Recommender Systems: Item-Customer Collaborative Filtering

Sparsity, Similarity, and implicit binary Collaborative Filtering explained step by step with Python Code

Jesko Rehberg Mar 8 · 13 min read ★

Recommender uses similarity as a complement to distance (image by author)

Knowing what items our customers should be highly interested in is vital in many businesses. Recapping our FPGrowth data mining, we experienced that quite often products are related to each other. So if one customer buys product A that customer will most likely also purchase product B, because both articles are related (for whatever reason, maybe like Beer and Diapers). Because of this **association** rule, we can estimate customers to purchase more than just one product within one sales transaction. To add some extra flavor, would it not be awesome to be able to recommend items to our customers, which they do not yet know themselves that they are interested in (but they are)? Let's take this next step and learn how to set up such a **recommending** system in Python.

Solution:

We will move out of the **market basket** (which was on sales transaction id level) into the more overall buying behaviour of our customers (which items our customers bought overall in the past), building a ML **recommending** system. Our aim is to recommend products to our customers, which they have not yet purchased but which they should be greatly interested in (without necessarily knowing about their desire by themselves yet). The trick here is to look at other **similar customers**' shopping habits. Since we do not have any labeled data available which tells us which customers are similar to each other, we will extract similarity between customers simply out of their **connection between** the same **items** they bought in the past. In general, similarity is a very interesting metric for lots of analysing purposes, not only linked to item-customer recommendations (see e.g. similarity between words/ sentences in our Chatbot). Looking for similarity in the purchase history, we anticipate that the similar preferences customers had will also extend into the future. So, if customer A purchases products 2,5 and 6 (not necessarily within the same basket) and customer B purchases products 2, 5, 6 and 9 then we can estimate that customer A is also interested in product 9. Please note that it could be that product 9 is very different to the products customer A has ever purchased before, because we are looking at similarity between customers (not between items). ML algorithms that build on this concept are called **collaborative filtering** (CF) algorithms.

Data Munging

```
import pandas as pd
data = pd.read_excel('/content/gdrive/MyDrive/DDDDFolder/DDDD.xlsx')
data.head()
```

	SalesDate	SalesValue	SalesAmount	Customer	SalesTransactionID	SalesItem
0	2018-09-28	8280.0	10	0	0	0
1	2018-09-28	7452.0	10	0	0	0
2	2019-04-23	21114.0	30	0	1	0
3	2019-04-23	7038.0	10	0	1	1
4	2019-04-23	7000.0	2	0	1	2

That's how our dataframe originally looks like. Hopefully it is easily adaptable to your data.

So what we have got is Sales per Sales Day, Customer (e.g. Customer code 0 is one specific customer) and Sales Item (similar to Customer, Sales Item 0 is one specific Sales Item). The other columns are not relevant for our product recommendation, so we will simply drop those:

```
DataPrep = data[['SalesItem', 'SalesAmount', 'Customer']]
DataPrep.head()
```

	SalesItem	SalesAmount	Customer
0	0	10	0
1	0	10	0
2	0	30	0
3	1	10	0

We will only use SalesItem, SalesAmount and Customer as basis for our recommender.

Using Panda's .info function we see that no NaN (not a number) values exist within our 341422 data rows (which is a good thing):

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 341422 entries, 0 to 341421
Data columns (total 3 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   SalesItem    341422 non-null  int64
 1   SalesAmount  341422 non-null  int64
 2   Customer     341422 non-null  int64
dtypes: int64(3)
memory usage: 7.8 MB
```

Now we want to figure out what items have been purchased per customer, using groupby:

```
DataGrouped =
DataPrep.groupby(['Customer','SalesItem']).sum().reset_index()
DataGrouped.head()
```

	Customer	SalesItem	SalesAmount
0	0	0	281
1	0	1	158
2	0	2	13
3	0	768	1
4	1	3	2

Our data, on which we will build our recommending engine on, has this shape: 36 unique customers (from 0 to 35) and 3751 distinct items. The interesting question is how many different items our customers actually bought? We will use Numpy's .list function to get the SalesAmount per distinct number of Customers and Items:

```
import numpy as np
customers = list(np.sort(DataGrouped.Customer.unique()))
# a list of values, so customers now stores 36 unique customers

products = list(DataGrouped.SalesItem.unique())
quantity = list(DataGrouped.SalesAmount)
```

Using Panda's DataFrame we can converge Numpy's list back into a Panda's dataframe:

```
from pandas import DataFrame
DfCustomerUnique = DataFrame(customers,columns=['Customer'])
DfCustomerUnique.head()
```

	Customer
0	0
1	1
2	2
3	3

DfCustomerUnique includes all unique Customers in one dataframe

Now we can build our sparse matrix:

```
from scipy import sparse
from pandas.api.types import CategoricalDtype
rows =
DataGrouped.Customer.astype(CategoricalDtype(categories=customers)).c
at.codes
# We have got 35 unique customers, which make up 13837 data rows
```

```
products)).cat.codes
# We have got unique 3725 SalesItems, making up 13837 data rows
(index)
# Get the associated column indices
#Compressed Sparse Row matrix
PurchaseSparse = sparse.csr_matrix((quantity, (rows, cols)), shape=
(len(customers), len(products)))
#len of customers=35, len of products=3725
#csr_matrix((data, (row_ind, col_ind)), [shape=(M, N)])
#where data, row_ind and col_ind satisfy the relationship
a[row_ind[k], col_ind[k]] = data[k]. , see [3]
PurchaseSparse
```

```
<35x3725 sparse matrix of type '<class 'numpy.longlong'>'
        with 13837 stored elements in Compressed Sparse Row format>
```

Sparse matrix (not a pandas df) are efficient for row slicing and fast matrix vector products

We have 35 customers and 3725 items overall. For these user/item interactions, 13837 of these items actually had a purchase. In terms of sparsity of the matrix, that makes:

```
MatrixSize = PurchaseSparse.shape[0]*PurchaseSparse.shape[1] # 130375
possible interactions in the matrix (35 unique customers * 3725
unique SalesItems=130375)

PurchaseAmount = len(PurchaseSparse.nonzero()[0]) # 13837 SalesItems
interacted with;

sparsity = 100*(1 - (PurchaseAmount/MatrixSize))

sparsity
```

```
89.38676893576223
```

Since we will use Matrix Factorization for our collaborative filtering it should not be a problem that 89.3% of the interaction matrix is sparse. In plain English, 89,3% in our case means that only 10,7% of our customer-item interactions are already filled, meaning that most items have not been purchased by customers. It is said that

good measure for sparse data, so we will stick to Cosine (instead of Pearson, Euclidean, Manhattan etc.).

Binary data vs SalesAmount

Our Collaborative Filtering will be based on **binary** data (a set of just two values), which is an important special case of categorical data. For every dataset we will add a 1 as purchased. That means, that this customer has purchased this item, no matter how many the customer actually has purchased in the past. We decided to use this binary data approach for our recommending example. Another approach would be to use the SalesAmount and normalize it, in case you want to treat the Amount of SalesItems purchased as a kind of taste factor, meaning that someone who bought SalesItem x 100 times- while another Customer bought that same SalesItem x only 5 times- does not like it as much. I believe that very often in Sales Recommendations a binary approach makes more sense, but of course that really depends on your data.

```
def create_DataBinary(DataGrouped):
DataBinary = DataGrouped.copy()DataBinary['PurchasedYes'] = 1
return DataBinary
DataBinary = create_DataBinary(DataGrouped)
DataBinary.head()
```

	SalesItem	SalesAmount	Customer	PurchasedYes
0	0	10	0	1
1	0	10	0	1
2	0	30	0	1
3	1	10	0	1

Finally, let's get rid of the column SalesAmount:

```
purchase_data=DataBinary.drop(['SalesAmount'], axis=1)
purchase_data.head()
```

0	0	0	1
1	0	0	1
2	0	0	1
3	1	0	1
4	2	0	1

That's our readily prepared data for building our recommendations on.

Collaborative Filtering

Trying to conduct collaborative filtering we could now have a look at Yifan Hu, Yehuda Koren and Chris Volinskys` paper ([1]). It's supposed that also Facebook and Spotify basically use the approach described in this paper. Have you read it? Have you understood it and know how to put this into Python code? Honestly, I can say yes to the first question, but definitely no to the second question. In case you`re answering similarly, I feel honored you to join me through all this step by step, for the sake of better understanding.

CF is one of the most widely researched and implemented recommendation algorithms (according to Aberger [2]). CF is a ML algorithm that collects preferences or taste information from many customers (collaborative, since it is combining collected information) and uses this information to make automatic predictions (filtering out less probable options to find the most probable prediction) about the interests of other customers. Of course, we will only recommend items which our customer has not yet purchased.

Explicit is better than Implicit (Zen of Python)

..but **implicit** is better than nothing. Using implicit, we cannot really judge if our customer liked a specific product or not. We can just tell that the customer did bought a specific item. And this information can furthermore be broken down to if the customer purchased a specific item at all (binary, yes or no) or how many times the customer bought this item (as a kind of taste factor: the more often the item was bought the more

us readers. I think that in real life very often we only have implicit feedback at our fingertips. Implicit feedback like amount of items purchased, clicks on a website, movies watched etc. is not an explicit feedback ("I like" etc). So it could be that even if we purchased a specific book, we do not really like it at all while reading it. Even though this is a disadvantage of implicit feedback very often we just do not have explicit customer feedback, so we have to stick to implicit feedback. Also if the amount of implicit feedback outweighs the amount of explicit feedback we might decide to build our recommending engine on implicit data (sometimes mass of data rules over less data even with better quality). We look for associations between customers, not between products. Therefore, collaborative filtering relies only on observed customer behavior to make recommendations — with no information about the customers or the items required.

In one sentence: our recommendation is based on the assumption that customers who purchased products in a similar quantity share one or more hidden preferences. Due to this shared **latent** or **hidden features** customers will likely purchase similar products.

Similarity metric

How can we tell how similar one customer is compared to another customer? Or how close one item is to other items? To answer this question it is helpful to state that the approach of similarity is a complement to distance. While the distance between two points can be obtained using the Euclidean distance -by calculating the length of a straight line connecting two points- , similarity is about "how close" the points are. The similarity value is usually expressed as a number between 0 and 1, where 0 means a low similarity, and 1 a high similarity. Let's break this down into simple Google Sheets/ Microsoft Excel formula (below sheet screenshots used Google Sheets, which you can also find in my Github repository):

imagine our 9 Customers have purchased these 7 Sales items as follows (1 means purchased, 0 means not purchased), then the "PurchaseHistory" Tab looks like this:

	A	B	C	D	E	F	G	H
1	Customer	Sales Item 1	Sales Item 2	Sales Item 3	Sales Item 4	Sales Item 5	Sales Item 6	Sales Item 7
2	Customer 1	0	0	1	0	0	0	0

6	Customer 5	0	0	0	1	0	0	0
7	Customer 6	0	1	0	0	0	0	1
8	Customer 7	0	1	0	0	0	0	0
9	Customer 8	0	0	1	0	0	0	0
10	Customer 9	1	0	1	1	1	1	1

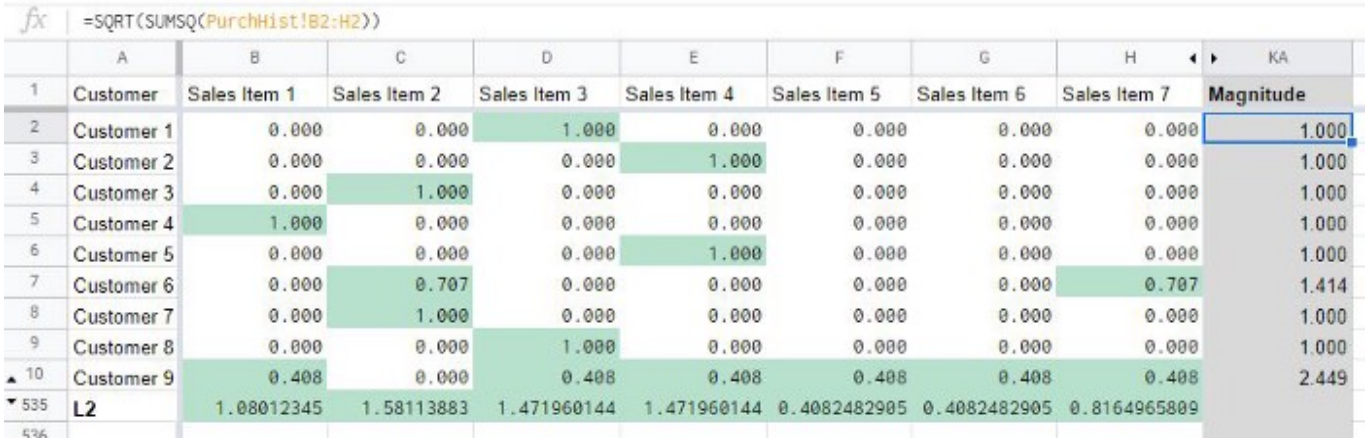
E.g., customer 3 bought Sales Item 2 once.

The normalized data will look like this in tab “PurchHistNorm”:

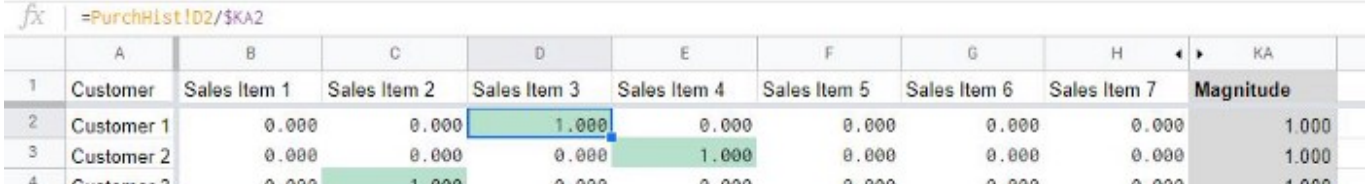
	A	B	C	D	E	F	G	H	KA
1	Customer	Sales Item 1	Sales Item 2	Sales Item 3	Sales Item 4	Sales Item 5	Sales Item 6	Sales Item 7	Magnitude
2	Customer 1	0.000	0.000	1.000	0.000	0.000	0.000	0.000	1.000
3	Customer 2	0.000	0.000	0.000	1.000	0.000	0.000	0.000	1.000
4	Customer 3	0.000	1.000	0.000	0.000	0.000	0.000	0.000	1.000
5	Customer 4	1.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000
6	Customer 5	0.000	0.000	0.000	1.000	0.000	0.000	0.000	1.000
7	Customer 6	0.000	0.707	0.000	0.000	0.000	0.000	0.707	1.414
8	Customer 7	0.000	1.000	0.000	0.000	0.000	0.000	0.000	1.000
9	Customer 8	0.000	0.000	1.000	0.000	0.000	0.000	0.000	1.000
10	Customer 9	0.408	0.000	0.408	0.408	0.408	0.408	0.408	2.449
535	L2	1.08012345	1.58113883	1.471960144	1.471960144	0.4082482905	0.4082482905	0.8164965809	

Does scaling of binary data makes sense? This will be answered later on.

..because for eg. customer 1 we have to chekc the magnitude for the sake of normalizing:



	A	B	C	D	E	F	G	H	KA
1	Customer	Sales Item 1	Sales Item 2	Sales Item 3	Sales Item 4	Sales Item 5	Sales Item 6	Sales Item 7	Magnitude
2	Customer 1	0.000	0.000	1.000	0.000	0.000	0.000	0.000	1.000
3	Customer 2	0.000	0.000	0.000	1.000	0.000	0.000	0.000	1.000
4	Customer 3	0.000	1.000	0.000	0.000	0.000	0.000	0.000	1.000
5	Customer 4	1.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000
6	Customer 5	0.000	0.000	0.000	1.000	0.000	0.000	0.000	1.000
7	Customer 6	0.000	0.707	0.000	0.000	0.000	0.000	0.707	1.414
8	Customer 7	0.000	1.000	0.000	0.000	0.000	0.000	0.000	1.000
9	Customer 8	0.000	0.000	1.000	0.000	0.000	0.000	0.000	1.000
10	Customer 9	0.408	0.000	0.408	0.408	0.408	0.408	0.408	2.449
535	L2	1.08012345	1.58113883	1.471960144	1.471960144	0.4082482905	0.4082482905	0.8164965809	



	A	B	C	D	E	F	G	H	KA
1	Customer	Sales Item 1	Sales Item 2	Sales Item 3	Sales Item 4	Sales Item 5	Sales Item 6	Sales Item 7	Magnitude
2	Customer 1	0.000	0.000	1.000	0.000	0.000	0.000	0.000	1.000
3	Customer 2	0.000	0.000	0.000	1.000	0.000	0.000	0.000	1.000

9	Customer 8	0.000	0.000	1.000	0.000	0.000	0.000	0.000	1.000
10	Customer 9	0.408	0.000	0.408	0.408	0.408	0.408	0.408	2.449
535	L2	1.08012345	1.58113883	1.471960144	1.471960144	0.4082482905	0.4082482905	0.8164965809	

=SQRT(SUMSQ(B$2:B$10))

	A	B	C	D	E	F	G	H	KA
1	Customer	Sales Item 1	Sales Item 2	Sales Item 3	Sales Item 4	Sales Item 5	Sales Item 6	Sales Item 7	Magnitude
2	Customer 1	0.000	0.000	1.000	0.000	0.000	0.000	0.000	1.000
3	Customer 2	0.000	0.000	0.000	1.000	0.000	0.000	0.000	1.000
4	Customer 3	0.000	1.000	0.000	0.000	0.000	0.000	0.000	1.000
5	Customer 4	1.000	0.000	0.000	0.000	0.000	0.000	0.000	1.000
6	Customer 5	0.000	0.000	0.000	1.000	0.000	0.000	0.000	1.000
7	Customer 6	0.000	0.707	0.000	0.000	0.000	0.000	0.707	1.414
8	Customer 7	0.000	1.000	0.000	0.000	0.000	0.000	0.000	1.000
9	Customer 8	0.000	0.000	1.000	0.000	0.000	0.000	0.000	1.000
10	Customer 9	0.408	0.000	0.408	0.408	0.408	0.408	0.408	2.449
535	L2	1.08012345	1.58113883	1.471960144	1.471960144	0.4082482905	0.4082482905	0.8164965809	

So finally the item-item similarity matrix looks like this:

=SUMPRODUCT(PurchHistNorm!B2:B10,PurchHistNorm!B2:B10)/(PurchHistNorm!B535*PurchHistNorm!B535)

	A	B	C	D	E	F	G	H
1		Sales Item 1	Sales Item 2	Sales Item 3	Sales Item 4	Sales Item 5	Sales Item 6	Sales Item 7
2	Sales Item 1	1						
3	Sales Item 2	0	1					
4	Sales Item 3	0.10482848	0	1				
5	Sales Item 4	0.10482848	0	0.07692307	1			
6	Sales Item 5	0.37796447	0	0.27735009	0.27735009	1		
7	Sales Item 6	0.37796447	0	0.27735009	0.27735009	1	1	
8	Sales Item 7	0.18898223	0.38729833	0.13867504	0.13867504	0.5	0.5	1

Certainly a product is always highest correlated with itself (Sales Item 1 with Sales Item 1 correlates perfectly scoring a 1). I hope this simple spreadsheet example helps you to get the basic idea behind setting up a Item-Item Similarity.

Which similarity measure to use?

It is recommended to use Pearson if the data is subject to grade-inflation (different customers may be using different scales). If our data is dense (almost all attributes have non-zero values) and the magnitude of the attribute values is important, use distance measures such as Euclidean or Manhattan. But for our sparse univariate data example we will stick to Cosine Similarity.

as a prefix before every SalesItem. Otherwise we would only have customer and SalesItem Numbers, which can become a little bit puzzling:

```
purchase_data['SalesItem'] = 'I' +
purchase_data['SalesItem'].astype(str)
DfMatrix = pd.pivot_table(purchase_data, values='PurchasedYes',
index='Customer', columns='SalesItem')
DfMatrix.head()
```

SalesItem	I0	I1	I10	I100	I1000	I1001	I1002	I1003	I1004	I1005	I1006	I1007	I1008	I1009	I101	I1010	I1011
Customer																	
0	1.0	1.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	1.0	1.0	1.0	NaN	NaN	NaN	NaN	1.0	1.0	NaN	NaN	NaN	1.0	NaN	NaN
3	NaN	NaN	1.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	1.0	NaN	NaN
4	NaN	NaN	NaN	1.0	NaN	NaN	NaN	NaN	NaN	NaN	1.0	NaN	NaN	NaN	1.0	NaN	NaN

5 rows × 3725 columns

Since we are only using 1 and 0, we do not need to think about normalization, which is one method of scaling. Normalization reduces the scale of the data to be in a range from 0 to 1:

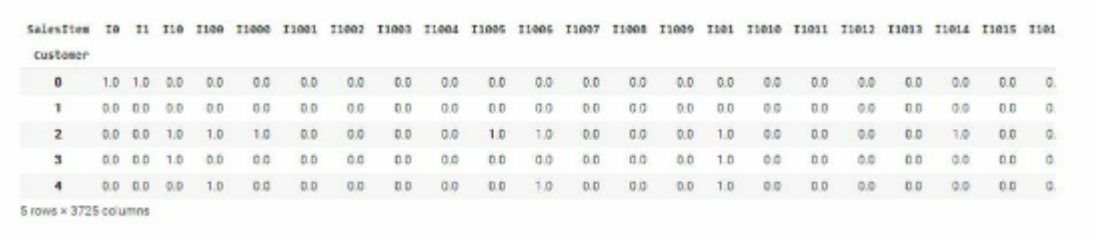
$$X_{normalized} = X - X_{min} / (X_{max} - X_{min})$$

Another form of scaling is standardization, which reduces the scale of the data to have a mean(μ) of 0 and a standard deviation(σ) of 1:

$$X_{standardized} = X - \mu / \sigma$$

But talk is cheap, let`s check to see that even if we would normalize, the result is the same, of course:

```
DfMatrix=DfMatrix.fillna(0)
#NaN values need to get replaced by 0, meaning they have not been
purchased yet.

DfMatrixNorm3 = (DfMatrix-DfMatrix.min())/(DfMatrix.max()-
DfMatrix.min())
```



SalesItem	I0	I1	I10	I100	I1000	I1001	I1002	I1003	I1004	I1005	I1006	I1007	I1008	I1009	I101	I1010	I1011	I1012	I1013	I1014	I1015	I101
Customer																						
0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
2	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.
3	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
4	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.

5 rows × 3725 columns

We need to bring our pivot table into the desired format, via reset_index and rename_axis:

```
DfResetted = DfMatrix.reset_index().rename_axis(None, axis=1)
DfResetted.head()
#Now each row represents one customer`s buying behaviour: 1 means the
customer has purchased, NaN the customer has not yet purchased it
```

	Customer	I0	I1	I10	I100	I1000	I1001	I1002	I1003	I1004	I1005	I1006	I1007	I1008	I1009	I101	I1010	I101
0	0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
1	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0
2	2	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0
3	3	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0
4	4	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0

5 rows × 3726 columns

```
DfMatrix.shape
```

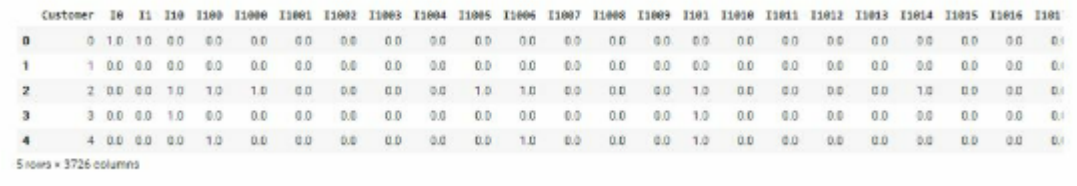
```
(35, 3725)
```

Customer must be nvarchar! If customer is int, then failure during CustItemSimilarity!

```
df=DfResetted
```

item, yes or no. That is called binary. If customer bought a specific item, that means 1. If not, then 0. Because of this binary problem there is no use in using any further scaling techniques.

```
df=df.fillna(0)
df.head()
```



	Customer	I0	I1	I10	I100	I1000	I1001	I1002	I1003	I1004	I1005	I1006	I1007	I1008	I1009	I101	I1010	I1011	I1012	I1013	I1014	I1015	I1016	I101
0	0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
1	1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
2	2	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.
3	3	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
4	4	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.

5 rows × 3726 columns

Now we create a dataframe which only includes Sales Items, while Customer is indexed instead.

```
DfSalesItem = df.drop('Customer', 1)
DfSalesItem.head()
```

	I0	I1	I10	I100	I1000	I1001	I1002	I1003	I1004	I1005	I1006	I1007	I1008	I1009	I101	I1010	I1011	I1012	I101
0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.
2	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.
3	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.
4	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.

5 rows × 3725 columns

So we can now calculate the Item based recommendation. We will normalize dataframe for comparising reasons:

```
import numpy as np
DfSalesItemNorm =
DfSalesItem/np.sqrt(np.square(DfSalesItem).sum(axis=0))
DfSalesItemNorm.head()
```

	I0	I1	I10	I100	I1000	I1001	I1002	I1003	I1004	I1005	I1006	I1007	I1008	I1009	I101	I1011
0	0.333333	0.301511	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.0	0.00000	0.0
1	0.000000	0.000000	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.0	0.00000	0.0
2	0.000000	0.000000	0.408248	0.288675	0.5	0.0	0.0	0.0	0.0	0.333333	0.288675	0.0	0.0	0.0	0.27735	0.0
3	0.000000	0.000000	0.408248	0.000000	0.0	0.0	0.0	0.0	0.0	0.000000	0.000000	0.0	0.0	0.0	0.27735	0.0
4	0.000000	0.000000	0.000000	0.288675	0.0	0.0	0.0	0.0	0.0	0.000000	0.288675	0.0	0.0	0.0	0.27735	0.0

5 rows × 3725 columns

Calculating with .dot Vectors to compute Cosine Similarities:

```
ItemItemSim = DfSalesItemNorm.transpose().dot(DfSalesItemNorm)
ItemItemSim.head()
```

	I0	I1	I10	I100	I1000	I1001	I1002	I1003	I1004	I1005	I1006
I0	1.000000	0.703526	0.136083	0.192450	0.333333	0.384900	0.235702	0.333333	0.384900	0.222222	0.192450
I1	0.703526	1.000000	0.123091	0.174078	0.150756	0.174078	0.000000	0.301511	0.174078	0.201008	0.174078
I10	0.136083	0.123091	1.000000	0.589256	0.408248	0.235702	0.000000	0.204124	0.235702	0.680414	0.589256
I100	0.192450	0.174078	0.589256	1.000000	0.577350	0.500000	0.408248	0.577350	0.500000	0.866025	0.833333
I1000	0.333333	0.150756	0.408248	0.577350	1.000000	0.866025	0.707107	0.750000	0.866025	0.666667	0.577350

5 rows × 3725 columns

Use ItemItemSim.to_excel(“ExportItem-Item.xlsx”) if you want to export this result.

Create a placeholder items for closest neighbours to an item

```
ItemNeighbours =
pd.DataFrame(index=ItemItemSim.columns,columns=range(1,10))
ItemNeighbours.head()
```

	1	2	3	4	5	6	7	8	9
I0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
I1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
I10	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
I100	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
I1000	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Create a placeholder items for closes neighbours to an item

```
for i in range(0,len(ItemItemSim.columns)): ItemNeighbours.iloc[i,:9]
= ItemItemSim.iloc[0:,i].sort_values(ascending=False)[:9].index #we
only have 9 items, so we can max recommend 9 items (itself included)
ItemNeighbours.head()
```

	1	2	3	4	5	6	7	8	9
I0	I0	I2	I1	I769	I1134	I705	I704	I1139	I1138
I1	I1	I768	I759	I758	I754	I757	I749	I750	I753
I10	I10	I1699	I1696	I1674	I2102	I19	I1242	I970	I254
I100	I161	I86	I146	I128	I71	I152	I193	I89	I200
I1000	I747	I962	I1041	I893	I930	I1000	I790	I975	I917

You can see that the first column stores the item itself.

```
ItemNeighbours.head().iloc[:11,1:9]
#it needs to start at position 1, because position 0 is item itself
```

	2	3	4	5	6	7	8	9
I0	I2	I1	I769	I1134	I705	I704	I1139	I1138
I1	I768	I759	I758	I754	I757	I749	I750	I753
I10	I1699	I1696	I1674	I2102	I19	I1242	I970	I254
I100	I86	I146	I128	I71	I152	I193	I89	I200
I1000	I962	I1041	I893	I930	I1000	I790	I975	I917

Now we got rid of column 1, which was just the item itself.

```
# You might want to export this result to Excel now
ItemNeighbours.to_excel("ExportItem-Item-data_neighbours.xlsx")
```

Now we will create a customer based recommendation which we need our item similarity matrix for. Then we will have a look which items our customers have bought

in the end we just have to recommend the items with the highest score. Therefore we will create a place holder matrix for similarities, and fill in the customer column:

```
CustItemSimilarity = pd.DataFrame(index=df.index,columns=df.columns)
CustItemSimilarity.iloc[:,:1] = df.iloc[:,:1]
CustItemSimilarity.head()
```

	Customer	I0	I1	I10	I100	I1000	I1001	I1002	I1003	I1004	I1005	I1006	I1007	I1008	I1009	I101	I1010	I1011	I1012	I1013	I1014	I101
0	0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
1	1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
2	2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
3	3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na
4	4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	Na

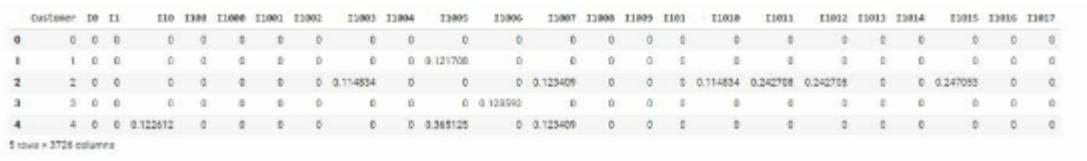
Finally we will calculate the similarity scores

```
def getScore(history, similarities):

return sum(history*similarities)/sum(similarities)
```

This takes ages (35 customers * 3725 items), because we now loop through the rows and columns filling in empty spaces with similarity scores (find an improved solution here). Please note that we score items that the customer has already consumed as 0, because there is no point in recommending these again.

```
from timeit import default_timer as timer
#to see how long the computation will take
start = timer() for i in range(0,len(CustItemSimilarity.index)): for
j in range(1,len(CustItemSimilarity.columns)): user =
CustItemSimilarity.index[i] product = CustItemSimilarity.columns[j]

if df.loc[i][j] == 1: CustItemSimilarity.loc[i][j] = 0 else: ItemTop
= ItemNeighbours.loc[product][1:9]
ItemTopSimilarity =
ItemItemSim.loc[product].sort_values(ascending=False)[1:9]
#here we will use the item dataframe, which we generated during item-
item matrix
```

```
print("\nRuntime: %0.2fs" % (end - start))
CustItemSimilarity.head()
```



If there occurs a strange error tz=getattr(series.dtype, 'tz', None) .. pandas index.. then this might come from using int # as column headers instead of string

Now we will generate a matrix of customer based recommendations as follows:

```
# Get the top SalesItems
CustItemRecommend = pd.DataFrame(index=CustItemSimilarity.index,
columns=['Customer','1','2','3','4','5','6']) #Top 1,2..6
CustItemRecommend.head()
```

	Customer	1	2	3	4	5	6
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Instead of having the matrix filled with similarity scores we want to see the product names. Therefore loop:

```
for i in range(0,len(CustItemSimilarity.index)):
CustItemRecommend.iloc[i,1:] =
CustItemSimilarity.iloc[i,:].sort_values(ascending=False).iloc[1:7,].
index.transpose()
CustItemRecommend.head()
```

2	2	I1165	I168	I169	I272	I299	I394
3	3	I192	I92	I61	I73	I108	I229
4	4	I1165	I280	I1179	I157	I6	I124

Customer 0's Top1 Recommendation is Item1134, Item999 is 2nd, Item2128 3rd and so on.

```
CustItemRecommend.to_excel("ExportCustomer-Item-
CustItemRecommend.xlsx")
```

Congratulations, we have coded a binary recommender engine, which works sufficient on a small data sets. Let us see if we can enhance the performance and scalability using the same dataset in another story. You can find the Jupyter Notebook and Excel example in my Github. Many thanks for reading, hope this was supportive! Any questions, please let me know. You can connect with me on LinkedIn or Twitter.

If you want to recommend using Scikit-Learn or Tensorflow Recommender you might find this post helpful:

Recommend using Scikit-Learn and Tensorflow Recommender
Collaborative Filtering for Sales Items sold (binary) per Customer
towardsdatascience.com

Originally published on my website DAR-Analytics.

Reference:

[1] Yifan Hu, Yehuda Koren and Chris Volinskys, Collaborative Filtering for Implicit Feedback Datasets: http://yifanhu.net/PUB/cf.pdf

[2] Christopher R. Aberger, Recommender: An Analysis of Collaborative Filtering Techniques: https://www.semanticscholar.org/paper/Recommender-%3A-An-Analysis-

[3] scipy.sparse.csr_matrix,
https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html

Collaborative Filtering Sparse Matrix Similarity Recommender Systems