Credit: NewsWire, The TakeOut.

How to Build a Recommendation System for Purchase Data (Step-by-Step)

An application of item-based collaborative filtering with Turicreate and Python

Moorissa Tjokro Follow
Oct 15, 2018 · 13 min read ★

Whether you are responsible for user experience and product strategy in a customer centric company, or sitting in your couch watching movies with loved ones, chances are you are already aware of some ways that recommendation technology is used to personalize your content and offers.

Recommendation systems are one of the most common, easily comprehendible applications of big data and machine learning. Among the

most known applications are Amazon's recommendation engine that provides us with a personalized webpage when we visit the site, and Spotify's recommendation list of songs when we listen using their app.

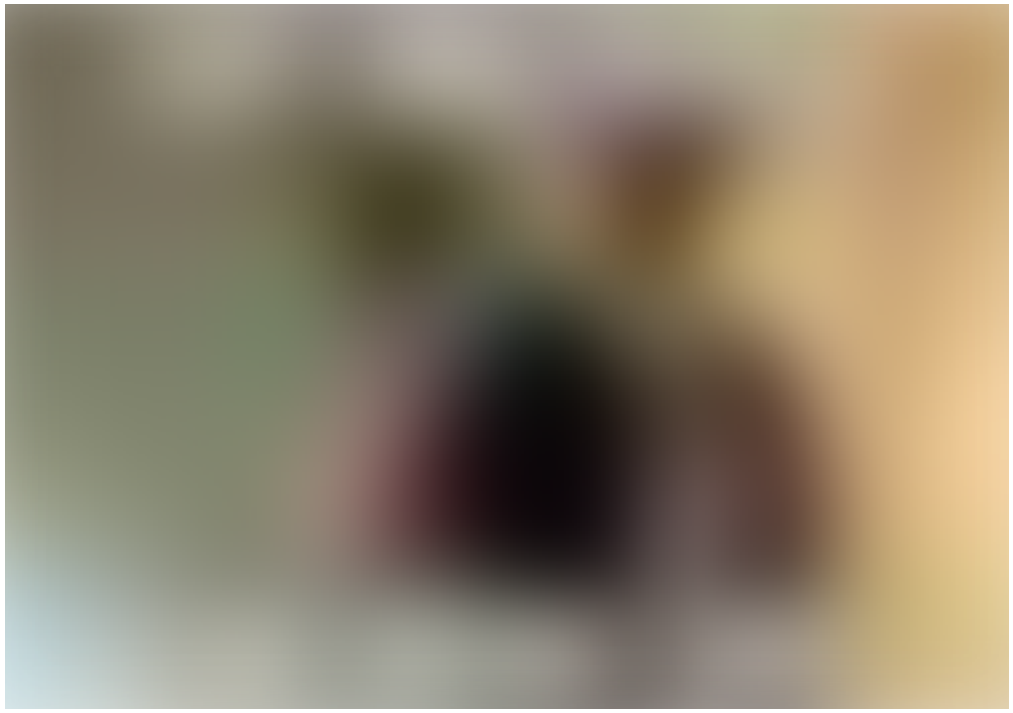
Last time, we got to build a Spotify's Discover Weekly with a bulk of audio data using Spark. This time, we'll build a recommendation engine for more tangible items.

The Challenge

If you look up online, there are many ways to build recommendation systems for *rating-based* data, such as movies and songs. The problem with rating-based models is that they couldn't be standardized easily for data with non-scaled target values, such as purchase or frequency data. For example, ratings are usually from 0–5 or 0–10 across songs and movies. However, purchase data is continuous and without an upper bound.

A lot of online resources unfortunately provide results without evaluating their models. For most data scientists and engineers, this is a dangerous

area when you're involving millions of data! For industries, results alone won't get your tools anywhere without any evaluation.



The Goal

In solving these problems, we will build collaborative filtering models for recommending products to customers using purchase data. In particular, we'll cover in details the step-by-step process in constructing a

recommendation system with Python and machine learning module Turicreate. These steps include:

- Transforming and normalizing data
- Training models
- Evaluating model performance
- Selecting the optimal model

Product Overview

Imagine a grocery chain releases a new mobile app allowing its customers to place orders before they even have to walk into the store.

There is an opportunity for the app to show recommendations: When a customer first taps on the “order” page, we may recommend top 10 items to be added to their basket, e.g. disposable utensils, fresh meat, chips, and and so on.

The tool will also be able to search for a recommendation list based on a specified user, such that:

- Input: customer ID
- Returns: ranked list of items (product IDs), that the user is most likely to want to put in his/her (empty) “basket”

Implementation

1. Import modules

- `pandas` and `numpy` for data manipulation
- `turicreate` for performing model selection and evaluation
- `sklearn` for splitting the data into train and test set

```
%load_ext autoreload
%autoreload 2

import pandas as pd
import numpy as np
import time
import turicreate as tc
from sklearn.cross_validation import train_test_split

import sys
sys.path.append("..")
```

2. Load data

Two datasets in .csv format are used below, which can be found in `data` folder *here*:

- `recommend_1.csv` consisting of a list of 1000 customer IDs to recommend as output
- `trx_data.csv` consisting of user transactions

```
customers = pd.read_csv('../data/recommend_1.csv')
transactions = pd.read_csv('../data/trx_data.csv')
```

3. Data preparation

Our goal here is to break down each list of items in the `products` column into rows and count the number of products bought by a user

3.1. Create data with user, item, and target field

- This table will be an input for our modeling later
- In this case, our user is `customerId`, `productId`, and `purchase_count`

```
data = pd.melt(transactions.set_index('customerId')
['products'].apply(pd.Series).reset_index(),
             id_vars=['customerId'],
             value_name='products') \
    .dropna().drop(['variable'], axis=1) \
    .groupby(['customerId', 'products']) \
    .agg({'products': 'count'}) \
    .rename(columns={'products': 'purchase_count'}) \
    .reset_index() \
    .rename(columns={'products': 'productId'})
data['productId'] = data['productId'].astype(np.int64)
```

3.2. Create dummy

- Dummy for marking whether a customer bought that item or not.
- If one buys an item, then `purchase_dummy` are marked as 1
- Why create a dummy instead of normalizing it, you ask? Normalizing the purchase count, say by each user, would not work because customers may have different buying frequency don't have the same taste. However, we can normalize items by purchase frequency across all users, which is done in section 3.3. below.

```
def create_data_dummy(data):
    data_dummy = data.copy()
```

```
    data_dummy['purchase_dummy'] = 1
    return data_dummy


data_dummy = create_data_dummy(data)
```

3.3. Normalize item values across users

- To do this, we normalize purchase frequency of each item across users by first creating a user-item matrix as follows

```
df_matrix = pd.pivot_table(data, values='purchase_count',
index='customerId', columns='productId')
```

```
df_matrix_norm = (df_matrix-df_matrix.min())/(df_matrix.max()-
df_matrix.min())
```

```
# create a table for input to the modeling


d = df_matrix_norm.reset_index()
d.index.names = ['scaled_purchase_freq']
data_norm = pd.melt(d, id_vars=['customerId'],
value_name='scaled_purchase_freq').dropna()


print(data_norm.shape)
data_norm.head()
```

The above steps can be combined to a function defined below:

```
def normalize_data(data):
    df_matrix = pd.pivot_table(data, values='purchase_count',
index='customerId', columns='productId')
    df_matrix_norm = (df_matrix-df_matrix.min())/(df_matrix.max()-
df_matrix.min())
    d = df_matrix_norm.reset_index()
    d.index.names = ['scaled_purchase_freq']
    return pd.melt(d, id_vars=['customerId'],
value_name='scaled_purchase_freq').dropna()
```

In this step, we have normalized the their purchase history, from 0–1 (with 1 being the most number of purchase for an item and 0 being 0 purchase count for that item).

4. Split train and test set

- Splitting the data into training and testing sets is an important part of evaluating predictive modeling, in this case a collaborative filtering model. Typically, we use a larger portion of the data for training and a smaller portion for testing.
- We use 80:20 ratio for our train-test set size.
- Our training portion will be used to develop a predictive model, while the other to evaluate the model's performance.

Let's define a splitting function below.

```
def split_data(data):
    '''
    Splits dataset into training and test set.

    Args:
        data (pandas.DataFrame)

    Returns
        train_data (tc.SFrame)
        test_data (tc.SFrame)
    '''
    train, test = train_test_split(data, test_size = .2)
    train_data = tc.SFrame(train)
    test_data = tc.SFrame(test)
    return train_data, test_data
```

Now that we have three datasets with purchase counts, purchase dummy, and scaled purchase counts, we would like to split each for modeling.

```
train_data, test_data = split_data(data)
train_data_dummy, test_data_dummy = split_data(data_dummy)
train_data_norm, test_data_norm = split_data(data_norm)
```

5. Define Models using Turicreate library

Before running a more complicated approach such as collaborative filtering, we should run a baseline model to compare and evaluate models. Since baseline typically uses a very simple approach, techniques used beyond this approach should be chosen if they show relatively better accuracy and complexity. In this case, we will be using popularity model.

A more complicated but common approach to predict purchase items is collaborative filtering. I will discuss more about the popularity model and collaborative filtering in the later section. For now, let's first define our variables to use in the models:

```
# constant variables to define field names include:


user_id = 'customerId'
item_id = 'productId'
users_to_recommend = list(customers[user_id])
n_rec = 10 # number of items to recommend
n_display = 30 # to display the first few rows in an output dataset
```

`Turicreate` has made it super easy for us to call a modeling technique, so let's define our function for all models as follows:

```
def model(train_data, name, user_id, item_id, target,
users_to_recommend, n_rec, n_display):
    if name == 'popularity':
        model = tc.popularity_recommender.create(train_data,
                                                 user_id=user_id,
                                                 item_id=item_id,
                                                 target=target)
    elif name == 'cosine':
        model = tc.item_similarity_recommender.create(train_data,
                                                 user_id=user_id,
                                                 item_id=item_id,
                                                 target=target,

similarity_type='cosine')


elif name == 'pearson':
        model = tc.item_similarity_recommender.create(train_data,
                                                 user_id=user_id,
```

```
                                                    item_id=item_id,
                                                    target=target,

similarity_type='pearson')

    recom = model.recommend(users=users_to_recommend, k=n_rec)
    recom.print_rows(n_display)
    return model
```

While I wrote python scripts for all the above process including finding similarity using python scripts (which can be found *here*, we use `turicreate` library for now to capture different measures faster and evaluate models.

6. Popularity Model as Baseline

- The popularity model takes the most popular items for recommendation. These items are products with the highest number of sells across customers.
- Training data is used for model selection

i. Using purchase count

```
name = 'popularity'
target = 'purchase_count'
popularity = model(train_data, name, user_id, item_id, target,
users_to_recommend, n_rec, n_display)
```

ii. Using purchase dummy

```
name = 'popularity'
target = 'purchase_dummy'
pop_dummy = model(train_data_dummy, name, user_id, item_id, target,
users_to_recommend, n_rec, n_display)
```

iii. Using scaled purchase count

```
name = 'popularity'
target = 'scaled_purchase_freq'
pop_norm = model(train_data_norm, name, user_id, item_id, target,
users_to_recommend, n_rec, n_display)
```

6.1. Baseline Summary

- Once we created the model, we predicted the recommendation items using scores by popularity. As you can tell for each model results above, the rows show the first 30 records from 1000 users with 10 recommendations. These 30 records include 3 users and their recommended items, along with score and descending ranks.
- In the result, although different models have different recommendation list, each user is recommended the same list of 10 items. This is because popularity is calculated by taking the most popular items across all users.
- If a grouping example below, products 132, 248, 37, and 34 are the most popular (best-selling) across customers. Using their purchase counts divided by the number of customers, we see that these products are at least bought 3 times on average in the training set of transactions (same as the first popularity measure on `purchase_count` variable)

7. Collaborative Filtering Model

In collaborative filtering, we would recommend items based on how similar users purchase items. For instance, if customer 1 and customer 2 bought similar items, e.g. 1 bought X, Y, Z and 2 bought X, Y, we would recommend an item Z to customer 2.

7.1. Methodology

To define similarity across users, we use the following steps:

1. Create a user-item matrix, where index values represent unique customer IDs and column values represent unique product IDs

2. Create an item-to-item similarity matrix. The idea is to calculate how similar a product is to another product. There are a number of ways of calculating this. In steps 7.2 and 7.3, we use `cosine` or `pearson` similarity measure, respectively.

- To calculate similarity between products X and Y, look at all customers who have rated both these items. For example, both X and Y have been rated by customers 1 and 2.
- We then create two item-vectors, v1 for item X and v2 for item Y, in the user-space of (1, 2) and then find the `cosine` or `pearson` angle/distance between these vectors. A zero angle or overlapping vectors with cosine value of 1 means total similarity (or per user, across all items, there is same rating) and an angle of 90 degree would mean cosine of 0 or no similarity.

3. For each customer, we then predict his likelihood to buy a product (or his purchase counts) for products that he had not bought.

- For our example, we will calculate rating for user 2 in the case of item Z (target item). To calculate this we weigh the just-calculated similarity-measure between the target item and other items that customer has already bought. The weighing factor is the purchase counts given by the user to items already bought by him.
- We then scale this weighted sum with the sum of similarity-measures so that the calculated rating remains within a predefined limits. Thus, the predicted rating for item Z for user 2 would be calculated using similarity measures.

7.2. `Cosine` similarity

- Similarity is the cosine of the angle between the 2 vectors of the item vectors of A and B
- It is defined by the following formula

- Closer the vectors, smaller will be the angle and larger the cosine

i. Using purchase count

```
name = 'cosine'
target = 'purchase_count'
cos = model(train_data, name, user_id, item_id, target,
users_to_recommend, n_rec, n_display)
```

ii. Using purchase dummy

```
name = 'cosine'
target = 'purchase_dummy'
cos_dummy = model(train_data_dummy, name, user_id, item_id, target,
users_to_recommend, n_rec, n_display)
```

iii. Using scaled purchase count

```
name = 'cosine'
target = 'scaled_purchase_freq'
cos_norm = model(train_data_norm, name, user_id, item_id, target,
users_to_recommend, n_rec, n_display)
```

7.3. `Pearson` similarity

- Similarity is the pearson coefficient between the two vectors.
- It is defined by the following formula

i. Using purchase count

```
name = 'pearson'
target = 'purchase_count'
pear = model(train_data, name, user_id, item_id, target,
users_to_recommend, n_rec, n_display)
```

ii. Using purchase dummy

```
name = 'pearson'
target = 'purchase_dummy'
pear_dummy = model(train_data_dummy, name, user_id, item_id, target,
users_to_recommend, n_rec, n_display)
```

iii. Using scaled purchase count

```
name = 'pearson'
target = 'scaled_purchase_freq'
pear_norm = model(train_data_norm, name, user_id, item_id, target,
users_to_recommend, n_rec, n_display)
```

8. Model Evaluation

For evaluating recommendation engines, we can use the concept of RMSE and precision-recall.

i. RMSE (Root Mean Squared Errors)

- Measures the error of predicted values
- Lesser the RMSE value, better the recommendations

ii. Recall

- What percentage of products that a user buys are actually recommended?
- If a customer buys 5 products and the recommendation decided to show 3 of them, then the recall is 0.6

iii. Precision

- Out of all the recommended items, how many the user actually liked?
- If 5 products were recommended to the customer out of which he buys 4 of them, then precision is 0.8

Why are both recall and precision important?

- Consider a case where we recommend all products, so our customers will surely cover the items that they liked and bought. In this case, we have 100% recall! Does this mean our model is good?
- We have to consider precision. If we recommend 300 items but user likes and buys only 3 of them, then precision is 0.1%! This very low precision indicates that the model is not great, despite their excellent recall.
- So our aim has to be optimizing both recall and precision (to be close to 1 as possible).

Let's first create initial callable variables for model evaluation:

```
models_w_counts = [popularity_model, cos, pear]
models_w_dummy = [pop_dummy, cos_dummy, pear_dummy]
models_w_norm = [pop_norm, cos_norm, pear_norm]


names_w_counts = ['Popularity Model on Purchase Counts', 'Cosine
Similarity on Purchase Counts', 'Pearson Similarity on Purchase
Counts']
names_w_dummy = ['Popularity Model on Purchase Dummy', 'Cosine
Similarity on Purchase Dummy', 'Pearson Similarity on Purchase
Dummy']
names_w_norm = ['Popularity Model on Scaled Purchase Counts', 'Cosine
Similarity on Scaled Purchase Counts', 'Pearson Similarity on Scaled
Purchase Counts']
```

Lets compare all the models we have built based on RMSE and precision-recall characteristics:

```
eval_counts = tc.recommender.util.compare_models(test_data,
models_w_counts, model_names=names_w_counts)


eval_dummy = tc.recommender.util.compare_models(test_data_dummy,
models_w_dummy, model_names=names_w_dummy)


eval_norm = tc.recommender.util.compare_models(test_data_norm,
models_w_norm, model_names=names_w_norm)
```

8.1. Evaluation Output

- *Based on RMSE*

- *Based on Precision and Recall*

8.2. Evaluation Summary

- **Popularity v. Collaborative Filtering:** We can see that the collaborative filtering algorithms work better than popularity model for purchase counts. Indeed, popularity model doesn't give any personalizations as it only gives the same list of recommended items to every user.
- **Precision and recall:** Looking at the summary above, we see that the precision and recall for Purchase Counts > Purchase Dummy > Normalized Purchase Counts. However, because the recommendation

scores for the normalized purchase data is zero and constant, we choose the dummy. In fact, the RMSE isn't much different between models on the dummy and those on the normalized data.

- **RMSE:** Since RMSE is higher using `pearson` distance than `cosine`, we would choose model the smaller mean squared errors, which in this case would be cosine.

> *Therefore, we select the **Cosine similarity on Purchase Dummy** approach as our **final model.***

9. Final Output

Finally, we would like to manipulate format for recommendation output to one we can export to *csv*, and also a function that will return recommendation list given a customer ID.

We need to first rerun the model using the whole dataset, as we came to a final model using train data and evaluated with test set.

```
final_model =
tc.item_similarity_recommender.create(tc.SFrame(data_norm),
                                                    user_id=user_id,
                                                    item_id=item_id,
                                                    target='purchase_dummy',
similarity_type='cosine')


recom = final_model.recommend(users=users_to_recommend, k=n_rec)
recom.print_rows(n_display)
```

9.1. CSV output file

Here we want to manipulate our result to a csv output. Let's see what we have:

```
df_rec = recom.to_dataframe()
print(df_rec.shape)
df_rec.head()
```

Let's define a function to create a desired output:

```
def create_output(model, users_to_recommend, n_rec, print_csv=True):
    recomendation = model.recommend(users=users_to_recommend,
k=n_rec)
    df_rec = recomendation.to_dataframe()
    df_rec['recommendedProducts'] = df_rec.groupby([user_id])
[item_id] \
        .transform(lambda x: '|'.join(x.astype(str)))
    df_output = df_rec[['customerId',
'recommendedProducts']].drop_duplicates() \
        .sort_values('customerId').set_index('customerId')
    if print_csv:
        df_output.to_csv('../output/option1_recommendation.csv')
        print("An output file can be found in 'output' folder with
name 'option1_recommendation.csv'")
    return df_output
```

Lets print the output below and set `print_csv` to *true*, this way we could literally print out our output file in *csv*, which you can also find it ***here***.

```
df_output = create_output(pear_norm, users_to_recommend, n_rec,
print_csv=True)
```

```
print(df_output.shape)
df_output.head()
```


9.2. Customer recommendation function

Let's define a function that will return recommendation list given a customer ID:

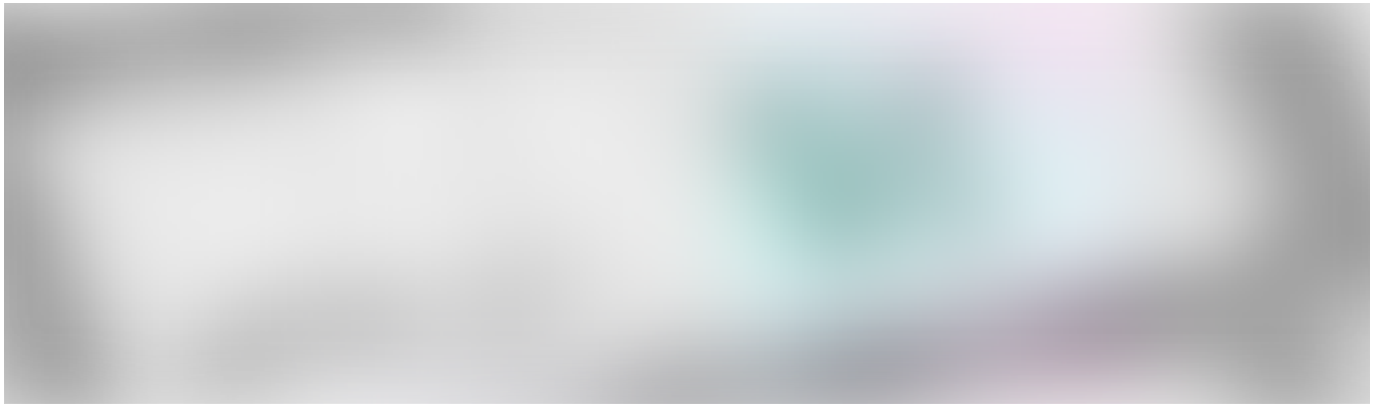
```
def customer_recomendation(customer_id):
    if customer_id not in df_output.index:
        print('Customer not found.')
        return customer_id
    return df_output.loc[customer_id]
```

Bingo!

Summary

In this article, we were able to traverse a step-by-step process for making recommendations to customers. We used Collaborative Filtering approaches with `Cosine` and `Pearson` measure and compared the models with our baseline popularity model.

We also prepared three sets of data that include regular buying count, buying dummy, as well as normalized purchase frequency as our target variable. Using RMSE, precision and recall, we evaluated our models and observed the impact of personalization. Finally, we selected the Cosine approach using dummy data as our best recommendation system model.



Hope you enjoy reading this article and are now ready to create your own “add-to-cart” button. Please give 50 claps and comment down below if you want more reads like this :) Enjoy hacking!

. . .

Moorissa is a mission-driven data scientist and social enterprise enthusiast. In December 2017, she graduated from Columbia University with a study in data science and machine learning. She hopes to always leverage her skills for making the world a better place, one day at a time.

Data Science | Recommendation System | Collaborative Filtering | Machine Learning | Artificial Intelligence

4.1K claps

WRITTEN BY

Moorissa Tjokro

Follow

Data Scientist at Tesla | http://moorissatjokro.com/

Data Driven Investor

Follow

from confusion to clarity, not insanity

See responses (35)

More From Medium

More from Data Driven Investor

Running your Deep Learning models in a browser using Tensorflow.js and ONNX.js

Quantum in Data Driven Investor
Oct 1 · 5 min read ★

223

More from Data Driven Investor

An Introduction to Python Counter

Chaitanya Baweja in Data Driven Investor
Sep 27 · 5 min read ★

174

More from Data Driven Investor

Trump Has No Legal or Factual Defense to Impeachment and Removal From Office

John Coble in Data Driven Investor
Oct 10 · 9 min read ★

117