MILESTONE 2

Objective:

The objective of this project is to develop a recommendation system that enhances user experience by suggesting relevant products based on historical interactions with products and available product data. The system aims to leverage machine learning techniques to analyse customer behaviour and provide personalised recommendations.

The 2nd milestone focuses on implementing approaches like TF-IDF and cosine similarity approach and machine learning algorithms like XGBoost and Random Forest to provide personalized recommendations to users. The report focuses on the feature engineering, feature selection, training process and the output interpretation carried out for each approach.

Type of tool:

A recommendation engine will be developed based on the user behaviour, product interactions and pricing trends.

Tech Stack

- **Programming Language:** Python
- **Libraries & Frameworks:** Pandas, NumPy, Scikit-learn, Matplotlib, Seaborn
- **Visualization Tools:** Matplotlib, Seaborn

Data Collection:

The data used is open-source datasets available on Kaggle. The data describes the user's interactions on an ecommerce website/application, providing information regarding the time spent on the website, the products viewed/bought, price, name of the product, etc through the following attributes:

1. **event_time** (timestamp of interaction)
2. **event_type** (type of user interaction)
3. **product_id** (unique identifier for products)
4. **category_id** (unique category identifier)
5. **category_code** (product classification)
6. **brand** (product brand)
7. **price** (product price)
8. **user_id** (unique identifier for users)
9. **user_session** (unique identifier for user sessions)

Three csv files are used, 2 csv files contain information related to user's interactions with an electronic store/website and the other csv file contains information related to users' interactions with a multi category store.

Project Timeline

Milestone 1: Data Collection, Preprocessing, and EDA (Feb 5, 2025 - Feb 21, 2025)

1. Week 1 (Feb 5 - Feb 11): Identify and acquire dataset, verify accessibility, and document dataset details.
2. Week 2 (Feb 12 - Feb 18): Handle missing values, address outliers, and apply feature scaling.
3. Week 3 (Feb 19 - Feb 21): Perform exploratory data analysis (EDA), generate visualizations, and identify patterns.

Milestone 2: Feature Engineering, Feature Selection, and Data Modeling (Feb 21, 2025 - March 21, 2025)

1. Week 4 (Feb 22 - Feb 28)**:** Engineer new features and encode categorical variables.
2. Week 5 (March 1 - March 7): Perform feature selection and dimensionality reduction.
3. Week 6 (March 8 - March 14): Split data into training and testing sets, train initial models.
4. Week 7 (March 15 - March 21): Tune hyperparameters and evaluate models using performance metrics.

Milestone 3: Evaluation, Interpretation, Tool Development, and Presentation (March 24, 2025 - April 23, 2025)

1. Week 8 (March 24 - March 30): Assess model performance and interpret results.
2. Week 9 (April 1 - April 7): Identify biases and refine models if needed.
3. Week 10 (April 8 - April 14): Develop an interactive tool (dashboard, conversational agent, or reporting system).
4. Week 11 (April 15 - April 23): Finalize report, prepare presentation, and deliver findings.

Feature Engineering:

Three different approaches have been used to create a recommendation system, TF-IDF vectorizer and Cosine similarity approach, XGBoost approach and the random forest approach. The process of feature engineering slightly differs for these approaches.

1. TF-IDF vectorizer and Cosine similarity:

The ‘merged_df’ csv has the following columns:

```
df.head()
```

	event_time	event_type	product_id	category_id	category_code	brand	price	user_id	user_session
0	2019-10-22 17:38:59 UTC	view	1005184	2053013555631882655	electronics.smartphone	samsung	1029.34	539358901	58a9ee1c-6df6-4d4a-847e-3fc93108b051
1	2019-10-01 08:19:45 UTC	view	1005105	2053013555631882655	electronics.smartphone	apple	1415.48	541796323	21673d95-29c3-4ffc-a0c3-f2a98bc98b02
2	2019-10-03 09:10:11 UTC	view	1004776	2053013555631882655	electronics.smartphone	xiaomi	182.50	556311667	d9507b2f-ec5b-4c56-9420-bce1aa3050cf
3	2019-10-03 19:23:20 UTC	view	1004655	2053013555631882655	electronics.smartphone	samsung	770.90	513276739	7d350fb0-49ac-4ad1-a2ba-34aae6f0172b
4	2019-10-02 13:03:45 UTC	view	1005072	2053013555631882655	electronics.smartphone	samsung	1039.80	540796372	b4967c5b-1f84-2dbe-616e-9ab638445935

Fig 1. Visualizing the dataframe

Before building the recommendation system the following feature engineering steps were carried out:

a. Creating ‘main_category’ and ‘sub_category’ columns from ‘category_code’ column:

```
[25]: #spliting the category code column values and storing the values in two new columns mainn_category and subcategory
df[['main_category','subcategory']]=df['category_code'].str.split('.', n=1, expand=True)
df.head()
```

[25]:

	event_time	event_type	product_id	category_id	category_code	brand	price	user_id	user_session	main_category	subcategory
0	2019-10-22 17:38:59 UTC	view	1005184	2053013555631882655	electronics.smartphone	samsung	1029.34	539358901	58a9ee1c-6df6-4d4a-847e-3fc93108b051	electronics	smartphone
1	2019-10-01 08:19:45 UTC	view	1005105	2053013555631882655	electronics.smartphone	apple	1415.48	541796323	21673d95-29c3-4ffc-a0c3-f2a98bc98b02	electronics	smartphone
2	2019-10-03 09:10:11 UTC	view	1004776	2053013555631882655	electronics.smartphone	xiaomi	182.50	556311667	d9507b2f-ec5b-4c56-9420-bce1aa3050cf	electronics	smartphone
3	2019-10-03 19:23:20 UTC	view	1004655	2053013555631882655	electronics.smartphone	samsung	770.90	513276739	7d350fb0-49ac-4ad1-a2ba-34aae6f0172b	electronics	smartphone
4	2019-10-02 13:03:45 UTC	view	1005072	2053013555631882655	electronics.smartphone	samsung	1039.80	540796372	b4967c5b-1f84-2dbe-616e-9ab638445935	electronics	smartphone

Fig 2. Dataframe after splitting the column category_code

This step will allow to recommend more details about the products like the main_category and subcategory (name of the product). Apart from this, using these two new columns the recommendation system will be more efficient and will be able to provide valid recommendations to the user.

b. Adding the ‘price_scaled’ and ‘interaction_score’ columns:

```
#scaling the price column and creating a new price_scaled column
scaler=MinMaxScaler()
price=pd.DataFrame(df['price'])
df['price_scaled']=scaler.fit_transform(price)
```

```
interaction_map = {'view': 1, 'cart': 2, 'purchase': 3}
df['interaction_score'] = df['event_type'].map(interaction_map)
```

Fig 3. Scaling and encoding the numerical and categorical columns

The ‘price’ column is scaled down to have a price range from 0 to 1, to decrease the training time and to deal with the outliers in the columns, scaling down these values will help mitigate the variance introduced by the outliers.

Adding interaction_score column quantifies user engagement, mapping event_type to values: 1 for "view," 2 for "cart," 3 for "purchase." This numeric representation reflects interest levels, enabling Random Forest and XGBRanker models to rank products by relevance. Interaction_score acts as target variable during training and ground truth for evaluation metrics like Precision@K and NDCG@K. Converting categorical data to continuous scale enhances personalization and accuracy, ensuring recommendations align with user behavior effectively.

c. Combining the categorical variables:

```
#combining all the categorical features into a single text for each instance
df['combined_text']=df['event_type'] + ' ' + df['category_code'] + ' ' + df['brand'] + ' ' + df['main_category'] + ' ' + df['subcateg
```

Fig 4. Combining the categorical columns

This is a crucial step as it transforms all the categorical variables into uniform strings, that enables TF-IDF vectorization for content-based recommendation. It enhances the similarity computation as the contextual relationships between event type and product attributes are captured. This feature engineering supports cosine similarity analysis, aligning product recommendations with user preferences effectively.

2. XGBoost and Random Forest:

a. Aggregating user_product_stats:

```
interaction_map = {'view': 1, 'cart': 2, 'purchase': 3}
df['interaction_score'] = df['event_type'].map(interaction_map)
```

```
user_product_stats = df.groupby(['user_id', 'product_id']).agg(
    total_interactions=('interaction_score', 'count'),
    avg_interaction_score=('interaction_score', 'mean'),
    last_interaction_score=('interaction_score', 'last')
).reset_index()

# Merge back with original data
df = df.merge(user_product_stats, on=['user_id', 'product_id'], how='left')
```

Fig 5. Feature engineering for XGBoost dataframe

This step involves grouping by user_id and product_id, calculating total_interactions as count, avg_interaction_score as mean, and last_interaction_score as latest value of interaction_score. This step captures user-product interaction patterns numerically and by combining them with behavioral features enhances the model input. Total_interactions reflects frequency, avg_interaction_score indicates average engagement, and last_interaction_score preserves recency.

b. Encoding Categorical variables:

```
#encoding the categorical variables
label_enc = LabelEncoder()
df['brand'] = label_enc.fit_transform(df['brand'])
df['category_code'] = label_enc.fit_transform(df['category_code'])
```

Fig 6. Encoding categorical variables

This is an important step as it involves converting the categorical variables(textual features) into numerical values, enabling machine learning models to process them effectively. The numeric format preserves categorical distinctions without assuming ordinality, facilitating feature use in Random Forest and XGBRanker.

Feature Selection:

Correlation Analysis:

Majority of the columns in the dataframe are text/categorical columns, although checking the correlation among the numerical columns is an important step while building a content based recommendation system as it measures the strength and direction of these relationships.

```
#correlation analysis
df_numeric=df.select_dtypes(include=np.number)
corr=df_numeric.corr(method='pearson')
plt.figure(figsize=(6,4))
sns.heatmap(corr, annot=True, cmap='coolwarm')
plt.title('Correlation Matrix')
plt.show()
```

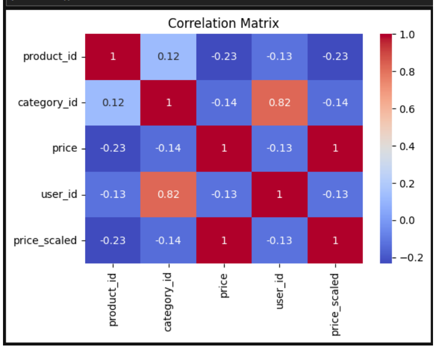

Fig 7. Heatmap for feature selection

From the matrix, we observe that category_id and user_id have a strong positive correlation (0.82), indicating that user preferences may align closely with specific categories. Similarly, price and price_scaled are perfectly correlated, as expected since one is a scaled transformation of the other. Other variables, such as product_id and price, show weak correlations (-0.23), suggesting minimal linear dependence. This analysis is crucial for identifying redundant features and understanding relationships between variables, which can guide feature selection or engineering in building a content-based recommendation system.

Feature selection process differs a bit for TF-IDF approach and the XGBoost and Random Forest approach. For TF-IDF, apart from the above-mentioned features, the column ‘combined_text’ (mentioned in feature engineering) is considered as it is a combination of the most important features. It summarizes all the textual imformation.

category_code	brand	price	user_id	user_session	main_category	subcategory	price_scaled	interaction_score	combined_text
kids.toys	orange	9.24	516207684	43e06b30-9a10-4ac5-91dd-ff7ed04aab82	kids	toys	0.001556	1	view kids.toys orange kids toys
appliances.kitchen.refrigerators	atlant	330.77	514498652	98c1ba90-5b20-4b48-ae64-e0223edc1627	appliances	kitchen.refrigerators	0.055713	1	view appliances.kitchen.refrigerators atlant a...
electronics.smartphone	samsung	196.83	528160375	29b1562d-8e2b-4298-ab2e-a186cbeaa7e1	electronics	smartphone	0.033153	1	view electronics.smartphone samsung electronic...
electronics.smartphone	oppo	153.04	525068636	03fdcfca-f8e1-40cf-928d-c104cf0de7ea	electronics	smartphone	0.025777	1	view electronics.smartphone oppo electronics s...
computers.notebook	acer	360.34	538423585	efb58f80-2ca8-4489-b672-9dc7961eaa7a	computers	notebook	0.060694	1	view computers.notebook acer computers notebook

Fig 8. Displaying the ‘combined_text’ column

For the Random Forest and XGBoost approach, it is necessary to consider the features created in the feature engineering process like the total_interactions, avg_interaction_score and last_interaction_score.

Data Splitting

1. TF-IDF

For the TF-IDF approach there is no splitting of the dataset because of the way the products are recommended. In this approach, the cosine similarity of the products with each other is calculated and on the basis of highest cosine similarity value, the products are recommended. The greater the number of instances, more accurate the cosine similarity value and thus the recommendations are more relevant.

2. **XGBoost**

Before feeding the data to the XGBoost model, the dataset was split into training and testing sets. The train_test_split class was used provided by sklearn. 20% of the original dataset was reserved for testing.

```
# Features for ranking
X = df[['user_id', 'product_id', 'category_code', 'brand', 'price',
        'total_interactions', 'avg_interaction_score', 'last_interaction_score']]
y = df['interaction_score']  # Target is the interaction score

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Calculate group sizes for test set
group_test = X_test.groupby('user_id').size().to_numpy()
```

Fig 9. Splitting the dataset into train and test sets

Computing group_test involves grouping X_test by user_id, calculating the size of each group, and converting results to a NumPy array. This process quantifies interactions per user in the test set, creating a group size vector. It is essential for XGBRanker's pairwise ranking objective, group_test ensures the model recognizes user-specific interaction counts, enabling accurate ranking of products within each user's context during evaluation, supporting effective recommendation generation. The same process is also carried out for the training set, discussed later in the training section.

3. **Random Forest**

Before feeding the data to the Random Forest model, the dataset was split into training and testing sets. The train_test_split class was used provided by sklearn. 20% of the original dataset was reserved for testing.

```
# Define features and target
features = ['category_code', 'brand', 'price', 'total_interactions', 'avg_interaction_score', 'last_interaction_score']
X = df1[['user_id', 'product_id'] + features]  # Include user_id and product_id for filtering
y = df1['interaction_score']

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Fig 10. Splitting the dataset into train and test sets for random forest

Model training

1. **TF-IDF**

The process of training in this approach is a bit different from the traditional machine learning model training. Following steps were to taken to get the product recommendations for users

a. Implementing the TF-IDF vectorizer:

```python
#implementing tfidf vectorizer on the combined text
tfidf_vectorizer=TfidfVectorizer(max_features=1200)
tfidf_matrix=tfidf_vectorizer.fit_transform(df['combined_text'])
```

Fig 11. Vectorizing the 'combined_test' column

The TfidfVectorizer is initialized with max_features=1200 (due to computational limitations) restricts vocabulary to 1200 key terms. The fit_transform methos is exeuted on combined_text transforms concatenated categorical data into a TF-IDF matrix, assigning weights based on term frequency and inverse document frequency. This process generates a sparse feature representation, capturing textual importance for training. Limiting features reduces dimensionality, enhancing computational efficiency while retaining critical information for similarity-based recommendation in the TF-IDF model training phase.

b. Extracting the feature names

```python
feature_names = tfidf_vectorizer.get_feature_names_out()

# Convert the TF-IDF matrix to a DataFrame for easier inspection
tfidf_df = pd.DataFrame(tfidf_matrix.toarray(), columns=feature_names)

# Display the first few rows of the TF-IDF matrix
print(tfidf_df.head())
```

```
   accessories  accord      acer  acme  acoustic  acqua  acv  adamex  adata  \
0          0.0     0.0  0.000000   0.0       0.0    0.0  0.0     0.0    0.0
1          0.0     0.0  0.000000   0.0       0.0    0.0  0.0     0.0    0.0
2          0.0     0.0  0.000000   0.0       0.0    0.0  0.0     0.0    0.0
3          0.0     0.0  0.000000   0.0       0.0    0.0  0.0     0.0    0.0
4          0.0     0.0  0.469178   0.0       0.0    0.0  0.0     0.0    0.0

   adidas  ...  zalman  zanussi  zebra  zelmer  zeppelin  zeta  zinc  zlatek  \
0     0.0  ...     0.0      0.0    0.0     0.0       0.0   0.0   0.0     0.0
1     0.0  ...     0.0      0.0    0.0     0.0       0.0   0.0   0.0     0.0
2     0.0  ...     0.0      0.0    0.0     0.0       0.0   0.0   0.0     0.0
3     0.0  ...     0.0      0.0    0.0     0.0       0.0   0.0   0.0     0.0
4     0.0  ...     0.0      0.0    0.0     0.0       0.0   0.0   0.0     0.0

   zte  zyxel
0  0.0    0.0
1  0.0    0.0
2  0.0    0.0
3  0.0    0.0
4  0.0    0.0
```

Fig 12. Displaying the feature vector

The feature_names from tfidf_vectorizer are extracted using get_feature_names_out(). The tfidf_matrix is then converted to a dense array and creating tfidf_df with feature_names as columns facilitates inspection. Displaying tfidf_df.head() shows the first few rows, revealing TF-IDF weights for each term per instance. This step aids in understanding feature representation, verifying vectorization output before training the TF-IDF recommendation model.

c. Combining tf-idf matrix with the numerical columns

```
#preparing the price_scaled column by converting it into a 2D array
price_features=df['price_scaled'].values.reshape(-1,1)

#combining the features using sparse matrices
combined_features=hstack([tfidf_matrix, price_features])

print(f'Final feature matrix shape: {combined_features.shape}')
```

```
Final feature matrix shape: (38408, 1201)
```

Fig 13. Stacking feature vector with price_scaled (numerical feature)

Combining tfidf_matrix with price_scaled enhances feature set for TF-IDF training. Converting price_scaled to a 2D array via reshape(-1,1) ensures compatibility. Using hstack merges sparse tfidf_matrix with numerical price_features, creating combined_features. Printing combined_features.shape confirms the resulting matrix dimensions, integrating textual and price data. This step enriches the model with both categorical and numerical insights, improving recommendation quality in the TF-IDF training phase.

d. Computing the cosine similarity

```
sparse_features = csr_matrix(combined_features)

def compute_top_k_similarities(features, batch_size=1000, top_k=10):
    n_samples = features.shape[0]

    # Store only the top-k similarities per row
    top_k_indices = np.zeros((n_samples, top_k), dtype=int)
    top_k_values = np.zeros((n_samples, top_k), dtype=float)

    for start in range(0, n_samples, batch_size):
        end = min(start + batch_size, n_samples)

        # Compute similarity only for the batch
        batch_similarities = cosine_similarity(features[start:end], features, dense_output=False)

        # Convert sparse matrix to dense for indexing (still efficient)
        batch_similarities = batch_similarities.toarray()

        # Get top-k values and indices
        for i in range(batch_similarities.shape[0]):
            row = batch_similarities[i]
            top_k_idx = np.argpartition(row, -top_k)[-top_k:]  # Get indices of top-k
            sorted_idx = top_k_idx[np.argsort(-row[top_k_idx])]  # Sort them
            top_k_indices[start + i] = sorted_idx
            top_k_values[start + i] = row[sorted_idx]

    return top_k_indices, top_k_values

# Run with sparse features
top_k_indices, top_k_values = compute_top_k_similarities(sparse_features)
```

Fig 14. Computing cosine similarity

Converting combined_features to csr_matrix creates sparse_features, optimizing memory usage for sparse data in our TF-IDF pipeline. We define compute_top_k_similarities to process features in batches of 1000, calculating cosine similarities efficiently. The function stores the top 10 similarities per row in top_k_indices and top_k_values, leveraging argpartition and sorting for speed. Executing this with sparse_features produces similarity rankings critical for training. This approach ensures scalability in similarity computation, enabling our TF-IDF model to identify and rank closely related products effectively for recommendation purposes.

e. User profile creation (for personal recommendation)

```
# Filter interactions based on event_type (e.g., 'view', 'purchase')
user_interactions = df[df['event_type'].isin(['view', 'purchase'])]

# Group by user_id and aggregate product features
user_profiles = {}
for user_id, group in user_interactions.groupby('user_id'):
    # Get indices of products interacted by the user
    product_indices = group.index

    # Aggregate TF-IDF features for these products
    aggregated_tfidf = np.mean(tfidf_matrix[product_indices].toarray(), axis=0)

    # Aggregate numerical feature (price_scaled)
    aggregated_price = np.mean(group['price_scaled'])

    # Combine aggregated TF-IDF and price into a single profile vector
    user_profile = np.hstack([aggregated_tfidf, aggregated_price])

    # Store the user profile
    user_profiles[user_id] = user_profile

print(f"Number of user profiles created: {len(user_profiles)}")
```

```
Number of user profiles created: 35601
```

Fig 15. User profile creation

Filtering df for event_type values "view" and "purchase" creates user_interactions, focusing on significant user actions. Grouping by user_id, we aggregate features into user_profiles. For each group, extracting product_indices allows averaging tfidf_matrix rows into aggregated_tfidf. Calculating aggregated_price from price_scaled adds numerical context. Combining these into user_profile via np.hstack forms a unified vector per user. Storing profiles in a dictionary and printing the count tracks profile creation, supporting personalized TF-IDF recommendation training.

f. Recommending products to users

```python
if user_id not in user_profiles:
    return {"error": f"No profile found for user {user_id}"}

# Get the user's profile vector
user_profile_vector = user_profiles[user_id]

# Compute similarity between user's profile and all product vectors
product_similarity_scores = cosine_similarity([user_profile_vector], combined_features.toarray())[0]

# Get indices of top K similar products
user_indices = data[data['user_id'] == user_id].index.values
top_indices = [idx for idx in np.argsort(product_similarity_scores)[::-1] if idx not in user_indices][:k]

# Get recommended products
recommendations = data.iloc[top_indices][['product_id', 'brand', 'subcategory']].copy()
recommendations['similarity_score'] = product_similarity_scores[top_indices]

# Ground truth interaction scores for recommended products
user_data = data[data['user_id'] == user_id]
true_scores = [user_data[user_data['product_id'] == pid]['interaction_score'].iloc[0]
               if pid in user_data['product_id'].values else 0 for pid in recommendations['product_id']]

# Binary relevance (relevant if interaction_score >= 2, i.e., cart or purchase)
binary_true = np.array([1 if score >= 2 else 0 for score in true_scores])
```

Fig 16. Recommendation algorithm for TF-IDF vector

```python
user_id_to_recommend = 516207684
result = recommend_and_evaluate_tfidf(user_id_to_recommend, df, user_profiles, combined_features, k=5)
if "error" not in result:
    print("TF-IDF Recommendations:\n", result["recommendations"])
    print(f"Precision@5: {result['precision@k']:.3f}")
    print(f"NDCG@5: {result['ndcg@k']:.3f}")
```

```
TF-IDF Recommendations:
        product_id   brand subcategory  similarity_score
13670      9001506  orange        toys          1.000000
2712       9000564  orange        toys          1.000000
11203      9000564  orange        toys          1.000000
15207      9001886  orange        toys          0.999999
26180      9001886  orange        toys          0.999999
Precision@5: 0.000
NDCG@5: 0.000
```

Fig 17. Displaying recommendations for a particular user

Defining recommend_and_evaluate_tfidf computes TF-IDF recommendations for user 516207684 by retrieving user_profile_vector from user_profiles, calculating cosine similarities with combined_features, and selecting top K products (excluding user's interactions). Extracting product_id, brand, and subcategory, adding similarity_score, and evaluating with interaction_score yields metrics. The output lists toys from brand "orange" with high similarity scores, but Precision@5 and NDCG@5 are 0.000. This occurs because recommended products lack interaction_score >= 2 (cart or purchase) in user_data. If the user only viewed these products (interaction_score = 1), binary relevance becomes 0, resulting in zero metrics, indicating poor alignment with significant user actions and highlighting recommendation challenges.

2. **XGBoost**

The XGBoost library in python provides a specific module- XGBRanker that is the most suitable for recommendation systems. The XGBRanker is a gradient boosting algorithm designed specifically for ranking tasks. In this implementation, the model will rank the products best suitable for recommendation. Using category_code, brand, price, total_interactions, avg_interaction_score, and last_interaction_score as features, with interaction_score as the target, recommend_and_evaluate_xgboost generates XGBRanker recommendations

a. Hyperparameter tuning

```
param_grid = {
    'learning_rate': [0.01, 0.1],
    'max_depth': [3, 5],
    'n_estimators': [50, 100],
}

# Use GroupKFold for cross-validation
gkf = GroupKFold(n_splits=5)

# Manual cross-validation
best_score = float('inf')
best_params = None
X_train_features = X_train.drop(columns=['user_id', 'product_id'])
```

Fig 18. Hyperparameter tuning XGBoost model

```
for params in ParameterGrid(param_grid):
    scores = []
    for train_idx, val_idx in gkf.split(X_train_features, y_train, groups=X_train['user_id']):
        # Split data for this fold
        X_tr, X_val = X_train_features.iloc[train_idx], X_train_features.iloc[val_idx]
        y_tr, y_val = y_train.iloc[train_idx], y_train.iloc[val_idx]

        # Calculate group sizes for this fold
        group_tr = X_train.iloc[train_idx].groupby('user_id').size().to_numpy()

        # Train the model
        model = xgb.XGBRanker(**params, objective='rank:pairwise', random_state=42)
        model.fit(X_tr, y_tr, group=group_tr)

        # Predict and evaluate
        preds = model.predict(X_val)
        mse = mean_squared_error(y_val, preds)
        scores.append(mse)
```

Fig 19. Iterating through the parameter grid

```
Params: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 50}, Avg MSE: 2.1441203594207763
Params: {'learning_rate': 0.01, 'max_depth': 3, 'n_estimators': 100}, Avg MSE: 3.025545930862427
Params: {'learning_rate': 0.01, 'max_depth': 5, 'n_estimators': 50}, Avg MSE: 2.1434604644775392
Params: {'learning_rate': 0.01, 'max_depth': 5, 'n_estimators': 100}, Avg MSE: 3.0230700969696045
Params: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 50}, Avg MSE: 10.617846298217774
Params: {'learning_rate': 0.1, 'max_depth': 3, 'n_estimators': 100}, Avg MSE: 11.191568565368652
Params: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 50}, Avg MSE: 10.525036811828613
Params: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 100}, Avg MSE: 11.062234497070312
Best Parameters: {'learning_rate': 0.01, 'max_depth': 5, 'n_estimators': 50} Best MSE: 2.1434604644775392
```

Fig 20. Displaying the best_model

Defining param_grid specifies hyperparameter combinations for learning_rate, max_depth, and n_estimators. Using GroupKFold with 5 splits ensures user-based cross-validation. Iterating through ParameterGrid, each fold splits X_train_features and y_train by user_id groups. Calculating group_tr sizes, training XGBRanker with rank:pairwise objective, and predicting on validation sets computes MSE scores. Averaging scores across folds identifies best_params with the lowest best_score. This process optimizes XGBRanker for ranking, enhancing recommendation accuracy.

b. Extracting the best model and training it on the grouped data

```
best_model = xgb.XGBRanker(**best_params, objective='rank:pairwise', random_state=42)
group_train = X_train.groupby('user_id').size().to_numpy()
best_model.fit(X_train_features, y_train, group=group_train)
```

```
XGBRanker

XGBRanker(base_score=None, booster=None, callbacks=None, colsample_bylevel=None,
          colsample_bynode=None, colsample_bytree=None, device=None,
          early_stopping_rounds=None, enable_categorical=False,
          eval_metric=None, feature_types=None, gamma=None, grow_policy=None,
          importance_type=None, interaction_constraints=None,
          learning_rate=0.01, max_bin=None, max_cat_threshold=None,
          max_cat_to_onehot=None, max_delta_step=None, max_depth=5,
          max_leaves=None, min_child_weight=None, missing=nan,
          monotone_constraints=None, multi_strategy=None, n_estimators=50,
          n_jobs=None, num_parallel_tree=None, objective='rank:pairwise', ...)
```

Fig 21. Training the model on grouped data

Initializing best_model as XGBRanker with best_params, rank:pairwise objective, and random_state=42 sets up the optimized model. Computing group_train by grouping X_train by user_id and converting sizes to a NumPy array defines user group sizes. Fitting best_model on X_train_features and y_train with group_train trains the model for ranking tasks. This step finalizes XGBRanker training, leveraging user-specific grouping to enhance product recommendation accuracy.

c. Recommending products using XGBoost

```
user_data = data[data['user_id'] == user_id].copy()  # Use .copy() to avoid SettingWithCopyWarning

if user_data.empty:
    return {"error": f"No data available for user {user_id}"}

# Add interaction_score from y_train to user_data
user_data['interaction_score'] = y_train.loc[user_data.index]

# Prepare input features (drop user_id, product_id, and interaction_score)
features = user_data.drop(columns=['user_id', 'product_id', 'interaction_score'])

# Predict interaction scores
user_data['predicted_score'] = model.predict(features)

# Ground truth interaction scores
true_scores = user_data['interaction_score'].values

# Sort by predicted score in descending order
recommendations = user_data.sort_values(by='predicted_score', ascending=False)

# Select top K products
top_k = recommendations[['product_id', 'predicted_score']].head(k)

# Deduplicate product_info to ensure unique product_id
unique_product_info = product_info[['product_id', 'category_code']].drop_duplicates(subset='product_id')

# Map the encoded category_code values
top_k['product_name_encoded'] = top_k['product_id'].map(unique_product_info.set_index('product_id')['category_code'])

# Decode the category_code back to original strings
top_k['product_name'] = label_encoder.inverse_transform(top_k['product_name_encoded'].astype(int))

# Drop the temporary encoded column
top_k = top_k.drop(columns=['product_name_encoded'])
```

Fig 22. Recommendation algorithm for XGBoost model

```
user_id_to_recommend = 516207684
result = recommend_and_evaluate_xgboost(user_id_to_recommend, X_train, best_model, df, label_enc, y_train, k=5)
if "error" not in result:
    print("XGBRanker Recommendations (Training Set):\n", result["recommendations"])
    print(f"Precision@5: {result['precision@k']:.3f}")
    print(f"NDCG@5: {result['ndcg@k']:.3f}")
else:
    print(result["error"])
```

```
XGBRanker Recommendations (Training Set):
    product_id  predicted_score product_name
0     9001245         -0.414667    kids.toys
Precision@5: 0.000
NDCG@5: 0.000
```

Fig 23. Displaying the recommendation from XGBoost

Defining recommend_and_evaluate_xgboost generates XGBRanker recommendations for user 516207684 using training data. We filter X_train, add interaction_score from y_train, predict scores with best_model, and rank products. Mapping category_code to names via label_encoder, we select top K products. Precision@5 and NDCG@5 evaluate relevance and ranking. The output shows one recommendation, product_id 9801245 (kids.toys), with a negative predicted_score. Precision@5 and NDCG@5 are 0.000, as the recommended product's interaction_score is likely below 2 (e.g., only viewed), making binary relevance zero. This indicates XGBRanker's recommendations fail to align with significant user actions like cart or purchase, highlighting potential model limitations on this training set.

3. Random Forest

The Random Forest algorithm, implemented via scikit-learn, excels in recommendation systems by modeling complex patterns. As an ensemble method, it predicts product rankings effectively. Using category_code, brand, price, total_interactions, avg_interaction_score, and last_interaction_score as features, with interaction_score as the target, recommend_and_evaluate_rf generates Random Forest recommendations.

a. Train the model

```
# Train Random Forest model (no hyperparameter tuning)
rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
rf_model.fit(X_train[features], y_train)
```

```
RandomForestRegressor
RandomForestRegressor(random_state=42)
```

Fig 24. Training the Random Forest model

The model was trained using a 100 estimators, the features it was trained on are 'category_code', 'brand', 'price', 'total_interactions', 'avg_interaction_score', 'last_interaction_score'.

b. Recommending products to user

```python
user_data = data[data['user_id'] == user_id].copy()

if user_data.empty:
    return {"error": f"No data available for user {user_id}"}

# Add interaction_score from y_train to user_data
user_data['interaction_score'] = y_train.loc[user_data.index]

# Prepare input features (drop user_id, product_id, and interaction_score)
features = user_data.drop(columns=['user_id', 'product_id', 'interaction_score'])

# Predict interaction scores
user_data['predicted_score'] = model.predict(features)

# Ground truth interaction scores
true_scores = user_data['interaction_score'].values

# Sort by predicted score in descending order
recommendations = user_data.sort_values(by='predicted_score', ascending=False)

# Select top K products
top_k = recommendations[['product_id', 'predicted_score']].head(k)

# Deduplicate product_info to ensure unique product_id
unique_product_info = product_info[['product_id', 'category_code']].drop_duplicates(subset='product_id')

# Map the encoded category_code values
top_k['product_name_encoded'] = top_k['product_id'].map(unique_product_info.set_index('product_id')['category_code'])

# Decode the category_code back to original strings
top_k['product_name'] = label_encoder.inverse_transform(top_k['product_name_encoded'].astype(int))

# Drop the temporary encoded column
top_k = top_k.drop(columns=['product_name_encoded'])

# True relevance scores for top K
top_k_true_scores = recommendations['interaction_score'].head(k).values
```

Fig 25. Recommendation algorithm for Random Forest

```python
# Example usage with training set
user_id_to_recommend = 528160375
result = recommend_and_evaluate_rf(user_id_to_recommend, X_train, rf_model, df1, label_enc, y_train, k=5)
if "error" not in result:
    print("Random Forest Recommendations (Training Set):\n", result["recommendations"])
    print(f"Precision@5: {result['precision@k']:.3f}")
    print(f"NDCG@5: {result['ndcg@k']:.3f}")
else:
    print(result["error"])
```

```
Random Forest Recommendations (Training Set):
   product_id  predicted_score          product_name
2     1004751              1.0  electronics.smartphone
Precision@5: 0.000
NDCG@5: 0.000
```

Fig 26. Recommendations from Random Forest

Defining recommend_and_evaluate_rf generates Random Forest recommendations for user 528160375 using training data with features category_code, brand, price, total_interactions, avg_interaction_score, and last_interaction_score, targeting interaction_score. We filter X_train, add interaction_score from y_train, predict scores with rf_model, and rank products. Mapping category_code to names via label_encoder, we select top K products. The output shows one recommendation, product_id 1004751 (electronics.smartphone), with predicted_score 1.0. Precision@5 and NDCG@5 are 0.000, as the interaction_score is likely below 2 (e.g., only viewed), making binary relevance zero. This indicates Random Forest's recommendations fail to capture significant user actions like cart or purchase.

Comparing Recommendations from each model

```
# Example usage
user_id_to_compare = 516207684
results = compare_recommendations(user_id_to_compare, df, user_profiles, combined_features,
                                  X_train, best_model, X_train, rf_model, df, label_enc, y_train, k=5)
```

```
TF-IDF Recommendations:
        product_id   brand subcategory  similarity_score
13670      9001506  orange        toys          1.000000
2712       9000564  orange        toys          1.000000
11203      9000564  orange        toys          1.000000
15207      9001886  orange        toys          0.999999
26180      9001886  orange        toys          0.999999
Precision@5: 0.000
NDCG@5: 0.000

XGBRanker Recommendations (Training Set):
    product_id  predicted_score product_name
0     9001245        -0.414667    kids.toys
Precision@5: 0.000
NDCG@5: 0.000

Random Forest Recommendations (Training Set):
    product_id  predicted_score product_name
0     9001245              1.0    kids.toys
Precision@5: 0.000
NDCG@5: 0.000
```

Comparing TF-IDF, XGBRanker, and Random Forest for user 516207684 reveals distinct recommendation outcomes on a smaller dataset, constrained by computational limitations, potentially impacting results. TF-IDF recommends toys from "orange" with high similarity scores but yields zero Precision@5 and NDCG@5, indicating no significant user actions like cart or purchase. XGBRanker suggests a kids.toys product with a negative score, also showing zero metrics due to low interaction_score. Random Forest proposes an electronics.smartphone with a score of 1.0, yet metrics remain zero, reflecting similar relevance issues. Using compare_recommendations, this analysis highlights TF-IDF's content-based strength, XGBRanker's ranking focus, and Random Forest's pattern recognition. The limited dataset may contribute to zero metrics, as sparse user interactions reduce the likelihood of capturing meaningful engagements, underscoring challenges in achieving accurate recommendations across all methods on the training set.