notebook

June 24, 2021

1 Using Recommender Systems to Identify Top Beauty Products

Student name: Jonathan Lee

Student pace: Full Time

Scheduled project review date/time: June 22, 2pm

Instructor name: James Irving

Blog post URL:

1.1 Overview

This project uses the Surprise package from scikit with Amazon review data of Luxury Beauty products to build a recommendation system. For this analysis, we will examine the performance of memory-based collaborative filtering in the form of K-Nearest Neighbors, as well as of model-based collaborative filtering in the form of Singular Value Decomposition. From our test results, we find that out of KNN methods, Singular Value Decomposition, and Alternating Least Squares methods, Singular Value Decomposition was the best performing model for our selected data. We also examine what the optimal hyperparameters are for this particular dataset.

1.2 Business Problem

Our client is a beauty product retailer that wants to know what the most popular products on Amazon are, as well as what other products customers would be likely to give high ratings to, under the assumption that they would give high ratings to these popular products. We want to optimize a recommender system based on Amazon reviews that as accurately as possible predicts other products that customers would be likely to enjoy. Using this optimized recommender system, we will move forward with the goal of using our client's customer preferences to extract insights into what other brands/products would be successful if our client were to add them to their product offering. ***Questions to address:*** **What is are the optimal model and hyperparameters to build a recommender system to work with Amazon ratings dataset to provide recommendations for our own customers? * What are Amazon's most popular products in terms of number of ratings? * Assuming that our client's customers currently give high ratings to the popular products on Amazon, what other products can we recommend adding to inventory?** *

1.3 Data Understanding and Preparation

In this analysis, we use Amazon review data and product metadata featured in the following paper:

Justifying recommendations using distantly-labeled reviews and fined-grained aspects

Jianmo Ni, Jiacheng Li, Julian McAuley

Empirical Methods in Natural Language Processing (EMNLP), 2019

Due to the large size of the complete dataset and hardware limitations, we will complete the analysis with only reviews and metadata from the luxury beauty product category.

Let's begin by loading in our data and doing some Exploratory Data Analysis.

[1]:
```
# Import standard packages
import sys
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline

# Set random seed
np.random.seed(27)
```

[2]:
```
# Set theme and style for plots
sns.set_theme('talk')
sns.set_style('darkgrid')
```

1.3.1 Loading in the Data

We have two tables to work with in this analysis: 1. Review data: contains product ASIN code, user code, and the rating that user provided. 2. Product metadata which includes all product metadata including price, product name, and product images paired with ASIN codes.

[3]:
```
# Load review dataset and metadata
review_df = pd.read_csv('data/Luxury_Beauty.csv', names=['asin', 'user', \
                                                         'rating', 'timestamp'])
meta_df = pd.read_json('data/meta_Luxury_Beauty.json.gz', lines=True)
display(review_df, meta_df)
```

	asin	user	rating	timestamp
0	B00004U9V2	A1Q6MUU0B2ZDQG	2.0	1276560000
1	B00004U9V2	A3HO2SQDCZIE9S	5.0	1262822400
2	B00004U9V2	A2EM03F99X3RJZ	5.0	1524009600
3	B00004U9V2	A3Z74TDRGD0HU	5.0	1524009600
4	B00004U9V2	A2UXFNW9RTL4VM	5.0	1523923200
…	…	…	…	…
574623	B01HIQEOLO	AHYJ78MVF4UQO	5.0	1489968000
574624	B01HIQEOLO	A1L2RT7KBNK02K	5.0	1477440000
574625	B01HIQEOLO	A36MLXQX9WPPW9	5.0	1475193600
574626	B01HJ2UY0W	A23DRCOMC2RIXF	1.0	1480896000

```
574627  B01HJ2UY1G   AJEDVHTLS9P3V     5.0  1484352000

[574628 rows x 4 columns]
      category tech1                                        description fit  \
0            []        [After a long day of handling thorny situation…
1            []        [If you haven't experienced the pleasures of b…
2            []        [Rich, black mineral mud, harvested from the b…
3            []        [This liquid soap with convenient pump dispens…
4            []        [Remember why you love your favorite blanket? …
…         …   …                                                 …   ..
12294        []        [, CND Craft Culture Collection: Patina Buckle…
12295        []        [CND Shellac was designed to be used as a syst…
12296        []        [CND Shellac was designed to be used as a syst…
12297        []        [The I AM JUICY COUTURE girl is once again tak…
12298        []        [I Love Juicy Couture Eau De Parfum Spray 3.4 …

                                                   title  \
0      Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
1                                        AHAVA Bath Salts
2          AHAVA Dead Sea Mineral Mud, 8.5 oz, Pack of 4
3      Crabtree &amp; Evelyn Hand Soap, Gardeners, 10…
4                                      Soy Milk Hand Crme
…                                                    …
12294              CND Shellac Power Polish, Patina Buckle
12295                 CND Shellac power polish denim patch
12296                         CND Shellac, Leather Satchel
12297  Juicy Couture I Love Juicy Couture, 1.7 fl. Oz…
12298  Juicy Couture I Love Juicy Couture, 3.4 fl. Oz…

                                                also_buy tech2 brand feature  \
0      [B00GHX7H0A, B00FRERO7G, B00R68QXCS, B000Z65AZ…                 []
1                                                     []                 []
2                                                     []                 []
3                                                     []                 []
4      [B000NZT6KM, B001BY229Q, B008J724QY, B0009YGKJ…                 []
…                                                    …   …    …      …
12294  [B003ONLAXQ, B00YDEZ9T6, B074KHRD13, B00R3PZK1…                 []
12295  [B003ONLAXQ, B003OHOKBA, B004LEMWGG, B01MT91G4…                 []
12296  [B003ONLAXQ, B003OHOKBA, B004LEMWGG, B01MT91G4…                 []
12297                                                 []                 []
12298                                       [B071NZZW3K]                 []

                                          rank  \
0              4,324 in Beauty & Personal Care (
1          1,633,549 in Beauty & Personal Care (
2      1,806,710 in Beauty &amp; Personal Care (
3                                           []
```

```
4         42,464 in Beauty &amp; Personal Care (
…                                          …
12294         88,740 in Beauty & Personal Care (
12295        122,331 in Beauty & Personal Care (
12296        168,028 in Beauty & Personal Care (
12297        490,755 in Beauty & Personal Care (
12298        181,383 in Beauty & Personal Care (

                                               also_view  \
0      [B00FRER07G, B00GHX7H0A, B07GFHJRMX, B00TJ3NBN…
1                                                     []
2                                                     []
3      [B00004U9V2, B00GHX7H0A, B00FRER07G, B00R68QXC…
4                                                     []
…                                                    …
12294  [B00D2VMUA2, B074KJZJYW, B074KHRD13, B073SB9JW…
12295  [B00D2VMUA2, B01L0EV8X2, B004LEMWGG, B00EFGDYZ…
12296  [B00D2VMUA2, B01L0EV8X2, B004LEMWGG, B00EFGDYZ…
12297  [B0757439SY, B01HJ2UY1G, B01KX3TK7C, B01LX71LJ…
12298  [B0757439SY, B01LX71LJV, B01HJ2UY0W, B07GBSC3L…

                                                 details       main_cat  \
0      {'
    Product Dimensions:
    ': '2.2 x 2.2 …  Luxury Beauty
1      {'
    Product Dimensions:
    ': '3 x 3.5 x …  Luxury Beauty
2      {'
    Product Dimensions:
    ': '5.1 x 3 x …  Luxury Beauty
3      {'
    Product Dimensions:
    ': '2.6 x 2.6 …  Luxury Beauty
4      {'
    Product Dimensions:
    ': '7.2 x 2.2 …  Luxury Beauty
…                                                    …             …
12294  {'
    Item Weight:
    ': '0.48 ounces', 'Sh…  Luxury Beauty
12295  {'Shipping Weight:': '1.4 ounces (', 'ASIN:': …  Luxury Beauty
12296  {'Shipping Weight:': '1.4 ounces (', 'Domestic…  Luxury Beauty
12297  {'
    Product Dimensions:
    ': '3.3 x 2.7 …  Luxury Beauty
12298  {'
    Product Dimensions:
```

```
    ': '3.3 x 3.2 …  Luxury Beauty

      similar_item date   price        asin  \
0                   NaT  $30.00  B00004U9V2
1                   NaT          B0000531EN
2                   NaT          B0000532JH
3                   NaT  $15.99  B00005A77F
4                   NaT  $18.00  B00005NDTD
…                …   …     …          …
12294               NaT  $15.95  B01HIQIEYC
12295               NaT  $15.95  B01HIQHQU0
12296               NaT  $15.95  B01HIQEOLO
12297               NaT  $76.00  B01HJ2UY0W
12298               NaT  $96.00  B01HJ2UY1G

                                                imageURL  \
0      [https://images-na.ssl-images-amazon.com/image…
1                                                     []
2      [https://images-na.ssl-images-amazon.com/image…
3      [https://images-na.ssl-images-amazon.com/image…
4      [https://images-na.ssl-images-amazon.com/image…
…                                                    …
12294                                                 []
12295                                                 []
12296  [https://images-na.ssl-images-amazon.com/image…
12297  [https://images-na.ssl-images-amazon.com/image…
12298  [https://images-na.ssl-images-amazon.com/image…

                                          imageURLHighRes
0      [https://images-na.ssl-images-amazon.com/image…
1                                                     []
2      [https://images-na.ssl-images-amazon.com/image…
3      [https://images-na.ssl-images-amazon.com/image…
4      [https://images-na.ssl-images-amazon.com/image…
…                                                    …
12294                                                 []
12295                                                 []
12296  [https://images-na.ssl-images-amazon.com/image…
12297  [https://images-na.ssl-images-amazon.com/image…
12298  [https://images-na.ssl-images-amazon.com/image…

[12299 rows x 19 columns]
```

1.3.2 Dropping Duplicates and Null Values

We are dealing with quite a large dataset, with the number of ratings being over 570,000. Therefore, it will be important to reduce the memory as much as possible by removing unnecessary features and reducing the memory usage. Since the timestamp data is unnecessary to our analysis, we will go ahead and drop that column from our ratings dataset. We also go through an initial iteration of removing duplicates and null values.

We will also write a function that displays the size of a dataframe, so that we can confirm that the transformations performed on the dataset are resulting in a reduced memory footprint.

[4]:
```python
def get_df_size(df):
    """
    Gets size of dataframe and prints value in MB.
    Function inspired by James Irving.

    Args:
        df (DataFrame) : DataFrame to print size of.
    Returns:

    """
    size = round((sys.getsizeof(df) * 1e-6), 2)

    print(f"Dataframe memory usage: {size} MB.")
```

[5]:
```python
# Drop duplicates and timestamp column from review table
review_df.drop_duplicates(inplace=True)
review_df.drop('timestamp', axis=1, inplace=True)
review_df
```

[5]:

	asin	user	rating
0	B00004U9V2	A1Q6MUU0B2ZDQG	2.0
1	B00004U9V2	A3HO2SQDCZIE9S	5.0
2	B00004U9V2	A2EM03F99X3RJZ	5.0
3	B00004U9V2	A3Z74TDRGD0HU	5.0
4	B00004U9V2	A2UXFNW9RTL4VM	5.0
…	…	…	…
574623	B01HIQEOLO	AHYJ78MVF4UQO	5.0
574624	B01HIQEOLO	A1L2RT7KBNK02K	5.0
574625	B01HIQEOLO	A36MLXQX9WPPW9	5.0
574626	B01HJ2UY0W	A23DRCOMC2RIXF	1.0
574627	B01HJ2UY1G	AJEDVHTLS9P3V	5.0

[538082 rows x 3 columns]

[6]:
```python
# Print size of original ratings df
get_df_size(review_df)
```

```
Dataframe memory usage: 82.72 MB.
```

Similarly with our metadata, we will go ahead and slice out the ASIN code and product names, since those are the pieces of data that will be used in our analysis. Then, we go on to drop duplicates from this table as well.

[7]:
```python
# Slice asin and title columns from metadata table
meta_df = meta_df[['asin','title']]
```

[8]:
```python
# Drop duplicates from metadata table
meta_df.drop_duplicates(inplace=True)
meta_df
```

[8]:
```
             asin                                              title
0      B00004U9V2  Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
1      B0000531EN                                    AHAVA Bath Salts
2      B0000532JH      AHAVA Dead Sea Mineral Mud, 8.5 oz, Pack of 4
3      B00005A77F  Crabtree &amp; Evelyn Hand Soap, Gardeners, 10…
4      B00005NDTD                                 Soy Milk Hand Crme
…             …                                                 …
12294  B01HIQIEYC          CND Shellac Power Polish, Patina Buckle
12295  B01HIQHQU0             CND Shellac power polish denim patch
12296  B01HIQEOLO                    CND Shellac, Leather Satchel
12297  B01HJ2UY0W  Juicy Couture I Love Juicy Couture, 1.7 fl. Oz…
12298  B01HJ2UY1G  Juicy Couture I Love Juicy Couture, 3.4 fl. Oz…

[12111 rows x 2 columns]
```

1.3.3 Merging Data Tables

Now, we will create a catalog_df which contains all of our ratings combined with their titles. This dataframe contains all of the information we will need for the purpose of our analysis. Let's also keep note of the size of our original catalog_df before we make transformations to reduce the memory allocation, and after dropping any duplicated or null values.

[9]:
```python
# Combine review data and metadata to create catalog table
catalog_df = review_df.merge(meta_df, how='left', on='asin')
catalog_df
```

[9]:
```
              asin            user  rating  \
0       B00004U9V2  A1Q6MUU0B2ZDQG     2.0
1       B00004U9V2  A3HO2SQDCZIE9S     5.0
2       B00004U9V2  A2EM03F99X3RJZ     5.0
3       B00004U9V2   A3Z74TDRGD0HU     5.0
4       B00004U9V2  A2UXFNW9RTL4VM     5.0
…              …               …     …
538077  B01HIQEOLO   AHYJ78MVF4UQO     5.0
538078  B01HIQEOLO  A1L2RT7KBNK02K     5.0
538079  B01HIQEOLO  A36MLXQX9WPPW9     5.0
538080  B01HJ2UY0W  A23DRCOMC2RIXF     1.0
538081  B01HJ2UY1G   AJEDVHTLS9P3V     5.0

                                                   title
```

```
0         Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
1         Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
2         Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
3         Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
4         Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
…                                                      …
538077                       CND Shellac, Leather Satchel
538078                       CND Shellac, Leather Satchel
538079                       CND Shellac, Leather Satchel
538080    Juicy Couture I Love Juicy Couture, 1.7 fl. Oz…
538081    Juicy Couture I Love Juicy Couture, 3.4 fl. Oz…

[538082 rows x 4 columns]
```

[10]:
```python
# Drop duplicates from merged catalog table
catalog_df.drop_duplicates(inplace=True)
catalog_df
```

[10]:
```
              asin            user  rating  \
0         B00004U9V2  A1Q6MUU0B2ZDQG     2.0
1         B00004U9V2  A3H02SQDCZIE9S     5.0
2         B00004U9V2  A2EM03F99X3RJZ     5.0
3         B00004U9V2   A3Z74TDRGD0HU     5.0
4         B00004U9V2  A2UXFNW9RTL4VM     5.0
…                …              …       …
538077    B01HIQEOLO   AHYJ78MVF4UQO     5.0
538078    B01HIQEOLO  A1L2RT7KBNK02K     5.0
538079    B01HIQEOLO  A36MLXQX9WPPW9     5.0
538080    B01HJ2UY0W  A23DRCOMC2RIXF     1.0
538081    B01HJ2UY1G   AJEDVHTLS9P3V     5.0

                                                    title
0         Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
1         Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
2         Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
3         Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
4         Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
…                                                      …
538077                       CND Shellac, Leather Satchel
538078                       CND Shellac, Leather Satchel
538079                       CND Shellac, Leather Satchel
538080    Juicy Couture I Love Juicy Couture, 1.7 fl. Oz…
538081    Juicy Couture I Love Juicy Couture, 3.4 fl. Oz…

[536295 rows x 4 columns]
```

[11]:
```
# Check for null values
catalog_df.isna().sum()
```

[11]:
```
asin        0
user        0
rating      0
title     184
dtype: int64
```

Since the number of null values in this catalog dataframe are quite small, we can go ahead and remove the observations where we do not have a product name paired with its ASIN code.

[12]:
```
# Drop null values
catalog_df.dropna(inplace=True)
catalog_df
```

[12]:
```
              asin            user  rating  \
0       B00004U9V2  A1Q6MUU0B2ZDQG     2.0
1       B00004U9V2  A3HO2SQDCZIE9S     5.0
2       B00004U9V2  A2EM03F99X3RJZ     5.0
3       B00004U9V2   A3Z74TDRGD0HU     5.0
4       B00004U9V2  A2UXFNW9RTL4VM     5.0
…              …              …     …
538077  B01HIQEOLO   AHYJ78MVF4UQO     5.0
538078  B01HIQEOLO  A1L2RT7KBNK02K     5.0
538079  B01HIQEOLO  A36MLXQX9WPPW9     5.0
538080  B01HJ2UY0W  A23DRCOMC2RIXF     1.0
538081  B01HJ2UY1G   AJEDVHTLS9P3V     5.0

                                                    title
0       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
1       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
2       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
3       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
4       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
…                                                  …
538077                          CND Shellac, Leather Satchel
538078                          CND Shellac, Leather Satchel
538079                          CND Shellac, Leather Satchel
538080  Juicy Couture I Love Juicy Couture, 1.7 fl. Oz…
538081  Juicy Couture I Love Juicy Couture, 3.4 fl. Oz…

[536111 rows x 4 columns]
```

[13]:
```
# Print size of initial catalog_df
get_df_size(catalog_df)
```

```
Dataframe memory usage: 141.67 MB.
```

1.3.4 Visualizing the Data

In this section, we will proceed to visualize the distribution of our ratings as well as how many users gave how many ratings each.

[14]:
```
# Check distribution of ratings
catalog_df['rating'].value_counts().sort_index(ascending=False)
```

[14]:
```
5.0    355360
4.0     65885
3.0     39428
2.0     27830
1.0     47608
Name: rating, dtype: int64
```

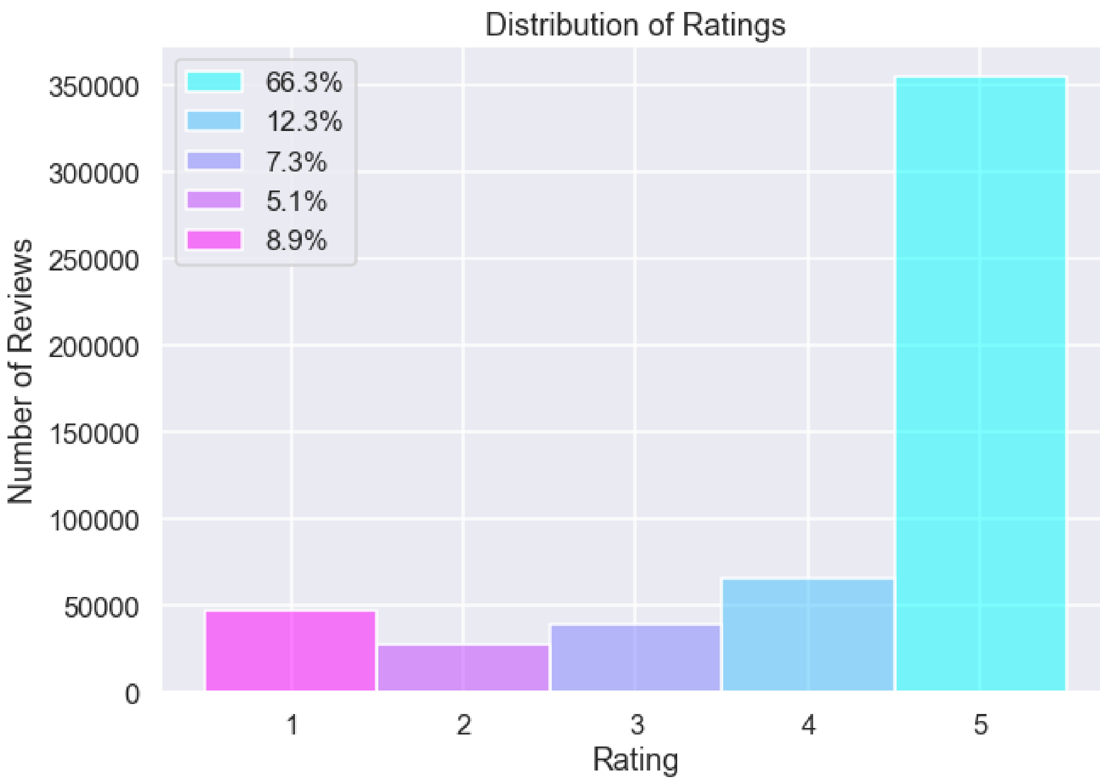
[15]:
```
# Check distribution of ratings in percent
catalog_df['rating'].value_counts(normalize=True).sort_index(ascending=False)
```

[15]:
```
5.0    0.662848
4.0    0.122894
3.0    0.073544
2.0    0.051911
1.0    0.088803
Name: rating, dtype: float64
```

[16]:
```
# Create bar plot of rating distribution
fig, ax = plt.subplots(figsize=(10,7))

g = sns.histplot(data=catalog_df, x='rating', hue='rating', palette='cool_r',\
                 discrete=True, legend=True)

ax.set_title('Distribution of Ratings')
ax.set_xlabel('Rating')
ax.set_ylabel('Number of Reviews')
ax.set_xticks([1,2,3,4,5])
ax.legend(['66.3%','12.3%','7.3%','5.1%','8.9%']);
```



[17]:
```
# Get number of ratings per user
freq_df = catalog_df.groupby('user').agg('count').reset_index()
freq_df
```

[17]:

	user	asin	rating	title
0	A0002708WFPIPQT73GK8	1	1	1
1	A0010876CNE3ILIM9HV0	1	1	1
2	A0026756LXIAIU5P6JUI	1	1	1
3	A0036810AKGSUKHOLV23	1	1	1
4	A004163085WKABQBPDOX	1	1	1
…	…	…	…	…
416072	AZZYUA6JI1MOO	2	2	2
416073	AZZYW4YOE1B6E	3	3	3
416074	AZZZ27Q95ZU80	1	1	1
416075	AZZZ3LGTCGUZF	1	1	1
416076	AZZZYAYJQSDOJ	1	1	1

[416077 rows x 4 columns]

[18]:
```
# Inspect measures of central tendency
freq_df.describe()
```

[18]:

	asin	rating	title
count	416077.000000	416077.000000	416077.000000
mean	1.288490	1.288490	1.288490
std	1.130142	1.130142	1.130142
min	1.000000	1.000000	1.000000
25%	1.000000	1.000000	1.000000
50%	1.000000	1.000000	1.000000
75%	1.000000	1.000000	1.000000
max	119.000000	119.000000	119.000000

[19]:
```
# Create table with number of users vs number of ratings per user
plot_df = freq_df.groupby('asin').agg('count')[:10]
plot_df
```

[19]:

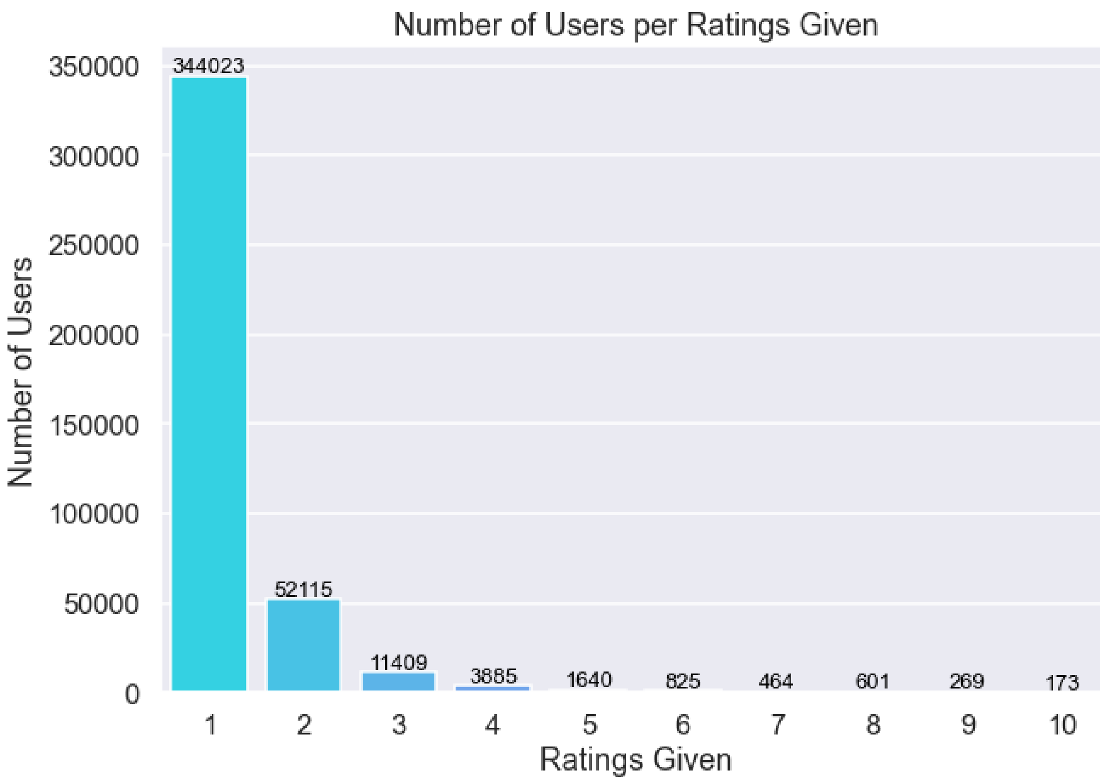
asin	user	rating	title
1	344023	344023	344023
2	52115	52115	52115
3	11409	11409	11409
4	3885	3885	3885
5	1640	1640	1640
6	825	825	825
7	464	464	464
8	601	601	601
9	269	269	269
10	173	173	173

[20]:
```
# Create bar plot of users per ratings given
fig, ax = plt.subplots(figsize=(10,7))

g = sns.barplot(data=plot_df, x=plot_df.index, y=plot_df['user'], \
                palette='cool')

ax.set_title('Number of Users per Ratings Given')
ax.set_xlabel('Ratings Given')
ax.set_ylabel('Number of Users')

for p in ax.patches:
             ax.annotate("%.0f" % p.get_height(), \
                         (p.get_x() + p.get_width() / 2., p.get_height()),\
                          ha='center', va='center', fontsize=13, \
                          color='black', xytext=(0, 5), \
                          textcoords='offset points');
```



[21]:
```
# Check measures of central tendency
catalog_df.describe()
```

[21]:

	rating
count	536111.000000
mean	4.219074
std	1.302025
min	1.000000
25%	4.000000
50%	5.000000
75%	5.000000
max	5.000000

1.3.5 Data Mapping

As mentioned before, due to the large size of this dataset, it is important to reduce the data to minimize the amount of memory being used. Hence, we map our ASIN and user codes to integer values in order to optimize memory allocation during the modeling process as well as converting our data types to the smallest possible integer type without losing any information.

[22]:
```
# Create list of unique asin codes
asin_list = catalog_df['asin'].unique()
```

[23]:
```python
# Create an array of integers to map asin codes to
np.arange(len(asin_list))
```

[23]:
```
array([    0,     1,     2, …, 12108, 12109, 12110])
```

[24]:
```python
# Construct dictionary using asin and corresponding product code
asin_map = dict(zip(asin_list, np.arange(len(asin_list))))
```

[25]:
```python
# Map asin to product code integer and check
catalog_df['asin'] = catalog_df['asin'].map(asin_map)
catalog_df
```

[25]:
```
         asin            user  rating  \
0           0  A1Q6MUU0B2ZDQG     2.0
1           0  A3HO2SQDCZIE9S     5.0
2           0  A2EM03F99X3RJZ     5.0
3           0   A3Z74TDRGD0HU     5.0
4           0  A2UXFNW9RTL4VM     5.0
…         …             …     …
538077   6007   AHYJ78MVF4UQ0     5.0
538078   6007  A1L2RT7KBNK02K     5.0
538079   6007  A36MLXQX9WPPW9     5.0
538080  12109  A23DRCOMC2RIXF     1.0
538081  12110   AJEDVHTLS9P3V     5.0

                                                  title
0       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
1       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
2       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
3       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
4       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
…                                                   …
538077                     CND Shellac, Leather Satchel
538078                     CND Shellac, Leather Satchel
538079                     CND Shellac, Leather Satchel
538080  Juicy Couture I Love Juicy Couture, 1.7 fl. Oz…
538081  Juicy Couture I Love Juicy Couture, 3.4 fl. Oz…

[536111 rows x 4 columns]
```

[26]:
```python
# Rename 'asin' column to 'product_code'
catalog_df = catalog_df.rename(columns={'asin': 'product_code'})
```

[27]:
```python
# Create list of unique users
user_list = catalog_df['user'].unique()
```

[28]:
```python
# Create an array of integers to map user codes to
np.arange(len(user_list))
```

[28]:
```
array([     0,      1,      2, …, 416074, 416075, 416076])
```

[29]:
```python
# Construct dictionary using user code and corresponding integer
user_map = dict(zip(user_list, np.arange(len(user_list))))
```

[30]:
```python
# Map asin to product code integer and check
catalog_df['user'] = catalog_df['user'].map(user_map)
catalog_df
```

[30]:
```
        product_code    user  rating  \
0                  0       0     2.0
1                  0       1     5.0
2                  0       2     5.0
3                  0       3     5.0
4                  0       4     5.0
…                …     …     …
538077          6007  194409     5.0
538078          6007  175285     5.0
538079          6007  416075     5.0
538080         12109  416076     1.0
538081         12110    4344     5.0

                                                    title
0       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
1       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
2       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
3       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
4       Crabtree &amp; Evelyn - Gardener's Ultra-Moist…
…                                                       …
538077                     CND Shellac, Leather Satchel
538078                     CND Shellac, Leather Satchel
538079                     CND Shellac, Leather Satchel
538080   Juicy Couture I Love Juicy Couture, 1.7 fl. Oz…
538081   Juicy Couture I Love Juicy Couture, 3.4 fl. Oz…

[536111 rows x 4 columns]
```

[31]:
```python
# Convert to more efficient integer types
catalog_df['rating']=catalog_df['rating'].astype(np.int8)
catalog_df['product_code']=catalog_df['product_code'].astype(np.int32)
catalog_df['user']=catalog_df['user'].astype(np.int32)
```

[32]:
```python
# Check data types
catalog_df.dtypes
```

[32]:
```
product_code     int32
user             int32
rating            int8
title           object
dtype: object
```

[33]:
```python
# Check datatype of columns
catalog_df.dtypes
```

[33]:
```
product_code     int32
user             int32
rating            int8
title           object
dtype: object
```

Now that we have reduced the datasize by converting each feature to its lowest possible integer type, let's take a look at the memory usage of our optimized catalog_df.

[34]:
```python
# Print size of transformed and optimized catalog_df
get_df_size(catalog_df)
```

```
Dataframe memory usage: 68.37 MB.
```

Great! We have successfully reduced the memory usage of this catalog_df from 141.67 MB to 68.37 MB without losing any essential information.

1.3.6 Slicing Data for Modeling

We're almost ready to enter the modeling process, so let's go ahead and slice out just the columns we need to do so.

[35]:
```python
# Create dataframe with user item rating
df = catalog_df[['user', 'product_code', 'rating']]
```

[36]:
```python
# Save csv file to use in Databricks ALS model
# catalog_df.to_csv(r'data/Luxury_Beauty_reduced.csv', index=False)
```

1.4 Data Modeling

In this section, we will take a look at using the Surprise scikit package to test which algorithm will be the best for building a recommender system using our Amazon review data.

The models we will look at are some K-Nearest Neighbor models and a series of gridsearched Singular Value Decomposition models. You can find the process behind modeling using Alternating Least Squares in PySpark, but we will leave this model out of our main analysis due to its poor performance on this specific dataset as well as the fact that we will need to use PySpark to perform the modeling process.

[37]:
```python
# If using Colab, install Surprise
# %pip install scikit-surprise
```

[38]:
```python
# Import necessary packages for building recommender system
from surprise import Dataset, Reader
from surprise import accuracy
from surprise.prediction_algorithms import knns
from surprise.similarities import cosine, msd, pearson
from surprise.model_selection import cross_validate, train_test_split
from surprise.prediction_algorithms import SVD
from surprise.model_selection import GridSearchCV
```

[39]:
```python
# Create reader object and format review data for processing
reader = Reader(line_format = 'user item rating', sep = ',')
data = Dataset.load_from_df(df, reader=reader)
```

[40]:
```python
# Create train test split
trainset, testset = train_test_split(data, test_size=0.25, random_state=27)
```

1.4.1 Memory-Based Item-Item Collaborative Filtering

As we see below, the number of unique items is much less than the number of unique users. Hence, for the following K-Nearest Neighbor models, it will be more effective to use item-based filtering in terms of computational efficiency as well as performance due to the fact that the average rating of each item is less likely to change as quickly as the ratings given by each user to different items.

For the KNN Basic and KNN with Means algorithms, we will examine performance based on cosine similarity and Pearson correlation coefficient. However, for the KNN with Z-score and KNN Baseline algorithms, we will only examine the Pearson baseline metric, since the Surprise documentation recommends this in order to achieve the best performance.

As we iterate through each model, we will save the resulting scores as dictionaries which we will combine in a dataframe to compare at the end.

[41]:
```python
# Write function to calculate average test metrics
def get_avg_metrics(score_dict):
    """
    Calculates average of each list in the specified dictionary.

    Inspired by solution by Jiby on StackOverflow:
    https://stackoverflow.com/questions/30687244/
↪python-3-4-how-to-get-the-average-of-dictionary-values

    Args:
        score_dict (dict) : Dictionary with model test scores.

    Returns:
        avgDict (dict) : Dictionary with calculated mean average values.
```

```
    """

    avgDict = {}
    for k,v in score_dict.items():
        avgDict[k] = sum(v)/ float(len(v))
    return avgDict
```

[42]:
```
# Check how many unique values for asin
catalog_df['product_code'].nunique()
```

[42]: 12111

[43]:
```
# Check how many unique values for user
catalog_df['user'].nunique()
```

[43]: 416077

KNN Basic We begin with the most basic form of the K-Nearest Neighbors algorithm.

[44]:
```
# KNN Basic with cosine similarity
KNN_basic_cos = knns.KNNBasic(sim_options={'name': 'cosine',
                                          'user_based': False}).fit(trainset)
```

```
Computing the cosine similarity matrix…
Done computing similarity matrix.
```

[45]:
```
# Get predictions on test data and print RMSE and MAE
predictions = KNN_basic_cos.test(testset)
accuracy.rmse(predictions)
accuracy.mae(predictions)
```

```
RMSE: 1.2579
MAE:  0.9413
```

[45]: 0.9413453103059339

[46]:
```
# Save dictionary with cross validated average scores
KNN_basic_cos_dict = cross_validate(KNN_basic_cos, data, verbose= True, \
                                    n_jobs=-1)
KNN_basic_cos_dict = get_avg_metrics(KNN_basic_cos_dict)
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.2632  1.2684  1.2633  1.2602  1.2582  1.2627  0.0035
MAE (testset)     0.9401  0.9429  0.9395  0.9383  0.9365  0.9395  0.0021
Fit time          13.79   13.34   12.81   11.93   9.86    12.35   1.39
Test time         2.11    1.56    1.22    1.06    0.97    1.38    0.41
```

[47]:
```
# KNN Basic with pearson correlation similarity
KNN_basic_pearson = knns.KNNBasic(sim_options={'name': 'pearson',
                                               'user_based': False})\
                        .fit(trainset)
```

```
Computing the pearson similarity matrix…
Done computing similarity matrix.
```

[48]:
```
# Get predictions on test data and print RMSE and MAE
predictions = KNN_basic_pearson.test(testset)
accuracy.rmse(predictions)
accuracy.mae(predictions)
```

```
RMSE: 1.2555
MAE:  0.9587
```

[48]: 0.9586864064951247

[49]:
```
# Save dictionary with cross validated average scores
KNN_basic_pearson_dict = cross_validate(KNN_basic_pearson, \
                                        data, verbose= True, n_jobs=-1)
KNN_basic_pearson_dict = get_avg_metrics(KNN_basic_pearson_dict)
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.2612  1.2494  1.2628  1.2675  1.2569  1.2596  0.0061
MAE (testset)     0.9569  0.9504  0.9583  0.9623  0.9552  0.9566  0.0039
Fit time          14.31   16.40   15.45   16.14   12.20   14.90   1.53
Test time         3.26    1.62    1.52    1.07    1.22    1.74    0.79
```

KNN With Means Next, we move onto a KNN algorithm which takes into account the mean ratings of each item.

[50]:
```
# KNN with Means with cosine similarity
KNN_mean_cos = knns.KNNWithMeans(sim_options={'name': 'cosine', \
                                              'user_based': False})\
                   .fit(trainset)
```

```
Computing the cosine similarity matrix…
Done computing similarity matrix.
```

[51]:
```
# Get predictions on test data and print RMSE and MAE
predictions = KNN_mean_cos.test(testset)
accuracy.rmse(predictions)
accuracy.mae(predictions)
```

```
RMSE: 1.2559
MAE:  0.9446
```

[51]: 0.9446222566929098

[52]:
```python
# Save dictionary with cross validated average scores
KNN_mean_cos_dict = cross_validate(KNN_mean_cos, data, verbose= True, \
                                   n_jobs=-1)
KNN_mean_cos_dict = get_avg_metrics(KNN_mean_cos_dict)
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.2635  1.2616  1.2630  1.2581  1.2585  1.2609  0.0023
MAE (testset)     0.9441  0.9444  0.9443  0.9425  0.9421  0.9435  0.0010
Fit time          11.76   14.62   13.00   12.69   10.07   12.43   1.50
Test time         2.97    1.37    1.34    1.21    1.23    1.63    0.68
```

[53]:
```python
# KNN with Means with pearson correlation similarity
KNN_mean_pearson = knns.KNNWithMeans(sim_options={'name': 'pearson', \
                                                  'user_based': False})\
                       .fit(trainset)
```

```
Computing the pearson similarity matrix…
Done computing similarity matrix.
```

[54]:
```python
# Get predictions on test data and print RMSE and MAE
predictions = KNN_mean_pearson.test(testset)
accuracy.rmse(predictions)
accuracy.mae(predictions)
```

```
RMSE: 1.2552
MAE:  0.9558
```

[54]: 0.9557535279674143

[55]:
```python
# Save dictionary with cross validated average scores
KNN_mean_pearson_dict = cross_validate(KNN_mean_pearson, data, verbose= True,\
                                       n_jobs=-1)
KNN_mean_pearson_dict = get_avg_metrics(KNN_mean_pearson_dict)
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.2587  1.2616  1.2565  1.2576  1.2662  1.2601  0.0035
MAE (testset)     0.9545  0.9561  0.9531  0.9522  0.9599  0.9551  0.0027
Fit time          15.13   15.91   15.40   15.07   12.55   14.81   1.17
Test time         2.53    2.26    1.74    1.37    1.33    1.84    0.48
```

KNN With Z-Score This algorithm takes into account the Z-score normalization of each item's ratings.

[56]:
```
# KNN with Z-score with pearson baseline correlation similarity
KNN_z_pearson = knns.KNNWithZScore(sim_options={'name': 'pearson_baseline', \
                                                'user_based': False})\
                    .fit(trainset)
```

```
Estimating biases using als…
Computing the pearson_baseline similarity matrix…
Done computing similarity matrix.
```

[57]:
```
# Get predictions on test data and print RMSE and MAE
predictions = KNN_z_pearson.test(testset)
accuracy.rmse(predictions)
accuracy.mae(predictions)
```

```
RMSE: 1.2548
MAE:  0.9510
```

[57]: 0.9510002056079127

[58]:
```
# Save dictionary with cross validated average scores
KNN_z_pearson_dict = cross_validate(KNN_z_pearson, data, verbose= True, \
                                    n_jobs=-1)
KNN_z_pearson_dict = get_avg_metrics(KNN_z_pearson_dict)
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.2555  1.2566  1.2563  1.2693  1.2604  1.2596  0.0051
MAE (testset)     0.9466  0.9488  0.9487  0.9563  0.9505  0.9502  0.0033
Fit time          13.17   14.68   14.45   13.20   12.48   13.60   0.83
Test time         2.00    1.51    1.34    1.10    1.01    1.39    0.35
```

KNN Baseline This final algorithm is a K-Nearest Neighbors algorithm that takes into account a baseline rating for each item.

[59]:
```
# KNN Baseline with pearson baseline similarity
KNN_base_pearson= knns.KNNBaseline(sim_options={'name': 'pearson_baseline', \
                                                'user_based': False})\
                      .fit(trainset)
```

```
Estimating biases using als…
Computing the pearson_baseline similarity matrix…
Done computing similarity matrix.
```

[60]:
```python
# Get predictions on test data and print RMSE and MAE
predictions = KNN_base_pearson.test(testset)
accuracy.rmse(predictions)
accuracy.mae(predictions)
```

```
RMSE: 1.2207
MAE:  0.9138
```

[60]:
```
0.9137729809312667
```

[61]:
```python
# Save dictionary with cross validated average scores
KNN_base_pearson_dict = cross_validate(KNN_base_pearson, data, \
                                       verbose= True, n_jobs=-1)
KNN_base_pearson_dict = get_avg_metrics(KNN_base_pearson_dict)
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.2227  1.2217  1.2241  1.2254  1.2270  1.2242  0.0019
MAE (testset)     0.9114  0.9096  0.9111  0.9130  0.9117  0.9114  0.0011
Fit time          12.39   12.85   12.92   12.44   11.32   12.38   0.57
Test time         2.41    1.94    1.48    1.16    1.10    1.62    0.50
```

We can see that the RMSE scores for our KNN are similar across the board, except for the KNN Baseline model, which did have a slightly lower score.

1.4.2 Model-Based Collaborative Filtering via Matrix Factorization

Singular Value Decomposition Now, let's move onto the SVD model where we will begin with a basic model and try to improve our score by using a series of gridsearches. This model-based approach takes a sparse matrix where we have users x items, and decomposes this utility matrix into item characteristics and user preferences that correspond to those characteristics. By utilizing a gridsearch, we can determine the optimal number of factors, or characteristics/preferences, as well as adjust learning and regularization rates.

[62]:
```python
# Train basic SVD model
svd1 = SVD(random_state=27)
svd1.fit(trainset)
```

[62]:
```
<surprise.prediction_algorithms.matrix_factorization.SVD at 0x7fbd087359d0>
```

[63]:
```python
# Get predictions on test data and print RMSE
predictions = svd1.test(testset)
accuracy.rmse(predictions)
accuracy.mae(predictions)
```

```
RMSE: 1.2343
MAE:  0.9513
```

[63]: 0.9513405014854374

[64]:
```
# Save dictionary with average scores
svd1_dict = cross_validate(svd1, data, verbose= True, n_jobs=-1)
svd1_dict = get_avg_metrics(svd1_dict)
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.2338  1.2398  1.2383  1.2373  1.2363  1.2371  0.0020
MAE (testset)     0.9486  0.9529  0.9531  0.9509  0.9498  0.9511  0.0017
Fit time          36.64   36.63   36.72   36.84   36.35   36.64   0.16
Test time         0.94    0.89    0.76    0.71    0.67    0.80    0.10
```

[65]:
```
# Gridsearch #1
param_grid = {'n_factors':[110, 130],'n_epochs': [25, 30], \
              'lr_all': [0.025, 0.05], 'reg_all': [0.1, 0.2]}
svd_grid1 = GridSearchCV(SVD,param_grid=param_grid,joblib_verbose=5, \
                         n_jobs=-1)
svd_grid1.fit(data)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 tasks      | elapsed:  1.3min
[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed: 10.9min
[Parallel(n_jobs=-1)]: Done  80 out of  80 | elapsed: 16.0min finished
```

[66]:
```
# Print results from gridsearch #1
svd_grid1.best_params
```

[66]:
```
{'rmse': {'n_factors': 130, 'n_epochs': 30, 'lr_all': 0.025, 'reg_all': 0.1},
 'mae': {'n_factors': 110, 'n_epochs': 30, 'lr_all': 0.05, 'reg_all': 0.1}}
```

[67]:
```
# Use best params to get RMSE and MAE on test data
svd2 = SVD(n_factors=130, n_epochs=30, lr_all=0.025, reg_all=0.1, \
           random_state=27)
svd2.fit(trainset)
predictions = svd2.test(testset)
accuracy.rmse(predictions)
accuracy.mae(predictions)
```

```
RMSE: 1.2182
MAE:  0.9285
```

[67]: 0.9285218562243839

[68]:
```
# Save dictionary with average scores
svd2_dict = cross_validate(svd2, data, verbose= True, n_jobs=-1)
svd2_dict = get_avg_metrics(svd2_dict)
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.2213  1.2221  1.2194  1.2214  1.2214  1.2211  0.0009
MAE (testset)     0.9263  0.9269  0.9270  0.9278  0.9290  0.9274  0.0009
Fit time          61.18   62.19   62.88   61.96   62.29   62.10   0.55
Test time         1.01    0.81    0.75    0.78    0.69    0.81    0.11
```

[69]:
```
# Gridsearch #2
param_grid = {'n_factors':[130, 150],'n_epochs': [30, 40], \
              'lr_all': [0.01, 0.025], 'reg_all': [0.05, 0.1]}
svd_grid2 = GridSearchCV(SVD,param_grid=param_grid,joblib_verbose=5, \
                         n_jobs=-1)
svd_grid2.fit(data)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 tasks      | elapsed:  1.8min
[Parallel(n_jobs=-1)]: Done  56 tasks      | elapsed: 14.9min
[Parallel(n_jobs=-1)]: Done  80 out of  80 | elapsed: 22.2min finished
```

[70]:
```
# Print results from gridsearch #2
svd_grid2.best_params
```

[70]:
```
{'rmse': {'n_factors': 130, 'n_epochs': 40, 'lr_all': 0.025, 'reg_all': 0.1},
 'mae': {'n_factors': 130, 'n_epochs': 40, 'lr_all': 0.025, 'reg_all': 0.05}}
```

[71]:
```
# Use best params to get RMSE and MAE on test data
svd3 = SVD(n_factors=150, n_epochs=40, lr_all=0.025, reg_all=0.1, \
           random_state=27)
svd3.fit(trainset)
predictions = svd3.test(testset)
accuracy.rmse(predictions)
accuracy.mae(predictions)
```

```
RMSE: 1.2174
MAE:  0.9259
```

[71]:
```
0.9258506393305158
```

[72]:
```
# Save dictionary with average scores
svd3_dict = cross_validate(svd3, data, verbose= True, n_jobs=-1)
svd3_dict = get_avg_metrics(svd3_dict)
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.2179  1.2205  1.2185  1.2207  1.2223  1.2200  0.0016
MAE (testset)     0.9243  0.9241  0.9238  0.9252  0.9256  0.9246  0.0007
```

```
Fit time          91.82   92.87   92.91   91.04   91.83   92.09   0.71
Test time         0.90    0.79    0.78    0.82    0.69    0.80    0.07
```

[73]:
```python
# Gridsearch #3
param_grid = {'n_factors':[150, 200],'n_epochs': [40, 50], 'lr_all': [0.025],
              'reg_all': [0.1]}
svd_grid_final = GridSearchCV(SVD,param_grid=param_grid,joblib_verbose=5, \
                              n_jobs=-1)
svd_grid_final.fit(data)
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done   2 tasks      | elapsed:  2.6min
[Parallel(n_jobs=-1)]: Done  10 out of  20 | elapsed:  5.9min remaining:  5.9min
[Parallel(n_jobs=-1)]: Done  15 out of  20 | elapsed:  6.6min remaining:  2.2min
[Parallel(n_jobs=-1)]: Done  20 out of  20 | elapsed:  8.5min remaining:    0.0s
[Parallel(n_jobs=-1)]: Done  20 out of  20 | elapsed:  8.5min finished
```

[74]:
```python
# Print results from final gridsearch
svd_grid_final.best_params
```

[74]:
```
{'rmse': {'n_factors': 200, 'n_epochs': 50, 'lr_all': 0.025, 'reg_all': 0.1},
 'mae': {'n_factors': 150, 'n_epochs': 50, 'lr_all': 0.025, 'reg_all': 0.1}}
```

[75]:
```python
# Use best params to get RMSE and MAE on test data
svd_final = SVD(lr_all=0.025, n_epochs=50, n_factors=150, reg_all=0.1, \
                random_state=27)
svd_final.fit(trainset)
predictions = svd_final.test(testset)
accuracy.rmse(predictions)
accuracy.mae(predictions)
```

```
RMSE: 1.2171
MAE:  0.9237
```

[75]:
```
0.9237444509387739
```

[76]:
```python
# Save dictionary with average scores
svd_final_dict = cross_validate(svd_final, data, verbose= True, n_jobs=-1)
svd_final_dict = get_avg_metrics(svd_final_dict)
```

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

                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    1.2206  1.2226  1.2185  1.2152  1.2223  1.2199  0.0027
MAE (testset)     0.9220  0.9250  0.9232  0.9200  0.9246  0.9230  0.0018
Fit time          114.02  114.80  112.94  114.20  114.78  114.15  0.68
Test time         1.01    0.85    0.80    0.72    0.67    0.81    0.12
```

1.5 Evaluation

In this section, we will begin by evaluating our test scores and then move on to build some functions to assist the client in looking up product codes. Finally, we will build a recommender system that takes a list of preferred products and returns a list of items that the user would likely give a high rating to.

Let's compare our test scores from all of the models that we've fit to this point:

[77]:
```
# Create and display dataframe of all models and mean test scores
res_list = [KNN_basic_cos_dict,
            KNN_basic_pearson_dict,
            KNN_mean_cos_dict,
            KNN_mean_pearson_dict,
            KNN_z_pearson_dict,
            KNN_base_pearson_dict,
            svd1_dict,
            svd2_dict,
            svd3_dict,
            svd_final_dict]

res_list_strings = ["KNN_basic_cos_dict",
                    "KNN_basic_pearson_dict",
                    "KNN_mean_cos_dict",
                    "KNN_mean_pearson_dict",
                    "KNN_z_pearson_dict",
                    "KNN_base_pearson_dict",
                    "svd1_dict",
                    "svd2_dict",
                    "svd3_dict",
                    "svd_final_dict"]

test_results_df = pd.DataFrame(res_list, index=res_list_strings)
test_results_df
```

[77]:

	test_rmse	test_mae	fit_time	test_time
KNN_basic_cos_dict	1.262665	0.939480	12.345936	1.383554
KNN_basic_pearson_dict	1.259578	0.956636	14.900020	1.738472
KNN_mean_cos_dict	1.260920	0.943496	12.428880	1.625561
KNN_mean_pearson_dict	1.260106	0.955121	14.808874	1.844759
KNN_z_pearson_dict	1.259619	0.950190	13.596971	1.392375
KNN_base_pearson_dict	1.224198	0.911362	12.382939	1.618132
svd1_dict	1.237101	0.951064	36.635049	0.795581
svd2_dict	1.221125	0.927406	62.099963	0.806985
svd3_dict	1.219963	0.924606	92.093888	0.797203
svd_final_dict	1.219865	0.922972	114.147337	0.810540

Great! We can see that by using our gridsearches, we were able to make some improvements in the RMSE score between iterations. It also looks like our final SVD model has a lower RMSE score than even our best performing KNN Baseline model, so we will move forward to building our recommender system using the SVD model with the best parameters found in our final gridsearch. We can also see that our MAE score is 0.9229, meaning that in terms of rating stars, the average error of our model is off by 0.9229 stars from the actual rating.

1.5.1 Searching Product Codes

Here, we create a reduced catalog of product names with their corresponding product codes. We then build a function to search the name of a product to assist our user in looking up product codes to input into the recommender system.

[78]:
```
# Set pandas options to increase max column width and row number
pd.options.display.max_colwidth = 100
pd.options.display.max_rows = 500
catalog_df
```

[78]:
```
        product_code    user  rating  \
0                  0       0       2
1                  0       1       5
2                  0       2       5
3                  0       3       5
4                  0       4       5
…                …       …       …
538077          6007  194409       5
538078          6007  175285       5
538079          6007  416075       5
538080         12109  416076       1
538081         12110    4344       5

        title
0       Crabtree &amp; Evelyn - Gardener's Ultra-Moisturising Hand Therapy Pump
- 250g/8.8 OZ
1       Crabtree &amp; Evelyn - Gardener's Ultra-Moisturising Hand Therapy Pump
- 250g/8.8 OZ
2       Crabtree &amp; Evelyn - Gardener's Ultra-Moisturising Hand Therapy Pump
- 250g/8.8 OZ
3       Crabtree &amp; Evelyn - Gardener's Ultra-Moisturising Hand Therapy Pump
- 250g/8.8 OZ
4       Crabtree &amp; Evelyn - Gardener's Ultra-Moisturising Hand Therapy Pump
- 250g/8.8 OZ
…
…
538077                                                                 CND Shellac,
Leather Satchel
538078                                                                 CND Shellac,
Leather Satchel
538079                                                                 CND Shellac,
Leather Satchel
```

```
538080                      Juicy Couture I Love Juicy Couture, 1.7 fl. Oz.,
perfume for women
538081                      Juicy Couture I Love Juicy Couture, 3.4 fl. Oz.,
perfume for women

[536111 rows x 4 columns]
```

[79]:
```
# Create lookup df to look up product codes and/or names
lookup_df = catalog_df.drop_duplicates('product_code')
lookup_df = lookup_df[['product_code', 'title']]
lookup_df
```

[79]:
```
        product_code  \
0                  0
559                1
567                2
637                3
653                4
…                …
538039         12106
538040         12107
538064         12108
538080         12109
538081         12110

                      title
0                     Crabtree &amp; Evelyn - Gardener's Ultra-Moisturising Hand
Therapy Pump - 250g/8.8 OZ
559                                        Crabtree &amp; Evelyn Hand
Soap, Gardeners, 10.1 fl. oz.
567
Soy Milk Hand Crme
637
Supersmile Powdered Mouthrinse
653     Supersmile Professional Teeth Whitening Toothpaste  Recommended By
Cosmetic Dentists, CLINICALLY…
…
…
538039                        St. Tropez Self Tan Bronzing Mousse, 8 fl. oz.
&amp; Applicator Mitt Bundle
538040                                      Klorane Conditioner with
Pomegranate - Color-Treated Hair
538064
CND Shellac, Brick Knit
538080                           Juicy Couture I Love Juicy Couture, 1.7
fl. Oz., perfume for women
538081                           Juicy Couture I Love Juicy Couture, 3.4
```

```
fl. Oz., perfume for women

[12111 rows x 2 columns]
```

[80]:
```python
# Create function to look up product codes
def product_search():
    """
    Prompts user to look up product name and returns product code.

    Args:

    Returns:
        search_results (DataFrame) : DataFrame including results of searched
        product name
    """

    # Prompt user for item name
    query_product = input('Search a brand or product: ')

    # Prompt user for number of results desired
    num_results = int(input('Up to how many results would you like to see? '))

    # Slice catalog_df to return DataFrame with results containing query
    search_results = lookup_df[lookup_df['title'].str\
                            .contains(query_product, case=False, na=False)]\
                            .head(num_results)

    return search_results
```

[81]:
```python
# Look up sample product codes
product_search()
```

```
Search a brand or product: ahava
Up to how many results would you like to see? 10
```

[81]:
```
        product_code  \
79526            597
79541            598
79556            599
79557            600
79562            601
79569            602
79586            605
94115            692
109827           813
193198          1743
```

```
 title
79526                                                  AHAVA Smoothing Moisturizer
(Day)
79541                                              AHAVA Mineral Hand Cream, 3.4
fl. oz.
79556                        AHAVA Muscle Soothing Mineral Bath Salts, Eucalyptus,
32 oz.
79557                                              AHAVA Mineral Foot Cream, 3.4
fl. oz.
79562                                    AHAVA Purifying Mud Soap for Oily Skin,
3.4 oz.
79569                                                              AHAVA Limited
Edition
79586                                                 AHAVA Dead Sea Mineral
Shower Gels
94115   AHAVA Pure Silk Multi-Vitamin Dry Oil Spray, Mandarin - Cedarwood, 3.4
fl. oz.
109827            AHAVA Sun Protection Anti-Aging Moisturizer with SPF 30, 8.5
fl. oz.
193198                             AHAVA Dead Mineral Botanic Velvet Cream Body
Washes
```

1.5.2 Building the Recommender System

In this section, we will take the hyperparameters from our best performing SVD model to build a usable recommender system. Upon running the function, the user will be prompted to enter a list of product codes of products that they gave high ratings to, and they will be given a list of products that our algorithm would recommend.

[82]:
```python
# Check last user number
df['user'].sort_values().tail()
```

[82]:
```
538073    416072
538074    416073
538075    416074
538079    416075
538080    416076
Name: user, dtype: int32
```

[83]:
```python
# Create function to train model on full dataset and return recommendations
def user_ratings(lr_all=0.025, n_epochs=50, n_factors=150, reg_all=0.1, \
                 random_state=27):
    """
    Prompts user to enter customer's preferred product codes, models SVD
    using ideal hyperparameters, and returns however many predictions
    the user requests.
```

```
    Args:
        lr_all : The learning rate for all parameters. Default is ``0.025``.
        n_epochs : The number of iteration of the SGD procedure. Default is
            ``50``.
        n_factors : The number of factors. Default is ``150``.
        reg_all : The regularization term for all parameters. Default is
            ``0.1``.
        random_state (int) : Determines the RNG that will be used for
            initialization. If int, ``random_state`` will be used as a seed
            for a new RNG. This is useful to get the same initialization over
            multiple calls to ``fit()``.  If RandomState instance, this same
            instance is used as RNG. If ``None``, the current RNG from numpy
            is used.  Default is``27``.

    Returns:
        rec_list (DataFrame) : DataFrame recommendations based on new user's
        preferred products.
    """

    # Prompt user for list of product codes
    list_of_products = [int(x) for x in \
                        input('Enter product codes preferred by customer␣
↪(separate by spaces): ')\
                        .split()]

    # Prompt user for desired number of product recommendations
    num_res = int(input('How many recommendations would you like? '))

    # Create list of ratings to add to dataset
    my_ratings = []
    for product in list_of_products:
        my_ratings.append({'user': 600000, 'product_code': product, \
                           'rating': '5'})

    # Add new ratings to full dataset
    new_ratings_df = df.append(my_ratings,ignore_index=True)

    # Format dataset for modeling
    reader = Reader(line_format='item user rating')
    new_data = Dataset.load_from_df(new_ratings_df,reader)

    # Train model on full dataset using preset hyperparameters
    svd_ = SVD(lr_all=lr_all, n_epochs=n_epochs, n_factors=n_factors, \
               reg_all=reg_all, random_state=random_state)
    svd_.fit(new_data.build_full_trainset())

    # Create total list of predictions for new user
```

```python
    list_of_predictions = []
    for item in df['product_code'].unique():
        list_of_predictions.append((item, svd_.predict(600000, item)[3]))

    # Sort predictions from high to low
    ranked_predictions = sorted(list_of_predictions, key=lambda x:x[1], \
                                reverse=True)

    # Create dataframe from ranked predictions
    ranked_df = pd.DataFrame(ranked_predictions, columns=['product_code', \
                                                          'rating'])

    # Merge predictions with lookup df to get product names
    merged_df = ranked_df.merge(lookup_df, how='inner', on='product_code')

    # Create dataframe with requested number of results
    rec_list = merged_df.head(num_res)



    return rec_list
```

[84]:
```python
# Test function
user_ratings()
```

```
Enter product codes preferred by customer (separate by spaces): 597 601 1743
How many recommendations would you like? 10
```

[84]:
```
   product_code  rating  \
0             2     5.0
1            35     5.0
2            61     5.0
3            87     5.0
4           147     5.0
5           172     5.0
6           201     5.0
7           203     5.0
8           225     5.0
9           233     5.0

                                                title
0                                  Soy Milk Hand Crme
1            jane iredale So-Bronze, Bronzing Powder, 0.35 oz
2      Borghese Cura-C Anhydrous Vitamin C Treatment, 1.7 oz.
3                          NEOVA Day Therapy SPF 30, 1.7 Fl Oz
4  Kneipp Lavender Mineral Bath Salt, Relaxing, 17.63 fl. oz.
5                                NEOVA Squalane, 1.0 Fl Oz
```

```
6                    Glycolix Elite Sunscreen SPF 30, 1.6 Fl Oz
7                            Archipelago Lanai Glass Jar Candle
8              Elizabeth Arden Fifth Avenue Eau de Parfum Spray
9             Paul Mitchell Soft Sculpting Spray Gel,16.9 Fl Oz
```

And there we have our product recommendations! Now, let's take a look at what the top products were by selecting the top 10 products in number of ratings.

[85]:
```python
# View top 10 products with most reviews
top_series = catalog_df['product_code'].value_counts().head(10)
top_df = pd.DataFrame(top_series)
top_df
```

[85]:

	product_code
1113	3427
129	3405
3203	3190
1230	3074
651	3013
14	2995
272	2734
744	2681
1249	2635
2980	2532

[86]:
```python
# Create list of top 10 products with most reviews
top_list = catalog_df['product_code'].value_counts().index[:10].tolist()
top_list
```

[86]:
```
[1113, 129, 3203, 1230, 651, 14, 272, 744, 1249, 2980]
```

[87]:
```python
# Merge top_df with lookup_df
new_df = top_df.merge(lookup_df, how='left', left_index=True, \
                      right_on='product_code')
new_df = new_df.groupby('title').agg({'product_code_x':'sum'})\
                                .sort_values(by='product_code_x', \
                                             ascending=False)
new_df
```

[87]:
```
              product_code_x
title
TOPPIK Hair Building Fibers
6832
HOT TOOLS Professional 24k Gold Extra-Long Barrel Curling Iron/Wand for Long
Lasting Results            5825
 Mario Badescu Facial Spray with Aloe, Herbs and Rosewater, 8 oz.
3074
```

```
OPI Nail Lacquer, Cajun Shrimp, 0.5 fl. oz.
3013
OPI Nail Lacquer, Not So Bora-Bora-ing Pink, 0.5 Fl Oz
2995
BaBylissPRO Ceramix Xtreme Dryer
2734
OPI Nail Envy Nail Strengthener
2681
Proraso Shaving Soap in a Bowl, Refreshing and Toning, 5.2 oz
2532
```

[88]:
```
# Limit title length to 45 characters
new_df.index = new_df.index.str[:45]
new_df = new_df.reset_index()
new_df
```

[88]:

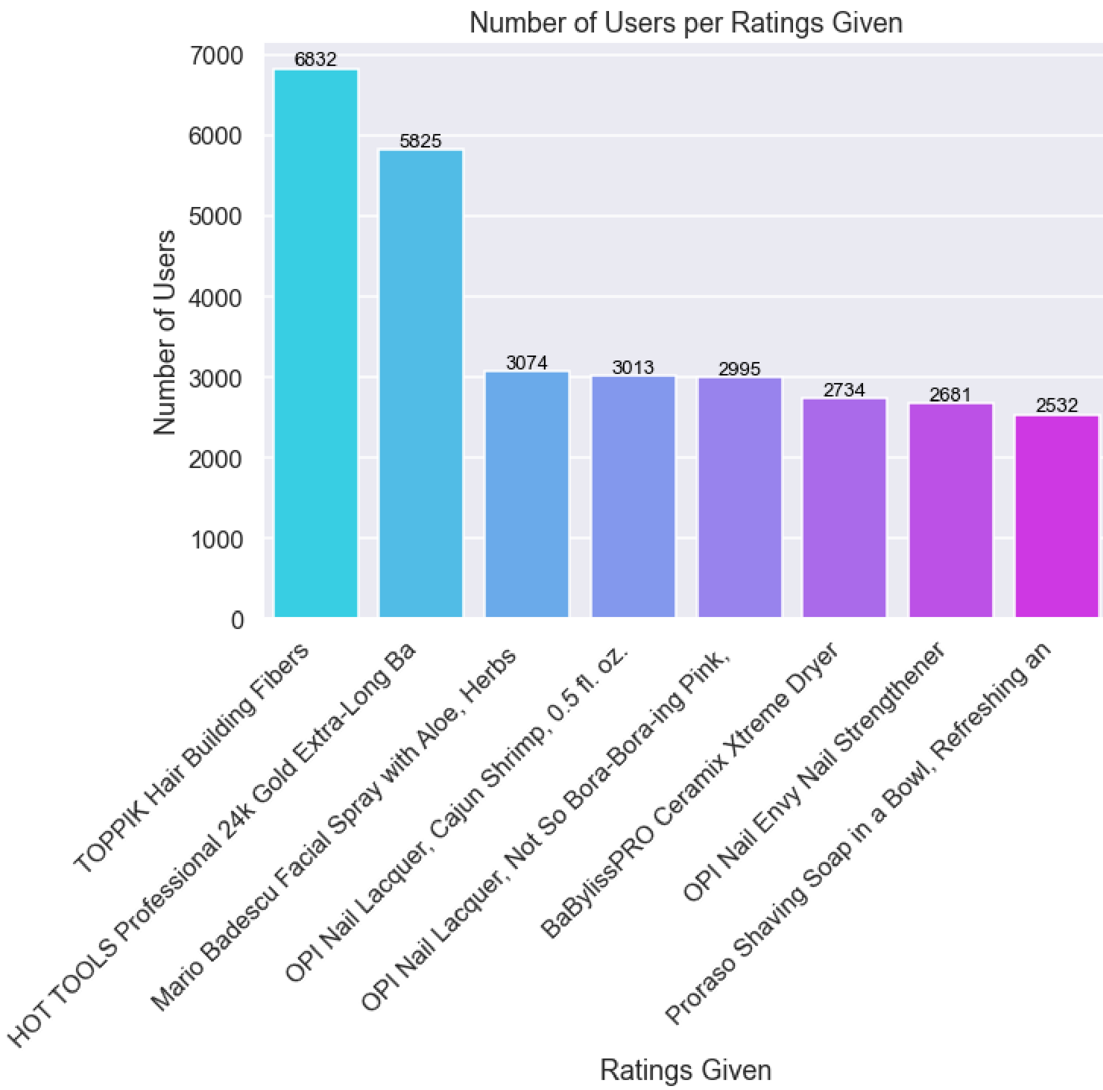
	title	product_code_x
0	TOPPIK Hair Building Fibers	6832
1	HOT TOOLS Professional 24k Gold Extra-Long Ba	5825
2	Mario Badescu Facial Spray with Aloe, Herbs	3074
3	OPI Nail Lacquer, Cajun Shrimp, 0.5 fl. oz.	3013
4	OPI Nail Lacquer, Not So Bora-Bora-ing Pink,	2995
5	BaBylissPRO Ceramix Xtreme Dryer	2734
6	OPI Nail Envy Nail Strengthener	2681
7	Proraso Shaving Soap in a Bowl, Refreshing an	2532

[89]:
```
# Create bar plot most popular products
fig, ax = plt.subplots(figsize=(10,7))

g = sns.barplot(data=new_df, x='title', y='product_code_x', palette='cool', \
                ci=None)

ax.set_title('Number of Users per Ratings Given')
ax.set_xlabel('Ratings Given')
ax.set_ylabel('Number of Users')
ax.set_xticklabels(ax.get_xticklabels(), rotation=45, ha='right')

for p in ax.patches:
             ax.annotate("%.0f" % p.get_height(), \
                         (p.get_x() + p.get_width() / 2., p.get_height()),
                 ha='center', va='center', fontsize=13, color='black', \
                         xytext=(0, 5),
                 textcoords='offset points');
```



Assuming that our client already carries these products which are popular on Amazon, let's see what other product recommendations we can get.

[90]:
```
# Get final recommendations
user_ratings()
```

```
Enter product codes preferred by customer (separate by spaces): 1113 3203 1230
651 14 272 744 2980
How many recommendations would you like? 10
```

[90]:
```
   product_code  rating  \
0             0     5.0
1             1     5.0
2             2     5.0
```

```
3             15     5.0
4             26     5.0
5             28     5.0
6             29     5.0
7             35     5.0
8             42     5.0
9             58     5.0

                 title
0                Crabtree &amp; Evelyn - Gardener's Ultra-Moisturising Hand
Therapy Pump - 250g/8.8 OZ
1                                             Crabtree &amp; Evelyn Hand Soap,
Gardeners, 10.1 fl. oz.
2
Soy Milk Hand Crme
3
Paul Mitchell Shampoo One
4                                                                     Glytone
Rejuvenating Mask, 3 oz.
5                                         PCA SKIN  Protecting Hydrator Broad
Spectrum SPF 30, 1.7 oz.
6                                                        jane iredale Amazing
Base Loose Mineral Powder
7                                                      jane iredale So-Bronze,
Bronzing Powder, 0.35 oz
8  Yu-Be: Japan&rsquo;s secret for dry skin relief. Deep hydrating moisturizing
cream for face, han…
9                                                      Calvin Klein ETERNITY Eau
de Parfum, 3.4 fl. oz.
```

1.6 Conclusions

And there we have our final product recommendations! We can see that the Singular Value Decomposition had the best performance with respect to RMSE. Upon running a series of gridsearches, we were also able to determine the optimal hyperparameters to further reduce the RMSE score.

To interpret our error, we looked at the MAE score which was 0.9237 on our final best model, meaning that the average error of our model is off by 0.9237 stars from the actual rating.

Finally, we built out functions to help us look up product codes to put into a recommender system which would then provide us with however many product recommendations the user desires.

The value of this project lies in the ability to use Amazon's huge amount of ratings data to identify what other products a smaller retailer might want to consider adding to their inventory. The only additional data that we would need from the retailer would be customer preferences on the products that the retailer currently carries and that the customer would give high ratings to, and we can place this information in the context of Amazon's ratings to determine what other products this customer would be likely to give high ratings to.

A limitation to this analysis is that the dataset only contains beauty products under the "Luxury Beauty" category, which is a collection of approved brands. Amazon also has a category labeled "All Beauty" whose data we have omitted in this analysis due to hardware limitations that would occur under the stress of dealing with the such a large size of these combined datasets.

To summarize, here are the final recommendations for our client:

1. In order to build a similar recommender system, SVD would be the best algorithm to use, with the following hyperparameters: lr_all=0.025, n_epochs=50, n_factors=150, reg_all=0.1
2. Client should carry the following products based on popularity on Amazon:

- TOPPIK Hair Building Fibers
- HOT TOOLS Professional 24k Gold Extra-Long Barrel Curling Iron/Wand
- Mario Badescu Facial Spray with Aloe, Herbs and Rosewater
- OPI Nail Lacquer, Cajun Shrimp
- OPI Nail Lacquer, Not So Bora-Bora-ing Pink
- BaBylissPRO Ceramix Xtreme Dryer
- OPI Nail Envy Nail Strengthener
- Proraso Shaving Soap in a Bowl, Refreshing and Toning

3. Assuming that our client's current customers would give high ratings to those products, our client should also consider carrying the following products:

- Crabtree & Evelyn - Gardener's Ultra-Moisturising Hand Therapy Pump
- Crabtree & Evelyn Hand Soap, Gardeners
- Soy Milk Hand Crme
- Paul Mitchell Shampoo One
- Glytone Rejuvenating Mask
- PCA SKIN Protecting Hydrator Broad Spectrum SPF 30
- jane iredale Amazing Base Loose Mineral Powder
- jane iredale So-Bronze, Bronzing Powder
- YU-Be: Japan's secret for dry skin relief. Deep hydrating moisturizing cream for face, hand and body
- Calvin Klein ETERNITY Eau de Parfum

Although ALS has been proven to be an effective algorithm in recommender systems, it was surprising to see such a poor performance score with the data used in this analysis. Moving forward, it might be a worthwhile investigation to see how the model performs if we combine data from the "All Beauty" category with the data used in this analysis.

[ ]: