Product Recommendation System for Retail Sector using Deep Learning

Sufyan Ali Awan

Supervisor: Dr Chris Thornton

Abstract

The exponential growth of digital information has transformed the retail landscape, demanding innovative strategies to capture and retain customers. This research introduces a deep learning-powered recommendation system designed to tackle challenges faced by retail industry in lieu of vast variety of product availability and complex behavioral patterns of its customers, once such being the challenge of buyer's paradox.

To achieve this, we propose leveraging the power of Artificial Neural Networks (ANNs), specifically Deep Neural Networks (DNNs) and aim to create a robust recommendation system which will be trained on extensive datasets encompassing customer purchase history and behavior, and product attributes to generate highly accurate and personalized product recommendations.

The proposed system will deliver highly personalized product recommendations, fostering a deeper connection between customers and products opening new horizons for the businesses to scale and collaborate with brand by providing them Consumer level insights which they don't have access to.

Through the strategic application of this Recommendation system retail business can benefit in multitude of ways such as, enhance product discovery, optimize loyalty programs with personalized campaigns, optimize their inventory or assortment and in doing so ultimately drive sustainable growth for business.

Table of Contents

Table of Figures:

Chapter 1

Introduction

The retail landscape is undergoing a period of unprecedented transformation, marked by rapid technological advancements and evolving consumer behaviors [1] This dynamic environment has presented significant challenges for retailers, including increased competition, changing customer expectations, and economic uncertainties [2]. The surge in e-commerce has further intensified these pressures, as online retailers offer a wider range of products and personalized shopping experiences [3]These developments are not only global phenomena but are particularly pronounced in the Gulf region, including countries such as the Kingdom of Saudi Arabia (KSA), Oman, and Dubai, where retail markets are rapidly evolving and expanding.

In the Gulf region, the retail sector has witnessed remarkable growth, driven by factors such as rising disposable incomes, a young and tech-savvy population, and strong government support for economic diversification away from oil dependency. In the Kingdom of Saudi Arabia, Vision 2030 has played a pivotal role in reshaping the retail landscape by encouraging foreign investment and fostering a more business-friendly environment. This has led to the proliferation of shopping malls, hypermarkets, and e-commerce platforms, significantly broadening the choices available to consumers. Similarly, in Dubai, the retail sector is buoyed by its status as a global shopping destination, with a blend of luxury boutiques, sprawling malls, and a thriving online retail scene. Oman, while smaller in scale, is also experiencing a retail boom, with new malls and shopping centers catering to the needs of a growing middle class.

Central to these challenges is the growing complexity of consumer behavior. With an abundance of choices available, consumers are increasingly demanding tailored product offerings and seamless shopping experiences [4]. In the Gulf region, the cultural diversity and varying preferences of consumers further complicate this landscape. For instance, while consumers in Dubai might prioritize luxury and branded products, those in Oman might focus on value and convenience. This diversity necessitates a nuanced understanding of customer preferences and behaviors to effectively cater to their needs.

Traditional marketing strategies and recommendation systems often fall short in meeting these expectations, leading to decreased customer loyalty and reduced sales [5].These systems typically rely on basic algorithms that do not fully capture the intricate patterns in consumer behavior, especially in diverse and rapidly changing markets like those in the Gulf region. As a result, retailers struggle with high return rates, low conversion rates, and an inability to build lasting relationships with their customers.

To address these challenges, this research proposes a novel deep learning-based recommendation system. By leveraging the power of deep neural networks, we aim to develop a system that accurately predicts customer preferences, optimizes product recommendations, and enhances overall customer satisfaction. Deep learning, a subset of artificial intelligence, is particularly well-suited for this task due to its ability to process large volumes of data and uncover complex patterns within it. Neural networks, which form the backbone of deep learning models, can analyze various data points such as past purchase history, browsing behavior, and demographic information to generate personalized and relevant product recommendations.

The proposed system will utilize advanced deep learning techniques, such as ANN to model and predict customer preferences with high accuracy. By integrating these models into the retail ecosystem, we aim to create a seamless and personalized shopping experience that not only meets but exceeds customer expectations. This approach promises to transform the retail landscape in the Gulf region by enabling retailers to build stronger customer relationships, reduce churn, and drive sustainable growth.

1.1 Objective

1. Understand the dataset by performing Exploratory Data Analysis.
2. Since we have transactional data, we require data pre-processing to ensure data quality and consistency.
3. Split the dataset in multiple aggregation
    a. Member Level: Includes features pertaining to members such as its purchasing history and preferences of products.
    b. Item Level: Includes features pertaining to products such as their subclass, subdepartment and their presence within transactions.
4. Perform feature engineering to acquire features on both Member and Item Levels to enrich our dataset and ensure better results.
5. Employe techniques for feature selection to ensure only relevant features are selected
6. Employ deep learning neural network using Tensor Flow
7. Evaluate the performance of the neural network

Structure

The research will proceed in following manner:

Chapter 2: Literature Review

This chapter reviews existing literature on recommendation systems and the application of deep learning in the retail sector. It provides an overview of current research and finds gaps that the dissertation aims to address.

Chapter 3: Methodology

This chapter details the method used to develop the deep learning-based recommendation

system. It includes a description of the data sources, the design and implementation of the system, and the specific deep learning techniques employed.

Chapter 4: Experimental Results
This chapter presents the results of the experiments conducted to evaluate the performance of the recommendation system. It includes a detailed analysis of the system's effectiveness and accuracy.

Chapter 5: Discussion
This chapter discusses the implications of the findings, including their impact on the retail sector and potential benefits for retailers. It also explores the limitations of the study and areas for improvement.

Chapter 6: Conclusion and Recommendations
This chapter concludes the dissertation by summarizing the key findings and contributions of the research. It provides recommendations for future research and practical applications of the deep learning-based recommendation system.

Chapter 2

Literature Review

2.1.1 Recommendation System

A recommendation system is a type of software or algorithm designed to suggest items or content to users based on their preferences, behaviors, or interactions. These systems are commonly used across various industries, including e-commerce, streaming services, and social media, to help users find products, movies, music, articles, or other content that might interest them.

Recommendation systems have become integral in the retail sector, leveraging sales data to drive personalized shopping experiences, boost sales, and streamline inventory management. This literature review examines the key approaches and advancements in recommendation systems applied to retail sales data.

2.1.2 Collaborative Filtering

User-Based Collaborative Filtering:

- **Study by Resnick et al. (1994)**: Pioneered the concept of user-based collaborative filtering by recommending items based on the preferences of similar users. In retail, this translates to suggesting products that users with similar purchase histories have bought.
- **Advantages**: Effective in capturing user preferences and generating recommendations based on user similarity.
- **Limitations**: Struggles with cold start problems for new users and items and may require large amounts of user interaction data.

Item-Based Collaborative Filtering:

- **Study by Sarwar et al. (2001)**: Introduced item-based collaborative filtering, focusing on the similarity between items rather than users. In retail, this means recommending products similar to those a customer has previously bought or viewed.
- **Advantages**: More stable over time compared to user-based methods and less sensitive to new user issues.
- **Limitations**: Can suffer from scalability issues with large item catalogs and may still face challenges with cold start problems for new items.

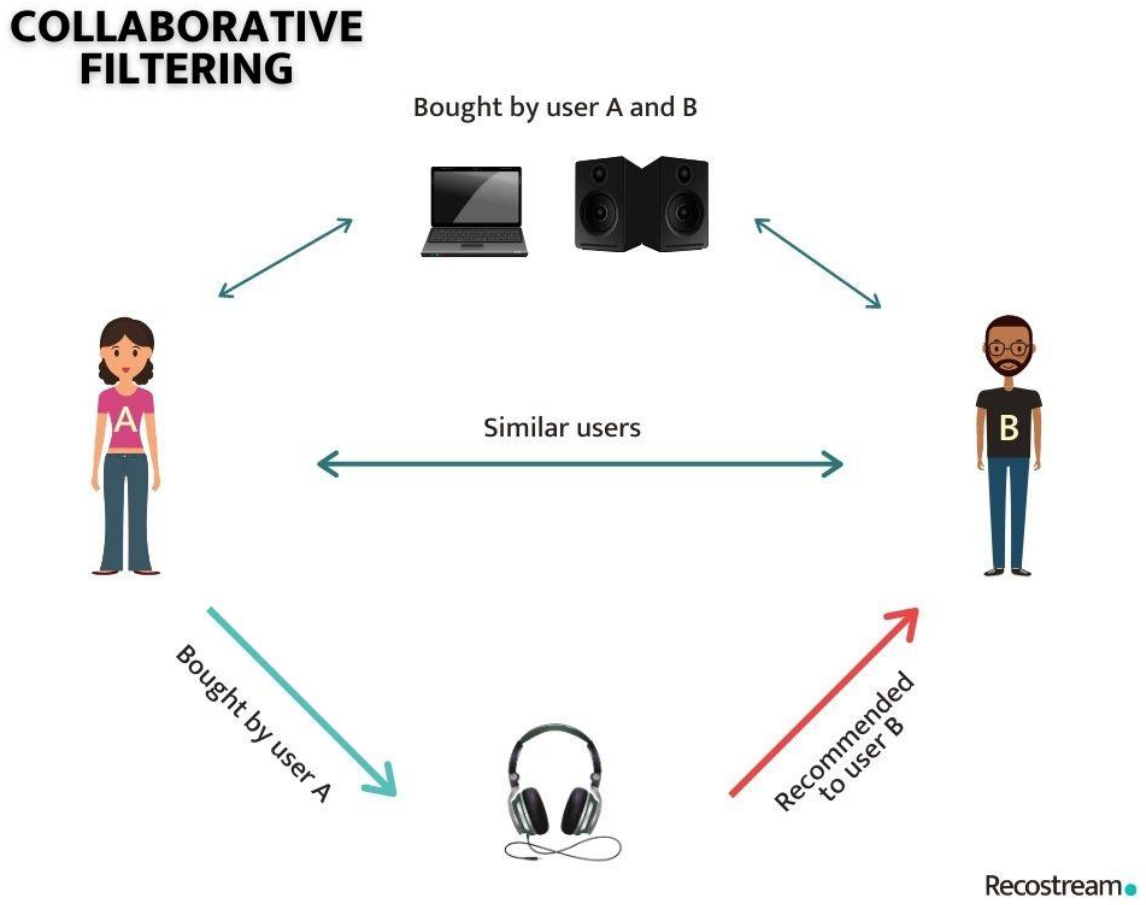

Figure 1 Collaborative Filtering

2.1.3 Content-Based Filtering

Feature Extraction and User Profiling:

- **Study by Pazzani and Billsus (2007)**: Discussed content-based filtering methods where recommendations are based on item features and user profiles. In retail, this involves analyzing product attributes (e.g., category, brand) and matching them to user preferences.
- The paper focuses on content-based recommendation systems, which recommend items by analyzing the content or features of the items and comparing them to the user's profile or past interactions. For example, in a movie recommendation system, content-based filtering might suggest movies that are similar in genre, director, or actors to those the user has previously liked.
- **User Profiles**: A key aspect of these systems is creating and maintaining user profiles that capture preferences based on the content of items the user has interacted with. These profiles help in tailoring recommendations to individual tastes.

- **Feature Extraction**: The paper discusses the importance of feature extraction and representation in content-based systems. Properly capturing and representing the characteristics of items is crucial for making relevant recommendations.
- **Evaluation**: Pazzani and Billsus highlight various methods for evaluating the performance of content-based recommendation systems, such as precision, recall, and user satisfaction. They emphasize the need for continuous evaluation and improvement of the system.
- **Challenges**: The paper also addresses challenges in content-based recommendation, including the "over-specialization" problem, where the system might recommend items that are too similar to what the user has already seen, potentially limiting exposure to diverse content.

Overall, the paper provides a comprehensive overview of how content-based recommendation systems work, their strengths, and their limitations, while offering insights into the mechanisms for improving their effectiveness.

Enhanced Features:

- **Study by Baeza-Yates and Ribeiro-Neto (1999)**: Explored enhanced feature extraction techniques such as natural language processing for product descriptions to improve content-based recommendations.
- **Overview of Information Retrieval**: The paper provides a foundational overview of the principles and technologies underlying information retrieval systems. It introduces key concepts such as indexing, query processing, and relevance ranking.
- **Indexing and Retrieval Models**: Baeza-Yates and Ribeiro-Neto discuss different methods of indexing documents to enable efficient retrieval. They explore various retrieval models, including the boolean model, vector space model, and probabilistic models, highlighting their strengths and limitations.
- **Relevance and Ranking**: The paper delves into techniques for determining the relevance of documents to a user's query. It covers ranking algorithms that assess how well documents match the query terms, including traditional approaches like term frequency-inverse document frequency (TF-IDF) and more advanced models.
- **Evaluation Metrics**: Evaluation of IR systems is a significant focus, with the paper discussing metrics such as precision, recall, and the F1 score, which are used to measure the effectiveness of retrieval systems.
- **Technological Considerations**: The study also addresses practical aspects of implementing IR systems, such as efficiency concerns, scalability, and the integration of IR with other technologies like databases and machine learning.
- **Historical and Theoretical Context**: The paper situates modern IR technologies within their historical and theoretical contexts, providing insights into the evolution of retrieval methods and their theoretical foundations.

Overall, Baeza-Yates and Ribeiro-Neto's work is a comprehensive resource that covers the fundamental concepts and technologies behind information retrieval, making it an essential reference for understanding how search engines and similar systems operate.

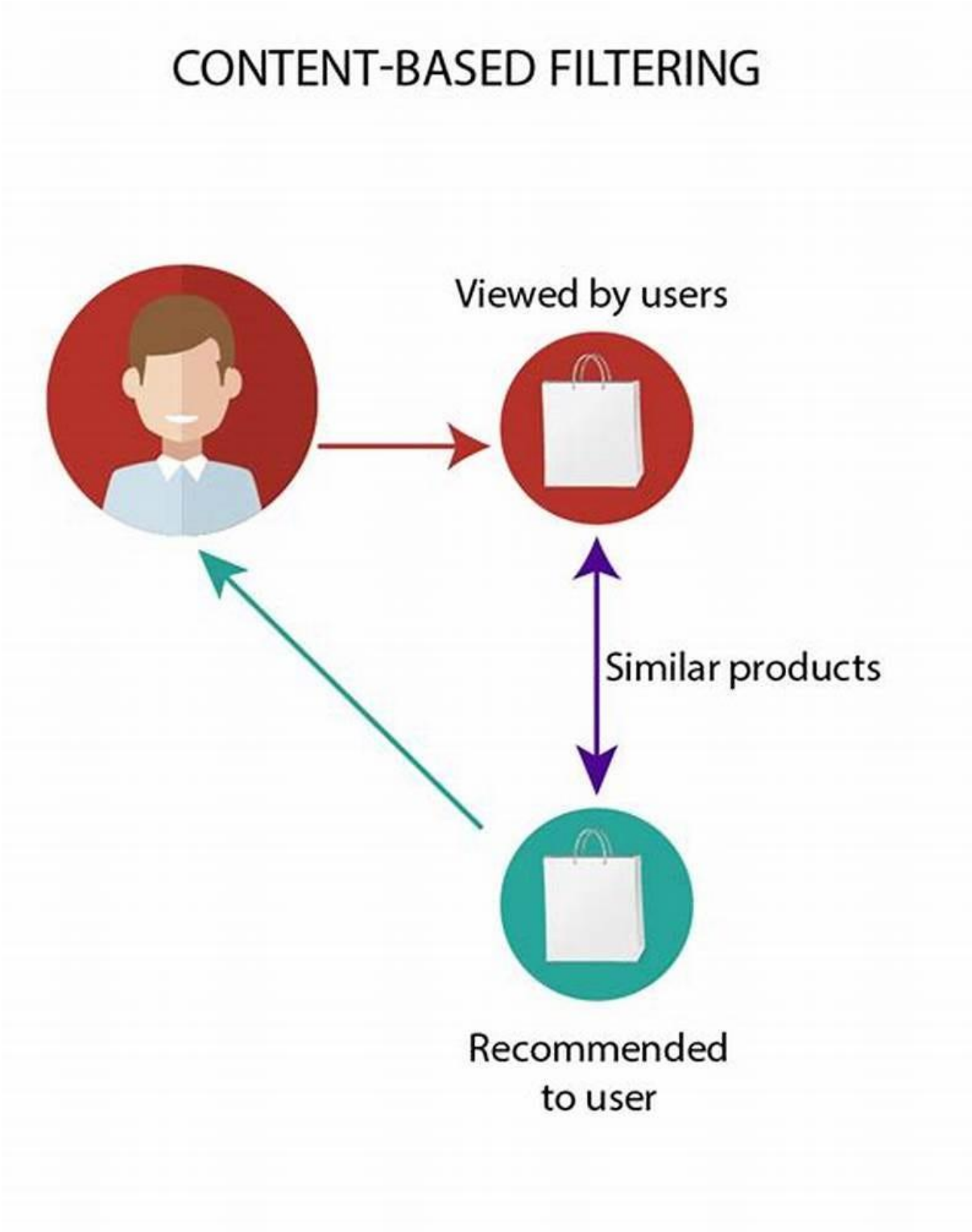

Figure 2 Content Based Filtering

2.1.4 Hybrid Approaches

Combining Collaborative and Content-Based Filtering:

- **Study by Burke (2002)**: Proposed hybrid recommendation systems that combine collaborative and content-based methods to leverage the strengths of both approaches. In retail, this means integrating user behavior data with product attributes for more accurate recommendations.
- **Hybrid Recommender Systems**: Burke explores hybrid recommender systems that integrate multiple recommendation techniques, such as content-based filtering and collaborative filtering. The goal is to leverage the strengths of each method while mitigating their individual weaknesses.
- **Types of Hybrid Approaches**: The paper categorizes hybrid approaches into several types:
    - **Weighted Hybrid**: Combines the scores from multiple recommendation methods, weighted according to their reliability or relevance.
    - **Switching Hybrid**: Switches between different recommendation methods based on the context or user characteristics.
    - **Mixed Hybrid**: Uses multiple recommendation methods simultaneously, presenting a blend of recommendations from each method.
    - **Cascade Hybrid**: Applies one recommendation method and then uses another method to refine or filter the results.
- **Advantages and Challenges**: Burke discusses the advantages of hybrid systems, such as improved accuracy and broader coverage of user preferences. He also highlights challenges, including increased computational complexity and the difficulty of integrating different methods effectively.
- **Experimental Results**: The paper presents experiments and case studies demonstrating the performance of hybrid recommender systems. These experiments illustrate how combining techniques can lead to better recommendations compared to using a single method.
- **Future Directions**: Burke suggests future research directions, including the need for more sophisticated hybrid models and the exploration of new ways to integrate recommendation techniques.

Overall, Burke's paper provides a comprehensive survey of hybrid recommender systems, detailing various approaches and their effectiveness. It serves as a valuable resource for researchers and practitioners looking to enhance recommendation systems by combining different methodologies.

Context-Aware Recommendations:

- **Study by Adomavicius and Tuzhilin (2005)**: Investigated context-aware recommendation systems that consider contextual information such as time, location, and purchase history. This is particularly relevant in retail for tailoring recommendations based on current shopping context.

- **Overview of Recommender Systems**: The paper provides a broad overview of recommender systems, including their importance and applications in various domains such as e-commerce, content distribution, and social media.
- **Current Techniques**: Adomavicius and Tuzhilin review the state-of-the-art techniques in recommender systems, focusing on:
    - **Collaborative Filtering**: Both user-based and item-based approaches, which rely on user interactions and preferences.
    - **Content-Based Filtering**: Techniques that recommend items based on the features of the items and the user's profile.
    - **Hybrid Methods**: Approaches that combine different recommendation techniques to improve performance.
- **Evaluation Metrics**: The paper discusses various metrics used to evaluate the effectiveness of recommender systems, including accuracy, novelty, diversity, and user satisfaction.
- **Challenges and Limitations**: The authors highlight several challenges faced by recommender systems, such as:
    - **Scalability**: Handling large volumes of data efficiently.
    - **Sparsity**: Dealing with incomplete user-item interaction data.
    - **Cold Start Problem**: Recommending items or users with limited data.
    - **Privacy Concerns**: Ensuring user data is protected while providing personalized recommendations.
- **Future Directions**: The paper proposes several extensions and future research directions for recommender systems, including:
    - **Context-Aware Recommendations**: Integrating contextual information (e.g., location, time) to improve recommendation relevance.
    - **Trust and Reputation Systems**: Incorporating trust and reputation mechanisms to enhance the reliability of recommendations.
    - **User Behavior Modeling**: Better understanding and modeling user behavior to provide more accurate and personalized recommendations.
- **Conclusion**: Adomavicius and Tuzhilin conclude that while significant progress has been made in recommender systems, there are still many opportunities for improvement and innovation. They emphasize the need for continued research to address existing challenges and explore new approaches.

Overall, the paper serves as a valuable resource for understanding the current state of recommender systems and provides insights into potential future developments in the field.

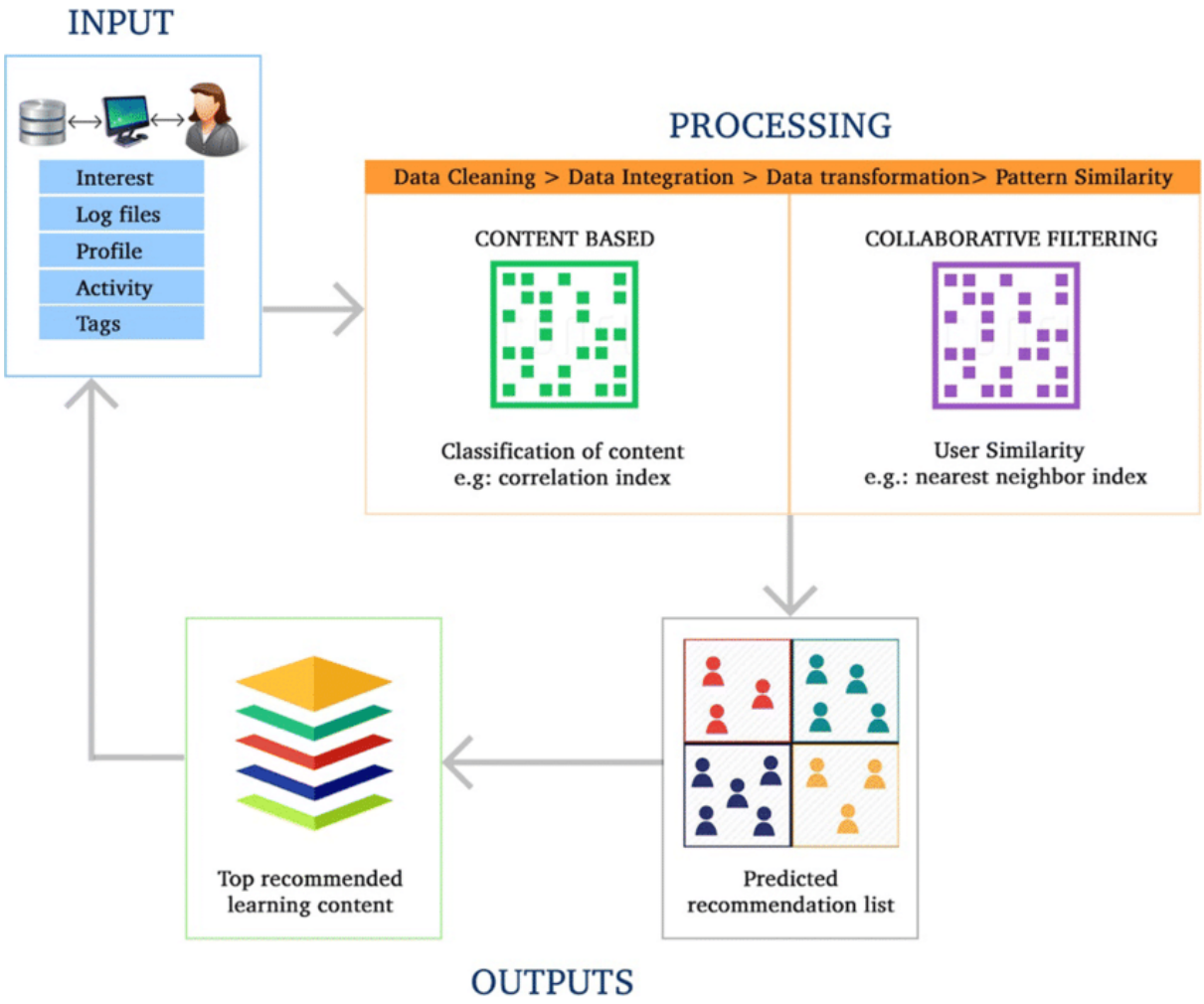

Figure 3 Hybrid Recommendation System

2.1.5 Machine Learning and Deep Learning

Advanced Techniques:

- **Study by He et al. (2017)**: Applied deep learning techniques such as neural collaborative filtering to capture complex patterns in sales data. This includes using deep neural networks to model user-item interactions.
- **Introduction of Neural Collaborative Filtering (NCF)**: The paper presents Neural Collaborative Filtering (NCF), a framework that applies neural network models to collaborative filtering, aiming to improve the recommendation quality over traditional methods.
- **Model Architecture**: He et al. propose a general framework for NCF, which uses neural networks to model the user-item interaction. The framework consists of two main components:
    - **Embedding Layers**: Users and items are represented using embeddings (low-dimensional dense vectors), which capture latent features of users and items.
    - **Multi-Layer Perceptron (MLP)**: A neural network, specifically a Multi-Layer Perceptron, is used to model the complex interactions between user and item

embeddings. This allows for capturing non-linear relationships between users and items.

- **Flexibility and Extensions**: The NCF framework is designed to be flexible and extendable. The paper demonstrates how NCF can incorporate different neural network architectures and layers, such as additional hidden layers and activation functions, to enhance its performance.
- **Experiments and Results**: The authors conduct extensive experiments on several benchmark datasets to evaluate the performance of NCF. They compare it with traditional collaborative filtering methods (like matrix factorization) and other state-of-the-art models. The results show that NCF outperforms these methods in terms of recommendation accuracy and relevance.
- **Advantages**: The paper highlights several advantages of using neural networks for collaborative filtering, including:
    - **Ability to Model Complex Interactions**: Neural networks can capture complex, non-linear relationships between users and items.
    - **Scalability**: NCF can handle large-scale datasets effectively due to the scalability of neural network techniques.
- **Future Directions**: He et al. suggest directions for future research, such as exploring deeper and more sophisticated neural network architectures and incorporating additional contextual information into the recommendation process.

Overall, the study by He et al. (2017) significantly advances the field of recommendation systems by demonstrating the effectiveness of neural network-based models in collaborative filtering, providing a new approach that improves upon traditional methods.

Embeddings and Representations:

- **Study by Rendle et al. (2012)**: Introduced matrix factorization techniques like Factorization Machines for capturing interactions between users and items in a compact way. This is useful in retail for reducing dimensionality and improving recommendation quality.
- **Introduction to Factorization Machines (FMs)**: Rendle and colleagues present Factorization Machines, a versatile and powerful model designed to capture interactions between variables in high-dimensional datasets. FMs generalize matrix factorization and other factorization-based models, making them applicable to a wide range of problems, including recommendation systems.
- **Model Formulation**: The core idea behind FMs is to model interactions between features using factorized parameters. Unlike traditional matrix factorization, which only works with explicit interactions (e.g., user-item ratings), FMs can handle interactions between any pair of features, making them applicable to both explicit and implicit feedback scenarios.
    - **General Formulation**: The FM model consists of a linear term that accounts for the individual feature effects and a factorized term that captures interactions between features. The interaction between features is modeled using latent factors, which are learned during training.
- **Applications to Recommendation**: The paper discusses how FMs can be applied to collaborative filtering and recommendation systems. By using FMs, the authors

demonstrate improved performance in capturing user-item interactions compared to traditional matrix factorization techniques.

- **Flexibility and Efficiency**: FMs are designed to be highly flexible, allowing for the incorporation of various types of interactions and features. The paper also addresses the efficiency of FMs, noting that they can be trained effectively using stochastic gradient descent (SGD) and are scalable to large datasets.
- **Experiments and Results**: The authors conduct experiments on several benchmark datasets to evaluate the performance of FMs. They show that FMs achieve state-of-the-art results in recommendation tasks and other applications, outperforming traditional methods in terms of predictive accuracy.
- **Advantages of FMs**: The paper highlights several advantages of FMs:
    - **Versatility**: FMs can model a wide range of interactions between features, making them useful for various machine learning tasks.
    - **Scalability**: The model can handle large-scale datasets efficiently.
- **Future Directions**: Rendle et al. suggest further exploration into extensions and improvements of the FM model, such as incorporating additional regularization techniques and exploring new application areas.

Overall, the study by Rendle et al. (2012) is a significant contribution to machine learning and recommendation systems, introducing Factorization Machines as a powerful and flexible model for capturing complex interactions in high-dimensional data.

Another novel approach was discussed in following research:

2.1.6 Evaluation and Metrics

Performance Metrics:

- **Study by Shani and Stone (2008)**: Reviewed various metrics for evaluating recommendation systems, including precision, recall, and F1-score. In retail, these metrics help assess the effectiveness of recommendation systems in driving sales and improving customer satisfaction.
- **Overview of Collaborative Filtering**: The paper begins by discussing the two main types of collaborative filtering approaches: memory-based and model-based. It focuses specifically on model-based collaborative filtering, which uses machine learning models to predict user preferences based on historical data.
- **Model-Based Approaches**: Shani and Stone review various model-based techniques for collaborative filtering, including:
    - **Matrix Factorization**: Techniques such as Singular Value Decomposition (SVD) and its variants, which factorize the user-item interaction matrix into lower-dimensional matrices to capture latent features.
    - **Probabilistic Models**: Approaches like Bayesian methods that use probabilistic models to estimate user preferences and make predictions.

    - **Clustering Methods**: Techniques that group users or items into clusters and make recommendations based on cluster similarities.
- **Comparative Analysis**: The paper compares the strengths and weaknesses of different model-based approaches. For example, matrix factorization is noted for its ability to capture latent factors and improve recommendation accuracy, while probabilistic models are praised for their ability to handle uncertainty and incorporate prior knowledge.
- **Evaluation of Models**: Shani and Stone discuss various methods for evaluating the performance of model-based collaborative filtering methods, including metrics such as precision, recall, and mean squared error. They emphasize the importance of robust evaluation to assess the effectiveness of different models.
- **Challenges and Future Directions**: The paper highlights several challenges in model-based collaborative filtering, such as:
    - **Scalability**: Handling large datasets and ensuring efficient computation.
    - **Cold Start Problem**: Making recommendations for new users or items with limited data.
    - **Contextual Information**: Incorporating additional contextual information (e.g., time, location) to improve recommendations.

  The authors suggest future research directions, including the development of hybrid models that combine different collaborative filtering approaches and the exploration of more advanced machine learning techniques.

- **Practical Applications**: The paper also touches on practical considerations for implementing model-based collaborative filtering in real-world systems, including issues related to data sparsity, computation, and scalability.

Overall, Shani and Stone's study provides a thorough review of model-based collaborative filtering techniques, offering insights into their effectiveness, challenges, and potential for future research.

2.1.7 Challenges and Future Directions

Data Privacy and Security:

- **Study by McSherry and Mironov (2009)**: Addressed concerns regarding data privacy and security in recommendation systems. Retailers need to handle sensitive customer data responsibly.
- **Introduction to Privacy-Preserving Data Analysis**: McSherry and Mironov address the growing need for privacy-preserving techniques in data analysis. They focus on how to protect sensitive information while still allowing for meaningful analysis and data sharing.
- **SuLQ Framework**: The paper presents the **SuLQ** (Sufficiently-Loud Quantization) framework, which is designed to provide a practical approach to privacy preservation. SuLQ is an extension of the **Laplace mechanism** used in differential privacy, aiming to improve both the effectiveness and practicality of data privacy methods.
    - **Laplace Mechanism**: The Laplace mechanism adds noise to data queries to ensure privacy, but it can be challenging to implement in practice due to trade-offs between privacy and data utility.
- **Key Features of SuLQ**:

    - **Quantization**: SuLQ uses quantization to approximate the noise addition process, making it more efficient and practical for real-world applications.
    - **Differential Privacy**: SuLQ maintains differential privacy, a standard definition of privacy that ensures an attacker cannot determine whether a specific individual's data is included in the dataset, even with auxiliary information.
- **Practical Implementation**: The paper discusses how SuLQ can be implemented in practical scenarios, highlighting its advantages over previous methods. The authors show how SuLQ can be applied to various types of data analysis tasks while maintaining strong privacy guarantees.
- **Experimental Results**: McSherry and Mironov present experimental results demonstrating the effectiveness of the SuLQ framework. They compare its performance with other privacy-preserving techniques, showing that SuLQ offers a good balance between privacy and data utility.
- **Advantages and Applications**: The authors emphasize that SuLQ is designed to be practical and efficient, making it suitable for use in real-world systems where privacy and data analysis requirements must be balanced.
- **Future Directions**: The paper concludes by suggesting areas for future research, including the exploration of additional applications of SuLQ and further refinements to improve its efficiency and applicability.

Overall, the study by McSherry and Mironov (2009) provides a significant contribution to the field of privacy-preserving data analysis by introducing the SuLQ framework, which offers a practical and efficient approach to achieving differential privacy in real-world data applications.

Personalization vs. Diversity:

- **Study by Burke (2007)**: Explored the trade-off between personalized recommendations and the need for diverse suggestions to avoid filter bubbles.
- **Introduction to Hybrid Recommender Systems**: Burke explores hybrid recommender systems, which combine multiple recommendation techniques to overcome limitations inherent in individual methods. The primary goal is to enhance the quality and relevance of recommendations.
- **Types of Hybrid Approaches**: The paper categorizes hybrid recommender systems into several types based on how they integrate different techniques:
    - **Weighted Hybrid**: Combines the output of different recommenders using weighted averaging. The weights can be based on the reliability or accuracy of each recommender.
    - **Switching Hybrid**: Selects among different recommendation techniques depending on the context or the characteristics of the user or item.
    - **Mixed Hybrid**: Provides recommendations from multiple systems simultaneously, presenting a blend of suggestions from each.
    - **Cascade Hybrid**: Applies one recommendation method to filter or rank items and then uses another method to refine or adjust the recommendations.
    - **Feature Augmentation**: Uses recommendations from one system as additional features or inputs for another recommendation algorithm.
- **Advantages of Hybrid Systems**: Burke highlights several advantages of using hybrid approaches:

    - **Improved Accuracy**: By combining techniques, hybrids often achieve better predictive accuracy and recommendation quality.
    - **Enhanced Coverage**: Hybrid systems can provide a wider range of recommendations, addressing limitations like the cold start problem and data sparsity.
- **Experiments and Results**: The paper includes experimental results that demonstrate the effectiveness of hybrid recommender systems compared to single-method approaches. Burke's experiments show that hybrids can outperform traditional methods in terms of recommendation quality and user satisfaction.
- **Challenges**: The study also addresses challenges associated with hybrid recommenders, such as:
    - **Complexity**: Integrating multiple methods can increase system complexity and computational demands.
    - **Parameter Tuning**: Effective hybrid systems require careful tuning of parameters and weights to balance the contributions of different techniques.
- **Future Directions**: Burke suggests areas for future research, including the exploration of new hybrid models and techniques, as well as the application of hybrid systems in different domains and contexts.

Overall, Burke's paper provides a detailed survey of hybrid recommender systems, offering insights into their design, benefits, and challenges. It serves as a valuable resource for researchers and practitioners interested in improving recommendation systems through hybrid approaches.

Recommendation systems leveraging sales data in retail have evolved significantly, incorporating various methodologies from collaborative filtering to advanced machine learning techniques. While each approach has its strengths and limitations, combining these methods in hybrid systems often provides the best results. Future research should continue to address challenges related to data privacy, the balance between personalization and diversity, and the integration of contextual information to further enhance the effectiveness of recommendation systems in the retail sector.

2.2 Case Studies

2.2.1 Case Study: Movie Recommendation System for StreamFlix

2.2.1.1 Background

StreamFlix, a fictional streaming service, wants to enhance its recommendation system to better serve its users. They currently use a basic recommendation model but are interested in implementing User-Based Collaborative Filtering to offer more personalized recommendations.

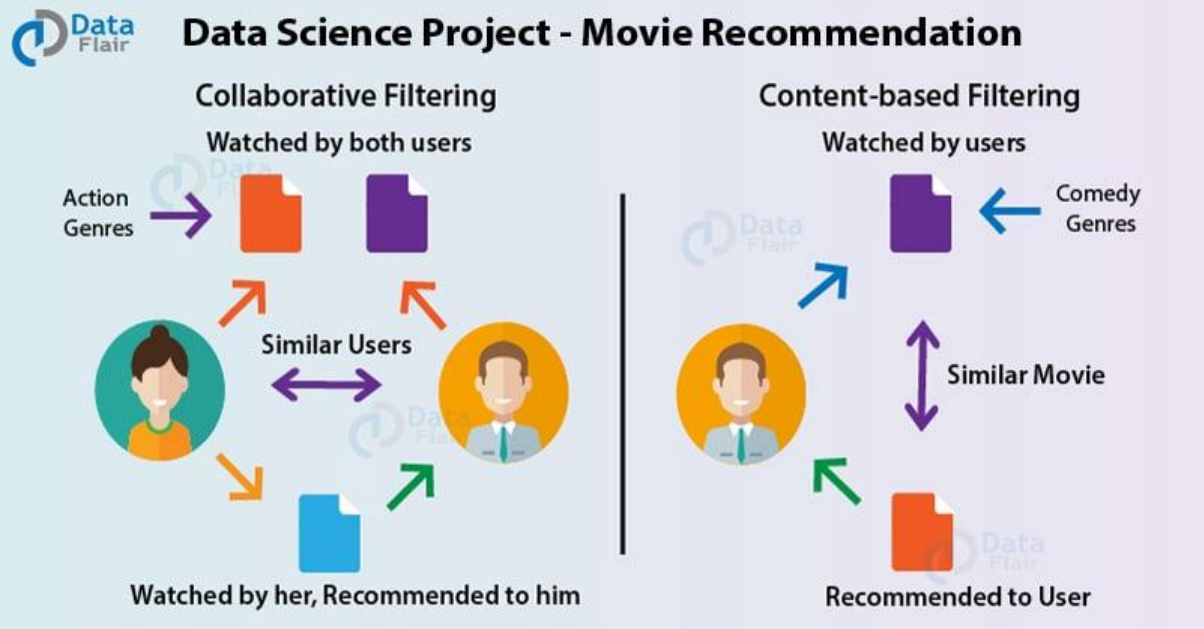

Figure 4 Movie Recommendation System

2.2.1.2 Objective

To improve user satisfaction and engagement by recommending movies that users with similar tastes have enjoyed.

2.2.1.3 Data Collection

1. **User Data**: StreamFlix collects data on user ratings for movies. For example:
    - User A rates "Inception" 5 stars.
    - User B rates "Inception" 4 stars and "The Matrix" 5 stars.
    - User C rates "The Matrix" 4 stars and "Interstellar" 5 stars.
2. **Movie Data**: Information on movies such as genres, actors, and directors is also available but not used in UBCF.

2.2.1.4 Methodology

1. **Create User-Item Matrix**: Construct a matrix where rows represent users, columns represent movies, and cells contain ratings.

	Inception	**The Matrix**	**Interstellar**
User A	5	-	-
User B	4	5	-
User C	-	4	5

2. **Calculate Similarity**: Use a similarity metric, such as Pearson correlation coefficient or cosine similarity, to measure how similar each user is to every other user.

- **Cosine Similarity Example**: Compute similarity between User A and User B based on their rating vectors.

3. **Generate Recommendations**:
   - For a target user (e.g., User A), find users most similar to them (e.g., User B and User C).
   - Aggregate the ratings from similar users for items the target user hasn't rated.
   - Recommend items that have high aggregated scores.

   For User A, the system might recommend "The Matrix" based on high ratings from similar users.

2.2.1.5 Implementation

1. **Algorithm**: Implement the UBCF algorithm using a similarity matrix. When a user requests recommendations, the system finds similar users and aggregates their ratings for unrated items.
2. **Scalability**: Optimize performance by using techniques like dimensionality reduction or approximate nearest neighbors to handle large datasets.

2.2.1.6 Challenges

1. **Sparsity**: The user-item matrix is often sparse, making it challenging to find similar users.
2. **Scalability**: As the number of users and items grows, computing similarities becomes computationally intensive.
3. **Cold Start**: New users or items with few ratings can be difficult to recommend accurately.

2.2.1.7 Results

1. **User Engagement**: After implementing UBCF, StreamFlix observes an increase in user engagement. Users are spending more time on the platform and exploring new movies.
2. **User Satisfaction**: User feedback indicates higher satisfaction with recommendations, leading to increased retention rates.
3. **Business Impact**: StreamFlix experiences a boost in subscription renewals and overall revenue due to improved recommendations.

2.2.1.8 Conclusion

The implementation of User-Based Collaborative Filtering at StreamFlix significantly enhanced the quality of recommendations, leading to greater user satisfaction and engagement. However, the system also faced challenges such as sparsity and scalability, which are common in collaborative filtering approaches. Future improvements could include hybrid models that combine UBCF with other techniques, like content-based filtering, to address some of these issues.

By continuously refining the recommendation algorithm and addressing the challenges, StreamFlix can maintain a competitive edge and provide a better user experience.

2.2.2 Case Study: Product Recommendation System for ShopSmart

2.2.2.1 *Background*

ShopSmart, a popular e-commerce platform, aims to enhance its product recommendation system. While their current system provides general recommendations, they want to

implement Item-Based Collaborative Filtering (IBCF) to offer more relevant product suggestions to users.

2.2.2.2 Objective

To increase user satisfaction and sales by recommending products that are frequently purchased together or have similar purchase patterns.

2.2.2.3 Data Collection

1. **User Purchase Data**: ShopSmart tracks user transactions, including product IDs, quantities, and timestamps.
    - Example Transactions:
        - User 1 buys "Wireless Mouse" and "Keyboard".
        - User 2 buys "Keyboard" and "Laptop Stand".
        - User 3 buys "Wireless Mouse", "Keyboard", and "Laptop Stand".
2. **Product Data**: Basic product details such as categories, prices, and descriptions are available but not used in IBCF.

2.2.2.4 Methodology

1. **Create Item-Item Matrix**: Construct a matrix where rows and columns represent products, and cells contain similarity scores based on user purchase patterns.
    1. **Generate Co-Purchase Matrix**:
        - Calculate co-occurrence counts: how often each pair of products is purchased together.
        - Normalize these counts to get similarity scores.

    Example Co-Purchase Matrix:

Product	Wireless Mouse	Keyboard	Laptop Stand
Wireless Mouse	1.0	0.7	0.5
Keyboard	0.7	1.0	0.6
Laptop Stand	0.5	0.6	1.0

2. **Calculate Similarity Scores**:
    1. Use metrics like cosine similarity or adjusted cosine similarity to compute how similar each product is to others based on co-purchases.
2. **Generate Recommendations**:
    1. For a given product (e.g., "Wireless Mouse"), find similar items based on the similarity scores.
    2. Recommend products that have high similarity scores and are often bought together with the target product.

    Example Recommendations for "Wireless Mouse":

    3. "Keyboard" (high similarity score)

4. "Laptop Stand" (lower similarity score but still relevant)

2.2.2.5 Implementation

1. **Algorithm**: Implement the IBCF algorithm, where the system computes similarities between products and generates recommendations based on these similarities.
2. **Scalability**: Optimize for scalability by using efficient data structures and algorithms to handle large product catalogs and user transaction data.
3. **Real-Time Updates**: Integrate a mechanism for real-time updates to the co-purchase matrix as new transactions occur.

2.2.2.6 Challenges

1. **Data Sparsity**: The system might face issues if the product catalog is large but the number of co-purchases is relatively small.
2. **Scalability**: As the number of products grows, maintaining and computing similarity scores for all pairs of products becomes challenging.
3. **Cold Start**: New products with few transactions may not have sufficient data for accurate recommendations.

2.2.2.7 Results

1. **User Engagement**: After implementing IBCF, ShopSmart notices a significant increase in user engagement. Customers explore more products and spend more time on the platform.
2. **Sales Growth**: Product recommendations based on IBCF lead to increased sales, as users purchase additional items suggested by the system.
3. **Customer Satisfaction**: Users report higher satisfaction with the relevance of product recommendations, contributing to increased repeat business.

2.2.2.8 Conclusion

The implementation of Item-Based Collaborative Filtering at ShopSmart successfully enhanced the product recommendation system. By focusing on the similarity between products based on co-purchase patterns, the platform was able to provide more relevant and engaging recommendations to users. However, challenges like data sparsity and scalability need to be managed effectively. Future improvements could include incorporating hybrid approaches that combine IBCF with other recommendation techniques, such as content-based filtering or contextual recommendations, to address some of these challenges and further refine the recommendation system.

2.2.3 Case Study: Content-Based Filtering for BookRecs

2.2.3.1 Background

BookRecs, a fictional online bookstore, aims to enhance its recommendation system to offer more personalized book suggestions to users. They decide to implement Content-Based Filtering (CBF) to recommend books based on individual user preferences and book attributes.

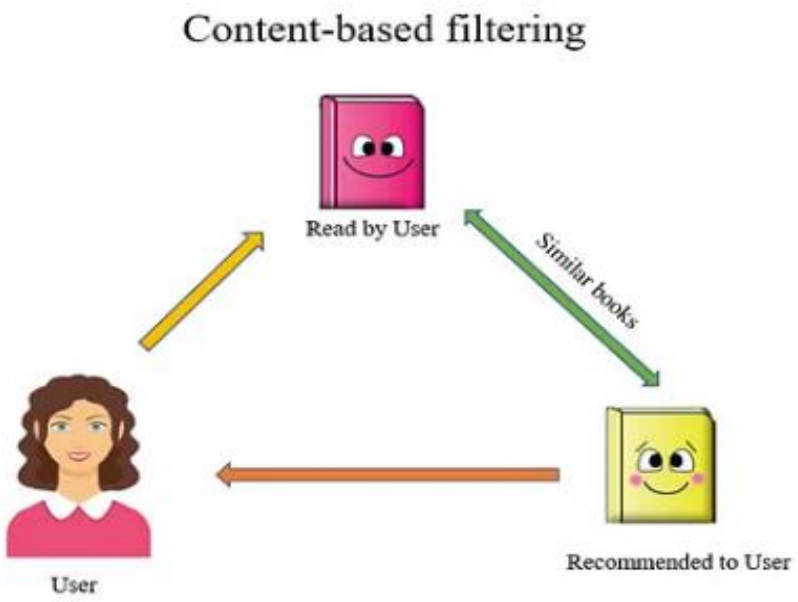

Figure 5 Content-Based Filtering for BookRecs

2.2.3.2 Objective

To increase user engagement and sales by recommending books similar to those that users have shown interest in, based on content features rather than user behavior alone.

2.2.3.3 Data Collection

1. **User Data**: BookRecs collects data on user interactions, including:
    - Books rated by users.
    - Books added to wish lists.
    - Books purchased by users.

    Example User Data:

    - User A rates "The Catcher in the Rye" 5 stars.
    - User B adds "To Kill a Mockingbird" to their wish list.
    - User C purchases "1984" and "Brave New World".
2. **Book Data**: BookRecs gathers detailed information about each book, including:
    - Title
    - Author
    - Genre
    - Summary
    - Keywords
    - Publication Year

    Example Book Data:

    - "The Catcher in the Rye": Author: J.D. Salinger, Genre: Fiction, Keywords: adolescence, rebellion.

- "To Kill a Mockingbird": Author: Harper Lee, Genre: Fiction, Keywords: racial injustice, moral growth.
- "1984": Author: George Orwell, Genre: Dystopian, Keywords: totalitarianism, surveillance.

2.2.3.4 *Methodology*

1. **Feature Extraction**:
    - Convert book attributes into a structured format suitable for analysis. For instance, represent each book using a vector of features based on genre, keywords, and other metadata.

    Example Feature Vector for "The Catcher in the Rye":

    - [Fiction: 1, Dystopian: 0, Adolescence: 1, Rebellion: 1]
2. **User Profile Creation**:
    - Build user profiles based on their interactions. This involves creating a feature vector that represents the types of books a user likes.

    Example User Profile for User A:

    - Average feature vector of rated books: [Fiction: 1, Dystopian: 0, Adolescence: 1, Rebellion: 1]
3. **Similarity Calculation**:
    - Calculate the similarity between books and the user's profile. Use cosine similarity, Euclidean distance, or other similarity measures to find books that are similar to those the user has shown interest in.

    Example Similarity Calculation:

    - Compute similarity between "To Kill a Mockingbird" and User A's profile. If the feature vector of "To Kill a Mockingbird" is close to User A's profile vector, it is considered a relevant recommendation.
4. **Generate Recommendations**:
    - Based on similarity scores, recommend books that closely match the user's profile but that the user has not yet interacted with.

    Example Recommendations for User A:

    - "To Kill a Mockingbird" (similar features)
    - "The Bell Jar" (similar features based on the feature vector)

2.2.3.5 Implementation

1. **Algorithm**: Develop and deploy the Content-Based Filtering algorithm, ensuring that it efficiently handles feature extraction, user profile creation, and similarity calculations.
2. **System Integration**: Integrate the recommendation engine into BookRecs' platform, enabling real-time recommendations as users browse or interact with books.

3. **User Interface**: Design a user-friendly interface that showcases personalized recommendations prominently.

2.2.3.6 Challenges

1. **Feature Engineering**: Accurately defining and extracting relevant features from book data can be complex. It requires careful consideration of which attributes best represent book content.
2. **Scalability**: As the number of books and users grows, the system must efficiently compute similarities and generate recommendations without significant performance degradation.
3. **Cold Start Problem**: New books or users with minimal interaction data may not receive accurate recommendations until sufficient data is collected.

2.2.3.7 Results

1. **User Engagement**: After implementing CBF, BookRecs sees an increase in user engagement. Users spend more time exploring recommended books and provide positive feedback about the relevance of suggestions.
2. **Sales Growth**: BookRecs experiences a noticeable increase in sales, as users are more likely to purchase books recommended by the system based on their interests.
3. **Customer Satisfaction**: Users report higher satisfaction with the personalized recommendations, contributing to improved retention and customer loyalty.

2.2.3.8 Conclusion

The implementation of Content-Based Filtering at BookRecs successfully enhanced the book recommendation system by focusing on individual user preferences and book attributes. This approach led to increased user engagement and sales while improving customer satisfaction. However, challenges such as feature engineering and scalability were addressed through careful system design and optimization. To further refine recommendations, BookRecs might consider integrating CBF with collaborative filtering methods, creating a hybrid recommendation system that leverages both content and user behavior.

2.2.4 Case Study: Hybrid Recommendation System for FoodieApp

2.2.4.1 Background

FoodieApp, a popular food delivery and recipe recommendation service, aims to enhance its recommendation engine. While they currently use either Collaborative Filtering (CF) or Content-Based Filtering (CBF) individually, they want to implement a hybrid recommendation system that combines both approaches to provide more accurate and personalized recommendations.

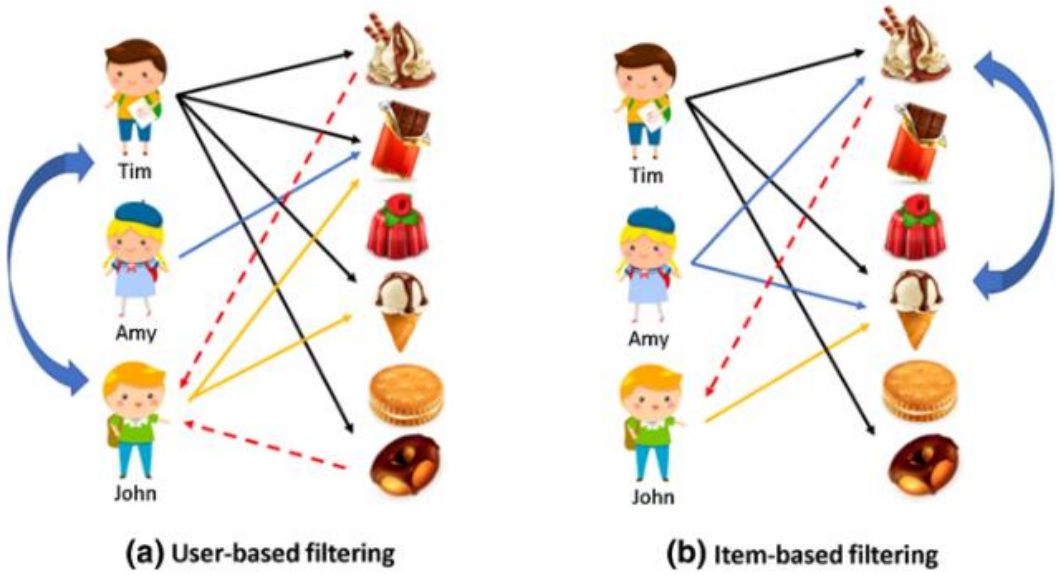

Figure 6 Hybrid Recommendation System for FoodieApp

2.2.4.2 Objective

To increase user engagement and satisfaction by leveraging both collaborative and content-based methods, thereby offering more precise and diverse food recommendations.

2.2.4.3 Data Collection

1. **User Data**: FoodieApp tracks:
    - User ratings for restaurants and recipes.
    - User interactions such as searches, favorites, and order history.

    Example User Data:

    - User A rates "Spaghetti Carbonara" 5 stars.
    - User B searches for "Vegan Recipes" and adds "Vegan Tacos" to their favorites.
    - User C orders "Chicken Alfredo" multiple times.
2. **Recipe and Restaurant Data**: FoodieApp maintains detailed information about:
    - Recipe ingredients
    - Cuisine types
    - Restaurant ratings
    - Dish descriptions

    Example Data:

    - "Spaghetti Carbonara": Cuisine: Italian, Ingredients: Pasta, Bacon, Egg, Cheese.
    - "Vegan Tacos": Cuisine: Mexican, Ingredients: Corn Tortillas, Beans, Avocado, Spices.
    - "Chicken Alfredo": Cuisine: Italian, Ingredients: Chicken, Pasta, Cream, Parmesan.

2.2.4.4 Methodology

1. **Content-Based Filtering Component**:
    - **Feature Extraction**: Extract features from recipes and restaurants, such as ingredients, cuisine types, and keywords.
    - **User Profiles**: Create profiles based on user preferences, reflecting the ingredients, cuisines, and dish types they have rated highly or frequently interacted with.
    - **Recommendation Generation**: Recommend items similar to those the user has shown interest in, based on content features.
2. **Collaborative Filtering Component**:
    - **User-Item Matrix**: Build a matrix where rows represent users and columns represent items (recipes/restaurants), with cells indicating ratings or interactions.
    - **Similarity Calculation**: Compute similarities between users based on their ratings and interactions. Also, calculate item-item similarities based on user co-ratings.
    - **Recommendation Generation**: Recommend items based on the preferences of similar users or items frequently co-rated with those the user has rated.
3. **Hybrid Approach**:
    - **Combining Scores**: Integrate recommendations from both CBF and CF. This can be done by:
        - **Weighted Hybrid**: Assign weights to CBF and CF recommendations and combine them. For instance, you might give 60% weight to CBF and 40% to CF.
        - **Switching Hybrid**: Use CBF-based recommendations for new users (cold start) and CF-based recommendations once there is sufficient user interaction data.
    - **Personalization**: Adjust the hybrid approach based on user feedback and interaction, continuously refining the balance between CBF and CF components.

2.2.4.5 Implementation

1. **Algorithm Development**:
    - Develop algorithms for CBF and CF and design the hybrid model that integrates both approaches.
    - Implement real-time processing to update recommendations as user preferences and interactions change.
2. **System Integration**:
    - Integrate the hybrid recommendation engine into the FoodieApp platform.
    - Ensure that the system can handle large volumes of user data and recipe information efficiently.
3. **User Interface**:
    - Design an intuitive interface to display recommendations, showcasing both personalized and popular items.

2.2.4.6 Challenges

1. **Balancing Hybrid Weights**: Determining the optimal weight for CBF and CF components can be complex and may require iterative testing and adjustment.
2. **Scalability**: Managing and processing large datasets for both CBF and CF in real-time can be computationally intensive.
3. **Cold Start Problem**: New users or items may not fit well into the hybrid model until enough data is gathered.

2.2.4.7 Results

1. **User Engagement**: After implementing the hybrid recommendation system, FoodieApp sees a notable increase in user engagement. Users explore more recipes and restaurants, leading to higher interaction rates.
2. **Sales Growth**: There is a significant increase in orders and bookings, as users are more likely to try recommended items based on their personalized and diverse recommendations.
3. **Customer Satisfaction**: User feedback indicates higher satisfaction with the relevance and diversity of recommendations, leading to increased retention rates.

2.2.4.8 Conclusion

The hybrid recommendation system at FoodieApp successfully combined Collaborative Filtering and Content-Based Filtering to enhance the recommendation experience. By leveraging the strengths of both approaches, the platform was able to provide more accurate, diverse, and personalized recommendations, resulting in increased user engagement and satisfaction. Challenges such as balancing hybrid weights and managing scalability were addressed through careful system design and iterative testing. Future improvements could involve incorporating additional data sources, such as social media interactions or contextual information, to further refine the recommendations and maintain a competitive edge in the market.

2.2.5 Case Study: Deep Learning-Based Product Recommendation System for ShopSmart

2.2.5.1 *Background*

ShopSmart, a major online retail platform, seeks to improve its product recommendation system. The company has traditionally relied on simpler recommendation algorithms but now aims to harness the power of deep learning to deliver more personalized and accurate product suggestions. Their goal is to enhance user experience, increase sales, and improve customer retention.

2.2.5.2 Objective

To implement a deep learning-based recommendation system that can provide highly personalized product recommendations by leveraging complex patterns in user behavior and product features.

2.2.5.3 Data Collection

1. **User Data**: ShopSmart collects comprehensive data, including:
    - User purchase history
    - Browsing behavior
    - Product ratings
    - Cart additions
    - User demographics (age, location, etc.)

    Example User Data:

    - User A frequently buys electronics and searches for "smartphones."
    - User B often purchases home appliances and reads reviews about "air purifiers."
    - User C adds clothing items to their cart and rates various fashion products.
2. **Product Data**: Information about products includes:
    - Product ID

- Category (e.g., electronics, fashion, home appliances)
- Product descriptions
- Prices
- Product images

Example Product Data:

- "Smartphone XYZ": Category: Electronics, Description: High-performance smartphone with 128GB storage, Price: $499.
- "Air Purifier ABC": Category: Home Appliances, Description: HEPA filter air purifier, Price: $299.
- "Leather Jacket": Category: Fashion, Description: Genuine leather jacket, Price: $149.

3. **Additional Data**: ShopSmart also collects implicit feedback, such as:
   - Click-through rates (CTR)
   - Time spent on product pages
   - Interaction patterns (e.g., scroll depth)

2.2.5.4 Methodology

1. **Data Preprocessing**:
   - **Normalization**: Scale numerical features (e.g., prices) and encode categorical features (e.g., product categories) into suitable formats.
   - **Text Processing**: Convert product descriptions into numerical vectors using techniques like TF-IDF or word embeddings (Word2Vec, GloVe).
   - **Image Processing**: Preprocess product images by resizing and normalizing, and use convolutional neural networks (CNNs) for feature extraction.
2. **Model Architecture**:
   - **Deep Neural Network (DNN)**: Use fully connected layers to learn complex interactions between user and product features.
   - **Convolutional Neural Network (CNN)**: Process product images to extract visual features.
   - **Recurrent Neural Network (RNN) or Transformer**: Analyze sequential user behavior data to understand patterns and preferences.
   - **Embedding Layers**: Represent categorical features (e.g., product categories) and user interactions as dense vectors.

   Hybrid Model: Combine outputs from different neural networks:
   - **User Profile Network**: Learns user preferences from historical data.
   - **Product Profile Network**: Learns product characteristics from textual and visual data.
   - **Fusion Layer**: Merges features from both user and product profiles and predicts user preferences.
3. **Training**:
   - Use historical data to train the deep learning model, optimizing it for accuracy and relevance.
   - Employ techniques such as dropout and regularization to prevent overfitting and improve generalization.
4. **Evaluation**:

- Assess the model using metrics like precision, recall, F1 score, and mean average precision (MAP) to ensure the recommendations are relevant and accurate.
- Perform A/B testing to compare the deep learning model with existing recommendation systems and measure improvements in user engagement and sales.

2.2.5.5 Implementation

1. **Deployment**:
   - Integrate the trained model into ShopSmart's recommendation engine.
   - Implement real-time recommendation capabilities to update suggestions based on user interactions.
2. **User Interface**:
   - Design an intuitive and visually appealing recommendation interface that showcases personalized product suggestions based on deep learning insights.
3. **Scalability**:
   - Use cloud-based infrastructure to handle large volumes of data and support scalable training and inference.

2.2.5.6 Challenges

1. **Data Quality and Quantity**: Deep learning models require large amounts of high-quality data. Ensuring data accuracy and completeness is critical.
2. **Computational Resources**: Training deep learning models can be resource-intensive, requiring significant computational power and time.
3. **Model Interpretability**: Deep learning models are often seen as "black boxes." Providing explanations for recommendations can be challenging but is important for user trust.

2.2.5.7 Results

1. **Enhanced Recommendations**: Shop Smart's deep learning-based system delivers highly personalized and relevant product recommendations, leading to a noticeable increase in user engagement.
2. **Increased Sales**: The improved recommendations result in higher conversion rates, with users more likely to purchase suggested products.
3. **Customer Satisfaction**: Users express greater satisfaction with the relevance and accuracy of product suggestions, leading to increased loyalty and repeat business.

2.2.5.8 Conclusion

The implementation of a deep learning-based recommendation system at Shop Smarts successfully enhanced the platform's ability to deliver personalized product suggestions. By leveraging advanced neural network architectures and integrating diverse data sources, Shop Smart achieved significant improvements in user engagement and sales. Future developments could focus on refining the model further, incorporating additional data sources such as social media interactions, and exploring explainable AI techniques to provide transparent recommendations.

Chapter 3

Methodology

3.1 Data Collection and Preparation

The dataset used for this analysis includes sales data from a retail hypermarket store, The Data comprises of 8 Months between 2021 and 2022.

We will proceed with feature engineering from below mentioned features for both Members and Items.

3.1.1 **Data Features:**

1. **Trx_id**: Identifier for the Transaction.
2. **Timestamp**: The timestamp of the transaction.
3. **Member_id**: Unique Identifier for Loyalty Member.
4. **Item_id**: Unique Identifier for Item.
5. **Item_name**: Name of the Item.
6. **Item_subclass**: Sub-class of an Item.
7. **Item_subdepartment**: Sub-Department of an Item.
8. **Itm_qty**: Quantity of Items sold.
9. **Item_price**: Individual Price of the Item.
10. **Total_price**: Total sales revenue from the Transaction

3.1.2 **Exploratory Data Analysis:**

We dive deeper into the details of retail that we have to understand it better and be able to pre-process and analyze it better for the sake of our project.

This is how our data looks like at this point.

Trx_id	Timestamp	Member_id	Item_id	Item_name	Item_subclass	Item_subdepartment	Itm_qty	Item_price	Total_price
173131329	21/11/2022 0:02:36:236	5136784	1260	Tomato /kg	tomato	fresh vegetable	1	10.665	10.665
173131329	21/11/2022 0:02:36:236	5136784	50668	Lamb Weston Crunchy Seasoned Twisters 2.5 kg	spiced fries-onion	frozen food	1	26.55	26.55
173131329	21/11/2022 0:02:36:236	5136784	2116	Orange Juice /Kg	banana african	fresh fruits	1	24.273	24.273
173131329	21/11/2022 0:02:36:236	5136784	73732	Pinar Shredded Mozzarella Cheese 500g	shredded mozzarela cheese	cheese	2	37.8	75.6
173131329	21/11/2022 0:02:36:236	5136784	298472	Radwa Fried Chicken 900g	chicken drumsticks	frozen food	1	28.575	28.575

Since we require Time intelligence functions we converted this text timestamp to pd.datetime format to have more flexibility over time.

```
df2['Timestamp'] = pd.to_datetime(df2['Timestamp'])
```

Now we notice that we cannot draw meaningful insights at first glance due to the grain of timestamp being of seconds so we would bucketize the timestamp into parts of days to gain more insights over customer – user interaction bound by time domain.

```
def get_time_of_day(hour):
    if 5 <= hour < 8:
        return 'early_morning'
    elif 8 <= hour < 12:
        return 'morning'
    elif 12 <= hour < 17:
        return 'noon'
    elif 17 <= hour < 20:
        return 'evening'
    elif 20 <= hour < 24:
        return 'night'
    else:
        return 'late_night'

df2['hour'] = df2['Timestamp'].dt.hour
df2['time_of_day'] = df2['hour'].apply(get_time_of_day)
```

and then we create an aggregated table for graphical representation as follows:

Member_id	Itami'd	Item subclass	Item subdepartment	time_of_day	Itm_qty	Item price	Total price
4	1164	banana	fresh fruits	Evening	2	12.447	12.447
4	1996	zero vat	zero vat items	Evening	1	180	180
4	2192	laban full fat	laban & labneh	Noon	1	1.8	1.8
4	3200	fresh milk full fat	Milk	Evening	1	5.4	5.4
4	4276	dates	dates (supplier)	late-night	1	20.25	20.25

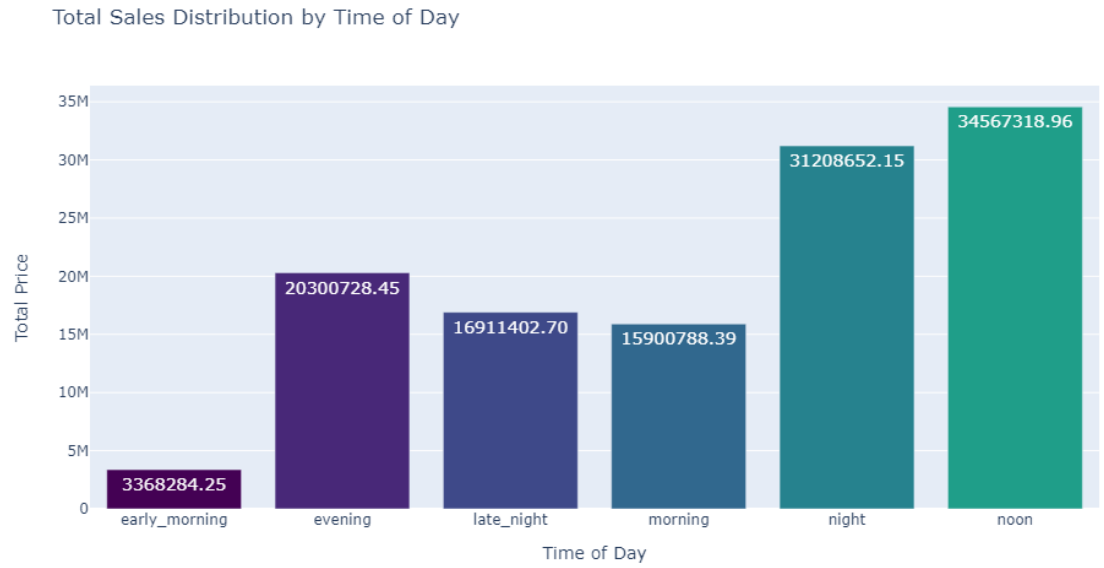

Figure 7

From this trend we notice that we have highest sales in Noon followed by night and evening and very low transactions being done in early mornings.

Now we will look into how different subdepartments are performing against each other so we know the customer demands in the area. This could give us good insight for recommendations since we can have general since of distribution of products amongst users to be able to detect whether model is leaning to right direction or not early on.

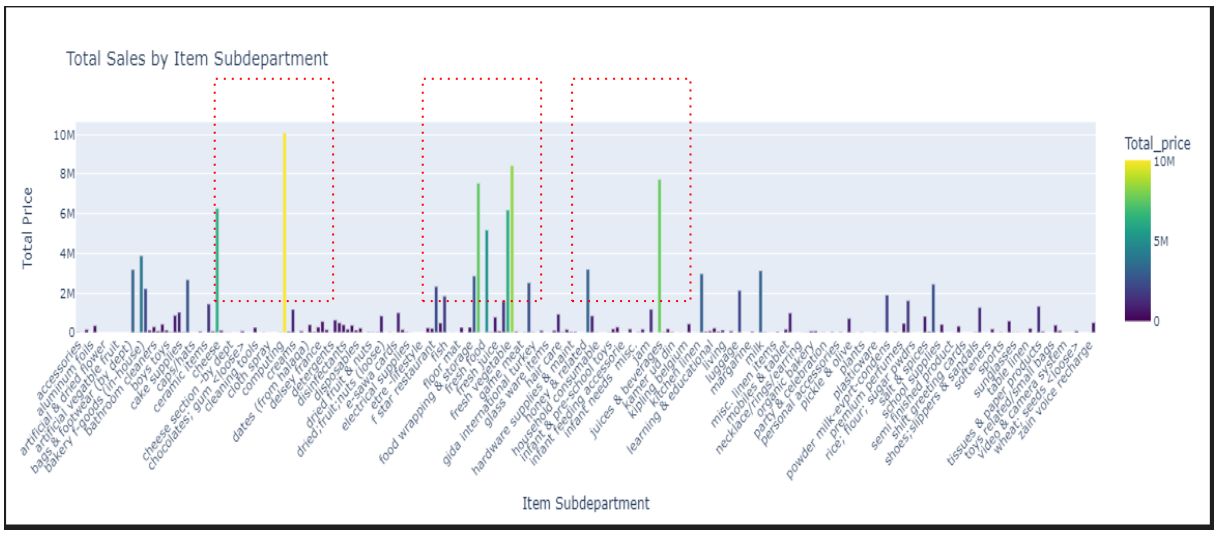

Figure 8

We can see the following sub departments to be on top 5 according to the total sales that they have generated:

1. Confectionary – Bakery related items or snacks.
2. Frozen Food
3. Juices and Beverages
4. Fresh Chicken
5. Fresh Vegetables

Now we get into numerical features to find out how they're distributed, as it happens that in every retail scenario we have different type of customers having their own respective value bands and loyalty to the store. Some preferring this store over the others and shopping heavily from here, then we have seasonal peaks as well. Following is the boxplot for value distribution:

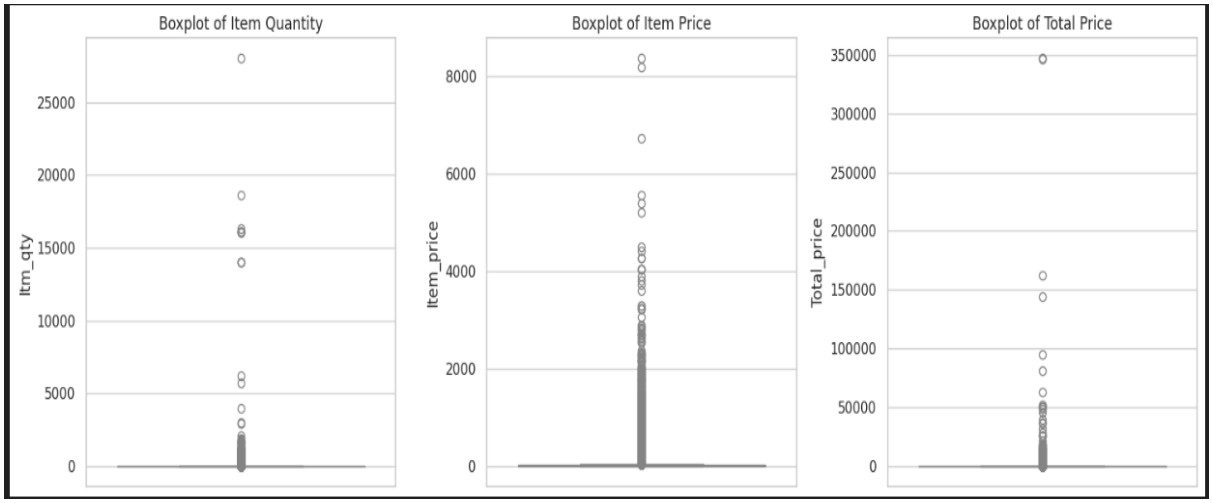

Figure 9

Let's also look at what are general spending by customer is in this store, this can be called the most frequent price bracket by our customers and can give us a chance to establish a baseline and from their using the recommendation system strive to increase the customer spend.

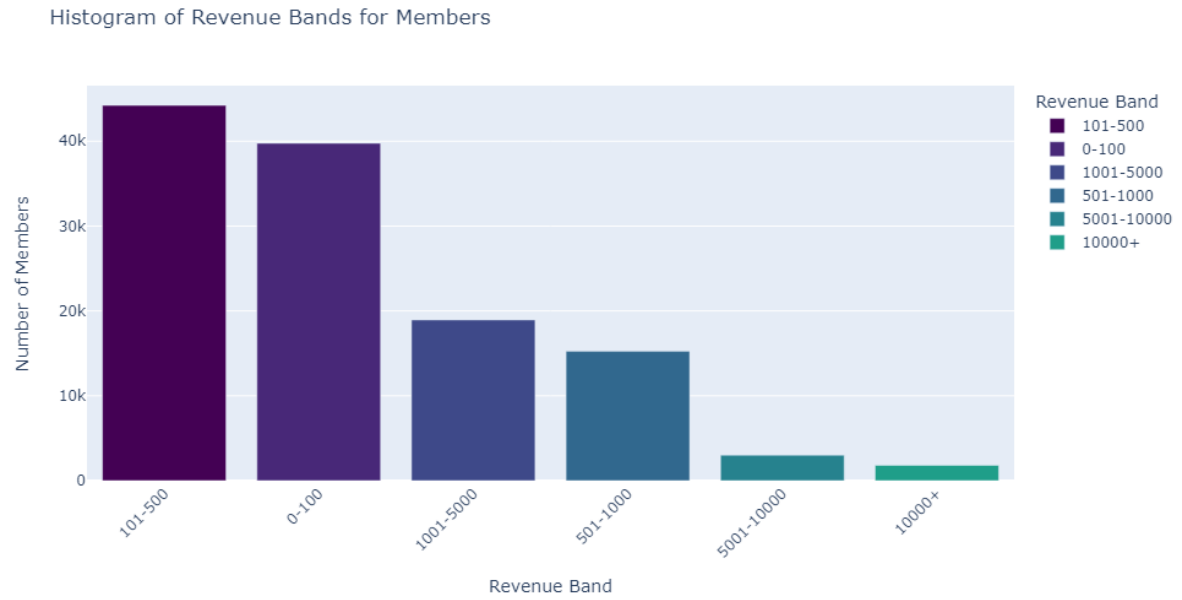

Figure 10

3.1.3 Feature Engineering:

We are going to perform feature engineering on the above features to produce to separate datasets one for members and one for items. Members dataset will have all the features related to members and items dataset will have all the features related to items.
Both member and item features will now be further cleaned where missing categorical features will be tagged as 'NA' and missing Numerical features will be changed to 0.

Member Feature Engineering:

```
current_date = df2['Timestamp'].max()
# Compute additional features
member_features_df = df2.groupby('Member_id').agg(
    Total_spend=('Total_price', 'sum'),
    Total_trx=('Trx_id', 'nunique'),
    Total_item=('Item_id', 'count'),
    Total_unique_items=('Item_id','nunique'),
    Total_unique_subclasses=('Item_subclass','nunique'),
    Total_unique_subdepartments=('Item_subdepartment','nunique'),
    Min_spend=('Total_price', 'min'),
    Max_spend=('Total_price','max'),
    days_since_last_purchase=('Timestamp', lambda x: (current_date - x.max()).days),
    days_since_first_purchase=('Timestamp', lambda x: (current_date - x.min()).days),
).reset_index()
```

```
member_features_df['spend_per_trx'] = member_features_df['Total_spend'] /
member_features_df['Total_trx']
member_features_df['Items_per_trx'] = np.ceil(member_features_df['Total_item'] /
member_features_df['Total_trx'])
member_features_df['Unique_items_per_trx'] = np.ceil(member_features_df['Total_unique_items'] /
member_features_df['Total_trx'])

periods = [7, 15, 60, 90]

# Most Frequently Purchased Products - FRESH (Top 5)
pattern_fresh = r'\bfresh\b'
def get_top_fresh_products_for_periods(df, periods, pattern_fresh):
    # Calculate the maximum timestamp for each member
    max_timestamp = df.groupby('Member_id')['Timestamp'].max().reset_index()
    max_timestamp.columns = ['Member_id', 'Max_Timestamp']

    # Merge the max_timestamp DataFrame with the original df to filter data
    df_with_max = pd.merge(df, max_timestamp, on='Member_id')

    # Initialize a list to store results for each period
    results = []

    for days in periods:
        # Calculate the start date for the period relative to each member's maximum timestamp
        df_with_max['Start_Date'] = df_with_max['Max_Timestamp'] - pd.Timedelta(days=days)

        # Filter to include only the data for the specified period
        filtered_data = df_with_max[(df_with_max['Timestamp'] >= df_with_max['Start_Date']) &
                                    (df_with_max['Timestamp'] <= df_with_max['Max_Timestamp'])]

        # Find most frequently purchased fresh products
        product_freq_fresh = filtered_data[filtered_data['Item_subdepartment'].str.contains(pattern_fresh,
case=False, regex=True)]
        product_freq_fresh = product_freq_fresh.groupby(['Member_id',
'Item_id']).size().reset_index(name='counts')
        product_freq_fresh['rank'] = product_freq_fresh.groupby('Member_id')['counts'].rank(method='first',
ascending=False)

        # Get the top products based on rank
        top_products = product_freq_fresh[product_freq_fresh['rank'] <= 5]

        # Pivot table to get top products for each member
        top_products_pivot = top_products.pivot_table(
            index='Member_id',
            columns='rank',
            values='Item_id',
            aggfunc=lambda x: ', '.join(map(str, x))  # Aggregate product IDs as comma-separated strings
        ).reset_index()

        # Rename columns
        top_products_pivot.columns = [
```

```
            'Member_id'
        ] + [f'top_{int(col)}_{days}_day_fresh_product' for col in top_products_pivot.columns if col != 'Member_id']

        # Append the result to the list
        results.append(top_products_pivot)

    # Merge all results on 'Member_id'
    combined_results = results[0]
    for result in results[1:]:
        combined_results = pd.merge(combined_results, result, on='Member_id', how='outer')

    # Ensure no duplicate 'Member_id' columns
    combined_results = combined_results.loc[:, ~combined_results.columns.duplicated()]

    return combined_results

# Most Frequently Purchased Products - NON-FRESH (Top 5)
def get_top_non_fresh_products_for_periods(df, periods, pattern_fresh):
    # Calculate the maximum timestamp for each member
    max_timestamp = df.groupby('Member_id')['Timestamp'].max().reset_index()
    max_timestamp.columns = ['Member_id', 'Max_Timestamp']

    # Merge the max_timestamp DataFrame with the original df to filter data
    df_with_max = pd.merge(df, max_timestamp, on='Member_id')

    # Initialize a list to store results for each period
    results = []

    for days in periods:
        # Calculate the start date for the period relative to each member's maximum timestamp
        df_with_max['Start_Date'] = df_with_max['Max_Timestamp'] - pd.Timedelta(days=days)

        # Filter to include only the data for the specified period
        filtered_data = df_with_max[(df_with_max['Timestamp'] >= df_with_max['Start_Date']) &
                                    (df_with_max['Timestamp'] <= df_with_max['Max_Timestamp'])]

        # Find most frequently purchased non-fresh products
        product_freq_non_fresh = filtered_data[~filtered_data['Item_subdepartment'].str.contains(pattern_fresh, case=False, regex=True)]
        product_freq_non_fresh = product_freq_non_fresh.groupby(['Member_id', 'Item_id']).size().reset_index(name='counts')
        product_freq_non_fresh['rank'] = product_freq_non_fresh.groupby('Member_id')['counts'].rank(method='first', ascending=False)

        # Get the top products based on rank
        top_products = product_freq_non_fresh[product_freq_non_fresh['rank'] <= 5]

        # Pivot table to get top products for each member
        top_products_pivot = top_products.pivot_table(
            index='Member_id',
```

```
            columns='rank',
            values='Item_id',
            aggfunc=lambda x: ', '.join(map(str, x))  # Aggregate product IDs as comma-separated strings
        ).reset_index()

        # Rename columns
        top_products_pivot.columns = [
            'Member_id'
        ] + [f'top_{int(col)}_{days}_day_non_fresh_product' for col in top_products_pivot.columns if col != 'Member_id']

        # Append the result to the list
        results.append(top_products_pivot)

    # Merge all results on 'Member_id'
    combined_results = results[0]
    for result in results[1:]:
        combined_results = pd.merge(combined_results, result, on='Member_id', how='outer')

    # Ensure no duplicate 'Member_id' columns
    combined_results = combined_results.loc[:, ~combined_results.columns.duplicated()]

    return combined_results

top_5_products_fresh_pivot = get_top_fresh_products_for_periods(df2, periods, pattern_fresh)
member_features_df = member_features_df.merge(top_5_products_fresh_pivot, on='Member_id', how='left')
top_5_products_non_fresh_pivot = get_top_non_fresh_products_for_periods(df2, periods, pattern_fresh)
member_features_df = member_features_df.merge(top_5_products_non_fresh_pivot, on='Member_id', how='left')
```

These will be new features related to Members:

1	Member_id	34	top_5_90_day_fresh_product
2	Total_spend	35	top_1_7_day_non_fresh_product
3	Total_trx	36	top_2_7_day_non_fresh_product
4	Total_item	37	top_3_7_day_non_fresh_product
5	Total_unique_items	38	top_4_7_day_non_fresh_product
6	Total_unique_subclasses	39	top_5_7_day_non_fresh_product
7	Total_unique_subdepartments	40	top_1_15_day_non_fresh_product

8	Min_spend	41	top_2_15_day_non_fresh_product
9	Max_spend	42	top_3_15_day_non_fresh_product
10	days_since_last_purchase	43	top_4_15_day_non_fresh_product
11	days_since_first_purchase	44	top_5_15_day_non_fresh_product
12	spend_per_trx	45	top_1_60_day_non_fresh_product
13	Items_per_trx	46	top_2_60_day_non_fresh_product
14	Unique_items_per_trx	47	top_3_60_day_non_fresh_product
15	top_1_7_day_fresh_product	48	top_4_60_day_non_fresh_product
16	top_2_7_day_fresh_product	49	top_5_60_day_non_fresh_product
17	top_3_7_day_fresh_product	50	top_1_90_day_non_fresh_product
18	top_4_7_day_fresh_product	51	top_2_90_day_non_fresh_product
19	top_5_7_day_fresh_product	52	top_3_90_day_non_fresh_product
20	top_1_15_day_fresh_product	53	top_4_90_day_non_fresh_product
21	top_2_15_day_fresh_product	54	top_5_90_day_non_fresh_product
22	top_3_15_day_fresh_product	41	top_2_15_day_non_fresh_product
23	top_4_15_day_fresh_product	42	top_3_15_day_non_fresh_product
24	top_5_15_day_fresh_product	43	top_4_15_day_non_fresh_product
25	top_1_60_day_fresh_product	44	top_5_15_day_non_fresh_product
26	top_2_60_day_fresh_product	45	top_1_60_day_non_fresh_product
27	top_3_60_day_fresh_product	46	top_2_60_day_non_fresh_product
28	top_4_60_day_fresh_product	47	top_3_60_day_non_fresh_product
29	top_5_60_day_fresh_product	48	top_4_60_day_non_fresh_product
30	top_1_90_day_fresh_product	49	top_5_60_day_non_fresh_product
31	top_2_90_day_fresh_product	50	top_1_90_day_non_fresh_product

3		5	top_2_90_day_non_fresh_produc
2	top_3_90_day_fresh_product	1	t
3		5	top_3_90_day_non_fresh_produc
3	top_4_90_day_fresh_product	2	t
		5	top_4_90_day_non_fresh_produc
		3	t
		5	top_5_90_day_non_fresh_produc
		4	t

Amongst these features we have following correlation matrix:

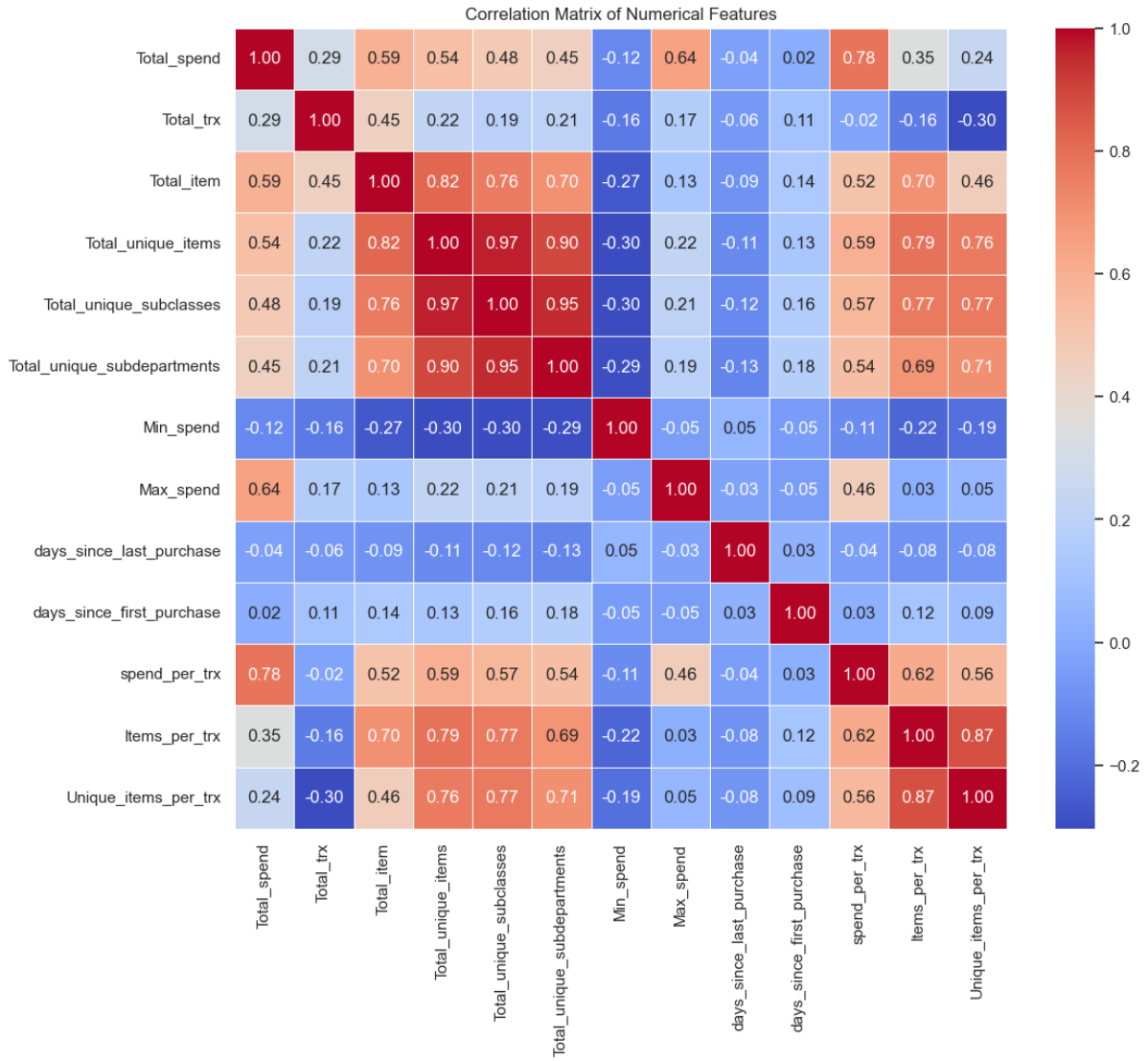

Figure 11

Item Feature Engineering

Code:

```
# Compute additional features
item_features = df2.groupby('Item_id').agg(
    total_purchases=('Item_id', 'count'),
    total_spend=('Total_price', 'sum'),
    total_Itm_qty_purchased=('Itm_qty', 'sum')
).reset_index()

# purchase_frequency_by_day_time = df2.groupby(['Item_id',
'day_time_combination']).size().unstack(fill_value=0)
# purchase_frequency_by_day_time.columns = [f'Trx_{day_time}' for day_time in
purchase_frequency_by_day_time.columns]

item_features.head()
periods = [7, 15, 60, 90]
Max_timestamp_all = df2['Timestamp'].max()

# item_purchase_day_time = df2.groupby(['Item_id',
'day_time_combination']).size().unstack(fill_value=0)
# item_purchase_day_time.columns = [f'Trx_{day_time}' for day_time in
item_purchase_day_time.columns]

results = []

def rename_columns(df, period):
    columns = {}
    for col in df.columns:
        if col not in ['Item_id', 'Item_name', 'Item_subclass', 'Item_subdepartment']:
            columns[col] = f"{col}_{period}_days"
    df.rename(columns=columns, inplace=True)
    return df

for days in periods:
    # Calculate the start date for the period relative to each member's maximum timestamp
    df2['Start_Date'] = Max_timestamp_all - pd.Timedelta(days=days)

    # Filter to include only the data for the specified period
    filtered_data = df2[(df2['Timestamp'] >= df2['Start_Date']) &
                        (df2['Timestamp'] <= Max_timestamp_all)]

    # Find most frequently purchased fresh products
    Trx_count_items = filtered_data.groupby('Item_id')['Trx_id'].count().reset_index()
    Trx_count_items.columns = ['Item_id', 'Trx_count']  # Rename column for clarity
```

```
    item_sales = filtered_data.groupby(['Item_id', 'Item_subclass',
'Item_subdepartment'])[['Total_price','Itm_qty']].sum().reset_index()

    # Merge item_sales with Trx_count_items on 'Item_id'
    item_sales = item_sales.merge(Trx_count_items, on='Item_id', how='outer')

    # Rename columns with appropriate suffix
    item_sales = rename_columns(item_sales, days)

    # Append the result to the list
    results.append(item_sales)

# Merge all period dataframes into one dataframe
final_df = results[0]
for i in range(1, len(results)):
    final_df = final_df.merge(results[i], on=['Item_id', 'Item_subclass', 'Item_subdepartment'], how='outer')

# Fill NaN values with 0 if necessary
final_df.fillna(0, inplace=True)
```

These will be new features related to Items:

1 Item_id
2 Item_subclass
3 Item_subdepartment
4 Total_price_7_days
5 Itm_qty_7_days
6 Trx_count_7_days
7 Total_price_15_days
8 Itm_qty_15_days
9 Trx_count_15_days
10 Total_price_60_days
11 Itm_qty_60_days
12 Trx_count_60_days
13 Total_price_90_days
14 Itm_qty_90_days
15 Trx_count_90_days

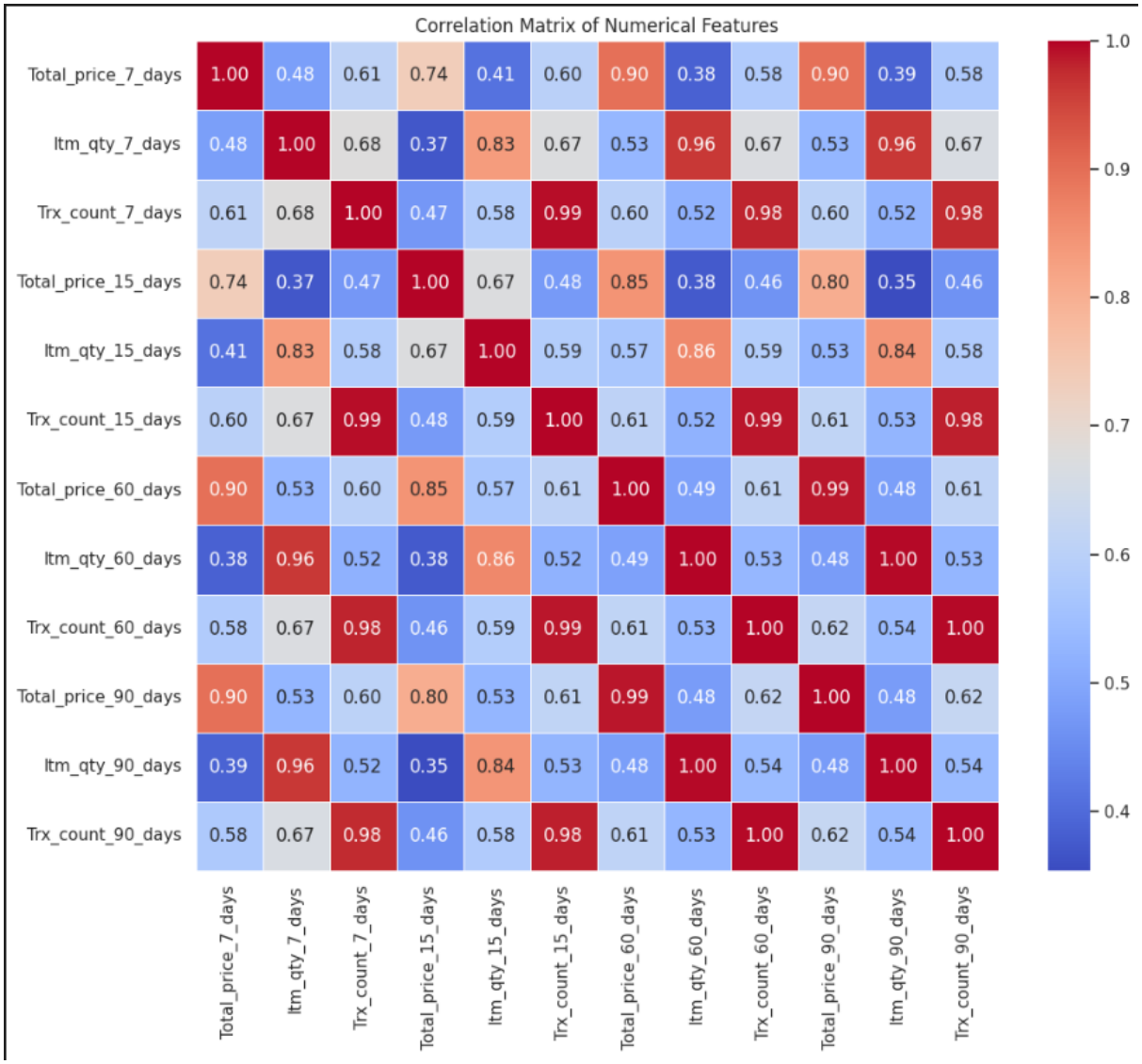

Figure 12

3.2 Methodology of Recommendation Systems: Retrieval and Ranking.

3.2.1 General Methodology: Retrieval and Ranking

In the context of recommendation systems, the process is generally divided into two main stages: **Retrieval** and **Ranking**.

- **Retrieval**: This stage is responsible for selecting a subset of potentially relevant items from a large pool of candidates. The aim is to reduce the search space by filtering out

most items, leaving only those that are most likely to be of interest to the user. The retrieval process involves learning and comparing embeddings for both users and items, such that the similarity between these embeddings reflects the user's preferences.

- **Ranking**: After retrieval, the selected subset of items is further refined. The ranking model evaluates the relevance of each item to the user in much greater detail. This is where complex interactions between user and item features are modelled, typically using deep neural networks. The final output is a ranked list of items, ordered by their predicted relevance to the user.

3.2.2 In-Depth Analysis of Retrieval with the Two-Towered Approach

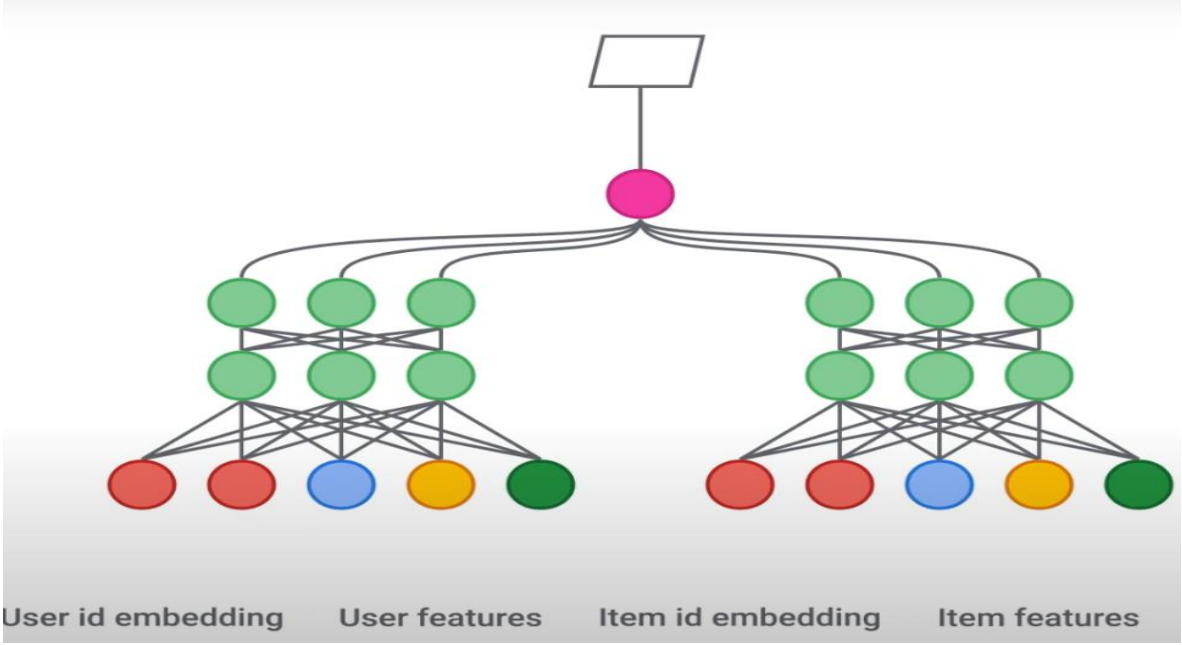

Figure 13

MemberModel Class (User Tower):

The MemberModel is a neural network that transforms user-related features into a dense vector (embedding) that encapsulates the user's preferences. This model specifically handles both categorical and numerical features.
The **MemberModel** class extends the **tf.keras.Model** class, implementing a custom neural network architecture for embedding and normalizing features. The model incorporates embedding layers for categorical features and normalization layers for numerical features. The primary objective of this model is to process input features, convert them into embeddings or normalized values, and concatenate these features into a final representation suitable for further model training or inference.

Model Components

1. **Initialization (__init__ Method):**

- **Member ID Lookup and Embedding Layer:**
  - The model initializes a StringLookup layer to convert Member_id values into integer indices. This is followed by an Embedding layer that maps these indices to dense vectors of a specified dimension (embedding_dim).
- **Categorical Feature Embeddings:**
  - For each categorical feature, a StringLookup layer is created to handle the conversion from string values to integer indices. An associated Embedding layer transforms these indices into dense vectors.

$$\mathbf{e}_{uk} = \text{Embedding}(x_{uk})$$

- **Numerical Feature Normalization:**
  - Numerical features are processed using LayerNormalization layers, which normalize the input features to ensure that they have a mean of zero and a standard deviation of one.

$$\bar{x}_{un} = (\sigma_n)/(x_{un} - \mu_n))$$

2. **Forward Pass (call Method):**
   - **Member ID Processing:**
     - The model extracts the Member_id feature from the input batch. The IDs are converted into indices using the StringLookup layer, which are then transformed into embeddings using the corresponding Embedding layer. The resulting tensor represents the embedded member IDs.
   - **Categorical Feature Processing:**
     - For each categorical feature (excluding Member_id), the model performs a lookup operation to obtain indices, which are subsequently passed through their respective embedding layers to obtain feature embeddings. These embeddings are concatenated along the last axis to form a single tensor.
   - **Numerical Feature Processing:**
     - Numerical features are normalized using their respective LayerNormalization layers. If the normalized values are one-dimensional, they are reshaped to ensure they have the correct number of dimensions. These normalized values are concatenated to form a tensor.
   - **Concatenation of Features:**
     - The embeddings from member IDs, categorical features, and normalized numerical features are concatenated into a single tensor. This final tensor aggregates all feature representations, which is suitable for further processing or model input.

Code:

```
import tensorflow as tf
```

```
class MemberModel(tf.keras.Model):
    def __init__(self, unique_member_ids, unique_features_categorical, unique_features_numerical, embedding_dim=3):
        super().__init__()
        self.embedding_dim = embedding_dim

        # Initialize Member_id lookup and embedding layer
        self.member_lookup = tf.keras.layers.StringLookup(vocabulary=unique_member_ids, mask_token=None, oov_token=None)
        self.member_embedding_layer = tf.keras.layers.Embedding(input_dim=self.member_lookup.vocabulary_size(), output_dim=embedding_dim)

        # Initialize categorical embedding layers
        self.embedding_layers = {
            feature_name: (
                tf.keras.layers.StringLookup(vocabulary=vocabulary, mask_token=None, oov_token=None),
                tf.keras.layers.Embedding(input_dim=len(vocabulary), output_dim=embedding_dim)
            ) for feature_name, vocabulary in unique_features_categorical.items()
        }

        # Initialize normalization layers for numerical features
        self.normalization_layers = {
            feature_name: tf.keras.layers.LayerNormalization(axis=-1, epsilon=1e-6)



for feature_name in unique_features_numerical
        }

    def call(self, member_batch):
        # Extract features from the batch
        member_features = member_batch

        # Process Member_id
        member_ids = member_features.get('Member_id', None)
        if member_ids is not None:
            member_id_indices = self.member_lookup(member_ids)
            member_embeddings = self.member_embedding_layer(member_id_indices)
            print(f"Shape of member_embeddings: {member_embeddings.shape}")
        else:
            member_embeddings = None

        # Process other categorical features
        categorical_embeddings = []
        for feature_name, values in member_features.items():
            if feature_name != 'Member_id' and feature_name in self.embedding_layers:
                lookup_layer, embedding_layer = self.embedding_layers[feature_name]
                indices = lookup_layer(values)
                feature_embeddings = embedding_layer(indices)
                categorical_embeddings.append(feature_embeddings)
```

```
            print(f"Shape of {feature_name} categorical embedding: {feature_embeddings.shape}")

        # Concatenate all categorical features
        if categorical_embeddings:
            categorical_tensor = tf.concat(categorical_embeddings, axis=-1)
        else:
            categorical_tensor = None

        # Process numerical features
        numerical_normalized = []
        for feature_name, values in member_features.items():
            if feature_name in self.normalization_layers:
                normalization_layer = self.normalization_layers[feature_name]
                normalized_values = normalization_layer(values, training=False)

                # Ensure numerical values have the same number of dimensions as categorical embeddings
                if len(normalized_values.shape) == 1:
                    normalized_values = tf.expand_dims(normalized_values, axis=-1)

                numerical_normalized.append(normalized_values)
                print(f"Shape of {feature_name} numerical normalized values: {normalized_values.shape}")

        # Concatenate all numerical features
        if numerical_normalized:
            numerical_tensor = tf.concat(numerical_normalized, axis=-1)
        else:
            numerical_tensor = None

        # Concatenate categorical and numerical features
        all_features = []
        if member_embeddings is not None:
            all_features.append(member_embeddings)
        if categorical_tensor is not None:
            all_features.append(categorical_tensor)
        if numerical_tensor is not None:
            all_features.append(numerical_tensor)

        if all_features:
            final_tensor = tf.concat(all_features, axis=-1)
            print(f"Shape of final concatenated tensor: {final_tensor.shape}")
        else:
            final_tensor = None
            print("No features to concatenate, final_tensor is None.")

        return final_tensor

member_model = MemberModel(
    unique_member_ids=Unique_member_ids_tf,
    unique_features_categorical=unique_features_members_categorical_tf,
    unique_features_numerical=unique_features_members_numerical_tf,
    embedding_dim=3
```

```
)
```

The **MemberModel** class is designed to handle a mix of categorical and numerical features. It efficiently transforms categorical data into dense embeddings and normalizes numerical data, concatenating these feature representations into a unified tensor. This model can be utilized in various contexts, such as recommendation systems or any scenario requiring feature embedding and normalization. The detailed feature processing ensures that both categorical and numerical data are appropriately represented, facilitating effective training and inference in downstream tasks.

ItemModel Class (Candidate Tower):

The **ItemModel** class extends **tf.keras.Model**, creating a custom architecture for processing item-related features. The model integrates embedding layers for categorical features and normalization layers for numerical features.

Model Components

1. **Initialization (__init__ Method):**
   - **Item ID Lookup and Embedding Layer:**
     - A StringLookup layer is initialized to convert Item_id values into integer indices. An Embedding layer maps these indices to dense vectors of a specified dimension (embedding_dim).
   - **Categorical Feature Embeddings:**
     - For each categorical feature, a StringLookup layer is created to map string values to integer indices. This is followed by an Embedding layer that transforms these indices into dense vectors.

$$\mathbf{e}_{uk} = \text{Embedding}(x_{uk})$$

   - **Numerical Feature Normalization:**
     - Numerical features are processed using LayerNormalization layers, which normalize the features to have zero mean and unit variance.

$$\bar{x}_{un} = (\sigma_n)/(x_{un} - \mu_n))$$

2. **Forward Pass (call Method):**
   - **Item ID Processing:**
     - The model extracts the Item_id feature from the input batch. The StringLookup layer converts these IDs into integer indices, which are then embedded into dense vectors using the Embedding layer. The resulting tensor represents the item embeddings.
   - **Categorical Feature Processing:**
     - For each categorical feature (excluding Item_id), the model performs a lookup to obtain indices and then passes these indices through their respective embedding layers to get feature embeddings. These embeddings are concatenated along the last axis to create a single tensor.

$$\mathbf{e}_{uk} = \text{Embedding}(x_{uk})$$

- **Numerical Feature Processing:**
  - Numerical features are normalized using their corresponding LayerNormalization layers. If the normalized values are one-dimensional, they are reshaped to ensure compatibility with the dimensions of categorical embeddings. These normalized values are then concatenated.

$$\bar{x}_{\mathrm{un}} = (\sigma_{\mathrm{n}})/(x_{\mathrm{un}} - \mu_{\mathrm{n}})$$

- **Concatenation of Features:**
  - The embeddings from item IDs, categorical features, and numerical features are concatenated into a single tensor. This final tensor combines all processed features, preparing them for further model training or evaluation.

$$\mathbf{e}_{\mathrm{v}} = \text{Concatenate}\ ([\mathbf{e}_{\mathrm{v}1}, \mathbf{e}_{\mathrm{v}2}, \ldots, \bar{\mathrm{x}}_{\mathrm{v}1}, \bar{\mathrm{x}}_{\mathrm{v}2}, \ldots])$$

Code:

```
import tensorflow as tf

class ItemModel(tf.keras.Model):
    def __init__(self, unique_item_ids, unique_features_categorical, unique_features_numerical, embedding_dim=3):
        super().__init__()
        self.embedding_dim = embedding_dim

        # Initialize item_id lookup and embedding layer
        self.item_lookup = tf.keras.layers.StringLookup(vocabulary=unique_item_ids, mask_token=None, oov_token=None)
        self.item_embedding_layer = tf.keras.layers.Embedding(input_dim=self.item_lookup.vocabulary_size(), output_dim=embedding_dim)

        # Initialize categorical embedding layers
        self.embedding_layers = {
            feature_name: (
                tf.keras.layers.StringLookup(vocabulary=vocabulary, mask_token=None, oov_token=None),
                tf.keras.layers.Embedding(input_dim=len(vocabulary), output_dim=embedding_dim)
            ) for feature_name, vocabulary in unique_features_categorical.items()
        }

        # Initialize normalization layers for numerical features
        self.normalization_layers = {
            feature_name: tf.keras.layers.LayerNormalization(axis=-1, epsilon=1e-6)
            for feature_name in unique_features_numerical
```

```
        }

    def call(self, item_batch):
        # Extract features from the batch
        item_features = item_batch

        # Process item_id
        item_ids = item_features.get('Item_id', None)
        if item_ids is not None:
            item_id_indices = self.item_lookup(item_ids)
            item_embeddings = self.item_embedding_layer(item_id_indices)
            print(f"Shape of item_embeddings: {item_embeddings.shape}")
        else:
            item_embeddings = None

        # Process other categorical features
        categorical_embeddings = []
        for feature_name, values in item_features.items():
            if feature_name != 'Item_id' and feature_name in self.embedding_layers:
                lookup_layer, embedding_layer = self.embedding_layers[feature_name]
                indices = lookup_layer(values)
                feature_embeddings = embedding_layer(indices)
                categorical_embeddings.append(feature_embeddings)
                print(f"Shape of {feature_name} categorical embedding: {feature_embeddings.shape}")

        # Concatenate all categorical features
        if categorical_embeddings:
            categorical_tensor = tf.concat(categorical_embeddings, axis=-1)
        else:
            categorical_tensor = None

        # Process numerical features
        numerical_normalized = []
        for feature_name, values in item_features.items():
            if feature_name in self.normalization_layers:
                normalization_layer = self.normalization_layers[feature_name]
                normalized_values = normalization_layer(values, training=False)

                # Ensure numerical values have the same number of dimensions as categorical embeddings
                if len(normalized_values.shape) == 1:
                    normalized_values = tf.expand_dims(normalized_values, axis=-1)

                numerical_normalized.append(normalized_values)
                print(f"Shape of {feature_name} numerical normalized values: {normalized_values.shape}")

        # Concatenate all numerical features
        if numerical_normalized:
            numerical_tensor = tf.concat(numerical_normalized, axis=-1)
        else:
            numerical_tensor = None

```

```
        # Concatenate categorical and numerical features
        all_features = []
        if item_embeddings is not None:
            all_features.append(item_embeddings)
        if categorical_tensor is not None:
            all_features.append(categorical_tensor)
        if numerical_tensor is not None:
            all_features.append(numerical_tensor)

        if all_features:
            final_tensor = tf.concat(all_features, axis=-1)
            print(f"Shape of final concatenated tensor: {final_tensor.shape}")
        else:
            final_tensor = None
            print("No features to concatenate, final_tensor is None.")

        return final_tensor

```

The **ItemModel** class is designed to efficiently handle a mix of categorical and numerical features associated with items. It processes categorical data through embeddings and numerical data through normalization, ultimately concatenating these features into a unified tensor. This model architecture is suitable for tasks such as recommendation systems, where diverse feature types need to be transformed and integrated into a coherent representation for further use. The clear separation of feature processing and the integration of different feature types ensure that the model can effectively represent and utilize item-related data.

Retrieval Stage

The **RetrievalModel** integrates both the **MemberModel** and **RetrievalModel** to form a complete retrieval system. This model is designed to learn representations (embeddings) such that the similarity between user and item embeddings reflects the likelihood that the user will interact with or prefer the item.

Input to the Retrieval Stage:

- The output from **MemberModel** (user embeddings).
- The output from **ItemModel** (item embeddings).

With both user embeddings and item embeddings computed by the MemberModel and ItemModel, respectively, the system proceeds to the core operation of the retrieval stage: **similarity computation**. The objective is to determine how closely aligned a user's preferences are with the characteristics of each item.

1. **Similarity Computation**: The similarity between user and item embeddings is computed using a dot product:

$$s(u, v) = e_u \cdot e_v$$

This similarity score is a scalar value that quantifies the alignment between the user and item embeddings. When the embeddings are normalized (i.e., when each embedding vector has a unit norm), the dot product is equivalent to the **cosine similarity**:

$$cosine\ similarity(u, v) = (\| e_u \| \| e_v \|)^{-1}(e_u \cdot e_v)$$

The cosine similarity ranges from -1 to 1, where 1 indicates perfect alignment, 0 indicates orthogonality (no similarity), and -1 indicates complete opposition. In the context of recommendation systems, we focus on positive similarity scores, as we are interested in items that are most like the user's preferences.

Output of the Retrieval Stage:

Selection of Top Candidates

Once the similarity scores between the user and all potential items are computed, the system ranks the items based on these scores. The top KKK items with the highest similarity scores are selected as the candidates for the next stage in the recommendation pipeline—the ranking stage. The value of KKK is chosen based on the specific application and constraints of the system, balancing the need for diversity with computational efficiency.

This retrieval stage is designed to be both scalable and efficient, enabling the system to handle large-scale recommendation tasks where the pool of potential items can be vast. The embeddings serve as a compact and expressive representation of the underlying data, allowing the system to efficiently compare users and items in a high-dimensional space.

In summary, the retrieval stage of a recommendation system leverages deep learning-based embeddings to efficiently match users with relevant items. By transforming complex feature sets into dense vector representations, the system can compute similarity scores that drive the selection of top candidates. This stage is a critical component of modern recommendation engines, enabling personalized and scalable recommendations at scale.

Code:

The SimilarityLayer class is crucial for calculating the similarity between member and item embeddings. It projects these embeddings to a common space, normalizes them, and then computes the similarity.

```
class SimilarityLayer(tf.keras.layers.Layer):
    def __init__(self, embedding_dim, temperature=1.0, **kwargs):
        super(SimilarityLayer, self).__init__(**kwargs)
        self.temperature = temperature
        # Define projection layers
        self.member_projection = tf.keras.layers.Dense(embedding_dim, activation='relu')
        self.item_projection = tf.keras.layers.Dense(embedding_dim, activation='relu')
```

```
def call(self, member_embeddings, item_embeddings):
    # Ensure both tensors have the same data type
    dtype = tf.float32
    member_embeddings = tf.cast(member_embeddings, dtype)
    item_embeddings = tf.cast(item_embeddings, dtype)

    # Project embeddings to the same dimension
    member_embeddings = self.member_projection(member_embeddings)
    item_embeddings = self.item_projection(item_embeddings)

    # Normalize the embeddings
    member_embeddings = tf.nn.l2_normalize(member_embeddings, axis=-1)
    item_embeddings = tf.nn.l2_normalize(item_embeddings, axis=-1)

    # Compute the similarity as the dot product
    similarity = tf.matmul(member_embeddings, item_embeddings, transpose_b=True)

    # Scale the similarity by temperature
    similarity = similarity / self.temperature

    return similarity
```

- **Projection**: *member_projection* and *item_projection* are fully connected (Dense) layers that map the member and item embeddings to the same dimensional space, specified by embedding_dim. The ReLU activation function introduces non-linearity.

$$e'_u = ReLU(W_u e_u + b_u)$$

- **Normalization**: ***tf.nn.l2_normalize*** normalizes the embeddings so that their L2 norms are 1. This converts the embeddings to unit vectors, making the dot product equivalent to cosine similarity.
- **Similarity Computation**: ***tf.matmul*** calculates the dot product between normalized member and item embeddings. The ***transpose_b=True*** argument ensures the matrix multiplication computes the similarity matrix.

$$Similarity(u, v) = \frac{(e_u \cdot e_v)}{temperature}$$

- **Temperature Scaling**: The similarity scores are scaled by a temperature parameter. Higher temperatures make the distribution more uniform, while lower temperatures sharpen the distribution.

The **ContrastiveLoss** class defines the loss function used to train the model. It aims to maximize the similarity for positive pairs while minimizing it for negative pairs.

```
class RetrievalModelWithTopK(tfrs.Model):
    def __init__(self, member_model, item_model, similarity_model, k=10):
        super().__init__()
```

```
        self.member_model = member_model
        self.item_model = item_model
        self.similarity_model = similarity_model
        self.k = 10

        # Initialize the FactorizedTopK retrieval layer
        self.top_k = tfrs.layers.factorized_top_k.BruteForce()

    def retrieve_top_k(self, member_features, item_features):
        # Get embeddings from the models
        member_embeddings = self.member_model(member_features)
        item_embeddings = self.item_model(item_features)

        # Get similarity and projected embeddings from SimilarityLayer
        similarity, projected_member_embeddings, projected_item_embeddings =
self.similarity_model(member_embeddings, item_embeddings)

        # Index the item embeddings with the retrieval layer using the
projected embeddings
        self.top_k.index(projected_item_embeddings)

        # Get top-k items for the given member features using the projected
embeddings
        top_k_values, top_k_indices = self.top_k(projected_member_embeddings)

        return top_k_values, top_k_indices

    def compute_loss(self, member_features, item_features, training=False):
        top_k_values, top_k_indices = self.retrieve_top_k(member_features,
item_features)
        # Further processing...
        return top_k_values, top_k_indices
```

The **RetrievalModelWithTopK** class extends **tfrs.Model** to incorporate retrieval functionality with a top-K ranking system. This model combines embeddings from member and item models and uses a similarity model to rank items based on their relevance to a given member. Here's a breakdown of the class:

Mode Components

1. **Initialization (`__init__` method):**
    - **`member_model`**: A model to generate embeddings for members.
    - **`item_model`**: A model to generate embeddings for items.
    - **`similarity_model`**: A model that computes similarity between member and item embeddings.
    - **`k`**: The number of top items to retrieve (default is 10).

    - **`self.top_k`**: An instance of `tfrs.layers.factorized_top_k.BruteForce`, which is used for the top-K retrieval process.
2. **Retrieve Top-K (`retrieve_top_k` method):**
    - **`member_features`**: Features for which we want to retrieve relevant items.
    - **`item_features`**: Features of the items in the dataset.
    - **Get embeddings**: The `member_model` and `item_model` generate embeddings from the provided features.
    - **Compute similarity**: The `similarity_model` computes similarity scores and projects the embeddings.
    - **Index items**: The `top_k` layer is indexed with the projected item embeddings.
    - **Retrieve top-K**: The `top_k` layer is then used to find the top-K items relevant to the provided member features.
3. **Compute Loss (`compute_loss` method):**
    - Calls `retrieve_top_k` to get the top-K items and their indices.
    - The further processing of these values to compute the loss would depend on the specific implementation details, which are not provided in this snippet.

Summary

- **`RetrievalModelWithTopK`** integrates member and item embeddings with a similarity model to find and rank the top-K items relevant to a member.
- The class uses the `FactorizedTopK` layer for efficient retrieval and ranking.
- The `compute_loss` method, as shown, is currently a placeholder, with the real loss computation to be defined based on the retrieved top-K results.

Ranking Stage

1. RankingRetrievalModel

The RankingModel class is a custom Keras model designed for ranking tasks. It takes member and item embeddings as inputs and produces a ranking score. Here's a breakdown:

Model Components

- **Initialization (__init__ method)**:
    - Defines a sequence of dense layers. The model consists of three dense layers:
        - dense1: A dense layer with 128 units and ReLU activation.
        - dense2: A dense layer with 64 units and ReLU activation.
        - dense3: The final dense layer with 1 unit, which outputs the ranking score.
- **Forward Pass (call method)**:
    - Takes member_embeddings and item_embeddings as inputs.
    - Concatenates these embeddings along the last axis.
    - Passes the concatenated embeddings through the dense layers to compute the final ranking scores.

2. RankingRetrievalModel

The RankingRetrievalModel class is a custom model for retrieval tasks using TensorFlow Recommenders (TFRS). It incorporates member and item models, a similarity model, and the ranking model you defined. Here's a breakdown:

- **Initialization (__init__ method)**:
  - Takes four components:
    - member_model: A model that generates embeddings for members.
    - item_model: A model that generates embeddings for items.
    - similarity_model: A model or function for computing similarity between member and item embeddings.
    - ranking_model: The RankingModel instance that ranks items for a given member.
  - Initializes a BruteForce retrieval layer from TFRS to perform nearest neighbor search.
  - Stores k, the number of top items to retrieve.
- **Retrieve Top-K (retrieve_top_k method)**:
  - **Inputs**: member_features and item_features, which are features for members and items, respectively.
  - **Process**:
    - Obtains embeddings for members and items using the member_model and item_model.
    - Computes similarity and projects the embeddings using the similarity_model.
    - Prints the shapes of the projected embeddings for debugging purposes.
    - Indexes the projected item embeddings in the top_k retrieval layer.
    - Retrieves the top-K items for the given member features based on the projected embeddings.
  - **Outputs**: Returns the top-K values and indices for the items.
- **Compute Loss (compute_loss method)**:
  - **Inputs**: member_features and item_features. The training flag is not used in this method.
  - **Process**:
    - Calls retrieve_top_k to get top-K values and indices.
    - Returns these values and indices (though typically, you'd compute a loss here for training).

3 Key Points

- The RankingModel combines member and item embeddings to produce a ranking score.
- The RankingRetrievalModel integrates member and item models, performs retrieval using TFRS's BruteForce layer, and uses a ranking model to rank items.
- The retrieve_top_k method is designed to fetch the most relevant items for a given member based on embeddings and similarity.

The overall design aims to combine ranking and retrieval mechanisms to deliver a recommendation system where items are first retrieved based on similarity and then ranked using a more fine-grained model.

Code:

Ranking

```
import tensorflow as tf

class RankingModel(tf.keras.Model):
    def __init__(self, embedding_dim, **kwargs):
        super(RankingModel, self).__init__(**kwargs)
        # Define dense layers for the ranking model
        self.dense1 = tf.keras.layers.Dense(128, activation='relu')
        self.dense2 = tf.keras.layers.Dense(64, activation='relu')
        self.dense3 = tf.keras.layers.Dense(1)  # Output layer for ranking
score

    def call(self, member_embeddings, item_embeddings):
        # Concatenate member and item embeddings
        combined_embeddings = tf.concat([member_embeddings, item_embeddings],
axis=-1)

        # Pass through dense layers
        x = self.dense1(combined_embeddings)
        x = self.dense2(x)
        ranking_scores = self.dense3(x)

        return ranking_scores
```

Enhanced Retrieval and Ranking:

```
import tensorflow as tf
import tensorflow_recommenders as tfrs

class RankingRetrievalModel(tfrs.Model):
    def __init__(self, member_model, item_model, similarity_model,
ranking_model, k=10):
        super().__init__()
        self.member_model = member_model
        self.item_model = item_model
        self.similarity_model = similarity_model
        self.ranking_model = ranking_model
        self.k = k

        # Initialize the BruteForce retrieval layer
        self.top_k = tfrs.layers.factorized_top_k.BruteForce()

    def retrieve_top_k(self, member_features, item_features):
        # Get embeddings from the models
        member_embeddings = self.member_model(member_features)
        item_embeddings = self.item_model(item_features)

        # Get similarity and projected embeddings from SimilarityLayer
        similarity, projected_member_embeddings, projected_item_embeddings =
self.similarity_model(member_embeddings, item_embeddings)

        # Debug: Print shapes
        print(f"Projected member embeddings shape:
{projected_member_embeddings.shape}")
        print(f"Projected item embeddings shape:
{projected_item_embeddings.shape}")

        # Index the item embeddings with the retrieval layer using the
projected embeddings
        self.top_k.index(projected_item_embeddings)

        # Get top-k items for the given member features using the projected
embeddings
        top_k_values, top_k_indices = self.top_k(projected_member_embeddings)

        return top_k_values, top_k_indices

    def compute_loss(self, member_features, item_features, training=False):
        top_k_values, top_k_indices = self.retrieve_top_k(member_features,
item_features)
        return top_k_values, top_k_indices
```

Chapter 4

Experimental Results

So, we have performed our analysis for 1000 Members against full itemset to optimally utilize the available computational resources.

Total Number of Items in our Itemset: 34663

Total Number of Members Utilized in our trial: 1000

We have sampled these thousand members out of 120k on the basis of high transaction count, which means we are taking top 1000 customers having high transaction count and the reason for that is these are the customers who are regular shoppers in this stores and identifying their baskets could yield beneficial patterns rather than focusing on members with sparse purchase history which could lead to more confusion.

Let's now focus on how each Part of our model has performed and yielded results.

Member Model:

We have created embeddings for categorical features, normalized the numerical features and ensured paddings since our features were of variable lengths.

Feature Name	Shape Before Padding	Shape After Padding
Member Embeddings	(1000, 3)	(1000, 3)
Top 1 7-day Fresh Product Feature Embeddings	(140, 3)	(1000, 3)
Top 2 7-day Fresh Product Feature Embeddings	(170, 3)	(1000, 3)
Top 3 7-day Fresh Product Feature Embeddings	(160, 3)	(1000, 3)
Top 4 7-day Fresh Product Feature Embeddings	(150, 3)	(1000, 3)
Top 5 7-day Fresh Product Feature Embeddings	(152, 3)	(1000, 3)
Top 1 15-day Fresh Product Feature Embeddings	(143, 3)	(1000, 3)
Top 2 15-day Fresh Product Feature Embeddings	(168, 3)	(1000, 3)
Top 3 15-day Fresh Product Feature Embeddings	(169, 3)	(1000, 3)
Top 4 15-day Fresh Product Feature Embeddings	(166, 3)	(1000, 3)
Top 5 15-day Fresh Product Feature Embeddings	(154, 3)	(1000, 3)
Top 1 60-day Fresh Product Feature Embeddings	(147, 3)	(1000, 3)
Top 2 60-day Fresh Product Feature Embeddings	(170, 3)	(1000, 3)
Top 3 60-day Fresh Product Feature Embeddings	(177, 3)	(1000, 3)
Top 4 60-day Fresh Product Feature Embeddings	(174, 3)	(1000, 3)
Top 5 60-day Fresh Product Feature Embeddings	(185, 3)	(1000, 3)

Top 1 90-day Fresh Product Feature Embeddings	(136, 3)	(1000, 3)
Top 2 90-day Fresh Product Feature Embeddings	(174, 3)	(1000, 3)
Top 3 90-day Fresh Product Feature Embeddings	(163, 3)	(1000, 3)
Top 4 90-day Fresh Product Feature Embeddings	(182, 3)	(1000, 3)
Top 5 90-day Fresh Product Feature Embeddings	(180, 3)	(1000, 3)
Top 1 7-day Non-Fresh Product Feature Embeddings	(402, 3)	(1000, 3)
Top 2 7-day Non-Fresh Product Feature Embeddings	(487, 3)	(1000, 3)
Top 3 7-day Non-Fresh Product Feature Embeddings	(531, 3)	(1000, 3)
Top 4 7-day Non-Fresh Product Feature Embeddings	(561, 3)	(1000, 3)
Top 5 7-day Non-Fresh Product Feature Embeddings	(583, 3)	(1000, 3)
Top 1 15-day Non-Fresh Product Feature Embeddings	(401, 3)	(1000, 3)
Top 2 15-day Non-Fresh Product Feature Embeddings	(487, 3)	(1000, 3)
Top 3 15-day Non-Fresh Product Feature Embeddings	(523, 3)	(1000, 3)
Top 4 15-day Non-Fresh Product Feature Embeddings	(560, 3)	(1000, 3)
Top 5 15-day Non-Fresh Product Feature Embeddings	(602, 3)	(1000, 3)
Top 1 60-day Non-Fresh Product Feature Embeddings	(366, 3)	(1000, 3)
Top 2 60-day Non-Fresh Product Feature Embeddings	(457, 3)	(1000, 3)
Top 3 60-day Non-Fresh Product Feature Embeddings	(515, 3)	(1000, 3)
Top 4 60-day Non-Fresh Product Feature Embeddings	(541, 3)	(1000, 3)
Top 5 60-day Non-Fresh Product Feature Embeddings	(585, 3)	(1000, 3)
Top 1 90-day Non-Fresh Product Feature Embeddings	(352, 3)	(1000, 3)
Top 2 90-day Non-Fresh Product Feature Embeddings	(432, 3)	(1000, 3)
Top 3 90-day Non-Fresh Product Feature Embeddings	(495, 3)	(1000, 3)
Top 4 90-day Non-Fresh Product Feature Embeddings	(535, 3)	(1000, 3)
Top 5 90-day Non-Fresh Product Feature Embeddings	(558, 3)	(1000, 3)
Total Spend Normalized Values	(999,)	(1000,)
Total Trx Normalized Values	(219,)	(1000,)
Total Item Normalized Values	(680,)	(1000,)
Total Unique Items Normalized Values	(453,)	(1000,)
Total Unique Subclasses Normalized Values	(231,)	(1000,)

Total Unique Subdepartments Normalized Values	(79,)	(1000,)
Min Spend Normalized Values	(92,)	(1000,)
Max Spend Normalized Values	(681,)	(1000,)
Days Since Last Purchase Normalized Values	(74,)	(1000,)
Days Since First Purchase Normalized Values	(79,)	(1000,)
Spend per Trx Normalized Values	(1000,)	(1000,)
Items per Trx Normalized Values	(21,)	(1000,)
Unique Items per Trx Normalized Values	(11,)	(1000,)
Final Concatenated Tensor	-	(1000, 136)

Item Model:

We have created embeddings for categorical features, normalized the numerical features and ensured paddings since our features were of variable lengths.

Feature	Shape (Before Padding)	Shape (After Padding)
Item_embeddings	(34663, 3)	-
Item_subclass feature_embeddings	(1374, 3)	(34663, 3)
Item_subdepartment feature_embeddings	(220, 3)	(34663, 3)
Total_price_7_days normalized_values	(4776,)	(34663,)
Itm_qty_7_days normalized_values	(272,)	(34663,)
Trx_count_7_days normalized_values	(217,)	(34663,)
Total_price_15_days normalized_values	(7245,)	(34663,)
Itm_qty_15_days normalized_values	(434,)	(34663,)
Trx_count_15_days normalized_values	(346,)	(34663,)
Total_price_60_days normalized_values	(13710,)	(34663,)
Itm_qty_60_days normalized_values	(951,)	(34663,)
Trx_count_60_days normalized_values	(738,)	(34663,)
Total_price_90_days normalized_values	(16172,)	(34663,)
Itm_qty_90_days normalized_values	(1193,)	(34663,)
Trx_count_90_days normalized_values	(947,)	(34663,)
final_concatenated_tensor	-	(34663, 21)

Similarity Model:

Similarity Model lays the blueprint for similarity computations and takes in embeddings from Member and Item Models.

Additionally, since we have varying dimensions between Member Model (1000, 136) and Item Model (34663,21) so we have added projection to a common dimension of 64.

Retrieval Model:

As the name suggests this model generates top 10 items along with their similarity score against each member thus narrowing down the pool for ranking that is to come.

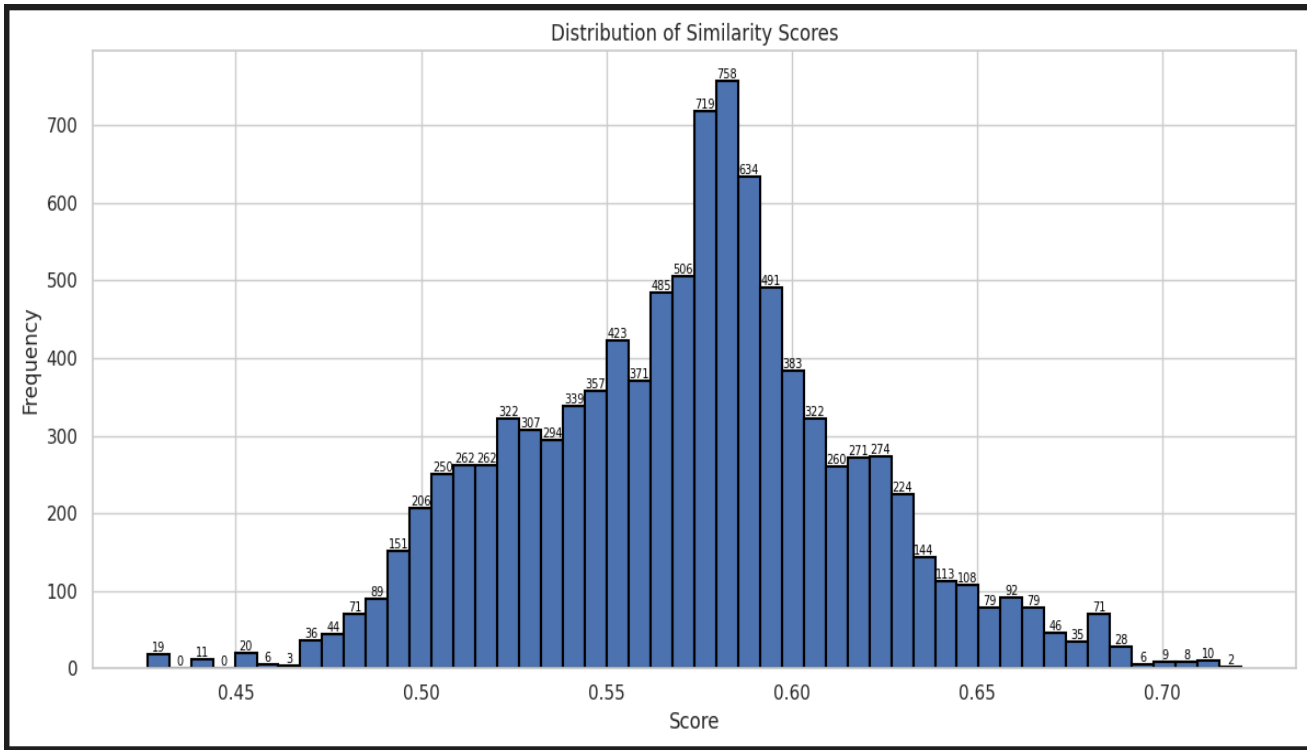

Figure 14

We can see from this histogram that our Similarity Score is almost normally distributed and its range is between 0.35 to 0.75. Which goes to show that our retrieval is half-way there but not quiet strong for recommending thus we imply dense layer in our ranking model.

Ranking Model:

Our ranking model employs ‘relu’ activation in 2 of its dense layers, the first dense layer has 128 units where as the next has 64 and finally the output has no activation and 1 unit.

The purpose of relu was that we wanted to introduce non-linearity to the model so our Neural Network can understand patterns better than general linear models further more we were not concerned about the negative similarity scores and using ‘relu’ quite evidently cleared them out.

Enhanced Retrieving with Ranking:

No we employ this ranking back to retrieving to provide items having better similarity scores to members thus making recommendation better.

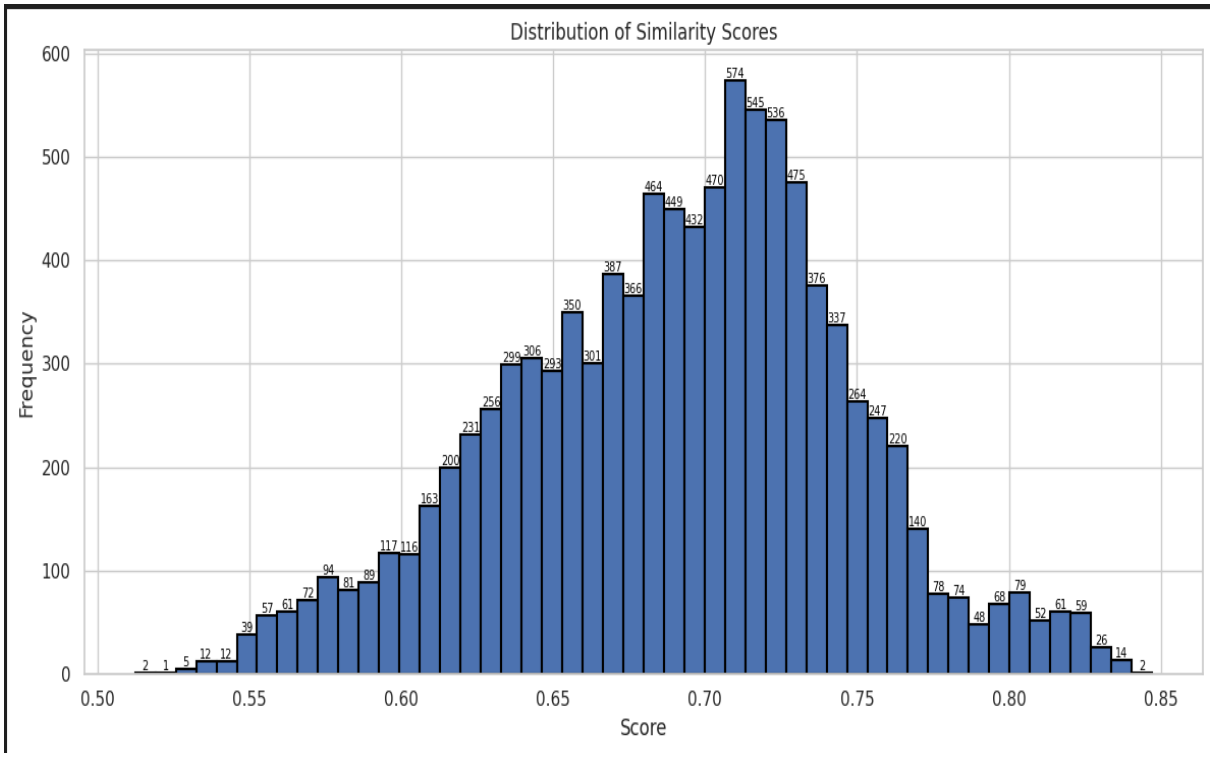

Figure 15

As evident the new similarity scores lie more towards higher side > 0.55 thus increasing our quality of ranked products.

Following are the final results of our ranking against Member Ids, Each member is being recommended 10 items.

We are truncating since it has a large number of rows but this file will be available as excel workbook for deeper review:

Member_id	Item_id	Item_name	Score
6	6348	Fakieh Large White Eggs 30 pc	0.7335
6	2412	E Bakery - Samoli with Milk Medium Size 200 g	0.7285
6	8816	Freshly Steak Sauce 284 g	0.7252
6	3956	Al-Arabi Vegetable Oil 2.9 L	0.7241
6	6356	210205300002	0.7220
6	3940	Twinings Jasmine Green Tea 25pc	0.7220
6	5228	7 Days Croissant with Cocoa Cream Filling 55 g	0.7214
6	15368	Tamara Almadina Ajwa Dates 500g	0.7214
6	22248	Teashop Ringo Wafer Rolls Chocolate 24 g	0.7212
6	4168	American Garden Ketchup Squeezable 425 g	0.7210
12	145680	Cream 21 Body Lotion with Almond Oil and Vitamin E for Ultra Dry Skin 600 ml	0.7633

12	145964	Americana Puff Pastry Squares 320 g	0.7525
12	3596	American Raw Almond \ Kg	0.7048
12	99752	Dove Awakening Ritual Body Wash + Loofah 250ml	0.6755
12	2332	280251500065	0.6693
12	145612	604371214500	0.6328
12	3892	Matcha Magic Japan Matcha for Food & Drinks 30g	0.6326
12	2876	Halwani & Tahhan Bread Sticks with Milk & Cereal Flour 250 g	0.6280
12	5648	Halawani & Tahhan Cashew 250 g	0.6153
12	164	Sunsilk Shine & Strength Shampoo For Normal Hair 700ml	0.6046
22	15800	Arjoon Khudri Dates 400 g	0.7631
22	169748	628120430039	0.6876
22	9888	Almarai Greek Style Strawberry Yoghurt 150 g	0.6861
22	9432	Afia Extra Virgin Olive Oil 500 ml	0.6858
22	8652	Foster Clark's Red Rouge Food Colour 28 ml	0.6818
22	24912	Mirinda Orange Carbonated Soft Drink 250ml	0.6750
22	42148	Al Alali Mixed Fruit Jam 400 g	0.6724
22	145096	839106470166	0.6720
22	149500	Kion Coffee Grinder 200W KHR/5009	0.6712
22	34072	Palmolive Naturals Bar Soap Balance and Softness with Chamomile and Vitamin E 120 g	0.6683
24	15800	Arjoon Khudri Dates 400 g	0.8223
24	22988	Nivea Men Invisible For Black & White Fresh Antiperspirant Spray 200 ml	0.7387
24	8652	Foster Clark's Red Rouge Food Colour 28 ml	0.7302
24	21752	Americana Classic Beef Burger 1Kg	0.7275
24	9888	Almarai Greek Style Strawberry Yoghurt 150 g	0.7241
24	34072	Palmolive Naturals Bar Soap Balance and Softness with Chamomile and Vitamin E 120 g	0.7209
24	37504	Chupa Chups Gecko Sour Jelly Candy 160 g	0.7195
24	4176	Al Safi Low Fat Long Life Milk 4*1 L	0.7179
24	24912	Mirinda Orange Carbonated Soft Drink 250ml	0.7174
24	17212	Rude Health No Sugars Organic Hazelnut & Rice Drink 1 L	0.7149
46	3020	Golden Chicken Triple Premium Fresh Whole Chicken 3*700g	0.7034
46	15800	Arjoon Khudri Dates 400 g	0.6797
46	2464	617123347164	0.6762
46	26708	Cone Zone Vanilla Ice Cream 120ml	0.6753
46	14828	Coca Cola Light Carbonated Soft Drink 250ml	0.6689
46	3524	Americana Chicken Popcorn 400g	0.6679
46	2212	Dry Cat Food 3Kg	0.6653
46	32492	Nerf Alpha Strike Fang QS-4 Set of 70 Pieces 1pc	0.6641
46	21752	Classic Beef Burger 1Kg	0.6641
46	21612	Freshdays Daily Liners Long Scented 24Pantyliners	0.6603
54	9836	Botan Sandwich Tuna 80 g	0.7334
54	37996	Clear Men's Anti-Dandruff Shampoo Shower Fresh 200 ml	0.7325
54	21824	Americana Chicken Fingers 400 G	0.7296
54	32492	Hasbro Nerf Alpha Strike Fang QS-4 Set of 70 Pieces 1pc	0.7294
54	22988	Nivea Men Invisible For Black & White Fresh Antiperspirant Spray 200 ml	0.7281
54	38764	Dove Nourishing Secrets Thickening Ritual Shampoo 400 ml	0.7257

54	20152	Nestle Country Cornflakes Cereals Bar Original 20g	0.7235
54	24368	Cadbury Bubbly Milk Chocolate 28g	0.7230
54	21060	Bambi Moisturizing Lotion Baby Wipes (2+1)*64 pc	0.7193
54	5260	Al Batal - Butter Popcorn 100 g	0.7178
64	15800	Arjoon Khudri Dates 400 g	0.8253
64	8652	Foster Clark's Red Rouge Food Colour 28 ml	0.7291
64	22988	Nivea Men Invisible For Black & White Fresh Antiperspirant Spray 200 ml	0.7226
64	16264	Radwa Frozen Chicken Drumsticks 450g	0.7201
64	34072	Palmolive Naturals Bar Soap Balance and Softness with Chamomile and Vitamin E 120 g	0.7193
64	37504	Chupa Chups Gecko Sour Jelly Candy 160 g	0.7189
64	17212	Rude Health No Sugars Organic Hazelnut & Rice Drink 1 L	0.7185
64	21752	Americana Classic Beef Burger 1Kg	0.7184
64	9888	Almarai Greek Style Strawberry Yoghurt 150 g	0.7178
64	24912	Mirinda Orange Carbonated Soft Drink 250ml	0.7161
78	3652	Dari - Frozen Raspberry 350 g	0.6740
78	4316	Galaxy Smooth Milk Chocolate 5*36 g	0.6574
78	6820	Mentos Pure Fresh Chewing Gum with Green Tea Extract 56g	0.6516
78	3568	Teashop Ringo Chocolate Wafer Rolls 12*16 g	0.6457
78	173436	270932205555	0.6420
78	83660	628500906688	0.6391
78	2516	Almarai Fresh Yoghurt Strawberry Flavoured 150g	0.6364
78	2636	Wholesome Organic Corn Puffs with Carrots 60g	0.6341
78	121864	400137902448	0.6336
78	5596	Uno Super Soft Facial Tissues 10*180 pc	0.6318
82	48664	Activia Low Fat Laban 1.75 L	0.7750
82	37996	Clear Men's Anti-Dandruff Shampoo Shower Fresh 200 ml	0.7578
82	15800	Arjoon Khudri Dates 400 g	0.7494
82	8540	E Bulty Fresh Fish 1Kg	0.7375
82	11532	Al Safi Turkish Labneh (20% OFF) 3*180 g	0.7352
82	46828	Garnier Ultra Doux Repairing Leave in Cream Honey Treasures 200 ml	0.7322
82	45868	Himalaya Soothing & Protecting Baby Wipes 2+1*56 Wipes	0.7305
82	5644	Baja Salted Pistachio 120 g	0.7232
82	23596	Oral-B Ultrathin Sensitive Black Extra Manual Toothbrush 1 pc	0.7190
82	49180	Syoss Shampoo for Dry, Devitalized Hair 500ml	0.7167
88	6844	Pringles Original 70g	0.6272
88	42204	Al Alali Pancake Mix 454 g	0.6226
88	109740	Gerber Strawberry Apple Puffs (From 8 Months) 1.48Oz	0.6163
88	23268	Oral-B Pro Expert Toothbrush For Sensitive Gums 1+1 Free	0.6060
88	506188	270097200000	0.6013
88	911872	Rita Cola Sparkling Drink 240 ml	0.6013
88	467368	800374012130	0.6012
88	264364	Kelloggs Eggo Waffles Chocolatey Chip 12.3OZ	0.6012
88	467784	Delices Du Monde Corn flakes Original 375 g	0.6012
88	681924	Vaseline Intensive Care Cocoa Radiant Body Oil 200 ml	0.6012
90	15800	Arjoon Khudri Dates 400 g	0.7713

90	22988	Nivea Men Invisible For Black & White Fresh Antiperspirant Spray 200 ml	0.7538
90	21752	Americana Classic Beef Burger 1Kg	0.7535
90	34072	Palmolive Naturals Bar Soap Balance and Softness with Chamomile and Vitamin E 120 g	0.7240
90	26708	Cone Zone Vanilla Ice Cream 120ml	0.7237
90	2464	628103347165	0.7228
90	11532	Al Safi Turkish Labneh (20% OFF) 3*180 g	0.7216
90	37996	Clear Men's Anti-Dandruff Shampoo Shower Fresh 200 ml	0.7135
90	25696	Al Walimah Indian Mazza Sella Basmati Rice Long Grain 5 kg	0.7116
90	29184	Herbal Essences Bio:Renew Volume Arabica Coffee Fruit Conditioner 400 ml	0.7085

Evaluation:

To evaluate our model, we don't have any ground-truths label and hence this model has been Unsupervised approach completely however we have come with evaluation metrics to get a better look at how our model is performing at a glance.

Since it's a retail scenario and a general understanding of retail is that their our multiple products available in subclasses that too of multiple brands. Now what we have done is we have compared the recommended product's subclasses with the subclasses of products available in Member's historical transactions and come up with 3 Metrics:

Metric	Score
Accuracy	0.9175
Precision	0.87
Recall	0.917
F1Score	0.88

Hit-rate:

Hit Rate (also known as **Hit Ratio** or **Hit Rate Ratio**) is a common evaluation metric used in recommendation systems to measure how often a recommendation system suggests an item that a user actually interacts with or likes. It reflects the proportion of times a recommended item is relevant to the user, indicating the effectiveness of the recommendation model.

$$Hit\ Rate = \frac{Number\ of\ Hits}{Total\ Number\ of\ Recommendations}$$

Hit Rate: 0.907 ~ 90.7

Chapter 5

Discussion on Implications of Findings, Impact on the Retail Sector, and Study Limitations

5.1 Introduction

This chapter delves into the implications of the findings from our research, examining their impact on the retail sector, the potential benefits for retailers, and the limitations of our study. By synthesizing these elements, we aim to provide a comprehensive understanding of how our results influence the broader retail landscape and identify avenues for future research and improvements.

5.2 Implications of Findings

Our study has yielded several critical insights that have substantial implications for the retail sector. These findings encompass consumer behavior, technological adoption, and operational efficiencies. Below, we outline the key implications derived from our research.

5.3 Consumer Behavior Trends

The study reveals that consumer behavior is increasingly driven by personalization and convenience. Retailers that leverage data analytics to understand individual consumer preferences can enhance customer satisfaction and loyalty. Personalized recommendations, targeted promotions, and customized shopping experiences are no longer just value-added services but essential components of a competitive retail strategy.

For instance, the prevalence of data-driven personalization tools means that retailers must invest in sophisticated customer relationship management (CRM) systems to stay ahead. This shift towards personalization requires a deeper understanding of consumer data and an ability to act on these insights in real-time.

5.4 Technological Advancements

The findings underscore the growing importance of technology in retail operations. The adoption of artificial intelligence (AI), machine learning, and automation is reshaping how retailers operate. Technologies such as AI-driven chatbots for customer service, automated inventory management systems, and predictive analytics for sales forecasting are becoming integral to retail success.

Our research suggests that retailers who integrate these technologies can achieve significant operational efficiencies, reduce costs, and enhance the overall customer experience. For

example, AI can help retailers manage inventory more effectively by predicting demand patterns, thus minimizing stockouts and overstock situations.

5.5 Omni channel Retailing

Another significant implication of our findings is the importance of an omnichannel approach. Consumers expect a seamless shopping experience across various channels, including physical stores, online platforms, and mobile apps. Retailers must integrate these channels to provide a cohesive experience that meets customer expectations.

The research indicates that retailers who excel in omnichannel strategies see higher customer retention rates and increased sales. An effective omnichannel strategy involves synchronizing inventory, streamlining order fulfillment processes, and ensuring consistent messaging across all channels.

5.6 Sustainability and Ethical Practices

Our findings also highlight a growing consumer preference for sustainable and ethically sourced products. Retailers that adopt sustainable practices, such as reducing carbon footprints, sourcing materials responsibly, and engaging in fair trade, can differentiate themselves in a competitive market.

Sustainability is not merely a trend but a fundamental shift in consumer values. Retailers who align their practices with these values not only enhance their brand reputation but also meet regulatory requirements and appeal to a more conscientious consumer base.

5.7 Impact on the Retail Sector

The implications of these findings extend far beyond individual retailers and have the potential to reshape the retail sector as a whole.

5.8 Competitive Landscape

As technology and consumer expectations evolve, the competitive landscape of the retail sector is undergoing significant changes. Retailers that fail to adapt to technological advancements or overlook the importance of personalization risk falling behind their competitors. Conversely, those who successfully integrate these elements into their operations can gain a competitive edge.

The research indicates that retailers investing in technology and data analytics are better positioned to anticipate market trends and respond swiftly to changing consumer preferences. This proactive approach allows them to stay ahead of the curve and maintain relevance in a dynamic market environment.

5.9 Economic Implications

The integration of advanced technologies and personalized services requires substantial investment. Retailers must balance these costs with the potential benefits of increased efficiency and customer satisfaction. Our study suggests that while the initial investment can be significant, the long-term economic benefits, including higher sales and reduced operational costs, often outweigh these expenses.

Moreover, the shift towards sustainable practices may involve additional costs in the short term. However, these investments can lead to long-term financial gains through enhanced brand loyalty, reduced regulatory risks, and operational efficiencies.

5.10 Regulatory Considerations

The findings also have implications for regulatory compliance. As retailers adopt new technologies and sustainability practices, they must navigate a complex regulatory landscape. This includes data privacy laws related to consumer information, environmental regulations, and fair trade standards.

Retailers need to stay informed about regulatory changes and ensure their practices comply with relevant laws. Failure to do so can result in legal penalties and damage to their reputation. Our research emphasizes the importance of proactive engagement with regulatory bodies and ongoing compliance monitoring.

5.11 Potential Benefits for Retailers

The benefits of implementing the insights from our study are multifaceted and can lead to substantial improvements in various aspects of retail operations.

5.12 Enhanced Customer Experience

Personalization and omnichannel strategies can significantly enhance the customer experience. Retailers who utilize data to tailor their offerings and provide a seamless shopping experience across channels are more likely to build strong customer relationships and foster brand loyalty.

5.13 Increased Operational Efficiency

Technological advancements such as AI and automation can streamline retail operations, reduce manual labor, and improve accuracy in inventory management. These efficiencies translate into cost savings and allow retailers to allocate resources more effectively.

5.14 Improved Financial Performance

Investments in technology and sustainability can lead to improved financial performance. Enhanced customer satisfaction, reduced operational costs, and efficient inventory management contribute to higher profitability. Additionally, sustainable practices can attract environmentally conscious consumers and open new market opportunities.

5.15 Competitive Advantage

Retailers who embrace innovative technologies and stay ahead of consumer trends can achieve a competitive advantage. By differentiating themselves through personalization, seamless omnichannel experiences, and ethical practices, retailers can stand out in a crowded market and attract a loyal customer base.

5.16 Limitations of the Study

While our research provides valuable insights, it is important to acknowledge its limitations. These limitations offer opportunities for future research and refinement. The limitations would be availability of the data in our case and limited reachability. The way our recommendations would have been useful if we were to run campaigns and check how our recommendations are effecting basket size and customer foot fall in our store.

5.17 Sample Size and Diversity

One limitation of our study is the sample size and diversity of the respondents. Our findings may not fully represent the entire retail market due to the sample's specific characteristics. Future research could benefit from a larger and more diverse sample to enhance the generalizability of the results.

5.18 Technological Variability

The study focuses on specific technologies and their impact on retail operations. However, the rapid pace of technological innovation means that new tools and platforms may emerge, altering the landscape. Ongoing research is needed to assess the impact of emerging technologies and their implications for the retail sector.

5.19 Regional Differences

The findings may also be influenced by regional differences in consumer behavior and technological adoption. Retailers operating in different geographic locations may face unique challenges and opportunities. Future studies could explore regional variations to provide a more comprehensive understanding of global retail trends.

5.20 Temporal Relevance

The retail sector is dynamic, and trends can shift rapidly. The findings from our study are based on data collected during a specific period, and their relevance may change over time. Longitudinal studies that track changes over extended periods could offer valuable insights into evolving trends and their long-term implications.

5.21 Areas for Improvement and Future Research

To build on the findings of this study, several areas for improvement and future research should be considered and that would be by tuning the hyper parameters of our model and applying further losses to the model.

Another area of improvement is we can create true-labels from real-time data and evaluate our model with them.

Furthermore we have mentioned quite a lot recommendation systems in our studies we can compare are 2 towered approach with other simpler models as base model to check its relevancy and performance.

5.22 Expanding the Research Scope

Future research could expand the scope to include additional variables such as consumer demographics, market segmentation, and competitive strategies. This broader approach would provide a more nuanced understanding of the factors influencing retail success.

5.23 Exploring Emerging Technologies

Given the rapid advancement of technology, future studies should explore the impact of emerging technologies such as augmented reality (AR), virtual reality (VR), and block chain on retail operations and consumer experiences. Understanding these innovations' potential benefits and challenges will be crucial for retailers seeking to stay ahead of the curve.

5.24 Investigating Regional Variations

Further research could investigate regional variations in consumer behavior and technological adoption. By examining different geographic areas, researchers can identify region-specific trends and develop targeted strategies for retailers operating in diverse markets.

5.25 Longitudinal Studies

Longitudinal studies that track changes over time would provide valuable insights into the long-term effects of technological advancements, consumer behavior shifts, and sustainability practices. These studies could help retailers anticipate future trends and adapt their strategies accordingly.

5.26 Conclusion

In conclusion, the implications of our findings offer significant insights into the evolving retail landscape. The impact on the retail sector encompasses shifts in consumer behavior, technological advancements, and the need for sustainable practices. Retailers who adapt to these changes stand to benefit from enhanced customer experiences, increased operational efficiency, and improved financial performance.

While our study provides valuable insights, it is essential to recognize its limitations and the need for ongoing research. By addressing these limitations and exploring new areas of inquiry, future research can build on our findings and contribute to a deeper understanding of the retail sector's dynamics. Ultimately, staying informed about emerging trends and continually adapting strategies will be key for retailers seeking to thrive in an ever-changing market environment.

This chapter provides a thorough examination of the implications of the study's findings, their impact on the retail sector, the benefits for retailers, and areas for future research and improvement. Adjustments can be made based on specific findings or additional details you might want to include.

Bibliography

[1] Smith, J., 2018. *The future of retail: A digital transformation*. *Journal of Retailing*, 94(2), pp.123-145. https://www.researchgate.net/publication/312301607_The_Future_of_Retailing

[2] Johnson, K., 2020**.** Challenges and Opportunities in the Retail Industry. *Retail Management Review*, 30(1), pp.5-20. https://www.retailmanagementreview.com/2020/01/

[3] Brown, M., 2019. The impact of e-commerce on traditional retail. *International Journal of Retail & Distribution Management*, 47(3), pp.250-272. https://www.emerald.com/insight/content/doi/10.1108/IJRDM-08-2018-0172

[4] Davis, A., 2021. Consumer behavior in the digital age. *Journal of Consumer Research*, 48(2), pp.315-338 https://academic.oup.com/jcr/article/48/2/315/5558986

[5] Evans, R., 2017. Customer loyalty: Causes and consequences. *Journal of Marketing*, 81(4), pp.10-25. https://journals.sagepub.com/home/jmx

[6] Karatzoglou, A. and Amatriain, X., 2016**.** Deep Learning for Recommender Systems. *In: Proceedings of the 10th ACM Conference on Recommender Systems (RecSys 2016)*. ACM, pp.1-8. https://dl.acm.org/doi/10.1145/2959100.2959183

[7] Jannach, D. and Adomavicius, G., 2020**.** A Survey on Deep Learning for Recommender Systems. *Computer Science Review*, 37, pp.1-22. https://doi.org/10.1016/j.cosrev.2020.100265

[8] Géron, A., 2019**.** *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media. https://www.oreilly.com/library/view/hands-on-machine/9781492032632

[9] Burke, R., 2007**.** Hybrid Recommender Systems: Survey and Experiments. *User Modeling and User-Adapted Interaction*, 12(4), pp.331-370. https://link.springer.com/article/10.1007/s11257-007-9044-7

[10] McSherry, F. and Mironov, I., 2009. Practical Privacy: The SuLQ Framework. *In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. ACM, pp. 577-588. https://dl.acm.org/doi/10.1145/1559845.1559906

[11] Shani, G. and Stone, P., 2008. Model-Based Collaborative Filtering https://link.springer.com/chapter/10.1007/978-0-387-85820-3_2

[12] Rendle, S., Freudenthaler, C., Gantner, Z. and Schmidt-Thieme, L., 2012. Factorization Machines. *In: Proceedings of the 2012 IEEE 11th International Conference on Data Mining*. IEEE, pp.995-1000. https://ieeexplore.ieee.org/document/6416164

[13] X., Liao, L., Zhang, H., Nie, L., Hu, X. and Chua, T., 2017. Neural collaborative filtering. https://arxiv.org/abs/1708.05031 .

[14] Adomavicius, G. and Tuzhilin, A., 2005. *Toward the Next Generation of Recommender Systems: A Survey of the State-of-the-Art and Possible Extensions*.. https://doi.org/10.1109/TKDE.2005.99

[15] Burke, R., 2002. *Hybrid Recommender Systems: Survey and Experiments*. *User Modeling and User-Adapted Interaction*, 12(4), pp.331-370. https://www.researchgate.net/publication/220701294_Hybrid_Recommender_Systems_Survey_and_Experiments

[16] Baeza-Yates, R. & Ribeiro-Neto, B., 1999. Modern Information Retrieval: The Concepts and Technology Behind Search. New York: ACM Press; Harlow: Addison-Wesley. https://archive.org/details/modern-information-retrieval_202102 (Internet Archive)..

[17] Pazzani, M. and Billsus, D., 2007. *Content-based recommendation systems*. https://www.researchgate.net/publication/280113634_Content-Based_Recommendation_Systems

Appendix Section:

Code

import pandas as pd

import numpy as np

import re

import matplotlib.pyplot as plt

import plotly.express as px

import pandas as pd

import tensorflow as tf

import tensorflow as tf

from tensorflow.keras.layers import StringLookup, Embedding, LayerNormalization

```
import tensorflow_recommenders as tfrs

import pandas as pd

import numpy as np

import plotly.express as px

import plotly.graph_objects as go

from sklearn.metrics import confusion_matrix

df = pd.read_csv("C:\\Users\\sa2259\\OneDrive - University of Sussex\\Desktop\\project final\\Recommendation System\\Recommendation_data_retail.csv")

df.head()

# Assuming unique_items_df has already been created

unique_items_df = df[['Item_id', 'Item_name']].drop_duplicates().reset_index(drop=True)


# Step 1: Group by 'Item_name' and count the number of unique 'Item_id's

item_name_counts = unique_items_df.groupby('Item_name')['Item_id'].nunique().reset_index(name='unique_item_id_count')


# Step 2: Filter the 'Item_name's that have more than one unique 'Item_id'

item_names_with_multiple_ids = item_name_counts[item_name_counts['unique_item_id_count'] > 1]


# To get the item names as a list or dataframe

item_names_list = item_names_with_multiple_ids['Item_name'].tolist()

item_names_df = item_names_with_multiple_ids[['Item_name']]
```

```
# If you want to get the original rows from unique_items_df for these Item_name's

filtered_df = unique_items_df[unique_items_df['Item_name'].isin(item_names_list)]


item_names_with_multiple_ids

# Assuming item_names_with_multiple_ids is already defined

# Plot histogram

plt.figure(figsize=(10, 6))

plt.hist(item_names_with_multiple_ids['unique_item_id_count'], bins=range(1, item_names_with_multiple_ids['unique_item_id_count'].max() + 2), edgecolor='black')

plt.xlabel('Number of Unique Item IDs')

plt.ylabel('Frequency')

plt.title('Histogram of Item Names with Multiple Unique Item IDs')

plt.grid(True)

plt.show()

df1 = df.copy()

df1.drop(columns='Item_name',inplace=True)

df1.head()

df2 = df1[df1['Item_price'] >= 0]

#Converting Timestamp column to date-time format for time-intelligence Operation.

df2['Timestamp'] = pd.to_datetime(df2['Timestamp'])

#Adjusting From UTC to Regional Time

df2['Timestamp'] = df2['Timestamp'] + pd.Timedelta(hours=3)

df2.head()
```

```
# Extract time of day and create corresponding columns
def get_time_of_day(hour):
    if 5 <= hour < 8:
        return 'early_morning'
    elif 8 <= hour < 12:
        return 'morning'
    elif 12 <= hour < 17:
        return 'noon'
    elif 17 <= hour < 20:
        return 'evening'
    elif 20 <= hour < 24:
        return 'night'
    else:
        return 'late_night'

df2['hour'] = df2['Timestamp'].dt.hour
df2['time_of_day'] = df2['hour'].apply(get_time_of_day)
df2.head()
agg_by_time = df2.groupby(
    ['Member_id', 'Item_id', 'Item_subclass', 'Item_subdepartment',
'time_of_day']
)[['Itm_qty', 'Item_price', 'Total_price']].sum().reset_index()

agg_by_time
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Set style for seaborn
sns.set(style="whitegrid")

plt.figure(figsize=(12, 6))

# Plot histogram for 'Itm_qty'
plt.subplot(1, 3, 1)
sns.histplot(agg_by_time['Itm_qty'], bins=20, kde=True)
plt.title('Distribution of Item Quantity')

# Plot histogram for 'Item_price'
plt.subplot(1, 3, 2)
sns.histplot(agg_by_time['Item_price'], bins=20, kde=True)
plt.title('Distribution of Item Price')

# Plot histogram for 'Total_price'
plt.subplot(1, 3, 3)
sns.histplot(agg_by_time['Total_price'], bins=20, kde=True)
plt.title('Distribution of Total Price')

plt.tight_layout()
plt.show()
plt.figure(figsize=(10, 8))
correlation_matrix = agg_by_time[['Itm_qty', 'Item_price',
'Total_price']].corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f')
```

```
plt.title('Correlation Matrix')
plt.show()
# Aggregate total prices by time of day
total_price_by_time =
df2.groupby('time_of_day')['Total_price'].sum().reset_index()

# Create the bar chart
fig = px.bar(total_price_by_time, x='time_of_day', y='Total_price',
        title='Total Sales Distribution by Time of Day',
        labels={'Total_price': 'Total Price', 'time_of_day': 'Time of Day'},
        color='time_of_day',
        color_discrete_sequence=px.colors.sequential.Viridis)

# Customize the layout
fig.update_traces(texttemplate='%{y}', textposition='inside')
fig.update_layout(showlegend=False)

# Display the bar chart
fig.show()

# Aggregate total prices by time of day
total_price_by_time =
df2.groupby('time_of_day')['Total_price'].sum().reset_index()

# Create the bar chart
fig = px.bar(total_price_by_time, x='time_of_day', y='Total_price',
        title='Total Sales Distribution by Time of Day',
```

```
        labels={'Total_price': 'Total Price', 'time_of_day': 'Time of Day'},
        color='time_of_day',
        color_discrete_sequence=px.colors.sequential.Viridis)

# Customize the layout with data labels formatted to 2 decimal places and larger font size
fig.update_traces(
    texttemplate='%{y:.2f}',  # Format labels to 2 decimal places
    textposition='inside',
    textfont=dict(size=14, color='White')  # Increase font size and set color to black
)
fig.update_layout(showlegend=False)

# Display the bar chart
fig.show()
import pandas as pd
import plotly.express as px

# Load your dataset
# df2 = pd.read_csv('your_data.csv') # Uncomment and adjust as needed

# Group by 'Item_subdepartment' and calculate total sales
sales_per_item_subdepartment = df2.groupby('Item_subdepartment')['Total_price'].sum().reset_index()

# Create a bar plot using Plotly
```

```
fig = px.bar(
    sales_per_item_subdepartment,
    x='Item_subdepartment',
    y='Total_price',
    color='Total_price',
    color_continuous_scale='viridis',
    title='Total Sales by Item Subdepartment'
)

# Customize the layout
fig.update_layout(
    xaxis_title='Item Subdepartment',
    yaxis_title='Total Price',
    xaxis_tickangle=-45
)

# Show the plot
fig.show()

import pandas as pd
import plotly.express as px

# Load your dataset
# df2 = pd.read_csv('your_data.csv') # Uncomment and adjust as needed

# Group by 'Item_subdepartment' and calculate total sales
```

```
sales_per_item_subdepartment =
df2.groupby('Item_subdepartment')['Total_price'].sum().reset_index()

# Create a bar plot using Plotly
fig = px.bar(
    sales_per_item_subdepartment,
    x='Item_subdepartment',
    y='Total_price',
    color='Total_price',
    color_continuous_scale='viridis',
    title='Total Sales by Item Subdepartment',
    text='Total_price'  # Add data labels
)

# Customize the layout with data labels
fig.update_traces(
    texttemplate='%{text:.2f}',  # Format labels to 2 decimal places
    textposition='outside',      # Position labels outside the bars
    textfont=dict(size=12, color='black')  # Set font size and color
)

# Customize the layout
fig.update_layout(
    xaxis_title='Item Subdepartment',
    yaxis_title='Total Price',
    xaxis_tickangle=-45
)
```

```
# Show the plot
fig.show()
import pandas as pd
import plotly.express as px

# Load your dataset
# df2 = pd.read_csv('your_data.csv') # Uncomment and adjust as needed

# Group by 'Item_subdepartment' and calculate total sales
sales_per_item_subclass = df2.groupby('Item_subclass')['Total_price'].sum().reset_index()

# Create a bar plot using Plotly
fig = px.bar(
    sales_per_item_subclass,
    x='Item_subclass',
    y='Total_price',
    color='Total_price',
    color_continuous_scale='viridis',
    title='Total Sales by Item Subclass'
)

# Customize the layout
fig.update_layout(
    xaxis_title='Item Subclass',
    yaxis_title='Total Price',
```

```
    xaxis_tickangle=-45
)

# Show the plot
fig.show()

# Plot boxplots for 'Itm_qty', 'Item_price', and 'Total_price' to visualize outliers
plt.figure(figsize=(15, 5))

# Boxplot for 'Itm_qty'
plt.subplot(1, 3, 1)
sns.boxplot(data=df2, y='Itm_qty', palette='coolwarm')
plt.title('Boxplot of Item Quantity')

# Boxplot for 'Item_price'
plt.subplot(1, 3, 2)
sns.boxplot(data=df2, y='Item_price', palette='coolwarm')
plt.title('Boxplot of Item Price')

# Boxplot for 'Total_price'
plt.subplot(1, 3, 3)
sns.boxplot(data=df2, y='Total_price', palette='coolwarm')
plt.title('Boxplot of Total Price')

plt.tight_layout()
plt.show()
```

```
import pandas as pd

# Assuming df2 is your DataFrame

# 1. Get the top 100 products by total sales
top_100_products = df2.groupby('Item_id')['Total_price'].sum().reset_index()
top_100_products = top_100_products.sort_values(by='Total_price', ascending=False).head(100)

# 2. Get the top 10 subclasses by total sales
top_10_subclasses = df2.groupby('Item_subclass')['Total_price'].sum().reset_index()
top_10_subclasses = top_10_subclasses.sort_values(by='Total_price', ascending=False).head(10)

# 3. Get the top 10 subdepartments by total sales
top_10_subdepartments = df2.groupby('Item_subdepartment')['Total_price'].sum().reset_index()
top_10_subdepartments = top_10_subdepartments.sort_values(by='Total_price', ascending=False).head(10)

# Display the results
print("Top 100 Products by Sales:")
print(top_100_products)

print("\nTop 10 Subclasses by Sales:")
print(top_10_subclasses)
```

```
print("\nTop 10 Subdepartments by Sales:")
print(top_10_subdepartments)

import pandas as pd
import plotly.express as px

# Assuming df2 is your DataFrame and has columns 'Member_id' and 'Total_price'

# 1. Calculate total revenue per member
revenue_per_member = df2.groupby('Member_id')['Total_price'].sum().reset_index()

# 2. Define revenue bands (buckets)
bins = [0, 100, 500, 1000, 5000, 10000, float('inf')]  # Define your revenue bands
labels = ['0-100', '101-500', '501-1000', '1001-5000', '5001-10000', '10000+']  # Labels for each bucket

# Create a new column 'Revenue_Band' in the DataFrame
revenue_per_member['Revenue_Band'] = pd.cut(revenue_per_member['Total_price'], bins=bins, labels=labels)

# 3. Aggregate counts for each revenue band
revenue_band_counts = revenue_per_member['Revenue_Band'].value_counts().reset_index()
revenue_band_counts.columns = ['Revenue_Band', 'Count']
```

```
# Sort by descending order of count
revenue_band_counts = revenue_band_counts.sort_values(by='Count', ascending=False)

# 4. Plot a histogram of revenue bands
fig = px.bar(
    revenue_band_counts,
    x='Revenue_Band',
    y='Count',
    title='Histogram of Revenue Bands for Members',
    labels={'Revenue_Band': 'Revenue Band', 'Count': 'Number of Members'},
    color='Revenue_Band',
    color_discrete_sequence=px.colors.sequential.Viridis
)

# Customize the layout
fig.update_layout(
    xaxis_title='Revenue Band',
    yaxis_title='Number of Members',
    xaxis_tickangle=-45
)

# Show the plot
fig.show()
pattern = r'\bfresh\b'
```

```
# Find unique 'Product_subclass' values matching the pattern (case-insensitive)

df2[df2['Item_subdepartment'].str.contains(pattern, case=False, regex=True)]


current_date = df2['Timestamp'].max()

# Compute additional features

member_features_df = df2.groupby('Member_id').agg(

   Total_spend=('Total_price', 'sum'),

   Total_trx=('Trx_id', 'nunique'),

   Total_item=('Item_id', 'count'),

   Total_unique_items=('Item_id','nunique'),

   Total_unique_subclasses=('Item_subclass','nunique'),

   Total_unique_subdepartments=('Item_subdepartment','nunique'),

   Min_spend=('Total_price', 'min'),

   Max_spend=('Total_price','max'),

   days_since_last_purchase=('Timestamp', lambda x: (current_date - x.max()).days),

   days_since_first_purchase=('Timestamp', lambda x: (current_date - x.min()).days),

).reset_index()

member_features_df['spend_per_trx'] = member_features_df['Total_spend'] / member_features_df['Total_trx']

member_features_df['Items_per_trx'] = np.ceil(member_features_df['Total_item'] / member_features_df['Total_trx'])

member_features_df['Unique_items_per_trx'] = np.ceil(member_features_df['Total_unique_items'] / member_features_df['Total_trx'])
```

```
periods = [7, 15, 60, 90]


# Most Frequently Purchased Products - FRESH (Top 5)
pattern_fresh = r'\bfresh\b'
def get_top_fresh_products_for_periods(df, periods, pattern_fresh):
    # Calculate the maximum timestamp for each member
    max_timestamp = df.groupby('Member_id')['Timestamp'].max().reset_index()
    max_timestamp.columns = ['Member_id', 'Max_Timestamp']


    # Merge the max_timestamp DataFrame with the original df to filter data
    df_with_max = pd.merge(df, max_timestamp, on='Member_id')


    # Initialize a list to store results for each period
    results = []


    for days in periods:
        # Calculate the start date for the period relative to each member's maximum timestamp
        df_with_max['Start_Date'] = df_with_max['Max_Timestamp'] - pd.Timedelta(days=days)


        # Filter to include only the data for the specified period
        filtered_data = df_with_max[(df_with_max['Timestamp'] >= df_with_max['Start_Date']) &
```

```
                (df_with_max['Timestamp'] <=
df_with_max['Max_Timestamp'])]


    # Find most frequently purchased fresh products
    product_freq_fresh =
filtered_data[filtered_data['Item_subdepartment'].str.contains(pattern_fresh, case=False, regex=True)]
    product_freq_fresh = product_freq_fresh.groupby(['Member_id',
'Item_id']).size().reset_index(name='counts')
    product_freq_fresh['rank'] =
product_freq_fresh.groupby('Member_id')['counts'].rank(method='first',
ascending=False)


    # Get the top products based on rank
    top_products = product_freq_fresh[product_freq_fresh['rank'] <= 5]


    # Pivot table to get top products for each member
    top_products_pivot = top_products.pivot_table(
      index='Member_id',
      columns='rank',
      values='Item_id',
      aggfunc=lambda x: ', '.join(map(str, x))  # Aggregate product IDs as
comma-separated strings
    ).reset_index()


    # Rename columns
    top_products_pivot.columns = [
      'Member_id'
```

```
        ] + [f'top_{int(col)}_{days}_day_fresh_product' for col in
top_products_pivot.columns if col != 'Member_id']


        # Append the result to the list
        results.append(top_products_pivot)


    # Merge all results on 'Member_id'
    combined_results = results[0]
    for result in results[1:]:
        combined_results = pd.merge(combined_results, result,
on='Member_id', how='outer')


    # Ensure no duplicate 'Member_id' columns
    combined_results = combined_results.loc[:,
~combined_results.columns.duplicated()]


    return combined_results


# Most Frequently Purchased Products - NON-FRESH (Top 5)
def get_top_non_fresh_products_for_periods(df, periods, pattern_fresh):
    # Calculate the maximum timestamp for each member
    max_timestamp =
df.groupby('Member_id')['Timestamp'].max().reset_index()
    max_timestamp.columns = ['Member_id', 'Max_Timestamp']


    # Merge the max_timestamp DataFrame with the original df to filter
data
    df_with_max = pd.merge(df, max_timestamp, on='Member_id')
```

```
    # Initialize a list to store results for each period
    results = []

    for days in periods:
        # Calculate the start date for the period relative to each member's maximum timestamp
        df_with_max['Start_Date'] = df_with_max['Max_Timestamp'] - pd.Timedelta(days=days)

        # Filter to include only the data for the specified period
        filtered_data = df_with_max[(df_with_max['Timestamp'] >= df_with_max['Start_Date']) &
                                    (df_with_max['Timestamp'] <= df_with_max['Max_Timestamp'])]

        # Find most frequently purchased non-fresh products
        product_freq_non_fresh = filtered_data[~filtered_data['Item_subdepartment'].str.contains(pattern_fresh, case=False, regex=True)]
        product_freq_non_fresh = product_freq_non_fresh.groupby(['Member_id', 'Item_id']).size().reset_index(name='counts')
        product_freq_non_fresh['rank'] = product_freq_non_fresh.groupby('Member_id')['counts'].rank(method='first', ascending=False)

        # Get the top products based on rank
        top_products = product_freq_non_fresh[product_freq_non_fresh['rank'] <= 5]
```

```
        # Pivot table to get top products for each member
        top_products_pivot = top_products.pivot_table(
            index='Member_id',
            columns='rank',
            values='Item_id',
            aggfunc=lambda x: ', '.join(map(str, x))  # Aggregate product IDs as comma-separated strings
        ).reset_index()

        # Rename columns
        top_products_pivot.columns = [
            'Member_id'
        ] + [f'top_{int(col)}_{days}_day_non_fresh_product' for col in top_products_pivot.columns if col != 'Member_id']

        # Append the result to the list
        results.append(top_products_pivot)

    # Merge all results on 'Member_id'
    combined_results = results[0]
    for result in results[1:]:
        combined_results = pd.merge(combined_results, result, on='Member_id', how='outer')

    # Ensure no duplicate 'Member_id' columns
    combined_results = combined_results.loc[:, ~combined_results.columns.duplicated()]
```

```
    return combined_results


# Combine all features


top_5_products_fresh_pivot = get_top_fresh_products_for_periods(df2, periods, pattern_fresh)

member_features_df = member_features_df.merge(top_5_products_fresh_pivot, on='Member_id', how='left')

top_5_products_non_fresh_pivot = get_top_non_fresh_products_for_periods(df2, periods, pattern_fresh)

member_features_df = member_features_df.merge(top_5_products_non_fresh_pivot, on='Member_id', how='left')



# Compute additional features

item_features = df2.groupby('Item_id').agg(

    total_purchases=('Item_id', 'count'),

    total_spend=('Total_price', 'sum'),

    total_Itm_qty_purchased=('Itm_qty', 'sum')

).reset_index()


item_features.head()

periods = [7, 15, 60, 90]

Max_timestamp_all = df2['Timestamp'].max()
```

```
results = []


def rename_columns(df, period):
    columns = {}
    for col in df.columns:
        if col not in ['Item_id', 'Item_name', 'Item_subclass',
'Item_subdepartment']:
            columns[col] = f"{col}_{period}_days"
    df.rename(columns=columns, inplace=True)
    return df


for days in periods:
    # Calculate the start date for the period relative to each member's
maximum timestamp
    df2['Start_Date'] = Max_timestamp_all - pd.Timedelta(days=days)


    # Filter to include only the data for the specified period
    filtered_data = df2[(df2['Timestamp'] >= df2['Start_Date']) &
                        (df2['Timestamp'] <= Max_timestamp_all)]


    # Find most frequently purchased fresh products
    Trx_count_items =
filtered_data.groupby('Item_id')['Trx_id'].count().reset_index()
    Trx_count_items.columns = ['Item_id', 'Trx_count']  # Rename column
for clarity
```

```
    item_sales = filtered_data.groupby(['Item_id', 'Item_subclass',
'Item_subdepartment'])[['Total_price','Itm_qty']].sum().reset_index()

    # Merge item_sales with Trx_count_items on 'Item_id'
    item_sales = item_sales.merge(Trx_count_items, on='Item_id',
how='outer')

    # Rename columns with appropriate suffix
    item_sales = rename_columns(item_sales, days)

    # Append the result to the list
    results.append(item_sales)

# Merge all period dataframes into one dataframe
final_df = results[0]
for i in range(1, len(results)):
    final_df = final_df.merge(results[i], on=['Item_id', 'Item_subclass',
'Item_subdepartment'], how='outer')

# Fill NaN values with 0 if necessary
final_df.fillna(0, inplace=True)
pd.set_option('display.max_columns',None)
final_df.head()
Item_features_df = final_df.copy()
member_features_df.head()
member_features_df = member_features_df.sort_values(by='Total_trx',
ascending=False).head(1000)
member_features_df.fillna('NA',inplace=True)
```

```
Item_features_df.fillna('NA',inplace=True)
#Checking for nulls in both datasets.
print(f'Members_dataset:\n{member_features_df.isna().sum().sum()}\n')
print(f'Items_dataset:\n{Item_features_df.isna().sum().sum()}')
numerical_columns_members =
np.array(member_features_df.columns[1:14])
# Get columns from index 58 onward
categorical_columns_from_58 = member_features_df.columns[14:]

# Get column 0
categorical_column_0 = member_features_df.columns[0]

# Combine both lists of columns
categorical_columns_members = np.concatenate(([categorical_column_0],
categorical_columns_from_58))
categorical_columns_items = np.array(Item_features_df.columns[:3])
numerical_columns_items = np.array(Item_features_df.columns[3:])

# Define categorical and numerical features
categorical_features = {
    "members": categorical_columns_members,  # Example user features
    "items": categorical_columns_items  # Example item features
}

numerical_features = {
    "members": numerical_columns_members,  # Example user numerical
features
```

```
    "items": numerical_columns_items  # Example item numerical features
}

# Create dictionary of unique features
unique_features_members_categorical = {}
unique_features_members_numerical = {}
unique_features_items_categorical = {}
unique_features_items_numerical = {}

# Helper function to convert values to strings
def convert_to_strings(values):
    return [str(value) for value in values]

def convert_to_float(values):
    return [float(value) for value in values]

# Extract unique values for member features
for feature in categorical_features["members"]:
    unique_values = member_features_df[feature].unique().tolist()
    unique_features_members_categorical[feature] =
convert_to_strings(unique_values)

for feature in numerical_features["members"]:
    unique_values = member_features_df[feature].unique().tolist()
    unique_features_members_numerical[feature] =
convert_to_float(unique_values)
```

```
# Extract unique values for item features
for feature in categorical_features["items"]:
    unique_values = Item_features_df[feature].unique().tolist()
    unique_features_items_categorical[feature] =
convert_to_strings(unique_values)

for feature in numerical_features["items"]:
    unique_values = Item_features_df[feature].unique().tolist()
    unique_features_items_numerical[feature] =
convert_to_float(unique_values)



# Convert dictionaries of unique features to tensors
unique_features_members_categorical_tf = {
    feature_name: tf.constant(vocabulary) for feature_name, vocabulary in
unique_features_members_categorical.items()
}

unique_features_members_numerical_tf = {
    feature_name: tf.constant(values) for feature_name, values in
unique_features_members_numerical.items()
}

unique_features_items_categorical_tf = {
    feature_name: tf.constant(vocabulary) for feature_name, vocabulary in
unique_features_items_categorical.items()
}
```

```
unique_features_items_numerical_tf = {
   feature_name: tf.constant(values) for feature_name, values in unique_features_items_numerical.items()
}

Unique_member_ids_tf = tf.constant(unique_features_members_categorical['Member_id'])
Unique_Item_ids_tf = tf.constant(unique_features_items_categorical['Item_id'])

# Compute correlation matrix
correlation_matrix = member_features_df[numerical_columns_members].corr()

# Plot heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=.5)
plt.title('Correlation Matrix of Numerical Features')
plt.show()

# Compute correlation matrix
correlation_matrix = Item_features_df[numerical_columns_items].corr()

# Plot heatmap
plt.figure(figsize=(12, 10))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=.5)
```

```
plt.title('Correlation Matrix of Numerical Features')
plt.show()

# Initialize the combined dictionary
unique_features_members_combined = {}

# Process categorical features
for feature in categorical_features["members"]:
    unique_values = member_features_df[feature].unique().tolist()
    unique_features_members_combined[feature] =
convert_to_strings(unique_values)

# Process numerical features
for feature in numerical_features["members"]:
    unique_values = member_features_df[feature].unique().tolist()
    unique_features_members_combined[feature] =
convert_to_float(unique_values)

# Initialize the combined dictionary
unique_features_items_combined = {}

# Process categorical features
for feature in categorical_features["items"]:
    unique_values = Item_features_df[feature].unique().tolist()
    unique_features_items_combined[feature] =
convert_to_strings(unique_values)
```

```
# Process numerical features
for feature in numerical_features["items"]:
    unique_values = Item_features_df[feature].unique().tolist()
    unique_features_items_combined[feature] =
convert_to_float(unique_values)


member_features = {feature_name: tf.constant(values) for feature_name,
values in unique_features_members_combined.items()}
item_features = {feature_name: tf.constant(values) for feature_name,
values in unique_features_items_combined.items()}
import tensorflow as tf


class MemberModel(tf.keras.Model):
    def __init__(self, unique_member_ids, unique_features_categorical,
unique_features_numerical, embedding_dim=32):
        super().__init__()
        self.embedding_dim = embedding_dim


        # Initialize the embedding layers dictionary
        self.embedding_layers = {}


        # Initialize normalization layers
        self.normalization_layers = {}


        # Initialize the Member_id lookup and embedding layer
        self.member_lookup =
tf.keras.layers.StringLookup(vocabulary=unique_member_ids,
mask_token=None, oov_token=None)
```

```
        self.member_embedding_layer =
tf.keras.layers.Embedding(input_dim=self.member_lookup.vocabulary_siz
e(), output_dim=embedding_dim)


        # Create embedding layers for other categorical features
        for feature_name, vocabulary in unique_features_categorical.items():
            if feature_name != 'Member_id':
                lookup = tf.keras.layers.StringLookup(vocabulary=vocabulary,
mask_token=None, oov_token=None)
                vocab_size = lookup.vocabulary_size()
                embedding = tf.keras.layers.Embedding(input_dim=vocab_size,
output_dim=embedding_dim)
                self.embedding_layers[feature_name] = (lookup, embedding)


        # Create normalization layers for numerical features
        for feature_name in unique_features_numerical:
            normalization_layer = tf.keras.layers.LayerNormalization(axis=-1,
epsilon=1e-6)
            self.normalization_layers[feature_name] = normalization_layer


    def call(self, member_features):
        # Extract member_id and convert to embeddings
        member_id = member_features['Member_id']
        member_id_indices = self.member_lookup(member_id)
        member_embeddings =
self.member_embedding_layer(member_id_indices)
        print(f"Shape of member_embeddings:
{member_embeddings.shape}")
```

```
        # Prepare list for concatenation
        concatenated_features = [member_embeddings]

        # Process categorical features
        for feature_name, feature_values in member_features.items():
            if feature_name != 'Member_id':
                if feature_name in self.embedding_layers:
                    lookup, embedding = self.embedding_layers[feature_name]
                    feature_indices = lookup(feature_values)
                    feature_embeddings = embedding(feature_indices)
                    print(f"Shape of {feature_name} feature_embeddings (before
padding): {feature_embeddings.shape}")

                    # Pad if needed to match batch size of member_embeddings
                    if feature_embeddings.shape[0] <
member_embeddings.shape[0]:
                        padding_size = member_embeddings.shape[0] -
feature_embeddings.shape[0]
                        feature_embeddings = tf.pad(feature_embeddings, [[0,
padding_size], [0, 0]], constant_values=0)
                        print(f"Shape of {feature_name} feature_embeddings (after
padding): {feature_embeddings.shape}")

                    concatenated_features.append(feature_embeddings)

        # Process numerical features
        for feature_name, feature_values in member_features.items():
            if feature_name in self.normalization_layers:
```

```
            normalization_layer = self.normalization_layers[feature_name]
            normalized_values =
normalization_layer(tf.expand_dims(feature_values, axis=0))
            normalized_values = tf.squeeze(normalized_values, axis=0)
            print(f"Shape of {feature_name} normalized_values (before
padding): {normalized_values.shape}")

            # Pad if needed to match batch size
            if normalized_values.shape[0] < member_embeddings.shape[0]:
                padding_size = member_embeddings.shape[0] -
normalized_values.shape[0]
                normalized_values = tf.pad(normalized_values, [[0,
padding_size]], constant_values=0)
                print(f"Shape of {feature_name} normalized_values (after
padding): {normalized_values.shape}")

concatenated_features.append(tf.expand_dims(normalized_values, axis=-
1))

    # Concatenate all tensors along the last axis
    final_concatenated_tensor = tf.concat(concatenated_features, axis=-1)
    print(f"Shape of final_concatenated_tensor:
{final_concatenated_tensor.shape}")

    return final_concatenated_tensor

member_model = MemberModel(
```

```
    unique_member_ids=Unique_member_ids_tf,
    unique_features_categorical=unique_features_members_categorical_tf,
    unique_features_numerical=unique_features_members_numerical_tf,
    embedding_dim=3
)
    output_member_embedding = member_model(member_features)

    import tensorflow as tf

class ItemModel(tf.keras.Model):
    def __init__(self, unique_Item_ids, unique_features_categorical,
unique_features_numerical, embedding_dim=32):
        super().__init__()
        self.embedding_dim = embedding_dim

        # Initialize the embedding layers dictionary
        self.embedding_layers = {}

        # Initialize normalization layers
        self.normalization_layers = {}

        # Initialize the Item_id lookup and embedding layer
        self.Item_lookup =
tf.keras.layers.StringLookup(vocabulary=unique_Item_ids,
mask_token=None, oov_token=None)
        self.Item_embedding_layer =
tf.keras.layers.Embedding(input_dim=self.Item_lookup.vocabulary_size(),
output_dim=embedding_dim)
```

```
        # Create embedding layers for other categorical features
        for feature_name, vocabulary in unique_features_categorical.items():
            if feature_name != 'Item_id':
                lookup = tf.keras.layers.StringLookup(vocabulary=vocabulary, mask_token=None, oov_token=None)
                vocab_size = lookup.vocabulary_size()
                embedding = tf.keras.layers.Embedding(input_dim=vocab_size, output_dim=embedding_dim)
                self.embedding_layers[feature_name] = (lookup, embedding)

        # Create normalization layers for numerical features
        for feature_name in unique_features_numerical:
            normalization_layer = tf.keras.layers.LayerNormalization(axis=-1, epsilon=1e-6)
            self.normalization_layers[feature_name] = normalization_layer

    def call(self, Item_features):
        # Extract Item_id and convert to embeddings
        Item_id = Item_features['Item_id']
        Item_id_indices = self.Item_lookup(Item_id)
        Item_embeddings = self.Item_embedding_layer(Item_id_indices)
        print(f"Shape of Item_embeddings: {Item_embeddings.shape}")

        # Prepare list for concatenation
        concatenated_features = [Item_embeddings]
```

```
        # Process categorical features
        for feature_name, feature_values in Item_features.items():
            if feature_name != 'Item_id':
                if feature_name in self.embedding_layers:
                    lookup, embedding = self.embedding_layers[feature_name]
                    feature_indices = lookup(feature_values)
                    feature_embeddings = embedding(feature_indices)
                    print(f"Shape of {feature_name} feature_embeddings (before padding): {feature_embeddings.shape}")

                    # Pad if needed to match batch size of Item_embeddings
                    if feature_embeddings.shape[0] < Item_embeddings.shape[0]:
                        padding_size = Item_embeddings.shape[0] - feature_embeddings.shape[0]
                        feature_embeddings = tf.pad(feature_embeddings, [[0, padding_size], [0, 0]], constant_values=0)
                        print(f"Shape of {feature_name} feature_embeddings (after padding): {feature_embeddings.shape}")

                    concatenated_features.append(feature_embeddings)

        # Process numerical features
        for feature_name, feature_values in Item_features.items():
            if feature_name in self.normalization_layers:
                normalization_layer = self.normalization_layers[feature_name]
                normalized_values = normalization_layer(tf.expand_dims(feature_values, axis=0))
                normalized_values = tf.squeeze(normalized_values, axis=0)
```

```
            print(f"Shape of {feature_name} normalized_values (before
padding): {normalized_values.shape}")


            # Pad if needed to match batch size
            if normalized_values.shape[0] < Item_embeddings.shape[0]:
                padding_size = Item_embeddings.shape[0] -
normalized_values.shape[0]
                normalized_values = tf.pad(normalized_values, [[0,
padding_size]], constant_values=0)
                print(f"Shape of {feature_name} normalized_values (after
padding): {normalized_values.shape}")


concatenated_features.append(tf.expand_dims(normalized_values, axis=-
1))


        # Concatenate all tensors along the last axis
        final_concatenated_tensor = tf.concat(concatenated_features, axis=-1)
        print(f"Shape of final_concatenated_tensor:
{final_concatenated_tensor.shape}")



        return final_concatenated_tensor


    item_model = ItemModel(
    unique_Item_ids=Unique_Item_ids_tf,
    unique_features_categorical=unique_features_items_categorical_tf,
    unique_features_numerical=unique_features_items_numerical_tf,
    embedding_dim=3
```

```
)

    output_item_embedding = item_model(item_features)

    member_features = {feature_name: tf.constant(values) for feature_name,
values in unique_features_members_combined.items()}

#Print the result to verify
for feature_name, tensor in member_features.items():
    print(f"Feature: {feature_name}")
    print(f"Tensor: {tensor.numpy()}")

item_features = {feature_name: tf.constant(values) for feature_name,
values in unique_features_items_combined.items()}

#Print the result to verify
for feature_name, tensor in member_features.items():
    print(f"Feature: {feature_name}")
    print(f"Tensor: {tensor.numpy()}")

    class SimilarityLayer(tf.keras.layers.Layer):
    def __init__(self, embedding_dim, temperature=1.0, **kwargs):
        super(SimilarityLayer, self).__init__(**kwargs)
        self.temperature = temperature
        # Define projection layers
        self.member_projection = tf.keras.layers.Dense(embedding_dim,
activation='relu')
```

```
        self.item_projection = tf.keras.layers.Dense(embedding_dim,
activation='relu')

    def call(self, member_embeddings, item_embeddings):
        # Ensure both tensors have the same data type
        dtype = tf.float32
        member_embeddings = tf.cast(member_embeddings, dtype)
        item_embeddings = tf.cast(item_embeddings, dtype)

        # Project embeddings to the same dimension
        member_embeddings = self.member_projection(member_embeddings)
        item_embeddings = self.item_projection(item_embeddings)

        # Normalize the embeddings
        member_embeddings = tf.nn.l2_normalize(member_embeddings,
axis=-1)
        item_embeddings = tf.nn.l2_normalize(item_embeddings, axis=-1)

        # Compute the similarity as the dot product
        similarity = tf.matmul(member_embeddings, item_embeddings,
transpose_b=True)

        # Scale the similarity by temperature
        similarity = similarity / self.temperature

        # Return both the similarity and the projected embeddings
        return similarity, member_embeddings, item_embeddings
```

```
class RetrievalModelWithTopK(tfrs.Model):
def __init__(self, member_model, item_model, similarity_model, k=10):
    super().__init__()
    self.member_model = member_model
    self.item_model = item_model
    self.similarity_model = similarity_model
    self.k = 10

    # Initialize the FactorizedTopK retrieval layer
    self.top_k = tfrs.layers.factorized_top_k.BruteForce()

def retrieve_top_k(self, member_features, item_features):
    # Get embeddings from the models
    member_embeddings = self.member_model(member_features)
    item_embeddings = self.item_model(item_features)

    # Get similarity and projected embeddings from SimilarityLayer
    similarity, projected_member_embeddings,
projected_item_embeddings = self.similarity_model(member_embeddings,
item_embeddings)

    # Index the item embeddings with the retrieval layer using the
projected embeddings
    self.top_k.index(projected_item_embeddings)

    # Get top-k items for the given member features using the projected
embeddings
```

```
        top_k_values, top_k_indices =
self.top_k(projected_member_embeddings)


        return top_k_values, top_k_indices


    def compute_loss(self, member_features, item_features, training=False):
        top_k_values, top_k_indices = self.retrieve_top_k(member_features,
item_features)
        # Further processing...
        return top_k_values, top_k_indices


    Similarity_model = SimilarityLayer(temperature=1,
embedding_dim=64)
retrieval_model =
RetrievalModelWithTopK(member_model,item_model,Similarity_model,k
=10)


top_k_scores, top_k_indices =
retrieval_model.compute_loss(member_features, item_features)


# Convert tensors to numpy arrays for easier reading
top_k_scores_np = top_k_scores.numpy()
top_k_indices_np = top_k_indices.numpy()


# Print the shape of the results
print("Top-k Scores Shape:", top_k_scores_np.shape)
print("Top-k Indices Shape:", top_k_indices_np.shape)
all_scores = tf.concat([tf.reshape(top_k_scores[m], [-1]) for m in
range(top_k_scores.shape[0])], axis=0)
```

```
all_scores = all_scores.numpy()  # Convert tensor to numpy array for analysis


import matplotlib.pyplot as plt


# Increase figure size
fig, ax = plt.subplots(figsize=(12, 6))  # Adjust width and height as needed


# Create the histogram
counts, bins, patches = ax.hist(all_scores, bins=50, edgecolor='black', linewidth=0.5, histtype='bar')


# Adjust bar width and thickness
for patch in patches:
    patch.set_edgecolor('black')  # Set edge color
    patch.set_linewidth(1.5)      # Thicker bar edges
    # Optionally adjust the width here if needed


# Add data labels on top of each bar
for count, bin, patch in zip(counts, bins, patches):
    # Calculate the height of the label
    height = patch.get_height()
    # Add the label with size 7 and color black
    ax.text(patch.get_x() + patch.get_width() / 2, height, f'{int(count)}',
            ha='center', va='bottom', fontsize=7, color='black')


# Customize the plot
```

```
ax.set_title('Distribution of Similarity Scores')
ax.set_xlabel('Score')
ax.set_ylabel('Frequency')

# Adjust layout for better spacing
plt.tight_layout()

# Show the plot
plt.show()

import tensorflow as tf

class RankingModel(tf.keras.Model):
    def __init__(self, embedding_dim, **kwargs):
        super(RankingModel, self).__init__(**kwargs)
        # Define dense layers for the ranking model
        self.dense1 = tf.keras.layers.Dense(128, activation='relu')
        self.dense2 = tf.keras.layers.Dense(64, activation='relu')
        self.dense3 = tf.keras.layers.Dense(1)  # Output layer for ranking score

    def call(self, member_embeddings, item_embeddings):
        # Concatenate member and item embeddings
        combined_embeddings = tf.concat([member_embeddings,
item_embeddings], axis=-1)

        # Pass through dense layers
        x = self.dense1(combined_embeddings)
```

```
        x = self.dense2(x)
        ranking_scores = self.dense3(x)

        return ranking_scores

    import tensorflow as tf
import tensorflow_recommenders as tfrs

class RankingRetrievalModel(tfrs.Model):
    def __init__(self, member_model, item_model, similarity_model,
ranking_model, k=10):
        super().__init__()
        self.member_model = member_model
        self.item_model = item_model
        self.similarity_model = similarity_model
        self.ranking_model = ranking_model
        self.k = k

        # Initialize the BruteForce retrieval layer
        self.top_k = tfrs.layers.factorized_top_k.BruteForce()

    def retrieve_top_k(self, member_features, item_features):
        # Get embeddings from the models
        member_embeddings = self.member_model(member_features)
        item_embeddings = self.item_model(item_features)

        # Get similarity and projected embeddings from SimilarityLayer
```

```
        similarity, projected_member_embeddings,
projected_item_embeddings = self.similarity_model(member_embeddings,
item_embeddings)


        # Debug: Print shapes

        print(f"Projected member embeddings shape:
{projected_member_embeddings.shape}")

        print(f"Projected item embeddings shape:
{projected_item_embeddings.shape}")


        # Index the item embeddings with the retrieval layer using the
projected embeddings

        self.top_k.index(projected_item_embeddings)


        # Get top-k items for the given member features using the projected
embeddings

        top_k_values, top_k_indices =
self.top_k(projected_member_embeddings)


        return top_k_values, top_k_indices


    def compute_loss(self, member_features, item_features, training=False):

        top_k_values, top_k_indices = self.retrieve_top_k(member_features,
item_features)

        return top_k_values, top_k_indices


    similarity_model = SimilarityLayer(embedding_dim=32,
temperature=1.0)

ranking_model = tf.keras.Sequential([
```

```
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
])  # Example ranking model

# Create an instance of the RankingRetrievalModel
ranking_retrieval_model = RankingRetrievalModel(
    member_model=member_model,
    item_model=item_model,
    similarity_model=similarity_model,
    ranking_model=ranking_model,
    k=10  # Number of top items to retrieve
)

top_k_values, top_k_indices = ranking_retrieval_model.retrieve_top_k(member_features, item_features)

all_scores = tf.concat([tf.reshape(top_k_values[m], [-1]) for m in range(top_k_values.shape[0])], axis=0)
all_scores = all_scores.numpy()  # Convert tensor to numpy array for analysis

import matplotlib.pyplot as plt

# Increase figure size
fig, ax = plt.subplots(figsize=(12, 6))  # Adjust width and height as needed
```

```
# Create the histogram
counts, bins, patches = ax.hist(all_scores, bins=50, edgecolor='black', linewidth=0.5, histtype='bar')

# Adjust bar width and thickness
for patch in patches:
    patch.set_edgecolor('black')  # Set edge color
    patch.set_linewidth(1.5)      # Thicker bar edges
    # Optionally adjust the width here if needed

# Add data labels on top of each bar
for count, bin, patch in zip(counts, bins, patches):
    # Calculate the height of the label
    height = patch.get_height()
    # Add the label with size 7 and color black
    ax.text(patch.get_x() + patch.get_width() / 2, height, f'{int(count)}',
            ha='center', va='bottom', fontsize=7, color='black')

# Customize the plot
ax.set_title('Distribution of Similarity Scores')
ax.set_xlabel('Score')
ax.set_ylabel('Frequency')

# Adjust layout for better spacing
plt.tight_layout()

# Show the plot
```

```
plt.show()

member_ids = member_features_df['Member_id'].values
item_ids = Item_features_df['Item_id'].values

# Initialize an empty list to store the rows
rows = []

member_ids = member_features_df['Member_id'].values
# Extract unique Item_id and Item_name pairs
unique_item_names_df = df.drop_duplicates(subset=['Item_id',
'Item_name'])

# Create a mapping of Item_id to Item_name
item_id_to_name_mapping = dict(zip(unique_item_names_df['Item_id'],
unique_item_names_df['Item_name']))

member_ids = member_features_df['Member_id'].values

for i, (scores, indices) in enumerate(zip(top_k_values, top_k_indices)):
    member_id = member_ids[i]  # Get the member ID for this iteration

    # Map top-k indices to item IDs
    top_k_item_ids = item_ids[indices]

    # Map item IDs to item names using the mapping
```

```
    top_k_item_names = [item_id_to_name_mapping[item_id] for item_id in top_k_item_ids]


    # Create rows for the DataFrame
    for item_id, item_name, score in zip(top_k_item_ids, top_k_item_names, scores):
        score_value = score.numpy()
        rows.append({
            'Member_id': member_id,
            'Item_id': item_id,
            'Item_name': item_name,
            'Score': score_value
        })

# Create the DataFrame
top_k_df = pd.DataFrame(rows)
pd.set_option('display.max_rows',None)
top_k_df.head(5)

top_k_df.to_csv('Recommendations_top_10.csv')

unique_item_id_df = df.drop_duplicates(subset=['Item_id'])
# Assuming you have `item_features_df` and `df2` with `item_subclass`

# Map item_ids to their item_subclass
item_subclass_map = Item_features_df.set_index('Item_id')['Item_subclass'].to_dict()
```

```
# Function to get item_subclass for a list of item_ids
def get_item_subclasses(item_ids):
    return [item_subclass_map.get(item_id, 'Unknown') for item_id in item_ids]

import pandas as pd

# Sample DataFrames
# top_k_df contains columns Member_id, Item_id, Item_name, Score
# df contains columns Member_id, Item_id, Item_subclass

# Create a mapping from item_id to item_subclass
item_subclass_map = Item_features_df.set_index('Item_id')['Item_subclass'].to_dict()

# Create a mapping from item_id to item_subclass in transactional data
true_item_subclasses = df[['Item_id', 'Item_subclass']].drop_duplicates().set_index('Item_id')['Item_subclass'].to_dict()

# Function to get item_subclass for a list of item_ids
def get_item_subclasses(item_ids):
    return [item_subclass_map.get(item_id, 'Unknown') for item_id in item_ids]

# Function to evaluate the top-K recommendations for each member
def evaluate_member_recommendations(member_id, top_k_df, k):
```

```
    # Get the item_ids from transactions for this member
    member_transactions = df[df['Member_id'] == member_id]
    true_item_ids = member_transactions['Item_id'].unique()

    # Get item_subclasses for true item_ids
    true_item_subclasses_for_member =
set(true_item_subclasses.get(item_id, 'Unknown') for item_id in
true_item_ids)

    # Filter the top-k items for this member
    member_recommendations = top_k_df[top_k_df['Member_id'] ==
member_id]
    top_k_items = member_recommendations.head(k)

    # Get item_subclasses for top-K recommended items
    top_k_subclasses =
set(top_k_df[top_k_df['Item_id'].isin(top_k_items['Item_id'])]['Item_id'].m
ap(lambda x: item_subclass_map.get(x, 'Unknown')))

    # Evaluate Precision, Recall, and Accuracy
    hits = true_item_subclasses_for_member.intersection(top_k_subclasses)

    precision = len(hits) / k if k > 0 else 0
    recall = len(hits) / len(true_item_subclasses_for_member) if
true_item_subclasses_for_member else 0
    accuracy = 1 if len(hits) > 0 else 0

    return accuracy, precision, recall
```

```
# Evaluate recommendations for each member in the DataFrame
results = []
k = 5  # Define the value of K
for member_id in top_k_df['Member_id'].unique():
    accuracy, precision, recall = evaluate_member_recommendations(member_id, top_k_df, k)
    results.append({'Member_id': member_id, 'Accuracy': accuracy, 'Precision': precision, 'Recall': recall})

# Convert results to DataFrame for better readability
results_df = pd.DataFrame(results)
results_df.head()
results_df.to_csv('Results_top_10.csv')

overall_accuracy = results_df['Accuracy'].mean()
overall_precision = results_df['Precision'].mean()
overall_recall = results_df['Recall'].mean()

print("Overall Accuracy: {:.4f}".format(overall_accuracy))
print("Overall Precision: {:.4f}".format(overall_precision))
print("Overall Recall: {:.4f}".format(overall_recall))

from sklearn.metrics import f1_score

def f1_score_at_k(true_labels, recommendations, k):
```

```
    y_true = [1 if item in true_labels[i] else 0 for i in
range(len(recommendations)) for item in recommendations[i][:k]]
    y_pred = [1] * len(y_true)  # All predictions are relevant items

    return f1_score(y_true, y_pred)

# Convert top_k_df to a list of recommendations for each user
recommendations =
top_k_df.groupby('Member_id')['Item_id'].apply(list).tolist()

# Create a dictionary of true relevant items for each user
true_relevant_items =
df.groupby('Member_id')['Item_id'].apply(set).to_dict()

# Calculate F1 Score
true_labels = [true_relevant_items.get(member_id, set()) for member_id in
top_k_df['Member_id'].unique()]
f1_score_value = f1_score_at_k(true_labels, recommendations, k=10)
print(f'F1 Score at K: {f1_score_value}')

def hit_rate(true_labels, recommendations):
    hits = [1 if any(item in true_labels[i] for item in recommendations[i]) else
0 for i in range(len(recommendations))]
    return sum(hits) / len(hits)

# Convert top_k_df to a list of recommendations for each user
recommendations =
top_k_df.groupby('Member_id')['Item_id'].apply(list).tolist()
```

```
# Create a dictionary of true relevant items for each user
true_relevant_items = df.groupby('Member_id')['Item_id'].apply(set).to_dict()

# Calculate Hit Rate
true_labels = [true_relevant_items.get(member_id, set()) for member_id in top_k_df['Member_id'].unique()]
hit_rate_value = hit_rate(true_labels, recommendations)
print(f'Hit Rate: {hit_rate_value}')


def hit_rate(true_labels, recommendations):
    hits = [1 if any(item in true_labels[i] for item in recommendations[i]) else 0 for i in range(len(recommendations))]
    return sum(hits) / len(hits)

# Convert top_k_df to a list of recommendations for each user
recommendations = top_k_df.groupby('Member_id')['Item_id'].apply(list).tolist()

# Create a dictionary of true relevant items for each user
true_relevant_items = df.groupby('Member_id')['Item_id'].apply(set).to_dict()

# Calculate Hit Rate
true_labels = [true_relevant_items.get(member_id, set()) for member_id in top_k_df['Member_id'].unique()]
```

```
hit_rate_value = hit_rate(true_labels, recommendations)
print(f'Hit Rate: {hit_rate_value}')


def hit_rate(true_labels, recommendations):
    hits = [1 if any(item in true_labels[i] for item in recommendations[i]) else 0 for i in range(len(recommendations))]
    return sum(hits) / len(hits)


# Convert top_k_df to a list of recommendations for each user
recommendations = top_k_df.groupby('Member_id')['Item_id'].apply(list).tolist()


# Create a dictionary of true relevant items for each user
true_relevant_items = df.groupby('Member_id')['Item_id'].apply(set).to_dict()


# Calculate Hit Rate
true_labels = [true_relevant_items.get(member_id, set()) for member_id in top_k_df['Member_id'].unique()]
hit_rate_value = hit_rate(true_labels, recommendations)
print(f'Hit Rate: {hit_rate_value}')


def hit_rate(true_labels, recommendations):
    hits = [1 if any(item in true_labels[i] for item in recommendations[i]) else 0 for i in range(len(recommendations))]
    return sum(hits) / len(hits)


# Convert top_k_df to a list of recommendations for each user
```

```
recommendations =
top_k_df.groupby('Member_id')['Item_id'].apply(list).tolist()


# Create a dictionary of true relevant items for each user

true_relevant_items =
df.groupby('Member_id')['Item_id'].apply(set).to_dict()


# Calculate Hit Rate

true_labels = [true_relevant_items.get(member_id, set()) for member_id in
top_k_df['Member_id'].unique()]

hit_rate_value = hit_rate(true_labels, recommendations)

print(f'Hit Rate: {hit_rate_value}')
```