Python Data Products

Course 4: Implementing and Deploying data-driven predictive systems

Lecture: Implementing a similarity-based recommender

Learning objectives

In this lecture we will...

- Implement a simple recommender system that recommends products based on the Jaccard similarity

Code: Reading the data

First we read the data. Note we use a larger dataset for this exercise, though could use a smaller one if running time is an issue.

In [1]:
```
import gzip
from collections import defaultdict
import random
import numpy
import scipy.optimize
```

In [2]:
```
path = "/home/jmcauley/datasets/mooc/amazon/amazon_reviews_us_Musical_Instruments_v1_00.tsv.gz"
```

In [3]:
```
f = gzip.open(path, 'rt', encoding="utf8")
```

In [4]:
```
header = f.readline()
header = header.strip().split('\t')
```

Code: Reading the data

Our goal is to make recommendations of products based on users' purchase histories. The only information needed to do so is **user and item IDs**

In [5]:
```
dataset = []
```

In [6]:
```
for line in f:
    fields = line.strip().split('\t')
    d = dict(zip(header, fields))
    d['star_rating'] = int(d['star_rating'])
    d['helpful_votes'] = int(d['helpful_votes'])
    d['total_votes'] = int(d['total_votes'])
    dataset.append(d)
```

In [7]:
```
dataset[0]
```

Out[7]:
```
{'marketplace': 'US',
 'customer_id': '45610553',
 'review_id': 'RMDCHWD0Y5OZ9',
 'product_id': 'B00HH62VB6',
 'product_parent': '618218723',
 'product_title': 'AGPtek® 10 Isolated Output 9V 12V 18V Guitar Pedal Board Power Supply Effect Pedals
with Isolated Short Cricuit / Overcurrent Protection',
```

Code: Useful data structures

To perform set intersections/unions efficiently, we first build data structures representing the set of items for each user and users for each item

In [8]:
```
# Useful data structures
```

In [9]:
```
usersPerItem = defaultdict(set)
itemsPerUser = defaultdict(set)
```

U_i

I_u

In [10]:
```
itemNames = {}
```

In [11]:
```
for d in dataset:
    user,item = d['customer_id'], d['product_id']
    usersPerItem[item].add(user)
    itemsPerUser[user].add(item)
    itemNames[item] = d['product_title']
```

Code: Jaccard similarity

The Jaccard similarity implementation follows the definition directly:

$$\text{Jaccard}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

In [12]:
```
def Jaccard(s1, s2):
    numer = len(s1.intersection(s2))
    denom = len(s1.union(s2))
    return numer / denom
```

Recommendation

We want a recommendation function that return **items similar to a candidate item *i*.** Our strategy will be as follows:

- Find the set of users who purchased *i*
- Iterate over all other items other than *i*
- For all other items, compute their similarity with *i (and store it)*
- Sort all other items by (Jaccard) similarity
- Return the most similar

Code: Recommendation

Now we can implement the recommendation function itself:

In [13]:
```
def mostSimilar(i):
    similarities = []
    users = usersPerItem[i]
    for i2 in usersPerItem:
        if i2 == i: continue
        sim = Jaccard(users, usersPerItem[i2])
        similarities.append((sim,i2))
    similarities.sort(reverse=True)
    return similarities[:10]
```

$$\text{Jaccard}(U_i, U_j) = \frac{|U_i \cap U_j|}{|U_i \cup U_j|}$$

Code: Recommendation

Next, let's use the code to make a recommendation. The query to the system is just a product ID:

```
In [14]: dataset[2]

Out[14]: {'marketplace': 'US',
          'customer_id': '6111003',
          'review_id': 'RIZR67JKUDBI0',
          'product_id': 'B0006VMBHI',
          'product_parent': '603261968',
          'product_title': 'AudioQuest LP record clean brush',
          'product_category': 'Musical Instruments',
          'star_rating': 3,
          'helpful_votes': 0,
          'total_votes': 1,
          'vine': 'N',
          'verified_purchase': 'Y',
          'review_headline': 'Three Stars',
          'review_body': 'removes dust. does not clean',
          'review_date': '2015-08-31'}

In [15]: query = dataset[2]['product_id']
```

Code: Recommendation

Next, let's use the code to make a recommendation. The query to the system is just a product ID:

```
In [16]: mostSimilar(query)

Out[16]: [(0.028446389496717725, 'B00006I5SD'),
          (0.01694915254237288, 'B00006I5SB'),
          (0.015065913370998116, 'B000AJR482'),
          (0.014204545454545454, 'B00E7MVP3S'),
          (0.008955223880597015, 'B001255YL2'),
          (0.008849557522123894, 'B003EIRVO8'),
          (0.008333333333333333, 'B0015VEZ22'),
          (0.00821917808219178, 'B00006I5UH'),
          (0.008021390374331552, 'B00008BWM7'),
          (0.007656967840735069, 'B000H2BC4E')]
```

Code: Recommendation

Finally, let's look at the items that were recommended:

In [17]:
```
itemNames[query]
```

Out[17]:
```
'AudioQuest LP record clean brush'
```

In [18]:
```
[itemNames[x[1]] for x in mostSimilar(query)]
```

Out[18]:
```
['Shure SFG-2 Stylus Tracking Force Gauge',
 'Shure M97xE High-Performance Magnetic Phono Cartridge',
 'ART Pro Audio DJPRE II Phono Turntable Preamplifier',
 'Signstek Blue LCD Backlight Digital Long-Playing LP Turntable Stylus Force Scale Gauge Tester',
 'Audio Technica AT120E/T Standard Mount Phono Cartridge',
 'Technics: 45 Adaptor for Technics 1200 (SFWE010)',
 'GruvGlide GRUVGLIDE DJ Package',
 'STANTON MAGNETICS Record Cleaner Kit',
 'Shure M97xE High-Performance Magnetic Phono Cartridge',
 'Behringer PP400 Ultra Compact Phono Preamplifier']
```

Recommending more efficiently

Our implementation was not very efficient. The slowest component is the iteration over all other items:

- Find the set of users who purchased *i*
- **Iterate over all other items other than *i***
- For all other items, compute their similarity with *i (and store it)*
- Sort all other items by (Jaccard) similarity
- Return the most similar

This can be done more efficiently as most items will have no overlap

Recommending more efficiently

In fact it is sufficient to iterate over **those items purchased by one of the users who purchased *i***

- Find the set of users who purchased *i*
- **Iterate over all users who purchased *i***
- Build a candidate set from all items those users consumed
- For items in this set, compute their similarity with *i (and store it)*
- Sort all other items by (Jaccard) similarity
- Return the most similar

Code: Faster implementation

Our more efficient implementation works as follows:

In [19]:
```
def mostSimilarFast(i):
    similarities = []
    users = usersPerItem[i]
    candidateItems = set()
    for u in users:
        candidateItems = candidateItems.union(itemsPerUser[u])
    for i2 in candidateItems:
        if i2 == i: continue
        sim = Jaccard(users, usersPerItem[i2])
        similarities.append((sim,i2))
    similarities.sort(reverse=True)
    return similarities[:10]
```

Code: Faster recommendation

Which ought to recommend the same set of items, but **much** more quickly:

```
In [20]: mostSimilarFast(query)

Out[20]: [(0.028446389496717725, 'B00006I5SD'),
          (0.01694915254237288, 'B00006I5SB'),
          (0.015065913370998116, 'B000AJR482'),
          (0.014204545454545454, 'B00E7MVP3S'),
          (0.008955223880597015, 'B001255YL2'),
          (0.008849557522123894, 'B003EIRVO8'),
          (0.008333333333333333, 'B0015VEZ22'),
          (0.00821917808219178, 'B00006I5UH'),
          (0.008021390374331552, 'B00008BWM7'),
          (0.007656967840735069, 'B000H2BC4E')]
```

Summary of concepts

- Implemented a similarity-based recommender based on the Jaccard similarity
- Showed how to make our implementation more efficient

On your own...

- Our code recommends *items* that are similar to a given item. Adapt it to recommend *users* who are similar to a given user
- Typcially we want to recommender items similar to one to which a user has already given a high rating (e.g. "you'll like X because you liked Y"). Adapt our code so that it takes as input a given *user*, and recommends items similar to those that user liked