In [ ]:

```
###load packages
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
from sklearn.metrics.pairwise import cosine_similarity
from surprise import Reader, Dataset, KNNBasic, NormalPredictor,BaselineOnly,KNNWithMeans,
from surprise import SVD, SVDpp, NMF, SlopeOne, CoClustering
from surprise.model_selection import cross_validate
from surprise.model_selection import GridSearchCV
from surprise import accuracy
import random
```

In [ ]:

```
### load datasets
purchase = pd.read_excel('UWL_Purchase Data - Expanded2.xlsx')
customer = pd.read_excel('UWL_Customer Data - Expanded.xlsx')
### store data removed
```

In [ ]:

```
### explore purchase data
display(purchase.head())
purchase.dtypes
np.where(pd.isnull(purchase))
```

In [ ]:

```
#change column names for clarity and ease of coding
purchase.columns = ['Customer_ID','Date','Day','TimeOfDay','DaySegment','TimeSegment','Sto
                    'PrdCatSubCat','UPCDescription','UPC','QtySold','Price']
```

In [ ]:

```
### explore customer data
display(customer.head())
customer.dtypes
```

In [ ]:

```
### exploration of store data removed
```

In [ ]:

```
num_customers = len(pd.unique(purchase['Customer_ID'])) ### identify number of unique cust
num_prod_purchases = len(purchase) ### of products purchased
num_products = len(pd.unique(purchase['UPCDescription'])) ### unique products in dataset
print("There are ", num_customers, "unique customers, among ", num_prod_purchases, "items
print("There are ", num_products, "products represented in the dataset.")
purchase['UniquePurchases'] = str(purchase['Customer_ID']) + purchase['Date'] + purchase['
num_purchases = len(pd.unique(purchase['UniquePurchases'])) ### number of unique transact
print('There are ',num_purchases,'unique purchases.')
```

In [ ]:

```
### remove fuel from dataset
purchase.drop(purchase.index[purchase["CatManDepartment"] == "Fuel"],inplace = True)
```

In [ ]:

```
### replace brand name prefixes
### code removed for anonymization purposes
```

In [ ]:

```
### change quantity sold of all items to 1, just want to know if they bought the item or n
purchase['QtySold'] = 1
```

In [ ]:
```
### identify number of unique transactions per customer
unique_purchases = purchase[['Customer_ID','UniquePurchases']]
x = unique_purchases.groupby('Customer_ID',as_index=False).nunique()
x
```

In [ ]:
```
### sum quantity of items purchased per unique product
item_purchases = purchase[['Customer_ID','UPCDescription','QtySold']]
y = item_purchases.groupby(['Customer_ID','UPCDescription'],as_index=False).sum()
y
```

In [ ]:
```
### create full table
full_table = pd.merge(x,y,on = "Customer_ID",how='inner')
full_table['rating'] = full_table['QtySold']/full_table['UniquePurchases'] ### create imp
full_table
```

In [ ]:
```
### get min and max ratings per customer
rating_transform = full_table[['Customer_ID','rating']]
rating_max = rating_transform.groupby('Customer_ID',as_index=False).max()
rating_max['rating'] = rating_max['rating'] + 0.01
rating_min = rating_transform.groupby('Customer_ID',as_index=False).min()
rating_max.rename(columns = {'rating':'rating_max'},inplace=True)
rating_min.rename(columns = {'rating':'rating_min'},inplace=True)
```

In [ ]:
```
###merge tables
full_table2 = pd.merge(full_table,rating_max, on = 'Customer_ID',how='inner')
full_table3 = pd.merge(full_table2,rating_min, on = 'Customer_ID',how='inner')
full_table3
```

In [ ]:
```
### get scaled rating
full_table3['scaled_rating'] = (10-1) * (full_table3['rating'] - full_table3['rating_min']
full_table3
```

In [ ]:
```
#minimum number of times an item has been purchased
min_item = 10
#minimum number of times a customer has purchased
min_purch = 10

### identify items that have been purchased more than the min
item_count = full_table3[["UPCDescription","Customer_ID"]].groupby("UPCDescription").count
item_count = item_count[item_count["Customer_ID"] >= min_item]

### identify customers that have purchased more than the min
customer_count = full_table3[["UPCDescription","Customer_ID"]].groupby("Customer_ID").cour
customer_count = customer_count[customer_count["UPCDescription"] >= min_purch]

### filter table on above criteria
full_table3 = full_table3[full_table3["Customer_ID"].isin(customer_count.index) & full_tak
```

In [ ]:
```
###preview table
full_table3
```

In [ ]:
```
### create final ratings table and explore
ratings = full_table3[['Customer_ID','UPCDescription','scaled_rating']]
```

```
ratings.dtypes
ratings
```

In [ ]:

```
### set up training data
X = ratings.copy()
y = ratings["Customer_ID"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25, stratify=y, ra
```

In [ ]:

```
### find median
print(f"The median of this rating range is {np.median(np.arange(np.min(ratings['scaled_rat

#define a baseline model to always return the median
def baseline(Customer_ID, UPC, scale_median, *args):
    return scale_median
```

In [ ]:

```
### defining score function
def score(cf_model, X_test, *args):
    #Construct a list of item-customer tuples from the testing dataset
    id_upc_pairs = zip(X_test[X_test.columns[0]], X_test[X_test.columns[1]])

    #Predict the rating for every customer-item tuple
    y_pred = np.array([cf_model(customer, item, *args) for (customer, item) in id_upc_pair

    #Extract the actual ratings given by the users in the test data
    y_true = np.array(X_test[X_test.columns[2]])

    #Return the final RMSE score
    return mean_squared_error(y_true, y_pred, squared=False)
```

In [ ]:

```
### define function for getting recommendations from base models
def base_recommendations(Customer_id, N, cf_model, X_test, *args):
  #Construct a list of user-item tuples from the dataset
    id_upc_pairs = zip(X_test[X_test.columns[0]], X_test[X_test.columns[1]])

    #Predict the rating for every customer-item tuple
    y_pred = np.array([cf_model(customer, item, *args) for (customer, item) in id_upc_pair

    Customer_ID = np.array(X_test[X_test.columns[0]])
    UPCDescription = np.array(X_test[X_test.columns[1]])

    ### join predictions with Customer_ID and UPC_Descriptionss
    recommendations = pd.DataFrame({'Customer_ID':Customer_ID,'UPCDescription':list(UPCDes
    #sort all predictions
    recommendations = recommendations.sort_values('Prediction', ascending=False)
    #remove all but brand name products
    recommendations = recommendations.loc[(recommendations.UPCDescription.str.startswith('
    # filter on Customer_ID
    recommendations = recommendations.loc[recommendations['Customer_ID'] == Customer_id]
    #Return the final recommendations
    return recommendations.head(N)
```

In [ ]:

```
### score basline model
baseline_rmse = score(baseline,X_test,5.5)
baseline_rmse
```

In [ ]:

```
### get recommendations from baseline mode
base_recommendations(2010, 10, baseline, X_test, 5.5)
```

In [ ]:
```
### set up ratings pivot
r_matrix = X_train.pivot_table(values='scaled_rating', index='Customer_ID', columns='UPCDe
```

In [ ]:
```
def mean_model(Customer_ID, UPCDescription, ratings_matrix, scale_median):
    #Check if UPCDescription exists in ratings_matrix
    if UPCDescription in ratings_matrix:
        #Compute the mean of all the ratings given to the item
        mean_rating = ratings_matrix[UPCDescription].mean()

    else:
        #Default to scale median
        mean_rating = scale_median

    return mean_rating
```

In [ ]:
```
### score mean model
mean_rmse = score(mean_model, X_test, r_matrix, 5.5)
mean_rmse
```

In [ ]:
```
base_recommendations(2010, 10, mean_model, X_train, r_matrix, 5.5)
```

In [ ]:
```
### creat item based based pivot
r_matrix_item = X_train.pivot(values='scaled_rating', index='UPCDescription', columns='Cus

#Create a dummy ratings matrix with all null values imputed to 0
r_matrix_item_dummy = r_matrix_item.copy().fillna(0)

#Compute the cosine similarity
cosine_sim_item = cosine_similarity(r_matrix_item_dummy, r_matrix_item_dummy)

#Convert to pandas dataframe
cosine_sim_item = pd.DataFrame(cosine_sim_item, index=r_matrix_item.index, columns=r_matri
```

In [ ]:
```
### preview similarity matrix
cosine_sim_item.head(10)
```

In [ ]:
```
#Item-Based Collaborative Filter using Weighted Mean Ratings
def wmean(Customer_ID, UPCDescription, ratings_matrix, c_sim_matrix, median_rating):

    #Check if Customer_ID exists in r_matrix
    if Customer_ID in ratings_matrix:

        #Get the similarity scores for the item in question with every other item
        sim_scores = c_sim_matrix[UPCDescription]

        #Get the user ratings for the item in question
        u_ratings = ratings_matrix[Customer_ID]

        #Extract the indices containing NaN in the m_ratings series
        idx = u_ratings[u_ratings.isnull()].index

        #Drop the NaN values from the Series
```

```
        u_ratings = u_ratings.dropna()

        #Drop the corresponding cosine scores from the sim_scores series
        sim_scores = sim_scores.drop(idx)

        #Compute the final weighted mean
        if sim_scores.sum() > 0:
            wmean_rating = np.dot(sim_scores, u_ratings)/ sim_scores.sum()
        else: # the book has zero cosine similarity with other items
            wmean_rating = median_rating

    else:
        #Default to a rating of 5.5 in the absence of any information
        wmean_rating = median_rating

    return wmean_rating
```

In [ ]:

```
### scored weighted mean model
wmean_rmse = score(wmean, X_test, r_matrix_item, cosine_sim_item, 5.5)
wmean_rmse
```

In [ ]:

```
### get recommendations from weighted mean model
base_recommendations(2010, 10, wmean, X_train, r_matrix_item, cosine_sim_item, 5.5)
```

In [ ]:

```
#examine distribution of ratings
ratings.scaled_rating.plot(kind='hist', bins=4, title='Actual Ratings')
```

In [ ]:

```
### Creating Normal Predictor Model
our_seed = 14

#Define a Reader object
reader = Reader(rating_scale=(1,11))

#Create the dataset
data = Dataset.load_from_df(ratings, reader)

#Define the normal predictor
normal_pred = NormalPredictor()

## apply the seeds before cross validating
random.seed(our_seed)
np.random.seed(our_seed)

#Evaluate RMSE
algo_cv = cross_validate(normal_pred, data, measures=['RMSE'], cv=5, verbose=True)
print(algo_cv)

#Extract average RMSE
algo_rmse = np.mean(algo_cv['test_rmse'])
print(f'\nThe RMSE across five folds was {algo_rmse}')
```

In [ ]:

```
#train on the whole dataset
trainset = data.build_full_trainset()
normal_pred.fit(trainset)
```

In [ ]:

```
## apply the seeds before predicting
random.seed(our_seed)
```

```
np.random.seed(our_seed)
#run predictions
pred_df = ratings.copy() #make a copy of the ratings that we can add columns to

#get all the predictions
pred_df['prediction'] = pred_df.apply(lambda x: normal_pred.predict(x['UPCDescription'], x

pred_df
```

In [ ]:

```
#### Creating KNN Model
our_seed = 14

#Define a Reader object
reader = Reader(rating_scale=(1,11)) # defaults to (0,5)

#Create the dataset
data = Dataset.load_from_df(ratings, reader)

sim_options = {'user_based':False}

#Define the algorithm object
knn = KNNBasic(k=3, verbose=False,sim_options = sim_options)

## apply the seeds before cross validating
random.seed(our_seed)
np.random.seed(our_seed)
#Evaluate Model
knn_cv = cross_validate(knn, data, measures=['RMSE'], cv=5, verbose=True)
print(knn_cv)

#Extract average RMSE
knn_rmse = np.mean(knn_cv['test_rmse'])
print(f'\nThe RMSE across five folds was {knn_rmse}')
```

In [ ]:

```
#Define a Reader object
reader = Reader(rating_scale=(1,11))

#Create the dataset
data = Dataset.load_from_df(ratings, reader)

#get the raw ratings
raw_ratings = data.raw_ratings

# shuffle ratings
random.seed(our_seed)
np.random.seed(our_seed)
random.shuffle(raw_ratings)

#A = 90% of the data, B = 10% of the data
threshold = int(.9 * len(raw_ratings))
A_raw_ratings = raw_ratings[:threshold]
B_raw_ratings = raw_ratings[threshold:]

data.raw_ratings = A_raw_ratings  # data is now the set A

# Select best algo with grid search.
print('Grid Search...')
param_grid = {'k': [3,5], 'min_k': [1,3]}
grid_search = GridSearchCV(KNNBasic, param_grid, measures=['rmse'], cv=3)
grid_search.fit(data)

knn_gs_algo = grid_search.best_estimator['rmse']
```

```
# retrain on the whole set A
trainset = data.build_full_trainset()
knn_gs_algo.fit(trainset)

# Compute biased accuracy on A
predictions = knn_gs_algo.test(trainset.build_testset())
print(f'Biased accuracy on A = {accuracy.rmse(predictions)}')

# Compute unbiased accuracy on B
testset = data.construct_testset(B_raw_ratings)  # testset is now the set B
predictions = knn_gs_algo.test(testset)
print(f'Unbiased accuracy on B = {accuracy.rmse(predictions)}')
```

In [ ]:

```
grid_search.best_params['rmse']
```

In [ ]:

```
#set seeds
random.seed(our_seed)
np.random.seed(our_seed)

#reset the data.raw_ratings to 100% of the data
data.raw_ratings = raw_ratings

#build a trainset
trainset = data.build_full_trainset()

#build the algorithm with best parameters
knn_gs_algo = grid_search.best_estimator['rmse']

#fit to the data
knn_gs_algo.fit(trainset)
```

In [ ]:

```
#Define the SVD model
svd = SVD()
## apply the seeds before cross validating
random.seed(our_seed)
np.random.seed(our_seed)
#Evaluate RMSE
svd_cv = cross_validate(svd, data, measures=['RMSE'], cv=5, verbose=True)
#Extract average RMSE
svd_rmse = np.mean(svd_cv['test_rmse'])
print(f'\nThe RMSE across five folds was {svd_rmse}')

#train on the whole dataset
trainset = data.build_full_trainset()
svd.fit(trainset)
```

In [ ]:

```
#build the recommendations function
def recommendations(ratings, Customer_ID, algo, N):
    #create dataframe of unique items
    sim_items = ratings.copy().drop(columns=['Customer_ID', 'scaled_rating']).drop_duplica
    #generate the predicted this customer's predicted rating for each item based on filte
    sim_items['Prediction'] = sim_items.apply(lambda x: algo.predict(Customer_ID, x['UPCDe
    #add back Customer_ID
    sim_items.insert(0, 'Customer_ID', Customer_ID)
    #sort all predictions
    sim_items = sim_items.sort_values('Prediction', ascending=False)
    #remove all but brand name products
    sim_items = sim_items.loc[(sim_items.UPCDescription.str.startswith('BN'))]
    return sim_items.head(N)
```

In [ ]:

```
### get recommendations from KNN model
recommendations(ratings, 2010, knn_gs_algo, 10)
```

In [ ]:

```
### get recommendations from SVD Model
recommendations(ratings, 2010, svd, 10)
```

In [ ]:

```
### get recommendations from normal predictor
recommendations(ratings, 2010, normal_pred, 10)
```

In [ ]:

```
### print all RMSE values
print("Baseline RMSE:",baseline_rmse)
print("Mean RMSE:",mean_rmse)
print("Weighted Mean RMSE:",wmean_rmse)
print("Normal Predictor RMSE",algo_rmse)
print("KNN RMSE:",knn_rmse)
print("SVD RMSE:",svd_rmse)
```