabasz Index is 6.198797956380378
For K = 5 with n_components = 12, the Silhouette Score is 0.042545192816081275 and the Calinski - Harabasz Index is 5.108810124277357
```

```
# Example code for exploring n_components = 16
pca_16 = PCA(n_components=16)
genres_pca_16 = pca_16.fit_transform(genres_data)
features_for_clustering_16 = np.concatenate((genres_pca_16,
original_language_encoded.reshape(-1, 1),
                                             production_companies_encoded,
keywords_tfidf,
                                             cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)
k_values_to_try = [3, 4, 5]
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering_16)
    silhouette_avg = silhouette_score(features_for_clustering_16, labels)
    calinski_harabasz = calinski_harabasz_score(features_for_clustering_16,
labels)
```

```
    print(f"For K = {k} with n_components = {16}, the Silhouette Score is
{silhouette_avg} and the Calinski - Harabasz Index is {calinski_harabasz}")
```

Figure 28 : Output_n_components = 16

```
For K = 3 with n_components = 16, the Silhouette Score is 0.05725171298519216 and the Calinski - Harabasz Index is 6.44432721772748
For K = 4 with n_components = 16, the Silhouette Score is 0.042164686476654306 and the Calinski - Harabasz Index is 5.469486874740024
For K = 5 with n_components = 16, the Silhouette Score is 0.041933532211163804 and the Calinski - Harabasz Index is 5.167813496424258
```

```
# Example code for exploring n_components = 18
pca_18 = PCA(n_components=18)
genres_pca_18 = pca_18.fit_transform(genres_data)
features_for_clustering_18 = np.concatenate((genres_pca_18,
original_language_encoded.reshape(-1, 1),
                                             production_companies_encoded,
keywords_tfidf,
                                             cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)
k_values_to_try = [3, 4, 5]
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering_18)
    silhouette_avg = silhouette_score(features_for_clustering_18, labels)
    calinski_harabasz = calinski_harabasz_score(features_for_clustering_18,
labels)
    print(f"For K = {k} with n_components = {18}, the Silhouette Score is
{silhouette_avg} and the Calinski - Harabasz Index is {calinski_harabasz}")
```

Figure 29 : Output_n_components = 18

```
For K = 3 with n_components = 18, the Silhouette Score is 0.05302212233296024 and the Calinski - Harabasz Index is 6.595530338945954
For K = 4 with n_components = 18, the Silhouette Score is 0.04166064358444136 and the Calinski - Harabasz Index is 5.177517257657455
For K = 5 with n_components = 18, the Silhouette Score is 0.03301082248139569 and the Calinski - Harabasz Index is 4.556633299748189
```

Comparing the n_components Scenarios

Overall, for K = 3, n_components = 12 seems to have the best combination of Silhouette Score and Calinski - Harabasz Index, indicating relatively better clustering quality in terms of both the compactness within clusters and the separation between clusters. However, it's important to note that the differences between some of the values are not extremely large, and other factors such as the interpretability of the clusters and the specific requirements of the movie recommendation system also need to be considered.

For higher values of K (4 and 5), the performance of different n_components values is more varied, and no clear winner emerges. The Silhouette Scores are generally lower, suggesting that increasing K might be leading to more complex and less well-defined clusters.

E. PCA n_components Analysis Summary

After exploring different values of n_components (12, 13, 14, 16, and 18) in PCA and evaluating the clustering results with K-Means for K values of 3, 4, and 5, we have the following observations:

For K = 3:

n_components = 12 shows the highest Calinski - Harabasz Index (7.366527254331932), indicating relatively better separation between clusters compared to other n_components values. The Silhouette Score (0.05711840408068229) is also reasonable.

n_components = 16 has a similar Silhouette Score (0.05725171298519216) but a lower Calinski - Harabasz Index (6.44432721772748).

n_components = 18 has a Calinski - Harabasz Index (6.595530338945954) and Silhouette Score (0.05302212233296024) that are also relatively good but not as high as for n_components = 12.

For K = 4 and K = 5: The performance of different n_components values is more inconsistent, and no single value clearly stands out as the best. The Silhouette Scores are generally lower for these K values, suggesting that increasing K might be leading to more complex and less well-defined clusters.

Overall, n_components = 12 seems to be a promising choice, especially when considering the separation between clusters for K = 3. However, the choice also depends on other factors such as the interpretability of the clusters and the specific requirements of the movie recommendation system.

4.5.2.2. Explorations

A. Revisit the keywords Column

Thoroughly review the code where the keywords were processed. Check if the column was correctly extracted, transformed (e.g., TF-IDF vectorization), and merged with the main dataset. Ensure that there are no naming or data type inconsistencies. Print out the merged_data columns before and after the relevant operations to identify the problem. If the column was dropped or renamed accidentally, correct the code to restore it.

Consider adjusting the TF-IDF parameters further. For example, try different values for min_df and max_df to see if it affects the clustering quality. You could also explore using other text preprocessing techniques such as stemming or lemmatization on the keywords before vectorization.

B. Enhance production_companies Feature

Instead of just grouping less frequent companies, create a weighted feature based on the production company's historical success. You could collect data on the box

office performance or critical acclaim of the company's previous movies and use it to assign weights to each production company in the dataset.

Explore more advanced encoding methods for production_companies. For example, you could use a frequency-based encoding where the value represents how often a production company appears in the dataset, or a binary encoding that indicates the presence or absence of a particular company in a movie's production history.

C. Improve cast and crew Features

For the cast feature, consider using a more sophisticated encoding method that takes into account the importance or popularity of each actor. For example, you could assign weights to actors based on their box office draw, awards, or the number of leading roles they have had.

Explore adding more detailed information about the crew members, such as their specific skills or experience levels. This could involve creating new features or enhancing the existing encoding to better capture the impact of the crew on the movie.

4.5.3. Explore other clustering algorithms

4.5.3.1. Hierarchical Clustering with Different Linkage Methods

Use the corrected code for Hierarchical Clustering with different linkage methods (single, average, and complete) and evaluate the clustering results. Compare the performance of Hierarchical Clustering with that of K-Means for the same set of features and n_components/K combinations. Analyze the cluster structures obtained and see if they provide more meaningful groupings for the movie data.

Consider visualizing the dendrograms generated by Hierarchical Clustering to understand the hierarchical relationships between the movies. This can help in determining the appropriate number of clusters and evaluating the stability of the clustering.

4.5.3.2. DBSCAN and Its Parameter Tuning

Explore DBSCAN as an alternative clustering algorithm. Tune the parameters eps and min_samples to find the optimal settings for the movie dataset. You can start with a grid search of different parameter values and evaluate the clustering results using appropriate metrics. For example:

```
from sklearn.cluster import DBSCAN
eps_values_to_try = [0.3, 0.5, 0.7]  # You can adjust these values
based on your data
min_samples_values_to_try = [3, 5, 7]
for eps in eps_values_to_try:
    for min_samples in min_samples_values_to_try:
```

```
        dbscan = DBSCAN(eps=eps, min_samples=min_samples)
        labels = dbscan.fit_predict(features_for_clustering_12)  # Use
features_for_clustering_12 as an example, can be changed
        # Calculate evaluation metrics for DBSCAN (note that DBSCAN may
have different evaluation considerations compared to K-Means)
        # You could calculate the number of clusters found (DBSCAN may
find different number of clusters automatically)
        n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
        print(f"For eps = {eps} and min_samples = {min_samples}, DBSCAN
found {n_clusters} clusters.")
        # You could also consider visualizing the clusters found by
DBSCAN to assess the quality (e.g., using a scatter plot or other
appropriate visualization methods)
```

Figure 30 : Output_DBSCAN

```
For eps = 0.3 and min_samples = 3, DBSCAN found 0 clusters.
For eps = 0.3 and min_samples = 5, DBSCAN found 0 clusters.
For eps = 0.3 and min_samples = 7, DBSCAN found 0 clusters.
For eps = 0.5 and min_samples = 3, DBSCAN found 0 clusters.
For eps = 0.5 and min_samples = 5, DBSCAN found 0 clusters.
For eps = 0.5 and min_samples = 7, DBSCAN found 0 clusters.
For eps = 0.7 and min_samples = 3, DBSCAN found 0 clusters.
For eps = 0.7 and min_samples = 5, DBSCAN found 0 clusters.
For eps = 0.7 and min_samples = 7, DBSCAN found 0 clusters.
```

The results obtained so far indicate that for the tested combinations of eps and min_samples values, DBSCAN is not identifying any clusters. This could be due to several reasons:

The dataset might have a relatively uniform density, and the chosen eps values are too small to capture the neighborhoods. You could try increasing the eps values further to see if it helps in identifying clusters. For example, try eps values like 1.0, 1.5, or 2.0.

The min_samples value might be too high considering the nature of the data. But we can try decreasing it to see if it allows DBSCAN to form clusters. For instance, try min_samples values of 2 or 1 (although a very low min_samples value might lead to over - clustering).

The features used for clustering might not be suitable for DBSCAN. Consider re - evaluating and potentially modifying the feature set. For example, you could try using only a subset of the features or applying different normalization techniques.

4.5.3.3.Combining Multiple Clustering Results

Consider combining the results of different clustering algorithms or different parameter settings. For example, you could use an ensemble approach where you take the consensus of multiple clustering results to obtain a more robust and accurate clustering. This could involve techniques such as cluster ensembles or voting methods.

Evaluate the combined clustering results using a combination of evaluation metrics to ensure that the final clustering provides better quality in terms of both cluster separation and compactness.

By exploring these additional aspects, we can further improve the clustering quality and potentially enhance the performance of the movie recommendation system. The goal is to find the best combination of feature engineering, clustering algorithm, and parameter settings that can effectively group movies based on their characteristics and provide useful recommendations to users.

4.6. Metrics of Success

4.6.1. Evaluation Metrics Overview

In the process of building and tuning our movie recommendation system based on clustering (using algorithms like K-Means and exploring alternatives such as Hierarchical Clustering and DBSCAN), several evaluation metrics play crucial roles in determining the success of the model. These metrics help us understand how well the clustering is performed and whether the resulting clusters are meaningful and useful for generating accurate recommendations.

Silhouette Score: This metric ranges from -1 to 1. A higher value indicates better clustering, meaning that data points within a cluster are close to each other and far from points in other clusters. For example, during our previous experiments with different values of K (the number of clusters) and n_components in PCA, we obtained various Silhouette Scores. When n_components = 12 and K = 3, the Silhouette Score was 0.05711840408068229, suggesting a somewhat reasonable clustering quality. However, as we adjusted K and n_components, the scores varied, indicating the sensitivity of the clustering quality to these parameters.

Calinski - Harabasz Index: It measures the ratio of between - cluster dispersion to within - cluster dispersion. A higher value suggests better separation between clusters. Our previous calculations showed different results for various parameter combinations. For instance, with n_components = 12 and K = 3, the Calinski - Harabasz Index was 7.366527254331932, indicating relatively good separation. But again, changes in K and n_components affected this index, highlighting the need for careful parameter tuning.

Davies - Bouldin Index: This is another useful metric that measures the average similarity between each cluster and its most similar cluster. A lower value indicates better clustering. We can calculate it as follows (using the appropriate clustering results and feature set):

```
from sklearn.metrics import davies_bouldin_score
# Calculate Davies - Bouldin Index for the current clustering (assuming
labels and features_for_clustering are available)
labels = kmeans.labels_  # Replace with the actual labels from the
clustering
davies_bouldin = davies_bouldin_score(features_for_clustering, labels)
print(f"The Davies - Bouldin Index is {davies_bouldin}")
```

4.6.2. Interpreting the Metrics for Success

Optimal K and n_components Selection: By considering the trends in the Silhouette Score, Calinski - Harabasz Index, and Davies - Bouldin Index across different values of K and n_components, we aim to find the optimal combination. For example, the results suggested that for K = 3 and n_components = 12, we had

relatively good clustering performance based on the Calinski - Harabasz Index. However, we also need to take into account the Silhouette Score and Davies - Bouldin Index. If the Davies - Bouldin Index is high or the Silhouette Score is very low for a particular combination, it indicates that there might still be issues with the clustering quality, and further adjustments might be needed.

Feature Contribution: These metrics also help us understand the contribution of different features to the clustering. If changing or adding a particular feature (such as adjusting the PCA components for the Genres feature or adding a new feature like a weighted production_companies feature) leads to an improvement in the evaluation metrics, it implies that the feature is valuable for creating distinct and meaningful clusters.

Model Improvement: Continuously monitoring and improving these metrics over different iterations of model tuning indicates the progress and success of our movie recommendation system. For example, if we can increase the Silhouette Score and the Calinski - Harabasz Index while decreasing the Davies - Bouldin Index through various adjustments like modifying clustering parameters, feature engineering, or using different clustering algorithms, it shows that our model is becoming more effective at separating movies into relevant groups for recommendations.

4.7. Show Model Work

4.7.1. Visualizing Clusters

To show how the model works and the resulting clusters, visualizations are extremely helpful. We have used two main types of visualizations:

Scatter Plot: We can plot two relevant features (after appropriate encoding or transformation) with data points colored according to their cluster labels. For example, using the first principal component of Genres (after PCA with the chosen n_components) and the Cast Popularity score:

```
# Assuming we have already clustered the data and have the labels
(replace with actual data and labels)
if 'genres_pca' in locals():
    plt.scatter(genres_pca[:, 0], merged_data['Cast Popularity'], c =
labels)
    plt.xlabel('First Principal Component of Genres (after PCA)')
    plt.ylabel('Cast Popularity')
    plt.show()
else:
    print("Error: Genres PCA not available. Please check the PCA
process.")
```

From the scatter plot, we can observe the distribution of data points across different clusters and assess the separation between them. If the points of different colors (representing different clusters) are well - separated and form distinct groups, it

indicates good clustering. However, if there is significant overlap, it suggests that further tuning of the model or features is required.

Parallel Coordinates Plot: Using the Plotly library, we can create an interactive visualization that shows how different features vary across the clusters. For instance:

```
import plotly.express as px
# Select the features for visualization (replace with actual relevant
features)
features = ['original_language', 'production_companies', 'keywords',
'Cast Popularity']
# Create a DataFrame with the selected features and cluster labels
(assuming labels are available)
df_visualization = merged_data[features + ['cluster_labels']]
# Create the parallel coordinates plot
fig = px.parallel_coordinates(df_visualization, color =
'cluster_labels')
fig.show()
```

This plot allows us to see the ranges and variations of multiple features within each cluster. If the lines for different clusters (colored differently) are clearly separated along most of the features, it indicates that the clusters are distinct in terms of those features.

4.7.2. Demonstrating Recommendations

Once the clustering is done and we have determined the optimal parameters and features, we can demonstrate how the movie recommendation system works based on these clusters. For example, if a user is interested in a movie from a particular cluster, we can recommend other movies from the same cluster that share similar characteristics in terms of the features used for clustering (such as similar genres, production companies, or cast).

Recommendation Logic: The recommendation logic could be as simple as selecting movies from the same cluster with high similarity scores based on a combination of feature similarities. For instance, if we have calculated feature vectors for each movie using the combined features for clustering, we can use distance metrics (like cosine similarity) to find the most similar movies within the cluster.

```
from sklearn.metrics.pairwise import cosine_similarity
# Assume we have feature vectors for all movies (could be based on
features_for_clustering)
movie_feature_vectors = np.array([...])  # Replace with actual feature
vectors
# Calculate cosine similarity matrix
similarity_matrix = cosine_similarity(movie_feature_vectors)
```

```
# If a user selects a movie (let's say movie_index is the index of the
selected movie)
movie_index = 0  # Replace with actual index
# Find the most similar movies within the same cluster
cluster_label = labels[movie_index]
cluster_indices = np.where(labels == cluster_label)[0]
similarity_scores = similarity_matrix[movie_index][cluster_indices]
sorted_indices = np.argsort(similarity_scores)[::-1]
recommended_indices = cluster_indices[sorted_indices[1:6]]  # Recommend
the top 5 most similar movies
recommended_movies = [... ]  # Get the actual movie details from the
dataset using the recommended indices
```

User Experience: From the user's perspective, the system can present these recommended movies in a user - friendly way, such as showing movie posters, titles, and short descriptions. This helps users understand the similarity between the recommended movies and the one they initially selected, and makes it easier for them to explore other movies they might like within the same cluster.

By effectively using evaluation metrics, visualizations, and demonstrating the recommendation process, we can showcase the work of our movie recommendation model and its ability to provide useful suggestions to users. Additionally, through the iterative process of adjusting parameters, exploring different clustering algorithms, and improving feature engineering, we can enhance the overall performance and quality of the model to better meet the needs of the movie recommendation application.

5. Conclusions

5.1. Summary

This research has delved into the development of a movie recommendation system using clustering methods. Through an extensive analysis of the movie dataset, which encompasses a rich variety of features such as genres, original language, production companies, keywords, cast, and crew, we have identified the suitability of these features for clustering. Genres, for instance, have shown great potential in grouping movies with similar thematic and stylistic elements, while the cast and crew information, when appropriately encoded, can provide insights into the creative aspects of the movies and aid in clustering.

We have explored multiple clustering algorithms, including K-Means, Hierarchical Clustering, DBSCAN, and Model-Based Clustering (GMM). Each algorithm has its own characteristics and performance trade-offs. K-Means, for example, is efficient for large datasets but is sensitive to the initial centroids and struggles in high dimensions. Hierarchical Clustering reveals the data structure without requiring a preset cluster number, yet it is computationally costly for big data. DBSCAN can handle noise and find clusters of arbitrary shapes but is sensitive to parameters and weak in varying density datasets. GMM can uncover hidden structures in the data but needs complex parameter estimation and is computationally intense.

In the process of parameter tuning for K-Means, we have observed the impact of different features on the clustering process. The one-hot encoding of genres, label and one-hot encoding of original language, and TF-IDF vectorization of keywords are among the techniques used to transform these features for clustering. The Elbow Method and Silhouette Score have been employed to determine the optimal number of clusters, and different initialization methods have been explored to improve the clustering results.

5.2. Reflection

The clustering-based recommendation system shows promise in enhancing the accuracy and personalization of movie recommendations. By grouping similar movies and users, it can potentially overcome some of the limitations of traditional recommendation systems, such as data sparsity and the cold start problem. The ability to capture the inherent characteristics of movies and user preferences through clustering can lead to more relevant and diverse recommendations. However, the effectiveness of the system is highly dependent on the proper selection and tuning of clustering algorithms and features. The visualizations of clusters, such as scatter plots and parallel coordinates plots, have provided insights into the distribution and separation of clusters, but also highlighted the need for

further improvement in cluster distinctiveness.

5.3. Limitations of the Study and Suggestions for Future Research

Despite the progress made, this study has several limitations. The dataset, although comprehensive, may not fully represent the entire movie universe, and there could be biases or limitations in the data that affect the clustering results. The feature engineering process, while extensive, may not have captured all the relevant aspects of movies, and further exploration of additional features or more sophisticated encoding methods could be beneficial. For example, the "keywords" feature could be further enhanced by using more advanced natural language processing techniques to extract more meaningful semantic information.

In terms of clustering algorithms, the performance of DBSCAN was not satisfactory in the initial exploration, suggesting that further parameter tuning and feature selection are required. Additionally, other clustering algorithms or combinations of algorithms could be explored to potentially improve the clustering quality. The evaluation metrics used, while providing valuable insights, may not fully capture the user experience and the quality of recommendations in all aspects. Future research could consider incorporating user feedback and behavior data to develop more comprehensive evaluation methods.

Overall, this research lays the foundation for further exploration and improvement of clustering-based movie recommendation systems. Future work could focus on addressing the limitations identified, exploring new algorithms and features, and conducting more extensive user studies to validate the effectiveness and usability of the system in real-world scenarios. This would contribute to the development of more accurate, personalized, and user-friendly movie recommendation systems in the future.

6. References

[1] LeCun,Y.,Bengio,Y.,&Hinton,G.(2015).Deep learning.Nature,521(7553),436-444.

[2] Alpaydin,E.(2004).Introduction to machine learning.The MIT Press.

[3] Brown,G.,Pocock,A.,Zhao,M.J.,&Luján,M.(2012).Conditional likelihood maximisation:A unifying framework for information theoretic feature selection.Journal of Machine Learning Research,13(1),27-66.

[4] Esteva,A.,Kuprel,B.,Novoa,R.A.,Ko,J.,Swetter,S.M.,Blau,H.M.,&Thrun,S.(2017).Dermatologist-level classification of skin cancer with deep neural networks.Nature,542(7639),115-118.

[5] Bojarski,M.,Del Testa,D.,Dworakowski,D.,Firner,B.,Flepp,B.,Goyal,P.,...&Zhang,J.(2016).End to end learning for self-driving cars.arXiv preprint arXiv:1604.07316.

[6] Adomavicius,G.,&Tuzhilin,A.(2011).Context-aware recommender systems.In Recommender Systems Handbook(pp.217-253).Springer,Boston,MA.

[7] Pazzani,M.J.,&Billsus,D.(2007).Content-based recommendation systems.In The Adaptive Web(pp.325-341).Springer Berlin Heidelberg.

[8] Porter,A.L.,Cohen,A.S.,&Roush,G.(1985).A guide to competitive technical intelligence for scientists and engineers.Journal of the American Society for Information Science,36(6),387-393.

[9] Sarwar,B.,Karypis,G.,Konstan,J.,&Riedl,J.(2001).Item-based collaborative filtering recommendation algorithms.In Proceedings of the 10th international conference on World Wide Web(pp.285-295).ACM.

[10] Burke,R.(2002).Hybrid recommender systems:Survey and experiments.User Modeling and User-Adapted Interaction,12(4),331-370.

[11] Zhang,Y.,&Callan,J.(2011).Maximum marginal relevance for mixed-initiative query-focused multi-document summarization.Information Processing&Management,47(2),270-286.

[12] Zhang,M.,Wang,W.,&Li,X.(2008).A paper recommender for scientific literatures.

[13] Linden,G.,Smith,B.,&York,J.(2003).Amazon.com recommendations:Item-to-item collaborative filtering.IEEE Internet Computing,7(1),76-80.

[14] Adomavicius,G.,&Tuzhilin,A.(2005).Toward the next generation of recommender systems:A survey of the state-of-the-art and possible extensions.IEEE Transactions on Knowledge and Data Engineering,17(6),734-749.

[15] Breese,J.S.,Heckerman,D.,&Kadie,C.(1998).Empirical analysis of predictive algorithms for collaborative filtering.In Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence(pp.43-52).Morgan Kaufmann Publishers Inc.

[16] Su,X.,&Khoshgoftaar,T.M.(2009).A survey of collaborative filtering

techniques.Advances in Artificial Intelligence,2009.AAAI'09.3rd Conference on,4-9.
[17] Ricci,F.,Rokach,L.,&Shapira,B.(2011).Introduction to Recommender Systems Handbook.In F.Ricci,L.Rokach,&B.Shapira(Eds.),Recommender Systems Handbook(pp.1-35).Springer US.
[18] Han,J.,Kamber,M.,&Pei,J.(2011).Data Mining:Concepts and Techniques(3rd ed.).Morgan Kaufmann.
[19] Xu,Z.,&Li,H.(2018).A clustering-based approach to enhancing the accuracy of recommender systems.Information,9(4),97.
[20] Zhao,T.,&Karypis,G.(20
[21] 04).Criterion functions for document clustering:Experiments and analysis.In Proceedings of the 2004 ACM CIKM international conference on Information and knowledge management(pp.469-476).ACM.
[22] Strehl,A.,&Ghosh,J.(2002).Cluster ensembles—a knowledge reuse framework for combining multiple partitions.Journal of Machine Learning Research,3(Dec),583-617.
[23] Wang,B.,&Zhang,S.(2018).A clustering-based method for improving the accuracy of recommender systems.In 2018 International Conference on Information Technology and Computer Science(ITCS),2018(pp.1-5).IEEE.
[24] Aggarwal,C.C.(2015).Data clustering:Algorithms and applications.In Data classification:algorithms and applications(pp.289-308).CRC Press.
[25] Ester,M.,Kriegel,H.P.,Sander,J.,Xu,X.,&Zhao,W.(2018).A density-based algorithm for discovering clusters in large spatial databases with noise.In Proceedings of the 2nd international conference on knowledge discovery and data mining(KDD-96)(pp.226-231).
[26] Agrawal,R.,&Srikant,R.(1994).Fast algorithms for mining association rules in large databases.In Proceedings of the 20th international conference on very large data bases,Vol.1215(pp.487-499).
[27] Milligan,G.W.,&Cooper,M.C.(1985).An examination of procedures for determining the number of clusters in a data set.Psychometrika,50(2),159-179.
[28] [28] Alyari, F.; Navimipour, N.J. Recommender systems: A systematic review of the state of the art literature and suggestions for future research. Kybernetes , 2018, 47, 985.
[29] [29] Caro-Martinez, M.; Jimenez-Diaz, G.; Recio-Garcia, J.A. A theoretical model of explanations in recommender systems. In Proceedings of the ICCBR , Stockholm, Sweden, 9–12 July 2018.
[30] [30] Aggarwal, C.C. An Introduction to Recommender Systems. In Recommender Systems ; Springer: Berlin/Heidelberg, Germany, 2016; pp. 1–28.
[31] [31] Deldjoo, Y.; Elahi, M.; Cremonesi, P.; Garzotto, F.; Piazzolla, P.; Quadrana, M. Content-Based Video Recommendation System Based on Stylistic Visual Features. J. Data Semant. , 2016, 5, 99–113.
[32] [32] Ghazanfar, M.A.; Prugel-Bennett, A. A scalable, accurate hybrid

recommender system. In Proceedings of the 2010 Third International Conference on Knowledge Discovery and Data Mining , Washington, DC, USA, 9–10 January 2010.
[33] [33] Schafer, J.B.; Konstan, J.A.; Riedl, J. E-commerce recommendation applications. Data Min. Knowl. Discov. , 2001, 5, 115–153.
[34] [34] Herlocker, J.L.; Konstan, J.A.; Terveen, L.G.; Riedl, J.T. Evaluating collaborative filtering recommender systems. ACM Trans. Inf. Syst. , 2004, 22, 5–53.
[35] [35] Koren, Y.; Bell, R.; Volinsky, C. Matrix factorization techniques for recommender systems. Computer , 2009, 42, 30–37.
[36] [36] Çano, E.; Morisio, M. Hybrid recommender systems: A systematic literature review. Intell. Data Anal. , 2017, 21, 1487–1524.
[37] [37] Abdulla, G.M.; Borar, S. Size recommendation system for fashion e-commerce. In Proceedings of the KDD Workshop on Machine Learning Meets Fashion , Halifax, NS, Canada, 14 August 2017.
[38] [38] Guy, I.; Zwerdling, N.; Ronen, I.; Carmel, D.; Uziel, E. Social media recommendation based on people and tags. In Proceedings of the 33rd International ACM SIGIR Conference on Research and Development in Information Retrieval , Geneva, Switzerland, 19–23 July 2010.
[39] [39] Katarya, R.; Verma, O.P. An effective collaborative movie recommender system with cuckoo search. Egypt. Inform. J. , 2017, 18, 105–112.
[40] [40] Kumar, B.; Sharma, N. Approaches, Issues and Challenges in Recommender Systems: A Systematic Review. Indian J. Sci. Technol. , 2016, 9, 1–12.
[41] Chen, M.; Liu, P. Performance evaluation of recommender systems. Int. J. Perform. Eng. , 2017, 13, 1246.
[42] Jena, K.K.; Bhoi, S.K.; Mallick, C.; et al. Neural model based collaborative filtering for movie recommendation system. Int. J. Inf. Tecnol. 2022, 14, 2067–2077.
[43] Behera, G.; Nain, N. DeepNNMF: Deep nonlinear non-negative matrix factorization to address sparsity problem of collaborative recommender system. Int. J. Inf. Tecnol. 2022, 14, 3637–3645.
[44] Choudhury, S.S.; Mohanty, S.N.; Jagadev, A.K. Multimodal trust-based recommender system with machine learning approaches for movie recommendation. Int. J. Inf. Tecnol. 2021, 13, 475–482.
[45] Patel, R.; Thakkar, P. Addressing Item Cold Start Problem in Collaborative Filtering-Based Recommender Systems Using Auxiliary Information. In IoT with Smart Systems: Smart Innovation, Systems and Technologies ; Choudrie, J., Mahalle, P., Perumal, T., Joshi, A. Eds.; Springer: Singapore, 2023; Vol. 312.
[46] Tr, M.; Vinoth Kumar, V.; Lim, S.J. UsCoTc: Improved Collaborative Filtering (CFL) recommendation methodology using user confidence, time context with impact factors for performance enhancement. PLoS One 2023, 18,

e0282904.
[47] Ricci, F., Rokach, L., & Shapira, B. (2015). Introduction to recommender systems handbook. In Recommender Systems Handbook (pp. 1-35). Springer, Cham. https://doi.org/10.1007/978-3-319-29659-3_1
[48] Jain, A. K. (2010). Data clustering: 50 years beyond K-means. Pattern Recognition Letters, 31(8), 651-666.
[49] Murtagh, F., & Contreras, P. (2012). Algorithms for hierarchical clustering: An overview. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 2(1), 86-97.
[50] Ester, M., Kriegel, H. P., Sander, J., & Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (pp. 226-231). AAAI Press.
[51] Nagesh, H., Goil, S., & Choudhary, A. (2001). Adaptive grids for clustering massive data sets. In Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (pp. 371-382). ACM.
[52] McLachlan, G. J., & Peel, D. (2000). Finite mixture models. Wiley series in probability and statistics. John Wiley & Sons.

7. Appendix

```
import os
os.environ["OMP_NUM_THREADS"] = '1'
import pandas as pd
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

movies_df = pd.read_csv('tmdb_5000_movies.csv')
credits_df = pd.read_csv('tmdb_5000_credits.csv')

mad_movies_df = movies_df.head( )
mad_movies_df

mad_credits_df = credits_df.head()
mad_credits_df

merged_data = pd.merge(movies_df, credits_df, left_on='id',
right_on='movie_id', suffixes=('', '_credits'))
merged_data.drop('movie_id', axis=1, inplace=True)
merged_data.drop('title_credits', axis=1, inplace=True)
mad_merged_data = merged_data.head()
mad_merged_data

# Check the shape of the dataset (number of rows and columns)
print(merged_data.shape)

# Examine the data types and the number of non-null values
print(merged_data.info())

print(merged_data.columns)

# Count the distribution of movie genres
genres_distribution = merged_data['genres'].value_counts().reset_index()
genres_distribution.columns = ['genres', 'Count']
display(genres_distribution)
```

```
# Count the distribution of original languages
original_language_distribution =
merged_data['original_language'].value_counts().reset_index()
original_language_distribution.columns = ['Original Language', 'Count']
display(original_language_distribution)

# Count the distribution of production companies
production_companies_distribution =
merged_data['production_companies'].value_counts().reset_index()
production_companies_distribution.columns = ['Production Companies',
'Count']
display(production_companies_distribution)

# Count the distribution of keywords
keywords_distribution =
merged_data['keywords'].value_counts().reset_index()
keywords_distribution.columns = ['Keywords', 'Count']
display(keywords_distribution)

# First, convert the string data in the cast column to a list format
cast_lists = merged_data['cast'].apply(lambda x: eval(x))
# Count the frequency of actors
cast_frequency = {}
for cast_list in cast_lists:
    for actor_dict in cast_list:
        # Use the actor's name as the unique entity
        actor_name = actor_dict.get('name', None)
        if actor_name is not None:
            if actor_name in cast_frequency:
                cast_frequency[actor_name] += 1
            else:
                cast_frequency[actor_name] = 1
# Convert the actor frequency dictionary to a DataFrame
cast_distribution = pd.DataFrame(list(cast_frequency.items()),
columns=['Actor', 'Count'])
cast_distribution = cast_distribution.sort_values(by='Count',
ascending=False)
display(cast_distribution)

# Check if the columns exist in the dataframe before attempting to drop
columns_to_delete = ['homepage', 'release_date','status']
```

```
# delete columns only if they exist
columns_to_delete = [col for col in columns_to_delete if col in
merged_data.columns]

# Drop the irrelevant columns from the merged_data dataframe
merged_data = merged_data.drop(columns_to_delete, axis=1)

print(merged_data.shape)
mad_movies_df

# Check for missing values
print(merged_data.isnull().sum())

# Delete rows with missing values in the 'overview' column
merged_data.dropna(subset=['overview'], inplace=True)

# Check for missing values again
print(merged_data.isnull().sum())

Text Feature Processing

# Extract all the unique movie genres
all_genres = set()
for genres_str in merged_data['genres']:
    print("Genres str:", genres_str)
    if isinstance(genres_str, str):  # Added this check to handle non-
string values like NaN
        genres_list = eval(genres_str)
        for genre in genres_list:
            all_genres.add(genre['name'])

# Create a genre matrix with one-hot encoding
genre_matrix = pd.DataFrame(0, index=merged_data.index,
columns=list(all_genres))
for i, genres_str in enumerate(merged_data['genres'].tolist()):
    if isinstance(genres_str, str):
        genres_list = eval(genres_str)
        for genre in genres_list:
            print("Processing genre:", genre['name'])
            genre_matrix.at[merged_data.index[i], genre['name']] = 1

# Reset index before merging to ensure consistency
```

```
merged_data.reset_index(drop=True, inplace=True)
genre_matrix.reset_index(drop=True, inplace=True)

# Merge the one-hot encoded genre matrix with the original data
merged_data = pd.concat([merged_data, genre_matrix], axis=1)

# Extract all the unique keywords
all_keywords = set()
for keywords_str in merged_data['keywords']:
    if isinstance(keywords_str, str):  # Checks if it is a string
        keywords_list = eval(keywords_str)
        for keyword in keywords_list:
            all_keywords.add(keyword['name'])

# Create a keyword matrix with one-hot encoding
keyword_matrix      =      pd.DataFrame(0,      index=merged_data.index,
columns=list(all_keywords))
for i, keywords_str in enumerate(merged_data['keywords']):
    if isinstance(keywords_str, str):  # Checks if it is a string
        keywords_list = eval(keywords_str)
        for keyword in keywords_list:
            keyword_matrix.at[i, keyword['name']] = 1

# Add an assert statement here to check for index consistency, and raise
a specified error message if they are inconsistent.
assert merged_data.index.equals(genre_matrix.index)

# Merge the one-hot encoded keyword matrix with the original data
merged_data = pd.concat([merged_data, keyword_matrix], axis=1)

def clean_original_language(text):
    if isinstance(text, str):
        return text.strip()
    return text

merged_data['original_language']                                    =
merged_data['original_language'].apply(clean_original_language)

def extract_company_names(companies_str):
    if isinstance(companies_str, str):
        companies_list = eval(companies_str)
        return [company['name'] for company in companies_list]
```

```
    return []

def clean_production_companies(company_names):
    cleaned_names = []
    for name in company_names:
        cleaned_name    =    name.lower().replace(',',    '').replace('.',
'').strip()
        cleaned_names.append(cleaned_name)
    return cleaned_names

merged_data['production_companies']                                    =
merged_data['production_companies'].apply(extract_company_names)
merged_data['production_companies']                                    =
merged_data['production_companies'].apply(clean_production_companies)

import nltk

from nltk.corpus import stopwords
nltk.download('stopwords')
stop_words = set(stopwords.words('english'))

def extract_cast_names(cast_str):
    if isinstance(cast_str, str):
        cast_list = eval(cast_str)
        return [actor['name'] for actor in cast_list]
    return []

def clean_cast_text(text):
    if isinstance(text, str):
        words = text.split()
        filtered_words = [word.lower() for word in words if word.lower()
not in stop_words and word.isalpha()]
        return ' '.join(filtered_words)
    return text

merged_data['cast'] = merged_data['cast'].apply(extract_cast_names)
merged_data['cast'] = merged_data['cast'].apply(clean_cast_text)

def extract_crew_names(crew_str):
    if isinstance(crew_str, str):
        crew_list = eval(crew_str)
        return [worker['name'] for worker in crew_list]
```

```
    return []

def clean_crew_text(text):
    if isinstance(text, str):
        words = text.split()
        filtered_words = [word.lower() for word in words if word.lower()
not in stop_words and word.isalpha()]
        return''.join(filtered_words)
    return text

merged_data['crew'] = merged_data['crew'].apply(extract_crew_names)
merged_data['crew'] = merged_data['crew'].apply(clean_crew_text)


merged_data.isnull().sum()

# Display the dataframe
display(merged_data)

Further Data Preprocessing Checks and Improvements

# Convert lists in 'production_companies', 'cast', and 'crew' columns to
strings
merged_data['production_companies']                                    =
merged_data['production_companies'].astype(str)
merged_data['cast'] = merged_data['cast'].astype(str)
merged_data['crew'] = merged_data['crew'].astype(str)

# Check if there are duplicate rows in the dataframe.
num_duplicates = merged_data.duplicated().sum()
if num_duplicates > 0:
    print(f"There are {num_duplicates} duplicate rows in the dataset.")
    # If there are duplicate rows, in the clustering analysis scenario,
they can usually be directly deleted.
    merged_data.drop_duplicates(inplace=True)
    print("Duplicate rows have been removed.")
else:
    print("No duplicate rows found in the dataset.")

# The describe() method provides basic statistical information for the
numerical columns of the dataframe, such as count, mean, standard
deviation, minimum value, and maximum value. This helps to comprehensively
```

```
understand the data distribution range and dispersion degree, and thus
detect potential outliers.
description = merged_data.describe()
print(description)
# Suppose there is a numerical feature column named 'numeric_feature' in
the dataset (it depends on your actual data). The following uses the
commonly used "3-sigma rule" in statistics to identify outliers.
numeric_feature_column = "numeric_feature"
if numeric_feature_column in merged_data.columns:
    # Extract the mean and standard deviation of this column from the
descriptive statistics results.
    mean_value = description.loc['mean', numeric_feature_column]
    std_value = description.loc['std', numeric_feature_column]
    # Calculate the lower and upper bounds of normal data according to
the "3-sigma rule".
    lower_bound = mean_value - 3 * std_value
    upper_bound = mean_value + 3 * std_value
    # Filter out the data rows outside this range to construct the outlier
dataset.
    outliers    =    merged_data[(merged_data[numeric_feature_column]    <
lower_bound) | (merged_data[numeric_feature_column] > upper_bound)]
    num_outliers = len(outliers)
    if num_outliers <= 0:
        print(f"No outliers found in column '{numeric_feature_column}'.")
    else:
        print(f"There    are    {num_outliers}    outliers    in    column
'{numeric_feature_column}'.")
        # Here, we just display the outlier situation. In actual business,
you can decide how to handle them according to your needs, such as deleting
them (be cautious as it may cause the loss of important special
information), correcting them (manually adjusting according to business
logic or using algorithms to estimate and correct), etc.
        print(outliers)


print(merged_data.columns)

genres_data = genre_matrix

# Check if there is data in the Genres columns
if genres_data.empty:
    print("Error: Genres data is empty. Please check the one-hot encoding
```

```
process.")
else:
    # Handling of missing data
    genres_data = genres_data.fillna(0)

    # Convert string based cells
    genres_data = genres_data.apply(pd.to_numeric, errors='coerce')

    # Applying PCA for dimensionality reduction
    from sklearn.decomposition import PCA
    pca = PCA(n_components=0.95)  # Retaining 95% of the variance
    try:
        genres_pca = pca.fit_transform(genres_data)
        print("PCA has been successfully applied on the genre data.")
    except ValueError as e:
        print(f"PCA error: {e}")

# Label encoding for Original Language
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
original_language_encoded                                              =
label_encoder.fit_transform(merged_data['original_language'])
print("Label Encoding for 'original_language' column has been successfully
performed.")

# One-hot encoding for Original Language
from sklearn.preprocessing import OneHotEncoder
onehot_encoder = OneHotEncoder()
original_language_onehot                                               =
onehot_encoder.fit_transform(merged_data[['original_language']])
print("OneHot  Encoding  for  'original_language'  column  has  been
successfully performed.")

# One-hot encoding for Production Companies
production_companies_encoded                                           =
onehot_encoder.fit_transform(merged_data[['production_companies']])
print("OneHot  Encoding  for  'production_companies'  column  has  been
successfully performed.")

# Grouping less frequent production companies
production_company_counts                                              =
merged_data['production_companies'].value_counts()
```

```
less_frequent_companies = production_company_counts[production_company_counts < 10].index
merged_data['production_companies'] = merged_data['production_companies'].apply(lambda x: 'Other' if x in less_frequent_companies else x)

print("Grouping less frequent production companies has been completed successfully.")

# Verifying the encoding process
less_frequent_companies_encoded = set(merged_data['production_companies'])
if 'Other' in less_frequent_companies_encoded:
    print("Encoding for less frequent production companies has been done successfully.")
else:
    print("Less frequent production companies are not encoded successfully, please check the code.")

# TF-IDF vectorization for Keywords
try:
    tfidf_vectorizer = TfidfVectorizer(min_df=0.01, max_df=0.9)  # Adjusting min and max document frequencies
    keywords_tfidf = tfidf_vectorizer.fit_transform(merged_data['keywords'])
    print("TF-IDF Vectorization completed successfully.")
except Exception as e:
    print(f"Error occurred: {e}")

# One-hot encoding for Cast (simplified example)
cast_lists = merged_data['cast'].apply(eval)  # Assume the data is in a format that can be directly evaluated as a list
all_actors = set([actor for sublist in cast_lists for actor in sublist])
actor_encoding = {actor: i for i, actor in enumerate(all_actors)}
cast_encoded_matrix = np.zeros((len(merged_data), len(all_actors)))

for i, cast_list in enumerate(cast_lists):
    for actor in cast_list:  # now we directly iterate over elements in 'cast_list'
        cast_encoded_matrix[i, actor_encoding[actor]] = 1
print("One-hot encoding for Cast has been successfully completed.")
```

```
# Creating a cast popularity score
actor_movie_counts = {}
for cast_list in cast_lists:
    for actor in cast_list:
        if actor in actor_movie_counts:
            actor_movie_counts[actor] += 1
        else:
            actor_movie_counts[actor] = 1

merged_data['Cast Popularity'] = merged_data['cast'].apply(lambda x:
sum([actor_movie_counts[actor] for actor in x]) if isinstance(x, list)
else 0)

print("Creation of Cast Popularity score has been successfully
completed.")

from sklearn.cluster import KMeans
def calculate_wcss(data, k):
    kmeans = KMeans(n_clusters=k)
    kmeans.fit(data)
    return kmeans.inertia_

# Generate a range of K values to test
k_values = range(1, 15)   # You can adjust the range based on your
understanding of the data

# One-hot encoding for Original Language
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
original_language_encoded                                          =
label_encoder.fit_transform(merged_data['original_language'])

# One-hot encoding for Production Companies
from sklearn.preprocessing import OneHotEncoder
onehot_encoder = OneHotEncoder()
production_companies_encoded                                       =
onehot_encoder.fit_transform(merged_data[['production_companies']])

# TF-IDF vectorization for Keywords
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer(min_df=0.01, max_df=0.9)  # Adjusting
min and max document frequencies
```

```
keywords_tfidf = tfidf_vectorizer.fit_transform(merged_data['keywords'])

# One-hot encoding for Cast (simplified example)
cast_lists = merged_data['cast'].apply(eval)  # Assume the data is in a
format that can be directly evaluated as a list
all_actors = set([actor for sublist in cast_lists for actor in sublist])
actor_encoding = {actor: i for i, actor in enumerate(all_actors)}
cast_encoded_matrix = np.zeros((len(merged_data), len(all_actors)))

for i, cast_list in enumerate(cast_lists):
    for actor in cast_list:  # now we directly iterate over elements in
'cast_list'
        cast_encoded_matrix[i, actor_encoding[actor]] = 1

# Creating a cast popularity score
actor_movie_counts = {}
for cast_list in cast_lists:
    for actor in cast_list:
        if actor in actor_movie_counts:
            actor_movie_counts[actor] += 1
        else:
            actor_movie_counts[actor] = 1

merged_data['Cast Popularity'] = merged_data['cast'].apply(lambda x:
sum([actor_movie_counts[actor] for actor in x]) if isinstance(x, list)
else 0)

# Combine the relevant features for clustering (assuming Genres has been
one-hot encoded)
try:
    genres_columns = [col for col in merged_data.columns if
col.startswith('genres_')]
    genres_data = genre_matrix
    # Handling of missing data
    genres_data = genres_data.fillna(0)
    # Convert string based cells
    genres_data = genres_data.apply(pd.to_numeric, errors='coerce')
    # Applying PCA for dimensionality reduction
    from sklearn.decomposition import PCA
    pca = PCA(n_components=0.95)  # Retaining 95% of the variance
    genres_pca = pca.fit_transform(genres_data)
    print("PCA has been successfully applied on the genre data.")
```

```
    features_for_clustering = np.concatenate((genres_pca,
original_language_encoded.reshape(-1, 1),
production_companies_encoded.toarray(), keywords_tfidf.toarray(),
cast_encoded_matrix, merged_data[['Cast Popularity']].values), axis=1)
except ValueError as e:
    print(f"PCA error: {e}")
    print("Error: Genres PCA not available. Please check the PCA
process.")
    features_for_clustering =
np.concatenate((original_language_encoded.reshape(-1, 1),
production_companies_encoded.toarray(), keywords_tfidf.toarray(),
cast_encoded_matrix, merged_data[['Cast Popularity']].values), axis=1)

wcss = [calculate_wcss(features_for_clustering, k) for k in k_values]

import matplotlib.pyplot as plt
plt.plot(k_values, wcss)
plt.xlabel('Number of Clusters (K)')
plt.ylabel('Within-Cluster Sum of Squares (WCSS)')
plt.show()

from sklearn.metrics import silhouette_score

# Start the loop from k=2 to avoid the error
for k in k_values[1:]:  # Start from the second element of k_values (k=2)
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering)
    silhouette_avg = silhouette_score(features_for_clustering, labels)
    print(f"For K = {k}, the Silhouette Score is {silhouette_avg}")

from sklearn.metrics import calinski_harabasz_score
# Start the loop from k=2 to avoid the error
for k in k_values[1:]:  # Start from the second element of k_values (k=2)
    # Corrected the init parameter to 'k-means++' (removed the extra space)
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering)
    calinski_harabasz = calinski_harabasz_score(features_for_clustering,
labels)
    print(f"For K = {k}, the Calinski - Harabasz Index is
{calinski_harabasz}")

if 'genres_pca' in locals():
```

```
    plt.scatter(genres_pca[:,    0],    merged_data['Cast    Popularity'],
c=labels)
    plt.xlabel('First Principal Component of Genres (after PCA)')
    plt.ylabel('Cast Popularity')
    plt.show()
else:
    print("Error:  Genres  PCA  not  available.  Please  check  the  PCA
process.")

import plotly.express as px

# Select the features for visualization
features  =  ['original_language',  'production_companies',  'keywords',
'Cast Popularity']

# Assuming 'labels' from your KMeans clustering is available:
merged_data['cluster_labels'] = labels  # Assign cluster labels to a new
column

# Create a DataFrame with the selected features and cluster labels
df_visualization = merged_data[features + ['cluster_labels']]

# Create the parallel coordinates plot
fig = px.parallel_coordinates(df_visualization, color='cluster_labels')
fig.show()

# Re - apply PCA with different n_components
from sklearn.decomposition import PCA
# Try different values for n_components, e.g., 0.90
pca = PCA(n_components=0.90)
genres_pca = pca.fit_transform(genres_data)

# Output reminder:
print("Shape of transformed data after PCA:", genres_pca.shape)

# Trying different values of K around the potentially good range (e.g.,
around 4)
k_values_to_try = [3, 4, 5]

for k in k_values_to_try:
    # Apply KMeans with the current k value and k-means++ initialization
    kmeans = KMeans(n_clusters=k, init='k-means++', random_state=42)   #
```

```
Added random_state for reproducibility
    labels = kmeans.fit_predict(features_for_clustering)

    # Calculate evaluation metrics
    silhouette_avg = silhouette_score(features_for_clustering, labels)
    calinski_harabasz = calinski_harabasz_score(features_for_clustering,
labels)

    # Print the results for the current k value
    print(f"For K = {k}, the Silhouette Score is {silhouette_avg:.3f} and
the Calinski-Harabasz Index is {calinski_harabasz:.3f}")

from sklearn.decomposition import PCA, IncrementalPCA
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, calinski_harabasz_score

# Assume this is your data for genres, you need to replace it with your
actual data
genres_data = np.random.rand(100, 20)  # Just a sample data with 100 rows
and 20 features for demonstration

# Calculate the number of components to explain 85% of the variance using
PCA for the first case
pca_085 = PCA(n_components=0.85)
pca_085.fit(genres_data)
n_components_085 = np.sum(pca_085.explained_variance_ratio_ > 0)

# Now use the calculated integer number of components with IncrementalPCA
for the first case
ipca_085 = IncrementalPCA(n_components=n_components_085)
genres_ipca_085 = ipca_085.fit_transform(genres_data)

# Assume these are your encoded features for other aspects. You need to
replace them with your actual data
original_language_encoded = np.random.randint(0, 2, size=(100, 1))   #
Sample encoded data for original language
production_companies_encoded = np.random.rand(100, 3)  # Sample encoded
data for production companies
keywords_tfidf = np.random.rand(100, 4)  # Sample TF-IDF data for keywords
cast_encoded_matrix = np.random.rand(100, 6)  # Sample encoded data for
cast
merged_data = pd.DataFrame({'Cast Popularity': np.random.rand(100)})   #
```

```
Sample data for merged_data

# Re - combine features for clustering for the first case
features_for_clustering_085 = np.concatenate((genres_ipca_085,
original_language_encoded.reshape(-1, 1),

production_companies_encoded, keywords_tfidf,
                                                cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)

# Calculate evaluation metrics for different values of K for the first
case
k_values_to_try = [3, 4, 5]
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering_085)
    silhouette_avg = silhouette_score(features_for_clustering_085,
labels)
    calinski_harabasz                                                =
calinski_harabasz_score(features_for_clustering_085, labels)
    print(f"For K = {k} with n_components = {n_components_085}, the
Silhouette Score is {silhouette_avg} and the Calinski - Harabasz Index is
{calinski_harabasz}")

# Calculate the number of components to explain 92% of the variance using
PCA for the second case
pca_092 = PCA(n_components=0.92)
pca_092.fit(genres_data)
n_components_092 = np.sum(pca_092.explained_variance_ratio_ > 0)

# Now use the calculated integer number of components with IncrementalPCA
for the second case
ipca_092 = IncrementalPCA(n_components=n_components_092)
genres_ipca_092 = ipca_092.fit_transform(genres_data)

# Re - combine features for clustering for the second case
features_for_clustering_092 = np.concatenate((genres_ipca_092,
original_language_encoded.reshape(-1, 1),

production_companies_encoded, keywords_tfidf,
                                                cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)
```

```
# Calculate evaluation metrics for different values of K for the second
case
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering_092)
    silhouette_avg    =    silhouette_score(features_for_clustering_092,
labels)
    calinski_harabasz                                                    =
calinski_harabasz_score(features_for_clustering_092, labels)
    print(f"For  K  =  {k}  with  n_components  =  {n_components_092},  the
Silhouette Score is {silhouette_avg} and the Calinski - Harabasz Index is
{calinski_harabasz}")

from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import silhouette_score, calinski_harabasz_score

# Use Hierarchical Clustering, for example, with Euclidean distance and
ward linkage
for k in k_values_to_try:
    hierarchical_clustering  =  AgglomerativeClustering(n_clusters  =  k,
linkage = 'ward')
    labels = hierarchical_clustering.fit_predict(features_for_clustering)
    silhouette_avg = silhouette_score(features_for_clustering, labels)
    calinski_harabasz = calinski_harabasz_score(features_for_clustering,
labels)
    print(f"For K = {k} with Hierarchical Clustering, the Silhouette Score
is   {silhouette_avg}   and   the   Calinski   -   Harabasz   Index   is
{calinski_harabasz}")

# Example code for exploring n_components = 13
pca_13 = PCA(n_components=13)
genres_pca_13 = pca_13.fit_transform(genres_data)
features_for_clustering_13        =        np.concatenate((genres_pca_13,
original_language_encoded.reshape(-1, 1),

production_companies_encoded, keywords_tfidf,
                                                cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)
k_values_to_try = [3, 4, 5]  # Use the correct variable name here
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
```

```
    labels = kmeans.fit_predict(features_for_clustering_13)
    silhouette_avg = silhouette_score(features_for_clustering_13, labels)
    calinski_harabasz                                                    =
calinski_harabasz_score(features_for_clustering_13, labels)
    print(f"For K = {k} with n_components = {13}, the Silhouette Score is
{silhouette_avg}    and    the    Calinski    -    Harabasz    Index    is
{calinski_harabasz}")

# Example code for exploring n_components = 14
pca_14 = PCA(n_components=14)
genres_pca_14 = pca_14.fit_transform(genres_data)
features_for_clustering_14        =        np.concatenate((genres_pca_14,
original_language_encoded.reshape(-1, 1),

production_companies_encoded, keywords_tfidf,
                                                cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)
k_values_to_try = [3, 4, 5]
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering_14)
    silhouette_avg = silhouette_score(features_for_clustering_14, labels)
    calinski_harabasz                                                    =
calinski_harabasz_score(features_for_clustering_14, labels)
    print(f"For K = {k} with n_components = {14}, the Silhouette Score is
{silhouette_avg}    and    the    Calinski    -    Harabasz    Index    is
{calinski_harabasz}")

# Example code for exploring n_components = 12
pca_12 = PCA(n_components=12)
genres_pca_12 = pca_12.fit_transform(genres_data)
features_for_clustering_12        =        np.concatenate((genres_pca_12,
original_language_encoded.reshape(-1, 1),

production_companies_encoded, keywords_tfidf,
                                                cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)
k_values_to_try = [3, 4, 5]
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering_12)
    silhouette_avg = silhouette_score(features_for_clustering_12, labels)
```

```
    calinski_harabasz =
calinski_harabasz_score(features_for_clustering_12, labels)
    print(f"For K = {k} with n_components = {12}, the Silhouette Score is
{silhouette_avg}    and    the    Calinski    -    Harabasz    Index    is
{calinski_harabasz}")

# Example code for exploring n_components = 16
pca_16 = PCA(n_components=16)
genres_pca_16 = pca_16.fit_transform(genres_data)
features_for_clustering_16        =        np.concatenate((genres_pca_16,
original_language_encoded.reshape(-1, 1),

production_companies_encoded, keywords_tfidf,
                                          cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)
k_values_to_try = [3, 4, 5]
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering_16)
    silhouette_avg = silhouette_score(features_for_clustering_16, labels)
    calinski_harabasz =
calinski_harabasz_score(features_for_clustering_16, labels)
    print(f"For K = {k} with n_components = {16}, the Silhouette Score is
{silhouette_avg}    and    the    Calinski    -    Harabasz    Index    is
{calinski_harabasz}")

# Example code for exploring n_components = 18
pca_18 = PCA(n_components=18)
genres_pca_18 = pca_18.fit_transform(genres_data)
features_for_clustering_18        =        np.concatenate((genres_pca_18,
original_language_encoded.reshape(-1, 1),

production_companies_encoded, keywords_tfidf,
                                          cast_encoded_matrix,
merged_data[['Cast Popularity']].values), axis=1)
k_values_to_try = [3, 4, 5]
for k in k_values_to_try:
    kmeans = KMeans(n_clusters=k, init='k-means++')
    labels = kmeans.fit_predict(features_for_clustering_18)
    silhouette_avg = silhouette_score(features_for_clustering_18, labels)
    calinski_harabasz =
calinski_harabasz_score(features_for_clustering_18, labels)
```

```
    print(f"For K = {k} with n_components = {18}, the Silhouette Score is {silhouette_avg}    and    the    Calinski    -    Harabasz    Index    is {calinski_harabasz}")

from sklearn.cluster import DBSCAN
eps_values_to_try = [0.3, 0.5, 0.7]  # can adjust these values based on your data
min_samples_values_to_try = [3, 5, 7]
for eps in eps_values_to_try:
    for min_samples in min_samples_values_to_try:
        dbscan = DBSCAN(eps=eps, min_samples=min_samples)
        labels = dbscan.fit_predict(features_for_clustering_12)   # Use features_for_clustering_12 as an example, can be changed
        # Calculate evaluation metrics for DBSCAN (note that DBSCAN may have different evaluation considerations compared to K-Means)
        # calculate the number of clusters found (DBSCAN may find different number of clusters automatically)
        n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
        print(f"For eps = {eps} and min_samples = {min_samples}, DBSCAN found {n_clusters} clusters.")

from sklearn.metrics import davies_bouldin_score
# Calculate Davies - Bouldin Index for the current clustering (assuming labels and features_for_clustering are available)
labels = kmeans.labels_   # Replace with the actual labels from the clustering
davies_bouldin = davies_bouldin_score(features_for_clustering, labels)
print(f"The Davies - Bouldin Index is {davies_bouldin}")

# Assuming we have already clustered the data and have the labels (replace with actual data and labels)
if 'genres_pca' in locals():
    plt.scatter(genres_pca[:, 0], merged_data['Cast Popularity'], c = labels)
    plt.xlabel('First Principal Component of Genres (after PCA)')
    plt.ylabel('Cast Popularity')
    plt.show()
else:
    print("Error: Genres PCA not available. Please check the PCA process.")

import plotly.express as px
```

```
# Select the features for visualization (replace with actual relevant features)
features = ['original_language', 'production_companies', 'keywords', 'Cast Popularity']
# Create a DataFrame with the selected features and cluster labels (assuming labels are available)
df_visualization = merged_data[features + ['cluster_labels']]
# Create the parallel coordinates plot
fig = px.parallel_coordinates(df_visualization, color = 'cluster_labels')
fig.show()

from sklearn.metrics.pairwise import cosine_similarity
# Assume we have feature vectors for all movies (could be based on features_for_clustering)
movie_feature_vectors = np.array([...])  # Replace with actual feature vectors
# Calculate cosine similarity matrix
similarity_matrix = cosine_similarity(movie_feature_vectors)
# If a user selects a movie (let's say movie_index is the index of the selected movie)
movie_index = 0  # Replace with actual index
# Find the most similar movies within the same cluster
cluster_label = labels[movie_index]
cluster_indices = np.where(labels == cluster_label)[0]
similarity_scores = similarity_matrix[movie_index][cluster_indices]
sorted_indices = np.argsort(similarity_scores)[::-1]
recommended_indices = cluster_indices[sorted_indices[1:6]]  # Recommend the top 5 most similar movies
recommended_movies = [... ]  # Get the actual movie details from the dataset using the recommended indices
```