Applied Recommender Systems — A Report

Lakshit Verma

April 14, 2024

Contents

1 Introduction

This report is a short summary of the book **Applied Recommender Systems with Python**. It covers chapters 1, 3, 4 and 5 of the text respectively. In the initial chapter, we get a brief introduction to the world of recommendation systems and their types. In the subsequent chapters, we get a deeper dive into three methods of recommendation— Content-Based, Collaborative Filtering, and the methods used to achieve and improve on Collaborative Filtering.

2 Introduction to Recommendation Systems

recommendation systems are typically built for one purpose— to maximize revenue by enhancing the user's experience and maximize their time spent on the site. To build a recommendation system, the most crucial element is user feedback. This comes in two forms, **explicit** and **implicit**.

- **Explicit Feedback:** It is feedback the user knowingly and explicitly puts on a product. Examples of this include likes, dislikes, star ratings, and reviews.
- **Implicit Feedback:** This is the feedback that is generated unconsciously, through the revealed preferences of the user. These can include links clicked on, pages visited, video watch time, etc.

2.1 Types of recommendation engines

There are several methods of creating a recommendation engine. The ones given in the text are as follows:

1. **Market basket analysis (association rule mining)**
2. **Content-based filtering**
3. **Collaborative-based filtering**
4. **Hybrid systems**
5. **ML clustering**
6. **ML classification**
7. **Deep learning and NLP**

2.1.1 Market basket analysis

This is method most often used by retailers to predict the popularity of a given item. It works through identifying pairs of items that are often put together.

A few important terms are used in context of this system, and one of them is **Association rule**. Their purpose is to identify and symbolize strong relationships between items. They are written in the form `{antecedent→consequent}`. For example, take `{bread→jam}`, which means in plain English **"There is a strong relationship between customers who bought bread and jam in the same purchase"**.

Support is the relative frequency of an association rule displaying. **Confidence** measures the reliability of the rule, where a confidence of 0.5 indicates the items in question were purchased together 50% of the time. Finally, **Lift** is a measure of the ratio of the expected support vs. the support if two rules are independent. A lift value close to one means the rules are independent, and lift values over 1 indicate more and more correlation between the two rules.

2.1.2 Content-Based Filtering

Content-based filtering method is a recommendation algorithm that suggests items similar to the ones other users have previously selected or shown interest in.

Take the example of Netflix. The popular streaming site saves all user viewing information in a vector-based format, known as the **profile vector**, which contains information on past viewings, liked and disliked shows, most frequently watched genres, and so on. Then there is another vector that stores all the information regarding the titles (movies and shows) available on the platform, known as the **item vector**. It stores information like the title, actors, genre, language, length, crew info, synopsis, etc.

The content-based filtering algorithm uses the concept of cosine similarity. In it, you find the cosine of the angle between two vectors — the profile and item vectors in this case. Suppose A is the profile vector and B is the item vector, then the (cosine) similarity between them is calculated as follows:

$$sim(A, B) = \cos\theta = \frac{A \cdot B}{\|A\|\|B\|}$$

This outcome always ranges between -1 and 1, and is calculated for multiple item vectors, keeping the profile vector constant. They are then ranked in descending order of similarity, and have one of two following approaches applied for recommendations:

- **Top-N approach:** The top N movies are recommended, where N limits the number of titles recommended.
- **Rating scale approach** A limit on the similarity value is set, and all the titles satisfying that threshold are recommended.

Other methods such as **Euclidean Distance** ($\sqrt{(x_1 - y_1)^2 + \cdots + (x_N - y_N)^2}$) and **Pearson's correlation** are also utilized in select cases.

The critical flaw of such a system is the fact that all suggestions emerging from this end up falling into the same sort of product "category", making it feel formless and repetitive.

2.1.3 Collaborative-Based Filtering

In this, a user-user similarity is considered along with item similarities, to address the issues with simple Content-Based filtering.

The process of finding similarities is much the same as that of Content-Based filtering, but the engine then recommends titles that the user has not watched but other users with the same interests have. There are two kinds of Collaborative-Based filtering algorithms.

- **User-user collaborative filtering:** Here, you find user-user correlations and offer recommendations based on what similar users chose before. Despite its effectiveness, it is highly compute-intensive and its use in large-scale databases is therefore discouraged.
- **Item-item collaborative filtering:** This involves finding similarities in item-item pairs instead of user-user pairs. This is far less computationally demanding, at the cost of less accurate results.

2.1.4 Hybrid Systems

A hybrid system offers the best of both worlds— combining both content-based and collaborative-based systems, drawing power from another when one fails to produce desirable results. They can be implemented in the following ways:

- Generating recommendations separately by using content- and collaborative-based systems and merging them subsequently.
- Adding features of the collaborative method to a content-based recommender engine.
- Adding features of the content method to a collaborative-based recommender engine.

Studies consistently show that hybrid recommender engines generally perform better, faster and provide more reliable recommendations.

2.1.5 ML clustering

Applying the novel field of Machine Learning to the recommendation systems seems like the logical next step. ML methods are of two types: supervised and unsupervised. Clustering is the unsupervised method, meaning it finds patterns in data sans any human intervention in labelling the data. It is the process of grouping objects into clusters, and generally an object belonging to a cluster is more similar to the objects inside the cluser than the objects outside of it.

Clustering based methods are usually implemented when there is little user data to go by. If a user is found to be similiar to a cluster of users, the user is added to that cluster. Users inside the cluster all share specific tastes, and recommendations are provided according to that.

Some popularly used clustering algorithms are:

- **K-means clustering**
- **Fuzzy mapping**
- **Self-organizing maps (SOM)**
- **Hybrids of two or more techniques**

2.1.6 ML classification

In a classification based system, the algorithm uses features of both the items and users to predict a user's affinity towards a given product. One application of this is the buyer propensity model.

Some flaws of classification-based systems are:

- Collection of data is tedious.
- Its classification is challenging, and again, time-consuming.
- Training the models to function in real time is difficult.

2.1.7 Deep Learning

Deep Learning and Deep Neural Networks (DNNs) are a more powerful form of Machine Learning. They work especially well on unstructured data such as text, images, and video.
A few DL-based systems are:

- **Restricted Boltzmann**
- **Autoencoder based**
- **Neural Attention based**

2.2 Applications

Here, we look at real examples of the construction of recommendation systems using Python and sci-py.

2.2.1 Popularity

This is the simplest form of recommendation— to sort products based on a measure of popularity (views, downloads, likes, etc.)

We import the required libraries and data first.

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

df = pd.read_csv('data.csv',encoding= 'unicode_escape')
```

Cleaning up the data by dropping NULLs and values with no description, we finally sort the most popular items.

```
global_popularity=df_new.pivot_table(index=["StockCode","Description"],
values="Quantity", aggfunc="sum").sort_values(by="Quantity", ascending=False)
print("10 most popular items globally....")
global_popularity.head(10)
```

To calculate the most popular items by country, we use the following code:

```
# Popular items by country
countrywise=df_new.pivot_table(index=["Country","StockCode","Description"],
values="Quantity", aggfunc="sum").reset_index()

# Vizualize top 10 most popular items in UK
sns.barplot(y="Description", x="Quantity",
```

```
    data=countrywise[countrywise["Country"] == "United Kingdom"]
        .sort_values(by="Quantity", ascending=False).head(10))
plt.title("Top 10 Most Popular Items in UK", fontsize=14)
plt.ylabel("Item")
```

2.2.2 Buy Again

This is another simple system that sorts items by number of repeated instances, and recommends the ones at the top.

```
from collections import Counter

def buy_again(customerid):
# Fetching the items bought by the customer for provided customer id
items_bought = df_new[df_new["CustomerID"] == customerid].Description

# Count and sort the repeated purchases
bought_again = Counter(items_bought)

# Convert counter to list for printing recommendations
buy_again_list = list(bought_again)

# Printing the recommendations
print("Items you would like to buy again :")

return(buy_again_list)
```

Using the function on a specific user, say 1252 (function call `buy_again(1252)`), we will recieve a list of the most likely items the user is to buy, based on their previous purchase history.

3 Content-Based Recommender Systems

Content-Based filtering systems recommend products based on their similarity to products already having piqued the user's interest, indicated through buying, liking, reviewing, etc.

The steps to build a Content-Based recommendation system are:

1. **Data collection**
2. **Data preprocessing**
3. **Conversion of text to features**
4. **Performing similarity measures**
5. **Recommendation of products**

3.1 Data Collection

To begin training, a bit of boilerplate code and imports are needed

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics.pairwise import cosine_similarity,
    manhattan_distances,
    euclidean_distances
from sklearn.feature_extraction.text import TfidfVectorizer

from gensim import models

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.style
```

```
from gensim.models import FastText as ft
from IPython.display import Image

import re
import os
```

A few models have already been trained on this, which we'll use. They are listed below

- **Word2vec:** https://drive.google.com/uc?id=0B7XkCwpI5KDYNlNUTTlSS21pQmM
- **GloVe:** https://nlp.stanford.edu/data/glove.6B.zip
- **fastText:** https://dl.fbaipublicfiles.com/fasttext/vectors-crawl/cc.en.300.bin.gz

We then import the data: `Content_df = pd.read_csv("Rec_sys_content.csv")`
The columns of the table are as follows:

```
>>> Content_df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3958 entries, 0 to 3957
Data columns (total 6 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   StockCode     3958 non-null   object
 1   Product Name  3958 non-null   object
 2   Description   3958 non-null   object
 3   Category      3856 non-null   object
 4   Brand         3818 non-null   object
 5   Unit Price    3943 non-null   float64
```

Now we load the pre-trained models.

```
word2vecModel = models.KeyedVectors.load_word2vec_format(
    'GoogleNews-vectors-negative300.bin.gz', binary=True
)

fasttext_model=ft.load_fasttext_format("cc.en.300.bin.gz")

glove_df = pd
    .read_csv('glove.6B.300d.txt', sep=" ", quoting=3, header=None, index_col=0)

glove_model = {key: value.values for key, value in glove_df.T.items()}
```

We now need to preprocess the text before turning it into features. This has the following steps.

1. Remove duplicates
2. Convert the string to lowercase
3. Remove special characters

3.2 Features

Now, we convert the text into Features. This can be done through several methods.

- **One-hot Encoding (OHE):** This is a simple technique. It converts all the tokens in the list of unique words to columns. Then, on every input, it turns the column of the word to the value 1 if present, and 0 if not.
- **CountVectorizer:** This is the same as OHE, except it keeps track of the number of times a word occurs and stores that in the column.

- **TF-IDF:** It addresses the problems in CountVectorizer by keeping track of two values, TF (term frequency) — the number of times a token appears in a corpus doc divided by the number of tokens, and IDF (inverse document frequency), which is the log of the total number of such corpus docs. It helps provide more weightage to rare words in the corpus. Multiplying this gives us a value referred to as the TF-IDF vector of the corpus.
- **Word Embeddings:** TF-IDF doesn't help in capturing the context of a sentence. To fix this, we use word embeddings. They are of several types, which are listed below.
  - **Word2vec**
  - **GloVe**
  - **fastText**
  - **Elmo**
  - **SentenceBERT**
  - **GPT**

3.3 Similarity Features

After converting text to features, we then employ a number of techniques to find similarity between words. They are of typically of three types.

- **Euclidean Distance:** It is calculated by taking the sum of the squares of two vectors and finding their square root.
- **Cosine Similarity:** This is the cosine of the angles between two vectors. It is essentially their dot product divided by the product of their magnitudes.
- **Manhattan Distance:** It's the sum of the absolute differences between two vectors.

3.4 Implementation

We use CountVectorizer here as an example conversion model.

```
# Comparing similarity to get the top matches using count Vec
def get_recommendation_cv(product_id, df, similarity, n=10):
    row = df.loc[df['Product Name'] == product_id]
    index = list(row.index)[0]
    description = row['desc_lowered'].loc[index]
    #Create vector using Count Vectorizer
    count_vector = cnt_vec.fit_transform(desc_list)
    if similarity == "cosine":
        sim_matrix = cosine_similarity(count_vector)
        products = find_similarity(sim_matrix , index)
    elif similarity == "manhattan":
        sim_matrix = manhattan_distances(count_vector)
        products = find_manhattan_distance(sim_matrix , index)
    else:
        sim_matrix = euclidean_distances(count_vector)
        products = find_euclidean_distances(sim_matrix , index)
    return products
```

The input for this is as follows

- **product_id:** The product name for which we are finding similar recomendations for.
- **df:** The preprocessed data.
- **similarity:** The similarity method to be run.
- **n:** Number of recomendations.

Now, lets take a word embedding of choice, say TF-IDF.

```
# Comparing similarity to get the top matches using TF-IDF
def get_recommendation_tfidf(product_id, df, similarity, n=10):
    row = df.loc[df['Product Name'] == product_id]
    index = list(row.index)[0]
    description = row['desc_lowered'].loc[index]
    #Create vector using tfidf
    tfidf_matrix = tfidf_vec.fit_transform(desc_list)
    if similarity == "cosine":
        sim_matrix = cosine_similarity(tfidf_matrix)
        products = find_similarity(sim_matrix , index)
    elif similarity == "manhattan":
        sim_matrix = manhattan_distances(tfidf_matrix)
        products = find_manhattan_distance(sim_matrix , index)
    else:
        sim_matrix = euclidean_distances(tfidf_matrix)
        products = find_euclidean_distances(sim_matrix , index)
    return products
```

The output it gives us for `product_id = 'Vickerman 14" Finial Drop Christmas Ornaments, Pack of 2'` is as follows:


```
[{'value': 'storefront christmas LED Decoration Light Gold Color Star Shape Vine Wedding Party event',
  'score': 458.13},
 {'value': '8 1/2 x 14 Cardstock - Crystal Metallic (500 Qty.)',
  'score': 488.19},
 {'value': 'Cavalier Spaniel St. Patricks Day Shamrock Mouse Pad&#44; Hot Pad Or Trivet',
  'score': 497.0},
 {'value': "Call of the Wild Howling the Full Moon Women's Racerback Alpha Wolf",
  'score': 509.22},
 {'value': 'Fringe Table Skirt Purple 9 ft x 29 inches Pkg/1',
  'score': 516.08},
 {'value': 'Trend Enterprises T-83315 1.25 in. Holiday Pals & Peppermint Scratch N Sniff Stinky Stickers&#44; Large Round',
  'score': 522.0},
 {'value': "Allwitty 1039 - Women's T-Shirt Ipac Pistol Gun Apple Iphone Parody",
  'score': 525.03},
 {'value': 'Clear 18 Note Acrylic Box Musical Paperweight - Light My Fire',
  'score': 526.08},
 {'value': 'Handcrafted Ercolano Music Box Featuring "Luncheon of the Boating Party" by Renoir, Pierre Auguste - New YorkNew
York',
  'score': 527.88},
 {'value': 'Platinum 5 mm Comfort Fit Half Round Wedding Band - Size 9.5',
  'score': 528.08}]
```


4 Collaborative Filtering

Collaborative Filtering is a recomendation method in which the algorithm uses the customer's history and ratings to find customers similar in that regard, and then recommend them items that they liked. There are two types of collaborative filtering methods — user-to-user and item-to-item. There is also a more popularly used KNN based algorithm.

4.1 User-to-user Filtering

Here, it recommends items that a particular user might like by finding similar users, using purchase history or ratings on various items, and then suggesting the items liked by these similar users.

A matrix is formed to describe all the users, corresponding to all the items. Using this, we can calculate the similarity metrics, such as cosine similarity to calculate user-user relations.

4.1.1 Implementation

We create a matrix consisting of purchase history. It corresponds to all users and all the items available. Then we encode the data, with a 1 indicating the customer has bought the item, and 0 if he hasn't.

```
purchase_df = (data1.groupby(['CustomerID', 'StockCode'])['Quantity'].
    sum().unstack().reset_index().fillna(0).set_index('CustomerID'))
purchase_df.head()
```

We then apply cosine similarity to the matrix, and store the user similarity in a different matrix. Values close to zero in this mean the similarity between customers is minimal, while those closer to 1 mean it is strong. We can then implement a recommendation system through this.

```
user_similarity = user_similarity_data[user_similarity_data.index == uid]

other_users_similarities = user_similarity_data[user_similarity_data.index != uid]

similarities = cosine_similarity(user_similarity,other_users_similarities)[0].tolist()
user_indices = other_users_similarities.index.tolist()
index_similarity_pair = dict(zip(user_indices, similarities))

sorted_index_similarity_pair = sorted(index_similarity_pair.items(), reverse=True)
top_k_users_similarities = sorted_index_similarity_pair[:k]
similar_users = [u[0] for u in top_k_users_similarities]

print(f"The users with behaviour similar to that of user {uid} are:", similar_users)
```

4.2 Item-to-item Filtering

This filtering method finds similiarity between items the user bought instead of similarity between other users who bought the same items like in user-to-user filtering.

After creating a similarity matrix of items, we apply similar cosine similiarity methods to it.

```
items_purchase_df = (data1.groupby(['StockCode','CustomerID'])['Quantity'].
    sum().unstack().reset_index().fillna(0).set_index('StockCode'))

items_purchase_df = items_purchase_df.applymap(encode_units)
item_similarities = cosine_similarity(items_purchase_df)
item_similarity_data = pd.DataFrame(item_similarities,index=items_purchase_df
    .index,columns=items_purchase_df.index)
```

We then use similar code to give recommendations.

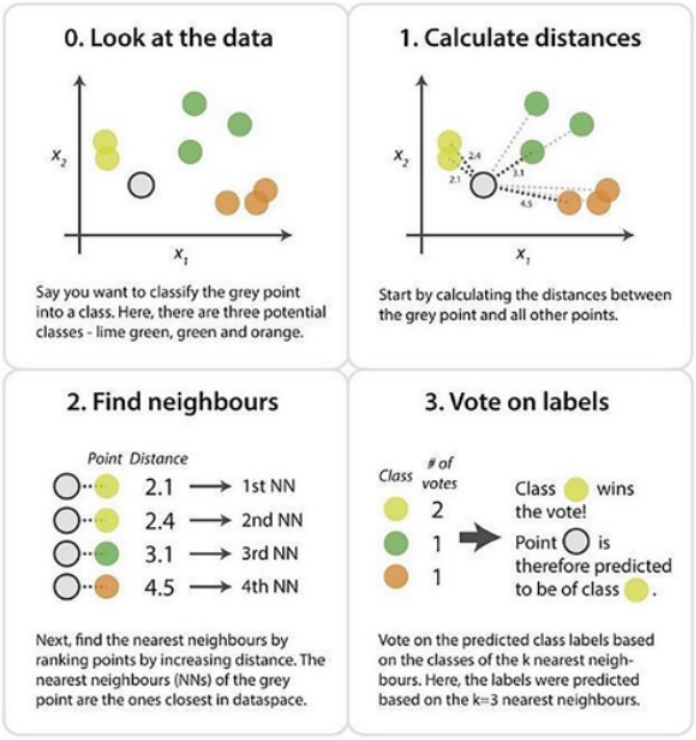
```
item_similarity = item_similarity_data[item_similarity_data.index == iid]

other_items_similarities = item_similarity_data[item_similarity_data.
index != iid]

similarities = cosine_similarity(item_similarity, other_items_
similarities)[0].tolist()

item_indices = other_items_similarities.index.tolist()

index_similarity_pair = dict(zip(item_indices, similarities))

sorted_index_similarity_pair = sorted(index_similarity_pair.items())

top_k_item_similarities = sorted_index_similarity_pair[:k]
similar_items = [u[0] for u in top_k_item_similarities]

print(similar_items)
```

This first separates the selected item from the other items and then takes a cosine similarity of the selected item with all other items to find the similarities.
For example, we get this as an output for a given input.

```
                        similar_items = fetch_similar_items('10002')
 ['10080', '10120', '10123C', '10124A', '10124G', '10125', '10133', '10135', '11001', '15030']
```

4.3 KNN

This approach uses principles of machine learning to make recommendations. The KNN method is a supervised learning model with low calculation times and easy implementation.



Since our matrix `purchase_df` is sparse, we need to convert it to a CSR matrix first, which divides a sparse matrix into three separate arrays.

- **values**
- **extent of rows**
- **index of columns**

Then we create the KNN model, using the Euclidean distance metric and fit it.

```
knn_model = NearestNeighbors(metric = 'euclidean', algorithm = 'brute')
knn_model.fit(purchase_matrix)
```

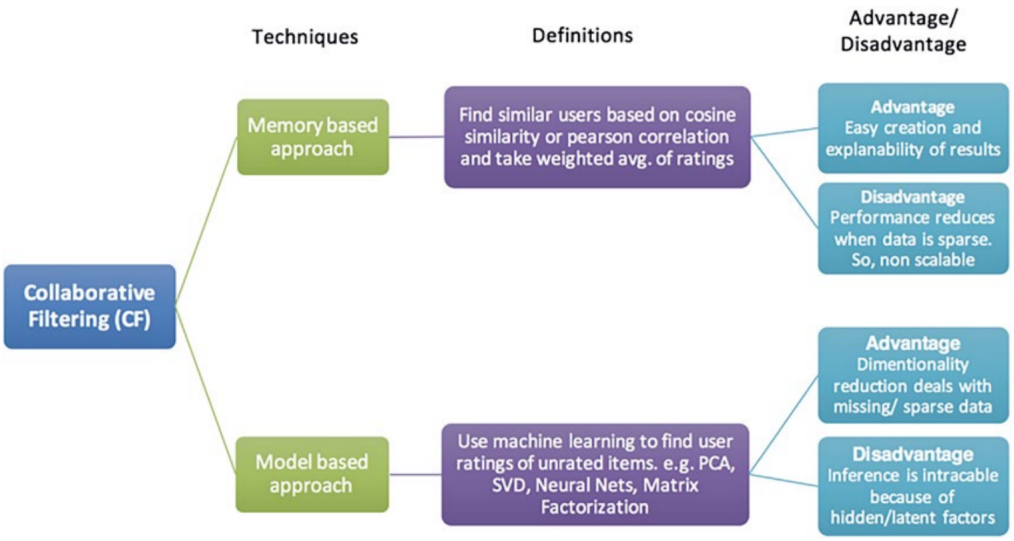
We can then write the necessary code to get similar users and their subsequent similar items.

```
similar_users_knn = []

distances, indices = knn_model.kneighbors(
    purchase_df.iloc[query_index,:].values.reshape(1, -1),
    n_neighbors = 5)

for i in range(0, len(distances.flatten())):
    if i == 0:
        print('Recommendations for {0}:\n'.format(purchase_
        df.index[query_index]))
    else:
        print('{0}: {1}, with distance of {2}:'.format(
            i,
            purchase_df.index[indices.flatten()[i]],
            distances.flatten()[i]))
        similar_users_knn.append(purchase_df.index[indices.flatten()[i]])
```

```
knn_recommnedations = []
for j in simular_users_knn:
    item_list = data1[data1["CustomerID"]==j]['StockCode'].to_list()
    knn_recommnedations.append(item_list)

flat_list = []

for sublist in knn_recommnedations:
    for item in sublist:
        flat_list.append(item)

final_recommendations_list = list(dict.fromkeys(flat_list))

ten_recs = random.sample(final_recommendations_list, 10)
print('Items bought by Similar users based on KNN', ten_recs)
```

5 Filtering using Matrix Factorizing, Singular Value Decomposition, and Co-Clustering

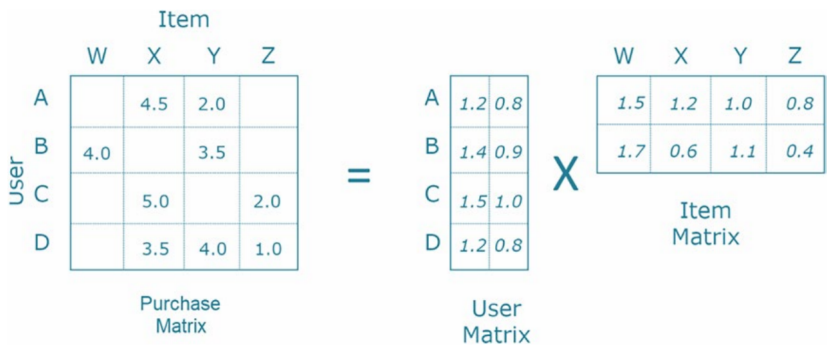
In this section we discuss more advanced methods of Collaborative Filtering. These methods are model-based approaches, while cosine similarity falls into the memory-based paradigm. The advantages and disadvantages of both are given below.



We now discuss the methods in detail and their implementations.

5.1 NMF

NMF stands for non-negative matrix factorization.



In it, hidden features (called embeddings) are generated from the user and item matrixes using matrix multiplication. This reduces the dimensionality of the full input matrix leading to space and compute savings. These embeddings are then used to fit an optimization problem (usually minimizing an error equation) to get to the predictions.

We train and fit the model as such.

```
algo1 = NMF()
algo1.fit(train_set)
pred1 = algo1.test(test_set)
```

After that, we can measure the performace of it using metrics such as **RMSE** (root-mean-squared error) and **MAE** (mean absolute error), where lower is better. We can also cross-validate them for complete confidence.

```
accuracy.rmse(pred1)
accuracy.mae(pred1)

RMSE: 428.3167
MAE:  272.6909

cross_validate(algo1, formated_data, verbose=True)
```

5.2 Co-Clustering

It is a data-mining technique that simultaneously clusters the columns and rows of a DataFrame/matrix. It differs from normal clustering, where each object is checked for similarity with other objects based on a single entity/type of comparison. This means you check for co-grouping of two different entities/types of comparison for each object simultaneously.

```
algo2 = CoClustering()
algo2.fit(train_set)
pred2 = algo2.test(test_set)

accuracy.rmse(pred2)
accuracy.mae(pred2)

RMSE: 6.7877
MAE:  5.8950
```

Using cross-validation, we can see the RMSE and MAE values are indeed low.

```
cross_validate(algo2, formated_data, verbose=True)

Evaluating RMSE, MAE of algorithm CoClustering on 5 split(s).

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    6.8485  6.6710  34.0950 11.0666 11.4735 14.0309 10.2338
MAE (testset)     5.6185  5.0401  7.0667  7.0208  5.9296  6.1352  0.7950
Fit time          0.19    0.17    0.18    0.20    0.18    0.18    0.01
Test time         0.01    0.01    0.02    0.01    0.01    0.01    0.00
```

5.3 Singular Value Decomposition (SVD)

SVD is a linear algebra concept generally used to reduce matrix dimensions. It is also a type of matrix factorization. An error equation is similarly minimized to get to the prediction.

```
algo3 = SVD()
algo3.fit(train_set)
pred3 = algo3.test(test_set)

accuracy.rmse(pred3)
accuracy.mae(pred3)

RMSE: 4827.6830
MAE:  4815.8341
```

This seems to perform the worst of all 3 methods we tested. This is verified with cross-validation. Performing a real-world test on it gives us similar results.

```
data1[(data1['StockCode']=='47590B')&(data1['CustomerID']==15738)].Quantity.sum()
algo2.test([['47590B',15738,78]])
```

```
[
    Prediction(uid='47590B', iid=15738, r_ui=78,
               est=133.01087456331527, details={'was_impossible': False})
]
```

5.4 Implementation of Co-Clustering

Now, we get the predictions using Co-Clustering.

```
predict_data = pd.DataFrame(pred2,
    columns = ['iid', 'cust_id', 'quantity', 'prediction', 'details'])
```

Add important information such as the number of item orders and customer orders for each record.

```
def get_item_orders(user_id):
    try:
        # for an item, return the no. of orders made
        return len(train_set.ur[train_set.to_inner_uid(user_id)])
    except ValueError:
        # user not present in training
        return 0


def get_customer_orders(iid):
    try:
        # for an customer, return the no. of orders made
        return len(train_set.ir[train_set.to_inner_iid(iid)])
    except ValueError:
        # item not present in training
        return 0


predict_data['item_orders'] = predict_data.iid.apply(get_item_orders)
predict_data['customer_orders'] = predict_data.cust_id.apply(get_customer_orders)
```

We calculate the error to get the worst and best predictions.

```
predict_data['error'] = abs(predict_data.prediction - predict_data.quantity)
```

	item_id	customer_id	quantity	prediction	details	item_orders	customer_orders	error
0	85014B	17228	130.0	119.183290	{'was_impossible': False}	459	31	10.816710
1	84406B	16520	156.0	161.858671	{'was_impossible': False}	459	29	5.858671
2	47590B	17365	353.0	352.777318	{'was_impossible': False}	457	32	0.222682
3	85049G	16755	170.0	159.540375	{'was_impossible': False}	450	32	10.459625
4	16156S	14895	367.0	368.129814	{'was_impossible': False}	440	30	1.129814
...	...	...	...	...	...	...	...	...
4539	47590B	15764	180.0	179.777318	{'was_impossible': False}	457	30	0.222682
4540	84970L	16222	137.0	144.853747	{'was_impossible': False}	458	34	7.853747
4541	84596F	16340	153.0	154.254839	{'was_impossible': False}	453	29	1.254839
4542	85099B	17511	745.0	748.576631	{'was_impossible': False}	447	32	3.576631
4543	85049E	16265	194.0	190.137590	{'was_impossible': False}	458	29	3.862410

```
best_predictions = predict_data.sort_values(by='error')[:10]
```

	item_id	customer_id	quantity	prediction	details	item_orders	customer_orders	error
334	16156S	17841	5095.0	5095.000000	{'was_impossible': False}	440	32	0.000000
3973	47590B	13230	457.0	456.777318	{'was_impossible': False}	457	29	0.222682
697	47590B	12415	601.0	600.777318	{'was_impossible': False}	457	30	0.222682
2339	47590B	13869	307.0	306.777318	{'was_impossible': False}	457	34	0.222682
1572	47590B	13078	276.0	275.777318	{'was_impossible': False}	457	32	0.222682
1608	47590B	17428	299.0	298.777318	{'was_impossible': False}	457	35	0.222682
1160	47590B	17799	343.0	342.777318	{'was_impossible': False}	457	31	0.222682
574	47590B	17337	543.0	542.777318	{'was_impossible': False}	457	29	0.222682
4000	47590B	14527	694.0	693.777318	{'was_impossible': False}	457	35	0.222682
516	47590B	14701	238.0	237.777318	{'was_impossible': False}	457	31	0.222682

```
worst_predictions = predict_data.sort_values(by='error')[-10:]
```

	item_id	customer_id	quantity	prediction	details	item_orders	customer_orders	error
4003	47599A	14286	141.0	125.720820	{'was_impossible': False}	471	34	15.279180
2939	47599A	15696	122.0	106.720820	{'was_impossible': False}	471	28	15.279180
2933	47599A	16393	214.0	198.720820	{'was_impossible': False}	471	32	15.279180
538	47599A	12662	157.0	141.720820	{'was_impossible': False}	471	32	15.279180
537	47599A	14040	178.0	162.720820	{'was_impossible': False}	471	31	15.279180
2180	47599A	14808	208.0	192.720820	{'was_impossible': False}	471	31	15.279180
1585	47599A	13555	136.0	120.720820	{'was_impossible': False}	471	30	15.279180
3252	47599A	14911	3648.0	3632.720820	{'was_impossible': False}	471	34	15.279180
1651	47599A	13089	1511.0	1495.720820	{'was_impossible': False}	471	31	15.279180
3033	47599A	12949	179.0	163.009478	{'was_impossible': False}	471	31	15.990522

Now, lets use the example of customer 12347. We get the customers who bought the same items as them

```
item_list = predict_data[predict_data['cust_id']==12347]['iid'].values.tolist()
print(item_list)

['82494L', '84970S', '47599A', '84997B', '85123A', '84997C', '85049A']

customer_list = predict_data[predict_data['iid'].isin(item_list)]['cust_id'].values
customer_list = np.unique(customer_list).tolist()
print(customer_list)

[12347,
 12362,
 12370,
 12378,
 ...,
 12415,
 12417,
 12428]
```

After filtering, we then finally get the top items to recommend to the user.

```
filtered_data = predictions_data[predictions_data['cust_id'].isin(customer_list)]
filtered_data = filtered_data[~filtered_data['iid'].isin(item_list)]

recommended_items = filtered_data
    .sort_values('prediction',ascending = False)
    .reset_index(drop = True)
    .head(10)['iid']
    .values.tolist()

printrecommended_items)
```

We then finally get this as output.

```
['16156S',
 '85049E',
 '47504K',
 '85099C',
 '85049G',
 '85014B',
 '72351B',
 '84536A',
 '48173C',
 '47590A']
```

The recommended list of items for **user 12347** is thus achieved.