Item Based Recommender System

Authors: Carleton Smith

Project Guide

Project Overview

EXPECTED TIME: 2.5 HRS

This project generally proceeds in 4 parts:

- General Familiarization with data.
- Mathematical foundations of, and simple examples of recommender systems.
- Execution of methods on data.
- Short introduction to `Surprise` package.

The methods used below should all be familiar from the lectures found in week 9. Some concepts will be reviewed before applying, but in some cases, SVD notably, particular will not be reviewed, only execution will be demonstrated.

The general goal is: given a collection of users, items and user reviews of items, predict what score a user would choose for an item they have yet to review.

We will be working with a synthetic review dataset, modeled on reviews from Amazon. For our examples we will work with a small amount of made-up data, to demonstrate techniques.

Activities include:

- Manipulating DataFrames
- Visualizing data
- Calculating distance in multiple ways
- Calculating and interpreting similarity scores
- Using `numpy`'s SVD function
- Learning about the `Surprise` package

Motivation: Recommender systems provide non-parametric comparisons between items. They are foundational in how individuals are served and thus navigate the web.

Objectives:

- Understand mathematical foundations of recommender systems.
- Translation of mathematical algorithm into code.

Imports

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

%matplotlib inline
```

Reading in the data

This assignments we will be using some synthetic data, modeled on Amazon product reviews. A description of the review data available and instructions to pull down actual Amazon review data can be found here.

Before turning to our synthetic data, see below the sample of actual Data from Amazon.

```
sample_data_path = "https://s3.amazonaws.com/amazon-reviews-pds/tsv/sample_us.tsv"

rev_df = pd.read_table(sample_data_path)

rev_df.head()
```

Looking at our raw data

While NLP might offer insight into the structure of reviews, for the purpose of this assignment, we are only interested in the customers, the product, and the star rating

```
rev_df = rev_df[['customer_id','product_id', 'star_rating']]
rev_df.rename(columns= {'star_rating':'score'}, inplace=True)
rev_df.head()
```

Question 1

```
### GRADED
### in the call to `.rename()` above, a dictionary was passed in to the `columns` parameter
### Which other object, when passed to 'columns', whould have yeilded the same result
### 'a') "reviewerID, productID, score"
### 'b') ['asin':'productID','overall':'score']
### 'c') ['customer_id', 'product_id', 'score']
### 'd') None of the above
### assign character - as string - associated with your choice to ans1
### YOUR ANSWER BELOW

ans1 = 'c'
```

While not strictly necessary, let's modify the values in product_id and customer_id to make them a little easier to read.

Below, building a dictionary for products.
Let the keys be all the unique values of productID, let the values be "P" followed by a number, 1-n, such that each product id is mapped to a unique string of the format ("P####"). Do the same for Reviewers, but use a "R" prefix.

```
pID_dict = {pID:'P'+ str(idx) for idx, pID in enumerate(rev_df['product_id'].unique()) }
rID_dict = {rID:'R'+ str(idx) for idx, rID in enumerate(rev_df['customer_id'].unique()) }

### Checking the values in the dictionaries below:
for k in list(pID_dict.keys())[:5]:
    print(k, ":",pID_dict[k])
print("length: ",len(pID_dict))
```

```
### Renaming of product and reviewer IDs

rev_df.loc[:, 'product_id'] = rev_df.loc[:,'product_id'].map(pID_dict)
rev_df.loc[:, 'customer_id'] = rev_df.loc[:,'customer_id'].map(rID_dict)

rev_df.head()
```

EDA

Having gone through the proess of cleaning the data, we will load in our more extensive synthetic data-set and perform some light EDA to look at the distributions of the data.

```
synth_data_path = "../resource/asnlib/publicdata/synthetic_reviews.csv"

rev_df = pd.read_csv(synth_data_path)
rev_df.head()
```

```
rev_df['score'].value_counts().plot(kind = 'bar');
```

```
rev_df['score'].value_counts()
```

Interesting preponderance of 5's.

```
print("Reviews per product, top and bottom reviewed\n")
print(rev_df['productID'].value_counts()[:15])
print(rev_df['productID'].value_counts()[-5:])
```

```
product_w_n_reviews = rev_df['productID'].value_counts().value_counts()
plt.figure(figsize = (10,7))
plt.scatter(product_w_n_reviews.index, product_w_n_reviews)
plt.xlabel("Number of Reviews", fontsize = 16)
plt.ylabel("Number of Products", fontsize = 16);
```

Of note, the above looks log-normal, however, in reality, the distribution will look like the distribution called "the hollow" curve, which is frequently found in nature in relative species abundance. the log-normal appearance is an artifact of the data synthesis.

```
print("Reviews per reviewer, top and bottom reviewers\n")
print(rev_df['reviewerID'].value_counts()[:15])
print(rev_df['reviewerID'].value_counts()[-5:])
```

```
product_w_n_reviews = rev_df['reviewerID'].value_counts().value_counts()
plt.figure(figsize = (10,7))
```

```
plt.scatter(product_w_n_reviews.index, product_w_n_reviews)
plt.xlabel("Number of Reviews", fontsize = 16)
plt.ylabel("Number of Users", fontsize = 16);
```

Most products are only reviewed a few times; most users only submit a few reviews. Very few products/users have/submit many many reviews. The above is more the look of a "hollow curve".

Functioning of Recommender Systems

Before tackling our extensive synthetic data, we will use a simpler dataset to demonstrate the functioning of recommender systems.

Below we have a set of rankings of six musicians made by six individuals.

```
ex = pd.read_csv("../resource/asnlib/publicdata/example.csv", index_col = 0)
ex
```

Every column and every row contains 1 value of "0", denoting an absence of ranking - NOT an extremely poor ranking.

To determine which products to recommend, the similarity of products will be used. We will use three different ways to calculate similarity: Euclidean distance, correlation coefficient (Pearson's), and a cosine similarity score. In order to compare these scores we will normalize each to range between 0 and 1.

To help review each, we will use a subset of our example DataFrame:

```
ex_hand = ex.iloc[:3,-3:]
ex_hand
```

Euclidean distance for points i, and j is $\sqrt{(a_j-a_i)^2+(b_j-b_i)^2+...+(n_j-n_i)^2}$

Question 2

```
### GRADED
### calculate the Euclidean distance between Brahms and Wagner in the `ex_hand` DataFrame
### Assign float answer to ans1.
### response graded to 3 decimal places
### Feel free to refer to KNN assignment
### YOUR ANSWER BELOW

ans1 = (1+1+4)**.5
```

To normalize this to between 0 and 1, the transformation from "distance" to "similarity" is $\frac1{1+dist}$. e.g. distance of 0 = similarity of 1, and $lim_{dist\rightarrow\infty}$ similarity approaches 0.

Question 3

```
### GRADED
### Code a function called "e_sim"
### ACCEPT two pandas series as inputs
### RETURN the euclidean similarity score

### Hint: remember `np.linalg.norm()`
### YOUR ANSWER BELOW

def e_sim(ser1, ser2):
    """
    Given two Pandas series, compute the euclidean similarity score;
        1 / 1+euclidean distance

    Positional Arguments --
        ser1: a Pandas Series of length n
        ser2: a Pandas Series of length n

    Example --
        ser1 = ex_hand.iloc[:,0]
        ser2 = ex_hand.iloc[:,1]
        print(e_sim(ser1, ser2)) #--> 0.28989794855663564
    """

    return 1/(1+np.linalg.norm(ser1-ser2))
```

Pearson's Correlation Coefficient

The correlation coefficient - ρ - that same one you learned about in intro to statistics - ranges from -1 to 1, thus in order to normalize into a difference score between 0 and 1, we need to reduce that range by half -- divide by two -- and increase the minimum score to 0 -- add .5:

$$\frac12 + \frac{\rho_{xy}}2$$

```
### Defining function for pearson's Correlation Coefficient
def p_sim(ser1, ser2):

    def normalize(raw):
        return .5 + (raw/2)

    corr = np.corrcoef(ser1, ser2)[0][1] ### returns 2x2 array with correlation to self(1) on diagonal

    return normalize(corr)
```

Cosine Similarity

Cosine similarity calculates the cosine of the angle between two vectors, once again, normalizing between 0 and 1.

Question 4

```
### GRADED
### True or False:
### Cosine similarity score will be unaffected by magnitudes
### e.g. an excited user who gives mostly 5's will be considered similar to a user who gives mostly 2's

### Assign boolean response to ans1
### YOUR ANSWER BELOW

ans1 = True
```

The equation for the cosine similarity score is the dot product of two vectors over the product of the norm of the two vectors: $$cos\theta = \frac{S1\cdot S2}{||S1||*||S2||}$$

```
### Define function for cosine similarity

def c_sim(ser1, ser2):
    def normalize(raw):
        return .5 + (raw/2)

    cosT = np.dot(ser1, ser2) / (np.linalg.norm(ser1)* np.linalg.norm(ser2))

    return normalize( cosT)
```

At this point we have defined three functions for calculating similarity between two vectors.

- c_sim: for cosine similarity
- p_sim: for pearson coorelation coefficient similarity
- e_sim: for euclidean similarity

Let's now calculate each of the three similarity scores for each pair of musicians in our `ex` DataFrame.

The function we will build must take into account that not all users have rated all musicians.

```
ex
```

So, for example, when "Mozart", and "Bach" are compared, the observations from both "Abel" and "Baker" must be disregarded.
Although our similarity functions will happily ingest the value of "0", The functions will take 0 to mean "less than 1" , *not* the "no information" which it actually represents.

Question 5

```
### GRADED
### Code a function called "drop_val"
### ACCEPT two inputs; A pandas DataFrame and a value, 'to_drop'
### Drop all of the *rows* that contain the value 'to_drop' in any column
### RETURN DataFrame with rows dropped

### e.g. As demonstrated in the doc-string;
### Passing a DataFrame based on 'ex' containing all rows but only the "Mozart" and "Bach" columns
### and a 'to_drop' of 5 should return the "Abel", "Erik", and "Frank" rows with all the original columns

### YOUR ANSWER BELOW

def drop_val (df, to_drop):

    """
    Drop rows from the DataFrame containing the specified values

    Positional Arguments --
        df: a Pandas DataFrame
        to_drop: a value found in some rows of df

    Example --

        df = ex.loc[:,"Mozart":"Bach"]
        to_drop = 5
        print(drop_val(df,to_drop)) # -->          Mozart  Bach
                                           Abel        0     1
                                           Erik        3     3
                                           Frank       2     2


    """
    keep = []
    for n in df.T:

        if to_drop not in df.T[n].unique():
            keep.append(n)
    return df.loc[keep,:]
```

The below cell defines a function "drop_rows_with_zeros" that ingests a DataFrame of two columns and returns a DataFrame where all the rows that contain a value of 0 have been removed

```
### Defining drop_rows_with_zeros

def drop_rows_with_zeros(df):
    keep = np.intersect1d( np.nonzero(df.iloc[:,0]), np.nonzero(df.iloc[:,1]))

    return df.iloc[keep,:]
```

Comparison of Similarity Scores

Below, the similarity scores for each each combination of musicians will be compared for each similarity score.

Find all the pairs of musicians:

```
import itertools
for c in itertools.combinations(ex.columns,2):
    print(c)
```

Check to see how the tuples can be used to subset the DataFrame.

```
ex[list(('Mozart', 'Bach'))]
```

Building a dictionary with a key of the musicians being compared, and Value of another dictionary.
In that second dictionary, keys of "Euclid", "Pearson", and "Cosine" with values of similarity score calculated with relevant method

```
sim_scores = dict()

for c in itertools.combinations(ex.columns, 2):
    df = drop_rows_with_zeros(ex[list(c)])
    ser1 = df.iloc[:,0]
    ser2 = df.iloc[:,1]
    scores = {"Euclid":e_sim(ser1, ser2), "Pearson":p_sim(ser1,ser2), "Cosine":c_sim(ser1,ser2)}
    key =", ".join(c)
    sim_scores[key] = scores

sims = pd.DataFrame.from_dict(sim_scores,orient = "index")
sims
```

```
### Note: X and y scales are not consistent in this visualization
sns.pairplot(sims );
```

```
### Visualization with consistent scales using `matplotlib`
plt.figure(figsize = (12,6))
for i, cols in enumerate([('Euclid','Pearson'), ("Euclid","Cosine"),("Pearson","Cosine")]):
    plt.subplot(1,3,i+1)

    plt.scatter(sims[cols[0]], sims[cols[1]])

    plt.xlabel(cols[0], fontsize = 14); plt.ylabel(cols[1], fontsize = 14)
    plt.xlim(0,1); plt.ylim(0,1)

plt.tight_layout()
```

Differences in Similarity Scores

One final technique - not implemented above - is to normalize scores before calculating a euclidean similarity score. For example, a the mean score might be subtracted to center all scores around 0. This would help to spread the Euclidean similarity scores out over the entire length of 0-1 instead of clustering them below ~.6

As should be evidenced by the above graphics, it is possible to see a range of similarity scores given different techniques. These similarity scores will majorly impact how a recommender system performs. Please refer to the lectures for the particular impacts and strengths of the different similarity calculations.

```
ex
```

Predicting Unknown Scores

Question 6

In predicting Abel's score for Mozart:

Mozart has a similarity score in relation to Bach.
Mozart has a similarity score in relation to Chopin.
Mozart has a similarity score in relation to Brahms.
etc. etc.

Abel has scored Bach, Chopin, Brahms, etc.

So Abel's predicted score for Mozart will **sum** Abel's score of every other musician times that musician's similarity score with respect to Mozart and **divide by** the sum of similarity scores.

e.g.

We can predict Abel's score for Mozart by:

$$\frac{\sum_{ mus = Bach}^{Liszt}A_{mus}*simScore(\text{Mozart, mus})}{\sum_{mus = Bach}^{Liszt}simScore{\text{(Mozart, mus)}}}$$

```
mozSimScores = {}
for mus in ex.columns[1:]:
    no_zeros = drop_rows_with_zeros(ex[['Mozart', mus]])

    mozSimScores[mus] = round(p_sim(no_zeros.iloc[:,0], no_zeros.iloc[:,1]),2)
```

```
print(mozSimScores)
```

```
### GRADED
### Given the similarity scores calculated and stored (and rounded) in `mozSimScores`,
### use the above formula for predicting
### a score [(sum of <ranking * similarity scores>)/ sum of similarity scores]
### to predict Abel's score for Mozart.

### USE ROUNDED SCORES PRINTED / SAVED ABOVE

### Store in ans1
### YOUR ANSWER BELOW

sScores = []; abelS = []
for k in mozSimScores:
    ss = mozSimScores[k]
    sScores.append(ss)
    abelS.append(ss*ex.loc['Abel',k])
ans1 = sum(abelS)/ sum(sScores)
```

Even though many other users rate Mozart very highly, Abel appears to be a bit more negative in his reviews. Thus, his predicted score for Mozart is also fairly low.

```
### Creating a function to create predictions in the manner described above.
### Note: "user" is a row, and "item" is a column in the df.
### simFunc defaults to p_sim

def scorePred(df, user, item, simFunc = p_sim, rev_items = None):
    """
    Positional Arguments --
        df: Pandas DataFrame
        user: Row-index in df
        item: Column-index in df
        simFunc: a similarity function (e_sim, p_sim, or c_sim)
        rev_items: For larger dfs, specifies all items reviewed by "user"
    """
    # Check to see if user has already scored item
    if df.loc[user,item] > 0:
        return "Already rated a "+str(df.loc[user,item])

    # rev_items used for larger dataframes,
    # when you have all the items a particular user has reviewed already
    # otherwise, if "None" (if statement below)
    # take all other items other than item to predict
    if not rev_items:
        rev_items = set(df.columns)
        rev_items.remove(item)

    sim_total, user_sim_total = 0,0

    for other_item in rev_items:
        user_score_other_item = df.loc[user, other_item] # grab user score

        if user_score_other_item == 0:
            print("no user score")
            continue

        # Use function built above to drop all other users with "other_item" score of "0"
        no_zeros = drop_rows_with_zeros(df[[item, other_item]])
        sh = no_zeros.shape

        # If no other users, move to next item
        if sh[0] == 0:
            continue

        #print(no_zeros.shape)

        # calculate similarity score using non-zero information.
        ser1 = no_zeros.iloc[:,0]
        ser2 = no_zeros.iloc[:,1]
        # print(ser1, ser2)
        ss = simFunc(no_zeros.iloc[:,0], no_zeros.iloc[:,1])
        # print(ss, user_score_other_item)

        # add up sim total and weighted sim total
        sim_total += ss
        user_sim_total += user_score_other_item * ss


    if sim_total == 0:
        return 0

    else:
        return user_sim_total / sim_total
```

```
ex
```

Use the below cells to play around with the `scorePred` function we have built out.

```
scorePred(ex, 'Baker', 'Bach')
```

Returning to the Data

Let's remind ourselves what the music rating data looks like

```
rev_df.head(10)
```

The data does not conform to the n-by-p matrix that we have been using, let's take a look at how we will need to manipulate our data. To do so we will use `Pandas'` built in `pivot()` method

```
# reviewers on rows, item on cols.
reshaped_reviews = rev_df.pivot(index = 'reviewerID', columns = 'productID')
reshaped_reviews.fillna(0, inplace = True) # turns NA into 0
reshaped_reviews.columns = reshaped_reviews.columns.droplevel() # removes artifact of pivot function


print("shape: ", reshaped_reviews.shape)
reshaped_reviews.head()
```

As should be clear from looking at even just the head of our new DataFrame, the majority of (reviewer, product) locations are empty. Thus, when trying to provide recommendations, it will not be possible to predict scores for every (reviewer, product) combination. To predict scores, a certain amount of overlap is required.

Below predictions are created for our reviewer called "R0"

```
### First, find the products that 'R0' has reviewed.
### These are the products that will be used for creating similarity scores.

r0 = reshaped_reviews.loc["R0", :]
r0_nonzero = np.nonzero(r0) # numpy function returns all indexes that are not "0"
r0_rev_prods = list(r0.iloc[r0_nonzero].index) # collect names of reviewed products
print(len(r0_rev_prods))
r0_rev_prods[:5]
```

```
### Creating a list of products not reviewed by R0

# return list of all prods not in the "reviewed" list
r0_not_rev = np.setdiff1d(reshaped_reviews.columns, r0_rev_prods)

print(len(r0_not_rev))
r0_not_rev
```

The below call requires too much processing for Vocareum. Output displayed below:

```
# %%time
# ### predict scores for products not yet reviewed by R0, using function built above: `scorePred`
# ### cosine similarity used because if a product all has only one value of review (e.g. all 5s)
# ### then the standard_deviation is 0 and the correlation coefficient is not calculable

# preds = {}
# for item in r0_not_rev:
#     preds[item] = scorePred(reshaped_reviews, 'R0', item, simFunc = c_sim, rev_items = r0_rev_prods)
```

```
# pd.Series(preds).sort_values(ascending = False).head(10)
```

Running the above cells took the local machine ~7.5'. results in the picture below.

```
In [379]:
%%time
### predict scores for products not yet reviewed by R0, using function built above: scorePred
### cosine similarity used because if a product all has only one value of review (e.g. all 5s)
### then the standard_deviation is 0 and the correlation coefficient is not calculable

preds = {}
for item in r0_not_rev:
    preds[item] = scorePred(reshaped_reviews, 'R0', item, simFunc = c_sim, rev_items = r0_rev_prods)

Wall time: 7min 23s
```

```
In [380]:
pd.Series(preds).sort_values(ascending = False).head(10)

Out[380]:
P664    5.0
P1400   5.0
P843    5.0
P3062   5.0
P2171   5.0
P966    5.0
P1444   5.0
P688    5.0
P3283   5.0
P3282   5.0
dtype: float64
```

Utilization Complexities

Certainly if there are products that a user is likely to score a "5", that product should be recommended. However, because we are dealing with very sparse data, the confidence in our predictions will vary between product.

Question 7

```
### GRADED
### consider 3 products: A, B, and X
### userN has ranked products A, and B, giving each a '5' and we are predicting their score for X
### user1 has ranked products A, and X, giving each a "3"
### user2 has ranked products B and X, giving each a '2'
```

```
### Assume this is all the data we have for a product.
### What will be the similarity score between prodA and prodX? assign number to ans1
### What will be the similarity score between prodB and prodX? assign number to ans2
### YOUR ANSWER BELOW

ans1 = 1
ans2 = 1
```

Question 8

```
### GRADED
### given your above answer, what will be the predicted score for userN's evaluation of prodX?
### assign number to ans1
### YOUR ANSWER BELOW

ans1 = 5
```

For many of these high predicted scores, some version of the above two questions will have occurred: high similarity scores due to few overlapping reviews, and high-user score for the user who is being predicted for.

So while the "5's" might be higher than a 4.89, if that 4.89 is based on more data, it's probably a better prediction.

SVD

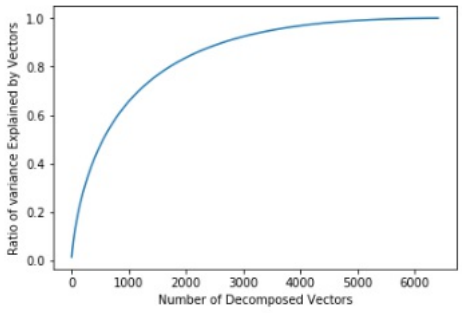
One way we will combat this sparsity of data, and increase the speed of our algorithm is by using Singular-Value Decomposition (SVD). SVD is a demensionality reduction technique which functions by "capturing" a majority of the information present in a data set in a smaller number of variables.

SVD relies upon some straight-forward linear algebra that is worth understanding, and has already been covered in lectures. Just incase you want more resources:

Here is a nice introduction.

Below, we will use `numpy's` implementation of SVD

The "Sigma" variable, when squared describes how much variance is described by each vector. Below a plot of % variance explained by n vectors as n -> total number of products



Around 1700 of these decomposed vectors will explain around 80% of the variance in the data. Computationally, this will make the recommendation calculations much faster: less than 1/3 of the computations will yeild 80% of the value.

```
# # Perform SVD
# u, sigma, _ = np.linalg.svd(reshaped_reviews)

# # Create function to decompose DataFrame
# def return_svd(df , u, sigma, n):

#     sigN = np.mat(np.eye(n) * sigma[:n]) #arrange Sig4 into a diagonal matrix

#     n_svd_vectors = np.dot(df.T, np.dot(u[:,:n] , sigN.I))  #create transformed items

#     return pd.DataFrame(n_svd_vectors).T

# # use function to create decomposed dataframe
# svd_df = return_svd(reshaped_reviews, u, sigma, 1700)
# svd_df.columns = reshaped_reviews.columns

# # Save dataframe

# svd_df.to_csv("svd_df.csv")
```

The above takes too long to run. Thus the results are provided from a .csv found below.

```
svd_df = pd.read_csv("../resource/asnlib/publicdata/svd_df.csv", index_col = 0)
svd_df.head()
```

As the shape of the `svd_df` indicates, these svd-vectors correspond to items. The calculation for recommendation will be the same, but just using these new values.

For example, to predict the score that reviewer "R0" would give to 'P1' (Remember "R0" already reviewed "P0") involves:

1. Find the products that R0 scored.
2. Find the similarity between those scored products and "P1"
3. Multiply those similarity scores by the scores from R0
4. Divide by sum of similarity scores.

The downside from SVD is that the SVD must be performed before the calculations, and as was seen above, those SVD calculations take some time. Potentially, SVD would need to be performed each time a new review is added.

Finally it will be possible to create a prediction and similarities for every pair of items, which will increase computation on its own, but will hopefully give more meaningful recommendations.

```
### Creating a function to create predictions using the SVD_df
### Note: "user" is a row, and "item" is a column in the df, and a column in svd_df.
### simFunc defaults to p_sim

def scorePredSVD(df, user, item, svd_df, simFunc = p_sim, rev_items = None):

    # Check to see if user has already scored item
    if df.loc[user,item] > 0:
        return "Already rated a "+str(df.loc[user,item])

    # Code below should be familiar from "scorePred" defined above
    if not rev_items:
        rev_items = set(df.columns)
        rev_items.remove(item)

    sim_total, user_sim_total = 0,0

    for other_item in rev_items:
        user_score_other_item = df.loc[user, other_item]

        if user_score_other_item == 0:
            print("no user score")
            continue

        ser1 = svd_df.loc[:,item]
        ser2 = svd_df.loc[:,other_item]
        # print(ser1, ser2)
        ss = simFunc(ser1, ser2)
        # print(ss, user_score_other_item)
        sim_total += ss
        user_sim_total += user_score_other_item * ss


    if sim_total == 0:
        return 0

    else:
        return user_sim_total / sim_total
```

```
%%time
### predict scores for products not yet reviewed by R0, using function built above: scorePred
### cosine similarity used because if a product all has only one value of review (e.g. all 5s)
### then the standard_deviation is 0 and the correlation coefficient is not calculable

SVDpreds = {}
for item in r0_not_rev:
    SVDpreds[item] = scorePredSVD(reshaped_reviews, 'R0', item, svd_df, simFunc = c_sim, rev_items = r0_rev_prods)
```

```
pd.Series(SVDpreds).sort_values(ascending = False).head()
```

Question 9

```
### GRADED
### SVD is a kind of:
### 'a') variance reduction
### 'b') special type of array in Python
### 'c') dimensionality reduction
### 'd') description of central tendency
### assign character associated with your choice as a string to ans1
### YOUR ANSWER BELOW

ans1 = 'c'
```

Surprise - Python Package

`Surprise` is a Python package which implements a number of tools for building and testing recommender systems.Surprise documentation

THE FOLLOWING CODE WILL NOT RUN ON VOCAREUM THEY ARE EXAMPLES SHOULD YOU CHOOSE TO USE `SURPRISE` IN YOUR OWN ENVIRONMENT

You may have to run:

`conda install -c conda-forge scikit-surprise`

In `Anaconda Prompt`, if it is not already installed on your computer, and possibly reinstall/update scipy, and restart Kernel:

`conda install -c anaconda scipy`

Below is example code from the surprise documentation

```
# from surprise import SVD
# from surprise import Dataset
# from surprise.model_selection import cross_validate


# # Load the movielens-100k dataset (download it if needed),
# data = Dataset.load_builtin('ml-100k')

# # We'll use the famous SVD algorithm.
```

```
# algo = SVD()

# # Run 5-fold cross-validation and print results
# cross_validate(algo, data, measures=['RMSE', 'MAE'], cv=5, verbose=True)
```

Loading in a df

```
# import os
# from surprise import BaselineOnly
# from surprise import Reader

# # when using a df to load in data, the columns must be:
# # user, product, rating, in that order
# # a reader must be defined to describe the rating_scale
# reader = Reader(rating_scale=(0,5))

# data = Dataset.load_from_df(rev_df, reader=reader)

# # We can now use this dataset as we please, e.g. calling cross_validate
# cross_validate(BaselineOnly(), data, verbose=True)
```

One final note:

This project has based its recommendations off of camparing the similarity of *items*. It is also possible to, using the same techniques, calculate the similarity of *users*. The reason that Items w as chosen here (and is usually chosen) is that there are usually few er items w hich demand separate similarity calculations. This w as seen in our data set as there w ere roughly 3.5k items and 5.5k users. When using complete data from a source, this difference w ill be even more significant. Of course, the more people w hich need to have similarity scores calculated from, the longer time to build a recommender system.