# E-commerce Product Recommendation System

# Application Context

In the competitive realm of e-commerce, delivering individualized suggestions to consumers is essential for improving the user experience and increasing sales. An efficient recommendation system evaluates user behavior, tastes, and historical purchases to provide goods customized for each person. Recommendation systems enhance customer engagement and conversion rates by detecting trends in user interactions to facilitate the discovery of new and relevant items. Numerous e-commerce systems depend on recommendation algorithms to enhance interactivity and personalize the purchasing experience. Data structures are crucial in this context, since they provide effective data organization, retrieval, and processing, which are needed for real-time suggestions. This recommendation system utilizes three data structures: a matrix for collaborative filtering, a hash map with linked lists for session tracking, and a K-D tree for similarity searches. Each data structure facilitates a unique recommendation method, enhancing a holistic system that delivers varied and individualized suggestions. The matrix facilitates collaborative filtering by depicting individuals and items in a two-dimensional framework, encapsulating interactions among users based on analogous interests. A hash map with linked lists records recent user interactions, hence facilitating session-based suggestions. The K-D tree organizes persons and goods according to feature vectors, facilitating content-based filtering by recognizing things similar to those a user has previously engaged with. Collectively, these data structures provide an effective and scalable basis for managing extensive information and providing real-time suggestions.

# Chosen Data Structures and Design Rationale

The selected data structures include a matrix for collaborative filtering, a hash map with linked lists for session tracking, and a K-D tree for clustering. Each structure is distinctly designed to facilitate certain roles inside the recommendation system, allowing for a combination of recommendation strategies that enhance accuracy and relevance.   
The Matrix is fundamental to the execution of collaborative filtering, a recommendation methodology that proposes items based on the preferences of individuals exhibiting like behaviors. This matrix is organized as a two-dimensional array, with rows denoting users and columns denoting goods. Each element in the matrix records a user's engagement with a product, including metrics such as ratings or purchase frequency. Collaborative filtering discerns trends inside this matrix by analyzing rows to identify people with comparable product evaluations or interests. If two individuals exhibit comparable interactions with a certain group of items, they are likely to possess analogous preferences generally. The temporal complexity for retrieving an item in the matrix is O(1), facilitating rapid retrieval of user-product interactions. Nonetheless, updating entries and computing suggestions for several people might become computationally demanding as the information expands. Studies on collaborative filtering validate the efficacy of matrices in managing user-product connections, highlighting that matrix-based approaches provide precise suggestions by using similarity scores across users (Koren et al., 2009).   
The Hash Map with Linked Lists is used for session tracking, cataloging each user's previous interactions with items. In this framework, each user ID serves as a key in the hash map, with the associated value being a linked list of product IDs that denote the user's recent activity. This configuration facilitates rapid retrieval of a user's session history, which may be used to provide suggestions based on recent views or acquisitions. The hash map offers O(1) average-case time complexity for both insertion and retrieval, facilitating rapid access to each user's interaction history. Linked lists facilitate the tracking of sequential interactions, enabling the system to sustain a sliding window of recent activities that can be refreshed with each new product engagement. For instance, when a user examines a product, it is appended to the front of their linked list, while the most antiquated element is eliminated to maintain the list within a certain length. This framework efficiently collects real-time session data, enabling suggestions based in current user behavior patterns. Research on session-based recommendation systems underscores the need of monitoring previous activities to forecast future interactions, given that users' interests may change over time (Hidasi et al., 2016).   
The K-D Tree facilitates content-based filtering by allowing effective grouping of people and goods according to feature vectors. A K-D tree is a binary search tree that partitions data at each level according to one of the k dimensions of the feature space. This recommendation system utilizes feature vectors for people and goods that include attributes such as category preferences, price range, and rating history. The K-D tree enables the system to efficiently locate items analogous to those a user has previously engaged with via nearest-neighbor searches. Upon a user's viewing or purchasing of a product, the K-D tree may identify additional items with similar attributes, therefore offering tailored suggestions. The time complexity for searching a balanced K-D tree is O(log n), making it an appropriate option for extensive recommendation systems. K-D trees are very efficient for high-dimensional data in recommendation scenarios, facilitating rapid similarity searches crucial for content-based suggestions (Arya & Mount, 2018).

# Python Implementation Overview

The Python implementation for this recommendation system involves defining classes for each data structure, with methods that support the operations needed to generate recommendations. Here’s a high-level overview of the code structure and some pseudocode snippets to illustrate key operations for each data structure.

The **Matrix** for collaborative filtering is implemented as a 2D array, where rows and columns represent users and products, respectively. The matrix stores ratings or interaction counts, which are used to calculate similarity scores between users. To update the matrix when a user interacts with a product, the relevant entry in the matrix is incremented or set to the rating value. The following pseudocode demonstrates how to update the matrix:

class CollaborativeFilteringMatrix:

def \_\_init\_\_(self, num\_users, num\_products):

self.matrix = [[0] \* num\_products for \_ in range(num\_users)]

def update\_interaction(self, user\_id, product\_id, rating):

self.matrix[user\_id][product\_id] = rating

To generate recommendations, similarity calculations can be performed by comparing rows to find users with similar ratings, allowing the system to suggest products that other similar users have liked.

The **Hash Map with Linked Lists** class tracks each user’s recent interactions. In this implementation, the UserSessionMap class includes a hash map for fast user access, with linked lists storing product IDs of recently viewed or purchased items. Each time a user interacts with a product, the interaction is added to their linked list, maintaining a record of their most recent activities. Here’s an example pseudocode for inserting a new interaction:

class LinkedListNode:

def \_\_init\_\_(self, product\_id):

self.product\_id = product\_id

self.next = None

class UserSessionMap:

def \_\_init\_\_(self):

self.sessions = {}

def add\_interaction(self, user\_id, product\_id):

if user\_id not in self.sessions:

self.sessions[user\_id] = LinkedListNode(product\_id)

else:

new\_node = LinkedListNode(product\_id)

new\_node.next = self.sessions[user\_id]

self.sessions[user\_id] = new\_node

This structure supports recommendations based on session-based interactions by quickly retrieving recent items a user has viewed, which helps to suggest related products.

The **K-D Tree** class clusters users and products based on feature vectors. When a user interacts with a product, the K-D tree can search for similar products using nearest-neighbor searches, leveraging the multi-dimensional feature space. In this pseudocode, the K-D tree is constructed, and a method for inserting products based on their feature vectors is provided:

class KDTreeNode:

def \_\_init\_\_(self, point, left=None, right=None):

self.point = point

self.left = left

self.right = right

class KDTree:

def \_\_init\_\_(self):

self.root = None

def insert(self, root, point, depth=0):

if root is None:

return KDTreeNode(point)

k = len(point)

axis = depth % k

if point[axis] < root.point[axis]:

root.left = self.insert(root.left, point, depth + 1)

else:

root.right = self.insert(root.right, point, depth + 1)

return root

The K-D tree’s structure supports content-based recommendations by grouping products with similar features and enabling efficient retrieval of related items when a user views or purchases a product.

# Challenges and Limitations

Developing a recommendation system for a substantial e-commerce platform has several issues, especially with the management of extensive datasets and the preservation of low-latency answers. A significant difficulty is scalability; as the user and product base expands, sustaining an updated matrix for collaborative filtering may become computationally burdensome. Matrix factorization approaches may mitigate some of this burden but may bring more complexity. A further problem lies in reconciling memory economy with real-time processing in the hash map using linked lists. While linked lists provide an effective method for monitoring recent user activity, they may become memory-intensive with a substantial user base. A notable constraint of using a K-D tree for similarity searches is its diminished performance with higher-dimensional data, resulting in reduced efficiency for nearest-neighbor searches. While the K-D tree is efficient with low-dimensional data, competing techniques like locality-sensitive hashing may be superior for high-dimensional feature vectors. Moreover, ensuring precision in suggestions while enhancing computing efficiency might be difficult. A trade-off often exists between the extent of customisation and system performance, necessitating meticulous adjustment of data structure characteristics and recommendation algorithms to satisfy user requirements.

# References

Arya, S., & Mount, D. M. (2018). Approximate nearest neighbor queries in fixed dimensions. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 271-280.

Hidasi, B., Karatzoglou, A., Baltrunas, L., & Tikk, D. (2016). Session-based recommendations with recurrent neural networks. *Proceedings of the International Conference on Learning Representations (ICLR)*.

Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, 42(8), 30-37.