Report on ShopSmart Project

1. Introduction

1.1 Project Objective

The ShopSmart project was developed with the aim of creating a recommendation system that utilizes collaborative filtering to suggest products to users based on their past preferences. This system aims to enhance users' shopping experience by providing personalized recommendations, thereby increasing the likelihood of purchases and customer satisfaction.

1.2 Project Overview

ShopSmart consists of several phases: data cleaning and preparation, exploratory data analysis (EDA), training of collaborative filtering models, and evaluation of their performance. The project has been structured in a modular way to facilitate maintenance and future expansion.

2. Project Structure

2.1 `notebook/` Folder

This folder contains Jupyter notebooks used for data preparation, exploratory analysis, and model comparison.

- **01-preprocessing.ipynb**: Notebook for data cleaning and preparation.
- **02-EDA.ipynb**: Notebook for data visualization and exploratory analysis.
- **03-compare.ipynb**: Notebook for comparing model performances and item recommendation.

2.2 `source/` Folder

This folder contains the source code of models and utilities required for the project.

- **models/**
  - **collabFilter_ItemItem.py**: Implementation of item-based collaborative filtering.
  - **collabFilter_UserUser.py**: Implementation of user-based collaborative filtering.
  - **base_model.py**: General management of collaborative filtering.
- **utils/**
  - **dataHandler.py**: Functions for data management and preparation.
  - **utils.py**: General utility functions.

2.3 Main Files

- **README.md**: Project description.
- **LICENSE**: Project license.
- **requirements.txt**: Project dependencies.

3. Installation

3.1 Cloning the Repository

To begin, you need to clone the GitHub repository and navigate to the project directory:

```
git clone https://github.com/DanieleT25/ShopSmart.git
cd ShopSmart
```

3.2 Creating and Activating the Virtual Environment

Next, create and activate a virtual environment to manage dependencies:

```
python3 -m venv venvShopSmart
source venvShopSmart/bin/activate  # On Windows, use `venvShopSmart\Scripts\activate`
```

3.3 Installing Dependencies

Finally, install the required dependencies:

```
pip install -r requirements.txt
```

4. Usage

4.1 Data Preprocessing

Data preprocessing is a crucial phase to ensure that the data used in the recommendation system is clean, consistent, and ready for analysis. In ShopSmart, data preprocessing has been implemented in the notebook `01-preprocessing.ipynb`, using various data manipulation and cleaning techniques. Here is a detailed description of the operations performed during this phase:

4.1.1 Loading and Preprocessing Data

The first step involved importing the `DataHandler` module from the directory `source/utils/`, which contains functions necessary for data management and preparation. Using this module, data was loaded from a CSV file (`../data/export.csv`) and preprocessed. This process includes data cleaning operations, such as removing missing values and correcting inconsistencies in the data.

```
from source.utils.dataHandler import DataHandler
data_handler = DataHandler('../data/export.csv')
data_handler.preprocess_data()
```

The `preprocess_data` function processes the data by converting the date attribute into a datetime format and applying a filtering function to remove null values.

```
def preprocess_data(self):
    """Preprocess the data"""

    self.df.columns = self.COLUMNS

    self.df['data'] = pd.to_datetime(self.df['data'])
    self.df['ora'] = pd.to_datetime(self.df['ora'], format='%H:%M').dt.time

    self.__filter_data()
```

4.1.2 Data Filtering by Quarter

To focus the analysis on a specific time period, a choice between four quarters was introduced:

- January-March
- April-June
- July-September
- October-December

In this example, the "Jan-Mar" quarter was selected.

The choice was made to select a quarter in order to recommend items from that period (e.g., not recommending ice cream in winter).

```
quarter = 'Jan-Mar'
data_handler.filter_month(quarter)
```

4.1.3 Selection of Index for Recommendation System

For implementing the recommendation system, it was necessary to identify the customer. The only feature providing this information is the loyalty card (`tessera`), but the number of customers with a loyalty card is very low. Therefore, the option was provided to choose between the loyalty card number and the receipt code (`id_sc`). In this example, the loyalty card (`tessera`) was chosen as the index.

```
index = 'tessera'  # or index = 'id_sc'
df = data_handler.process(index, 'liv3', 'descr_liv3')
data_handler.descriptiveStats(df)
```

Additionally, to avoid an explosion of recommended items, the attribute `cod_prod` was not chosen but rather `liv3`, which is a code representing the level 3 category. Level 4 was not chosen because subsequent tests showed that using level 3 codes slightly improved the recommendation system performance.

> **Tip**
>
> The dataset includes 4 attributes (`liv1`, `liv2`, `liv3`, `liv4`) that together represent a merchandise hierarchy (where `liv1` codes are broader categories and `liv4` codes are more specific categories).

4.1.4 Normalization and Data Saving

Subsequently, the data was further processed to normalize values on a scale from 1 to 10 using Min-Max normalization. This step ensures that the data is on a comparable scale, thereby improving the performance of the recommendation models.

```
df = data_handler.process(index, 'liv3', 'descr_liv3', 100)
df_MinMaxNorm_1_10 = data_handler.minMax_normalization_1_10(df)
data_handler.descriptiveStats(df_MinMaxNorm_1_10)
```

Min-Max normalization formula with predefined ranges:

$$\text{normalized\_value} = 1 + \frac{(\text{value} - \min(\text{value})) \cdot 9}{\max(\text{value}) - \min(\text{value})} \tag{1}$$

Finally, the normalized data was split into training and test sets and saved for later use in recommendation models. The split was done with a ratio of 75% for training and 25% for testing.

```
data_handler.split_and_save_data(df_MinMaxNorm_1_10, '../data/', 0.25)
```

> **Note**
>
> In addition to splitting the data into two datasets (`train_data.csv` and `test_data.csv`), `split_and_save_data` also saves the entire cleaned dataset as `data_clean.csv`.

4.2 Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is an essential phase to better understand the available data, identify significant patterns, and prepare the data for subsequent modeling processes. In the ShopSmart project, EDA was performed in the notebook `02-EDA.ipynb` using various techniques and visualizations. Here is a detailed description of the operations carried out during this phase:

4.2.1 Loading and Preprocessing Data

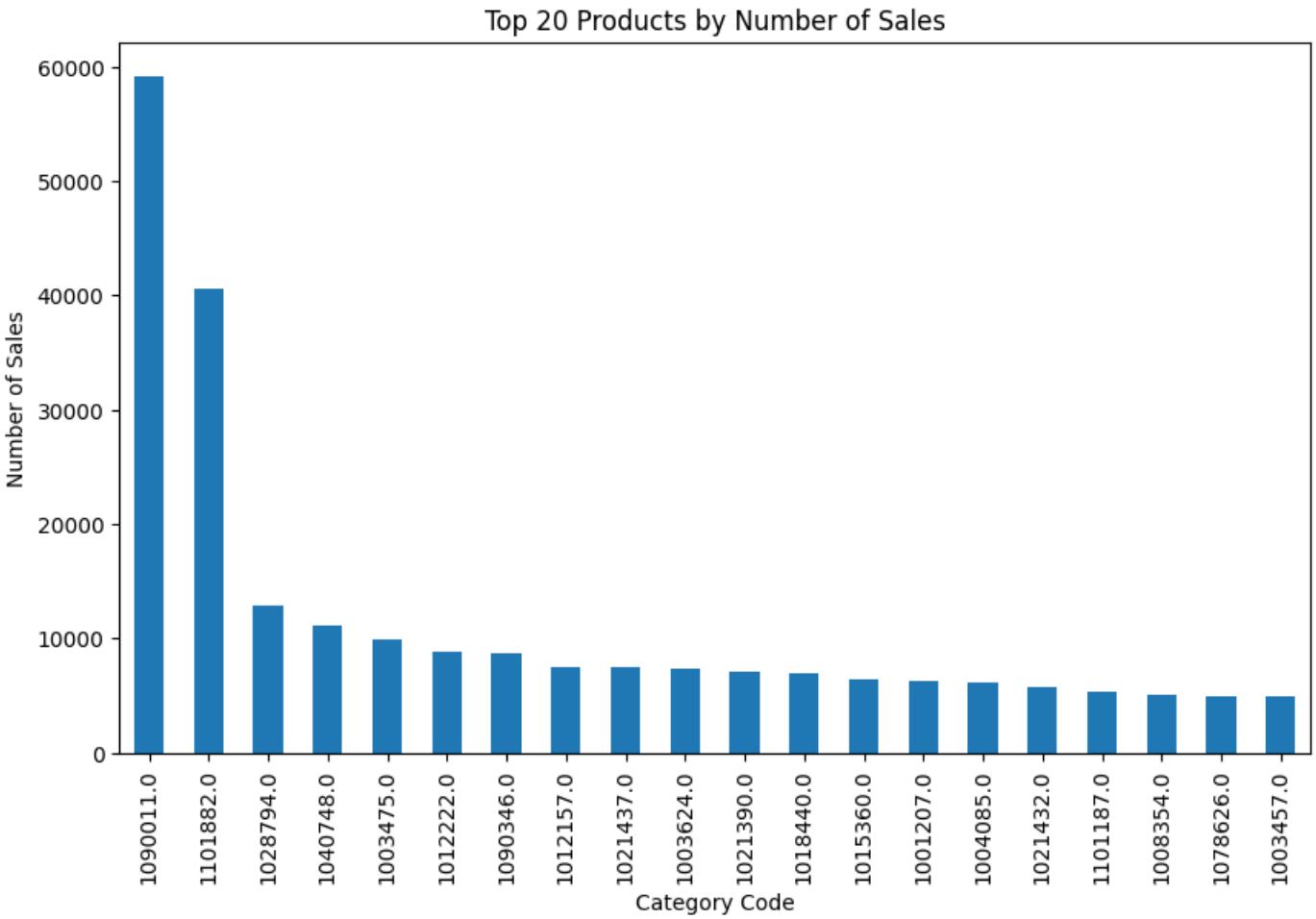
Similar to the preprocessing phase, the `DataHandler` module was used to load cleaned data from the CSV file (`../data/clean_data.csv`).

```
from source.utils.dataHandler import DataHandler
data_handler = DataHandler('../data/clean_data.csv')
data_handler.get_data().head()
```

4.2.2 Distribution of Sales by Product

A key aspect of EDA is understanding how sales are distributed among different products. This analysis helps identify the best-selling and least popular products, providing valuable insights for the recommendation system. The distribution of sales was visualized using specific functions from the `DataHandler` module.

```
## 1. Distribution of sales by product
data_handler.distribSales()
```

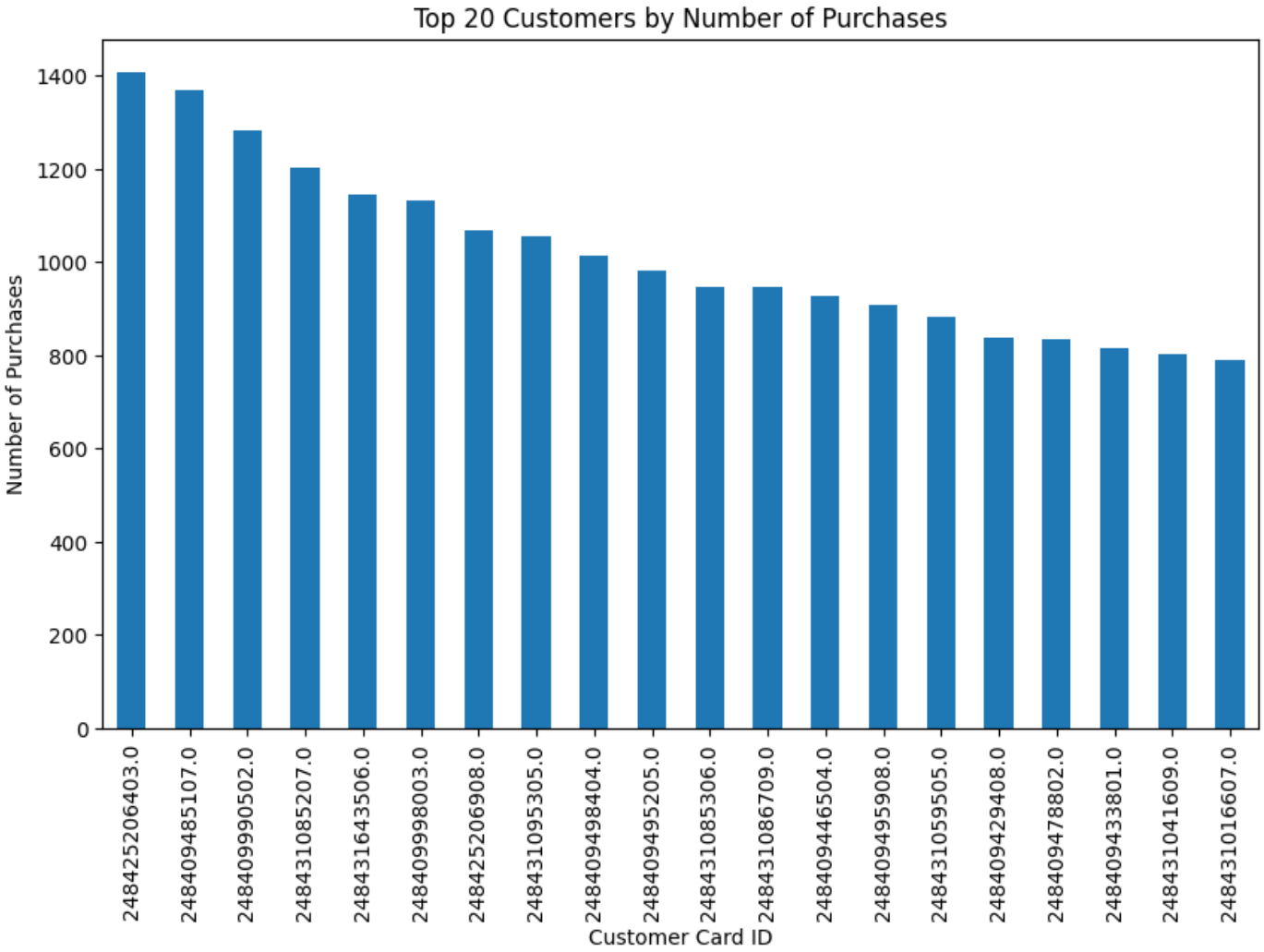



```
def distribSales(self):
        """Plot the distribution of sales by product"""

        product_sales = self.df['cod_prod'].value_counts()

        plt.figure(figsize=(10, 6))
        product_sales.head(20).plot(kind='bar')
        plt.title('Top 20 Products by Number of Sales')
        plt.xlabel('Category Code')
        plt.ylabel('Number of Sales')
        plt.show()
```

4.2.3 Frequency of Customer Purchases

Analyzing the frequency of customer purchases is crucial for understanding user behavior. This analysis can reveal insights into how often customers make purchases.

```
## 2. Frequency of customer purchases
data_handler.freqCustPurch()
```

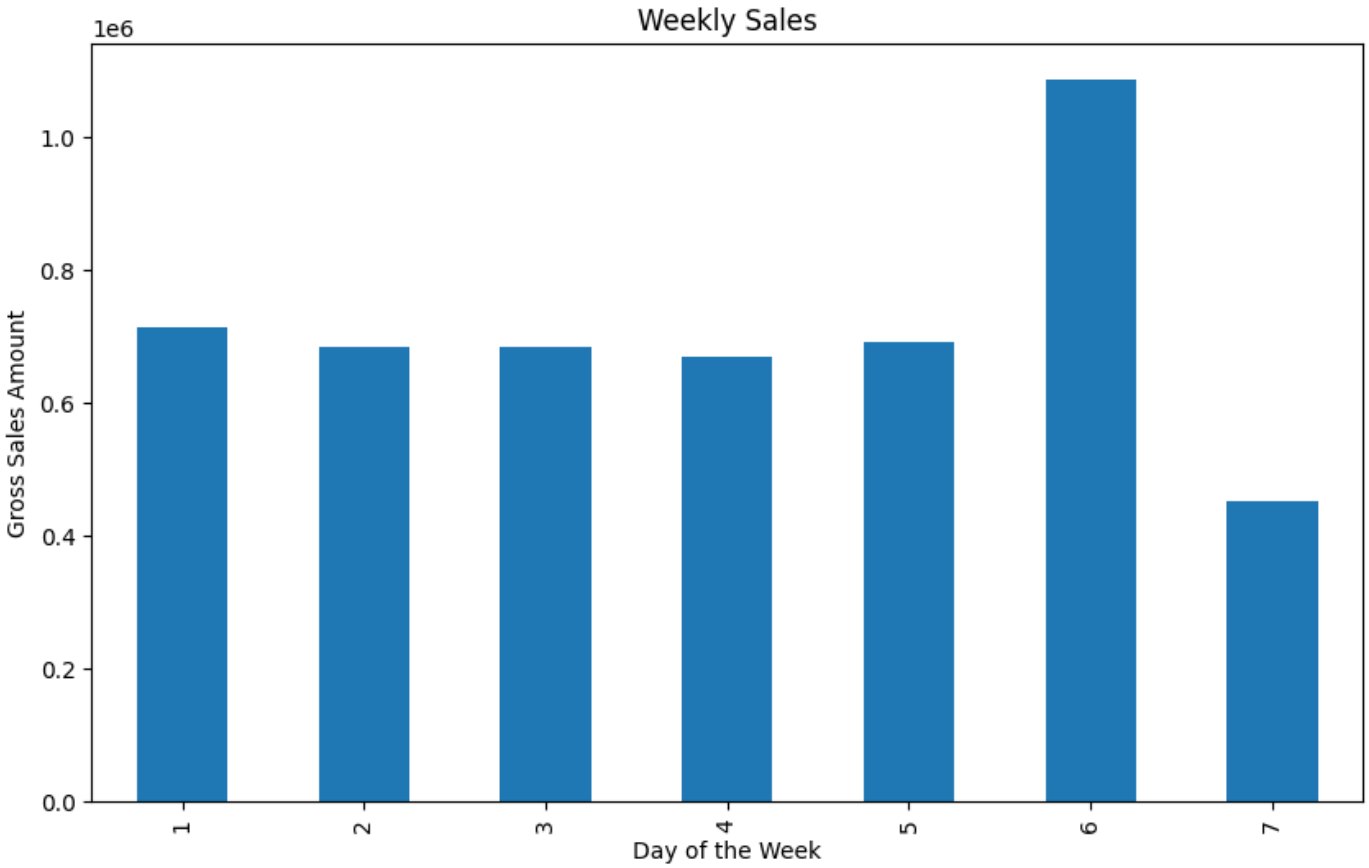



```
def freqCustPurch(self):
        """Plot the frequency of customers by number of purchases"""

        customer_frequency = self.df['tessera'].value_counts()

        plt.figure(figsize=(10, 6))
        customer_frequency.head(20).plot(kind='bar')
        plt.title('Top 20 Customers by Number of Purchases')
        plt.xlabel('Customer Card ID')
        plt.ylabel('Number of Purchases')
        plt.show()
```

4.2.4 Identification of Temporal Patterns in Sales

Another important aspect of EDA is identifying temporal patterns in sales. Recognizing these patterns can help better predict future sales and optimize recommendations based on weekly periods. Visualization of weekly patterns was performed using temporal charts.

```
## 3. Identification of seasonal or temporal patterns in sales
data_handler.plot_sales()
```




```
def plot_sales(self):
        """Plot weekly sales"""

        self.df['data'] = pd.to_datetime(self.df['data'])
        self.df['day_of_week'] = self.df['data'].dt.dayofweek

        self.df['day_of_week'] = self.df['day_of_week'].apply(lambda x: x + 1)
        weekly_sales = self.df.groupby('day_of_week')['r_importo_lordo'].sum()

        plt.figure(figsize=(10, 6))
        weekly_sales.plot(kind='bar')
        plt.title('Weekly Sales')
        plt.xlabel('Day of the Week')
        plt.ylabel('Gross Sales Amount')
        plt.show()
```

4.3 Model Comparison

Model comparison is a critical phase to determine which recommendation model offers the best performance in terms of accuracy and reliability. In the ShopSmart project, various models were compared using the notebook `03-compare.ipynb`. Here is a detailed description of the operations carried out during this phase:

4.3.1 Data Loading

For comparison purposes, the data was split into training and test sets. Cleaned data was loaded from CSV files (`../data/data_clean.csv`, `../data/train_data.csv`, `../data/test_data.csv`) to ensure a clear separation between model training and evaluation.

```
import pandas as pd
import sys
import os

sys.path.append(os.path.abspath(os.path.join(os.getcwd(), '..')))

from source.models.base_model import baseModel
from source.models.collabFilter_UserUser import CollaborativeFilter_UserUser
from source.models.collabFilter_ItemItem import CollaborativeFilter_ItemItem
import source.utils.utils as utils

data = pd.read_csv('../data/data_clean.csv')
train_data = pd.read_csv('../data/train_data.csv')
test_data = pd.read_csv('../data/test_data.csv')
```

4.3.2 Utility Matrix Creation

The utility matrix was created using the training data, where rows represent users (identified by loyalty card) and columns represent level 3 product categories (`liv3`). The values in the matrix represent how many times a user purchased a product category.

```
U = train_data.pivot_table(index='tessera', columns='liv3', values='value')
```

4.3.3 Model Training

Two collaborative filtering models were trained with consideration of 100 items for recommendation:

- **CollaborativeFilter_UserUser**: User-based collaborative filtering.
- **CollaborativeFilter_ItemItem**: Item-based collaborative filtering.

Each model was trained using the matrix utility created earlier.

```
cfUser = CollaborativeFilter_UserUser(N=100)
cfUser.fit(U)

cfItem = CollaborativeFilter_ItemItem(N=100)
cfItem.fit(U)
```

> **Important**
>
> On the GitHub code you will see that there is a third extra model (SVD) but since it wasn't added we decided not to include it in this relationship.

4.3.4 Model Prediction and Evaluation

The models' performances were evaluated using Mean Absolute Error (MAE) on the test data. Additionally, precision, recall, and F1-score metrics were calculated for further evaluation. The following code demonstrates the metrics applied to the item-item model.

Prediction

Prediction can be computed:

Sequentially

```
predicted_ratings_item = utils.predict(cfItem, test_data)
```

```
def predict(cf, test_data):
    predicted_ratings = []
    for i, row in tqdm(test_data.iterrows(), total=len(test_data)):
        try:
            rating = cf.predict(row['tessera'], row['liv3'])
        except:
            rating = np.nan
        predicted_ratings.append(rating)
    return np.array(predicted_ratings)
```

In Parallel

```
predicted_ratings_user, predicted_ratings_item = utils.parallel_predict(cfUser, cfItem,
test_data)
```

```
def parallel_predict(cfUser, cfItem, test_data):
    with ProcessPoolExecutor(max_workers=2) as executor:
        futures = {
            executor.submit(predict, cfUser, test_data): 'user',
            executor.submit(predict, cfItem, test_data): 'item'
        }

        predicted_ratings_user = None
        predicted_ratings_item = None

        for future in as_completed(futures):
            model_type = futures[future]
            if model_type == 'user':
                predicted_ratings_user = future.result()
            else:
                predicted_ratings_item = future.result()

        return predicted_ratings_user, predicted_ratings_item
```

Metrics

```
mae_score_item = utils.mae(test_data['value'], predicted_ratings_item)

precision_item = utils.precision_at_k(test_data['value'], predicted_ratings_item, k)
recall_item = utils.recall_at_k(test_data['value'], predicted_ratings_item, k)
f1_item = utils.f1_score_at_k(test_data['value'], predicted_ratings_item, k)

display(f"Precision Item: {precision_item}")
display(f"Recall Item: {recall_item}")
display(f"F1 Item: {f1_item}")
```

```
def mae(y_true, y_pred):
    return (y_true - y_pred).abs().mean()

def precision_at_k(y_true, y_pred, k):
    top_k_pred = y_pred[:k]
    relevant = sum([1 for i in top_k_pred if i in y_true])
    return relevant / k

def recall_at_k(y_true, y_pred, k):
    top_k_pred = y_pred[:k]
    relevant = sum([1 for i in top_k_pred if i in y_true])
    return relevant / len(y_true)

def f1_score_at_k(y_true, y_pred, k):
    precision = precision_at_k(y_true, y_pred, k)
    recall = recall_at_k(y_true, y_pred, k)
    if precision + recall == 0:
        return 0
    return 2 * (precision * recall) / (precision + recall)
```

4.3.5 Recommendations

Finally, the model with the best performance was used to generate recommendations. For an example user (identified by loyalty card), five products were recommended, with product descriptions retrieved from the original DataFrame.

```
user_index = U.index[0]
recommendations = cfItem.recommend(user_index, 5)

data['liv3'] = data['liv3'].astype(str)
descriptions = []
for item, _ in recommendations:
    item_str = str(item)
    if not data.loc[data['liv3'] == item_str, 'descr_liv3'].empty:
        descriptions.append(data.loc[data['liv3'] == item_str, 'descr_liv3'].values[0])
    else:
        print(f"Item {item_str} not found in DataFrame")

recommendations_str = "\n".join(descriptions)
print(f"Top 5 recommendations for user {user_index}:\n{recommendations_str}")
```

5. Conclusions

5.1 Results

In comparing the two collaborative models, the user-user collaborative filter took too long to complete the test, so it was not tested in a large dataset like the one shown above.

The item-item collaborative filter, on the other hand, was quite fast, taking 6 minutes on a dataset of 40,800 lines. Unfortunately, the metrics reported very low values:

- Precision: 0.03
- Recall: 7.352941176470588e-05
- F1: 0.00014669926650366747

To improve the metrics, the recommendation system was tested with various normalizations/standardizations, but the most effective one was found to be min-max normalization with a range from 1 to 10.

5.2 Future Developments

Possible future developments include integrating hybrid filtering methods to further improve recommendation accuracy.