Politecnico Di Milano

Graduate School of Management

MASTER'S THESIS

Product Recommendation in Fashion

Author: Makki Fourati

Tutor: Mauricio Abel Soto Gomez

Master in Business Analytics and Big Data

October 2022

Contents

I Introduction

1 - What is a recommendation system

Recommender Systems (RS) is one of the most powerful machine learning algorithms used widely in E-Commerce, video-on-demand, and music stream. Recommender Systems are software tools that aim to suggest products to a user by considering their interest. [1]

There exist two types of recommendations when it comes to data: Implicit vs explicit feedback. Explicit is when we have explicit ratings of items from users like in movies platforms. Implicit feedback is when we only have implicit interaction like whether the user purchased the item, likes, number of clicks etc…
The competition that this project is based on is, as a matter of fact, an implicit feedback recommendation system problem.

2 - About the competition

This project is based on a Kaggle competition made by H&M and is called 'H&M personalized Fashion Recommendations'. The goal of the competition is to develop product recommendations based on data from previous transactions, as well as from customer and product meta data. [2]

3 - Goals

The goals of this project are twofold:

i. Working on the competition's problem hence building a recommender system for H&M online shopping
ii. Using the same data, we will solve a different problem outside the scope of the competition which is recommending categories of products

II Data and resources

The data that was used comes exclusively from the competition and consists of three datasets:

- transctions_train.csv: Purchase history consisting of customer_id and article_id pairs with some transactions metadata (31788324 rows × 5 columns)
- articles.csv: Products metadata available in both categorical and numerical form and having as key article_id (105542 rows × 25 columns)

• customers.csv: Customer metadata with customer_id as key (1371980 rows × 7 columns)

Most features are more or less straightforward except for one in customers.csv ‘FN’ which means whether the customer is getting fashion news or not and ‘sales_channel_id’ where 2 means that the purchase was made online and 1 in store. This ambiguity was cleared by the host and communicated in this thread [3].

For this project I worked on Jupyter notebook for some light data manipulation and used Colab for the rest of the heavy-duty work. At some point I had to upgrade to Colab Pro, due to the size of some dataframes (70M rows) that couldn’t fit in memory.

For learning material, I followed two courses, one given by google cloud on Coursera [4] and another one on Udemy [5]. However, I had to search for information from many other websites and articles like towards data science, medium and many others.

III Personal recommendation

Personal recommendation aims at selecting for a user, a set of items that they might be interested in in the future, and this in a personalized way, meaning this set is tailored for this user according to their purchase history, personal information and any interaction with the website or platform.

1 - Turicreate : a very handy library

Turicreate is a machine learning library created by Apple that has a recommender system dedicated component [6]. I chose it to be my go to library for this project because it is complete and also pretty user friendly. It gives a great visibility over the models and allows you to fine tune them at a low level without having to deal with a black box. Logs and results are printed in a very clear way and documentation is also a big plus as every method and algorithm is explained in detail with all the eventual mathematics behind.

2 - Exploring Recommandation Systems

2.1 - Item-based Collaborative Filtering (item-item similarity)

Collaborative filtering is one of the most popular techniques in recommendation systems. It uses user-item interaction data to find similarities and make recommendations.

The item-item similarity is one of the CF techniques and is considered memory-based. The goal is to compute how similar a new item is to the set of items a certain user has already

purchased, using purchase history. The similarity itself between two items is computed using a certain similarity metric like Cosine, Jaccard or Pearson. Below is shown the formula for the Jaccard similarity between an item i and an item j, which is considered the best choice for implicit data:

$$\mathrm{JS}(i,j) = \frac{|U_i \cap U_j|}{|U_i \cup U_j|}$$

U_i : Set of users who purchased item i

U_j : Set of users who purchased item j

To see how good is item j for user u, we compute the following prediction score which is the mean of the individual similarity scores between j and the all the items that were purchased by user u (this applies for all similarity measures)

$$y_{uj} = \frac{\sum_{i \in I_u} \mathrm{SIM}(i,j)}{|I_u|}$$

I_u : user's previous observations

Turicreate item-base CF class:

`turicreate.recommender.item_similarity_recommender.ItemSimilarityRecommender`

2.2 - Collaborative Filtering with Matrix Factorization

Matrix factorization is a class of collaborative filtering (CF) that became very famous during the Netflix prize challenge [7]. It is considered model based. The goal is to reduce the dimensionality of our big sparse matrix by decomposing it into two smaller embedding matrices. These embeddings are learned such that the product of the two matrices is a very good approximation of the original interaction matrix. [8] We can train our model with either SGD or ALS (Alternating Least Squares). The latter is a faster technique that is based on a least square optimization problem where on every iteration we fix the values of a matrix and optimize the second, then do the reversed operation.

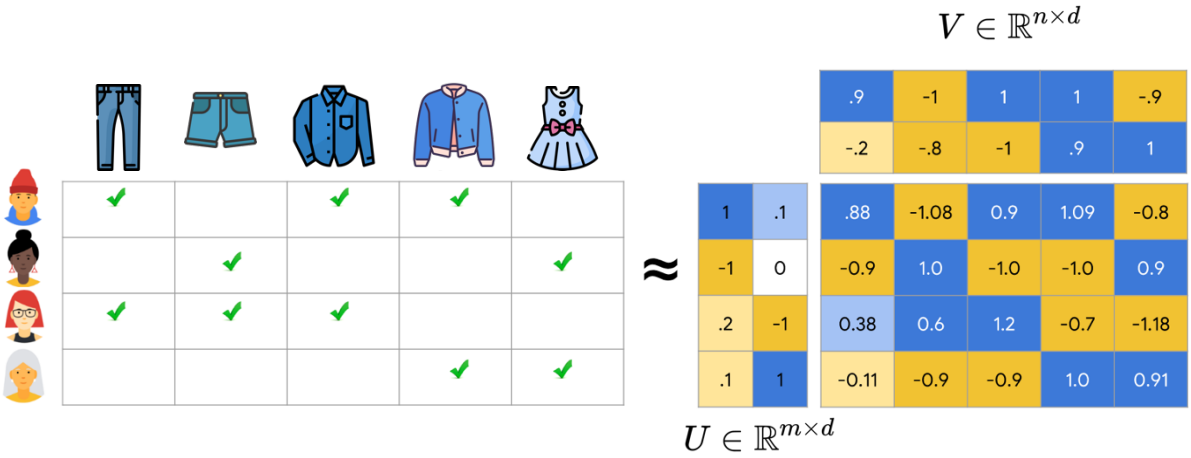


Figure 1: Matrix Factorization [8]

Where U is the user embedding matrix and V the item embedding matrix

This model has k as hyper parameter, which is the number of latent factors or in other words the dimensionality d. Now to make a prediction, all we have to do is compute the dot product between the embedding vector of a user u and the embedding vector of an item i, to know if we should recommend i to u.

Turicreate MF class:

`turicreate.recommender.ranking_factorization_recommender.RankingFactorizationRecommender`

N.B: Due to its nature, collaborative filtering comes with an issue called cold start, which is basically the fact that we cannot make recommendation for new users, the ones with little to no purchase history. Also with CF in general, the sparser the data the poorer the performance.

2.3 - Popularity

Popularity based recommendation is recommending the best-selling or most popular items. Although very simple and almost naïve, this method can improve our recommendation considerably when combined with other techniques. It also has as advantage the fact that we can recommend products to new users

2.4 - Repurchase

Another simple way of recommending is to recommend items that have already been purchased by a user hoping that they would buy them again. Like popularity, this is a technique that is usually combined with other ones.

3 - Metric: MAP@K

To get a better idea of how recommendation models are performing, one should first

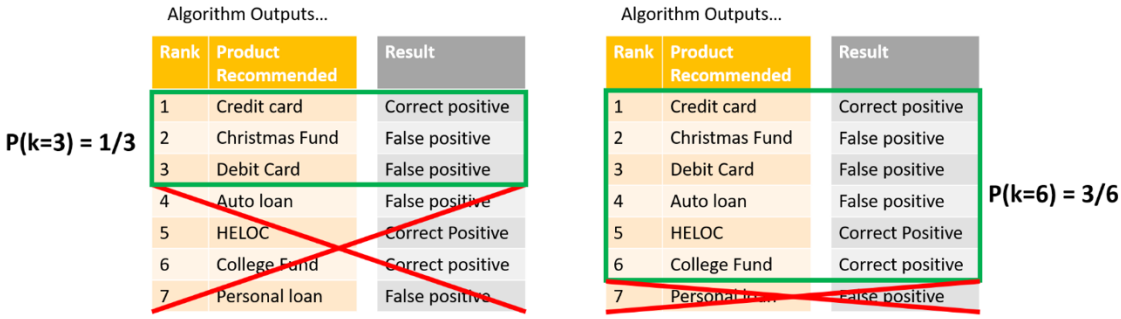
understand the evaluation metrics used in this topic. For an implicit recommendation problem like ours, a very commonly used metric is MAP@k or Mean Average Precision at k. To break it down we start by introducing the precision at k with the following formula:

$$P(k) = \frac{|a \cap p_k|}{k},$$

p_k : vector of the first k elements in the recommendation

a : set of items that the user actually purchased (or liked)

Here is an illustration of a real scenario



Now that we understand how is computed the precision at cutoff k we can look at the MAP@K with K set to 12:

$$\text{MAP@12} = \frac{1}{U}\sum_{u=1}^{U}\frac{1}{\min(m,12)}\sum_{k=1}^{\min(n,12)} P(k) \times \text{rel}(k)$$

U : number of customers,

$P(k)$: precision at cutoff k,

n : number predictions per customer

m : is the number of ground truth values per customer

$rel(k)$: indicator function equaling 1 if the item at rank k is a relevant (correct) label, zero otherwise.

In the competition, the scoring is done on the week that comes after the last day of transaction data using MAP@K with K equals to 12. During the competition the scoring is computed only on 1% of the testing data which is called public score, and towards the end of the competition a private score is revealed which is computed on the remaining 99%.

4 - Making recommendation and first submission

File: *Turicreate_RS.ipynb* [9]

We start by loading the transactions history, this will be the only data source we are using for now as we are working with collaborative filtering. As a first trial, I wanted to work only on a portion of the dataset. I grouped the transactions by customer_id to count the number of purchases for each user and kept the most active ones, then did the same thing with items by keeping most purchased ones. The goal is to have a fuller interaction matrix and see the effect of sparsity later on when we apply the same algorithms on sparser data and compare the results. We can split the data into train and test using Turicreate dedicated method that is recommender friendly, with which you can control how many users there will be in the test set.

Then we need to encode the ids of both customers and articles into ascending integers starting from 0, which is a crucial step for CF. We save the conversion result in a dictionary so we can convert them back to original ids when needed. Finally, we convert the pandas dataframe to an Sframe which is specific to Turicreate so we can use their models.

To make recommendations with Turicreate we first use the create () function of the corresponding algorithm class. Then we call the recommend() function using the model created to output the recommendation for the k best products (a parameter to specify in recommend())

For item similarity we only have to choose the similarity metric. Here is on the right an example of the recommendation output of the model

user_id	item_id	score	rank
0	19240	0.0026329703952955165	1
0	152	0.002247720956802368	2
0	1744	0.0020862435517103777	3
0	23080	0.0017583804286044576	4
0	19023	0.0016552216332891712	5
0	3921	0.0015621729519056237	6
0	27017	0.0015527029400286467	7
0	22885	0.0015468228122462397	8
0	3724	0.0015233625536379607	9
0	31797	0.0014299957648567531	10
0	571	0.001376674227092577	11
0	17142	0.0013392334399016006	12
1	33170	0.004627175403363777	1
1	8268	0.004484389767502294	2
1	6233	0.0038152319012266216	3

Figure 2 Model recommendation output

Now we can evaluate our model on the test data using evaluate_precision_recall() function at specified cutoffs that we call with the model object. We can either get the scores for each user or the overall one like on these figures:

cutoff	precision	recall
1	0.21536768873428827	0.01587973337186984
2	0.1783154221446756	0.025670259055042
3	0.15619223317515207	0.033290830165462124
4	0.14034629532901308	0.0394048224693976
5	0.12855825118636124	0.04469243680376137
6	0.11909180827063523	0.04936432860409997
7	0.111166325589371	0.053399794585660965
8	0.10478323201564761	0.05720018142130301
9	0.09933578570290616	0.06075331231667819
10	0.09457218746179773	0.0639932696481935

Figure 3 Cosine Similarity

cutoff	precision	recall
1	0.2573543303586333	0.017858598646577264
2	0.2102666118402772	0.02867344931811066
3	0.18100117432198679	0.036651172809033274
4	0.16064502506084544	0.04308605539516399
5	0.14541514319689575	0.04858322855534344
6	0.13356338189914185	0.05333618991773916
7	0.12418018407060619	0.057680893461563576
8	0.11630788722063547	0.061558464398518746
9	0.10970093686444717	0.06514621632861176
10	0.10394972271924	0.06841639538060973

Figure 4 Jaccard similarity

```
num_factors=100,
regularization=0.001,
max_iterations=40
solver='ials'
```

cutoff	precision	recall
1	0.09585356908680727	0.006501282302369816
2	0.08093930940976461	0.010798494390268351
3	0.07190406863671084	0.014282448224905202
4	0.06595281226036602	0.01738284922433971
5	0.061026216645733114	0.019982959170105556
6	0.056995365688311914	0.022319540760934186
7	0.05366370995781855	0.0244382007234695
8	0.05099687711850266	0.026469120454665228
9	0.04861149699504703	0.02832457338107626
10	0.046595392360609195	0.03007670389688931

Figure 5 Matrix factorization

We can see that the Jaccard similarity got better results as expected and that the matrix factorization is not performing as well as the item similarity. It is worth mentioning though that for limited computational power reasons, I was not able to increase the number of factors past 100 and eventually further improve the model.

File: *H&M_RS.ipynb* [9]

Next step is trying to make a first submission to the competition. This time around we are not selecting the most active customers but rather using CF on the last five weeks of transactions. Also, data splitting is done through a more recommender system way where we take as validation set the last week of purchases and as training the remaining previous weeks. Jaccard item similarity was again the best performing method here, but as expected, we had a huge drop in performance due to the sparsity of the data:

cutoff	precision	recall
1	0.008494723414125003	0.003570061259889739
2	0.0074510031311608455	0.005941017692509008
3	0.007620124473307754	0.008646510234893927
4	0.006682709033978888	0.01014408738840736
5	0.006146352777455469	0.011696354785841507
6	0.006214001314314529	0.013806236571934492
7	0.005823296499395445	0.014927338591182563
8	0.005566508175808903	0.01631117632187379
9	0.005286249951679516	0.017316036386251468
10	0.005015655804244501	0.018175726490476905

Next, we take the remaining customers that haven't received recommendations yet and recommend to them the twelve most popular or best-selling items. We concatenate the results of both methods and format it so we can make a submission. Here are the results:

	Private	Public
my_submission.csv 8 days ago by Makki Fourati Trained Jaccard similarity on both train and val	0.00985	0.00828
my_submission.csv 11 days ago by Makki Fourati add submission details	0.00750	0.00644

The top submission is the more recent one and the bottom one is the first. It is interesting to see how including the validation set into the item similarity method rather than the popularity gives a big jump in score.

To get a better idea on these numbers, here are the final scores of some winning teams

- 1st : 0.03792
- 100th : 0.02648
- 300th : 0.02393

5 - Retrieval and Ranking

In a real-world recommendation problem, we usually refer to a technique called retrieval and ranking. As the name suggests, it consists of two different steps:

Candidate retrieval or candidate generation: from all the items that we have (100k in our case) we are going to select a subset (100 to 1000 for ex) of items called candidates that most likely contains the ones that we want to recommend. In other words, we're reducing down the set of items to recommend for each user by ruling out the ones that are not relevant.

Ranking: takes the output of the retrieval step and applies a ranking machine learning algorithm to rank the retrieved candidates from worst to best for each customer, at which point we can select the k best that we want to recommend.

This technique has as advantages the fact that it usually gives better results, overcomes the issues of cold start and sparsity, and gets the best out of each technique mentioned above while also using users and items metadata and not only transactions.

5.1 - Candidate Retrieval

In candidate retrieval there exist two types of candidates that we should generate, positive and negative samples. The goal here is to give priority to a certain set of candidates, the positive samples, which are the items that were purchased last by a customer. To mimic the competition testing scheme, we take for each customer in the training data, the items that they bought during the last week of their purchase history. These samples are the ones that are going to be labeled ‘1’ later on when passed to the ranking algorithm to say that they are going to be purchased next week by this user.

Negative samples on the other hand are items that we generate using techniques that we discussed above. The samples are going to be labeled ‘0’ instead, which basically means that they have ‘lower priority’ over positive samples (see Ranking).
For each technique used, we a create a column with the name of the technique and assign ‘1’ if the corresponding sample is extracted using that technique and ‘0’ otherwise. We also create a column with the score of the strategy if it has one.

File: *Retrieval_Ranking.ipynb* [9]
First things first, one should define their training data. In my case I started with the last month of transactions for my first submissions, then I extended it to 7 weeks in order to improve my score.
Then I encoded the date column into weeks, from 1 to 7 (1 being the oldest and 7 the most recent), so that we can select for every customer their positive. I ended up keeping this variable for ranking even after sample selection because I thought it might help to have a time representing variable and, in my opinion, it would be more significant than having the dates.
Now we group our transactions by customer_id and for each customer only keep the transactions that happened during the most recent week of their transaction history (the ones that have the highest number in week columns). We add to these a column called label and assign ‘1’ to all transaction. And there we have our positive samples.

Repurchase is a strategy that I only used in my latest submission. Basically, I went with this scheme:

Training transactions = Positive samples + Repurchase samples

So, the thing to do here is to take the remaining transactions after taking our positive samples and label them as negative samples with ‘0’ in the label column, while assigning 1 to them in a new column named ‘repurchase’.

Next is to generate the rest of our negative samples. Since Item based collaborative filtering using Jaccard similarity gave good results in the previous experiments, I chose it to be my next negative samples generation technique.
As done previously and since we are using collaborative filtering, we need to encode customer and article ids with ascending integers starting from 0, and create dictionaries to keep track of the conversion.
Now instead of taking the best 12 outputs, we are going to select a higher number that we can tweak through a constant called N_CF_items. I started with high values like 100 and 80 but had to reduce it down to 20 for my latest submission for memory constraints reason, as I was adding more and more features.
I also tested the cosine similarity, but as I was expecting, it didn’t give better results.

Popularity is the last negative samples retrieval technique. Like the item similarity we are now selecting the N_MOST_POPULAR items, instead of just 12, and fine tune the number as we try to improve results. I started with 50 than I increased it all the way up to 100 while decreasing N_CF_items. I also created a dictionary of article_id to popularity score (which is basically the number of times an item was purchased) so that we can assign these scores to all the other candidates from the other strategies.

5.2 - Ranking

In a ranking problem we have as input data a set of queries where each query has its own set of documents. These documents have labels that indicates their relevance. The goal of the ranking model is to learn how to rank for each query the documents from least to most relevant.

In our case, the customers are the queries, the products are the documents and the relevance is the binary label we assigned to negative and positive sample to say if they are going to be bought next week or not.

Now that we have generated our set of candidates, it is time to train our ranking algorithm. To do so we need to first prepare and assemble all our data from candidate retrieval. We start with the output of the item similarity algorithm by converting it back to a pandas dataframe and converting the encoded ids back to the original one. We then create a column named ‘CF’ where these candidates are going to receive a value of ‘1’, and in contrast ‘0’ for all the others.

Next, we concatenate the repurchase and positive samples to the CF candidates. Note that item user pairs coming from these three different techniques are mutually exclusive due to the nature of their retrieval. For positive samples and repurchase candidates it is straightforward, as for the item similarity the reason is that collaborative filtering recommends items that have never been bought by the user.

However, when it comes to popularity candidates, there exist intersections with the beforementioned samples. Therefore, we must make an outer join between the two dataframes.
Since it is an outer join, we will have to fill the NaN values that come with the new popularity candidates. It is going to be ‘0’ everywhere: columns representing the other first three retrieval techniques, label column since these are also negative samples and columns related to transactions, like price and week.

All that is left to do now before training the model, is to add customer and articles metadata. For customer metadata I loaded the already cleaned customer dataset and join it to our table to add age and postal code features.
For articles data I chose the following features:
‘product_type_no’, ‘perceived_colour_master_id’ and ‘index_group_name’
Articles data in my opinion can be split into ‘categories’, mainly product related, department related and color related. I chose what I thought was the best combination of features by having one from each ‘category’ while trying to capture as much information as possible. Also all of them were in their numerical form. For limited memory reasons, I wasn’t able to add more features as the dataframe’s size was already considerable at this point.

One last thing to do is to sort our data by customer_id (required for ranking model) and reset the index. I also saved the dataframe in a csv file so I can restart the kernel and free up memory for the remaining of the work.
Here is how our final huge table looks like:

customer_id	article_id	price	sales_channel_id	week	label	CF	CF_score	repurchase	popularity_score	age	postal_code	product_type_no	perceived_colour_master_id	index_group_no
a17956c6d1c3...	915526001	0.000000	0.0	0.0	0.0	0.0	0.000000	0.0	2678	49.0	112978	252	9	1
a17956c6d1c3...	706016002	0.000000	0.0	0.0	0.0	0.0	0.000000	0.0	1151	49.0	112978	272	2	2
a17956c6d1c3...	858856005	0.000000	0.0	0.0	0.0	1.0	0.097682	0.0	174	49.0	112978	275	19	1
a17956c6d1c3...	572998009	0.000000	0.0	0.0	0.0	0.0	0.000000	0.0	1164	49.0	112978	272	2	2

Now we made it to the actual ranking. I chose to use LightGBM which is very powerful gradient boosting ML model. I made this choice for two reasons: the first one is that this model is generally very accurate and its ranking component is one of the best within ranking tasks. The second one is that in spite of its excellent performance, it is nonetheless computationally efficient compared to other powerful models, which is very important in such situation with this amount of data. It was also worth taking advantage of the GPU resource of Colab Pro to buy some time, so I installed a package that is a GPU accelerated version of LightGBM.

The model takes as input not two but three data variables:

- X: The equivalent of our independent variables or features which is everything in the table minus customer id and label.
- y: The equivalent of a target variable which is our ‘label’ column
- group: Here is what makes a ranking model different. This ordered list represents the queries, in our case the customers, encoded in a way where the ith element of this list represents how many documents query i has. This allows the model to recognize at what point in our data we go from query i to query i+1, which is why our table had to be sorted by customer_id earlier.

Cust_id	Art_id
1	1
1	2
1	3
2	2
2	4
3	5
4	1

[3,2,1,1]

For ranking we have to use the class LGBMRanker() from lightgbm, and when creating the model, we have to set objective to ‘lambdarank’. Throughout my initial submissions of retrieval ranking, I haven’t done any hyper parameter tuning in the ranking model, and the model I created each time was using the default values. This is because I was focusing on tuning the candidate retrieval part first, get the best overall strategy, and then I started tuning the ranker. After the ranker finished training, I saved the model using pickle.

When making a prediction, the model outputs a list of scores corresponding to user item pairs in the same order as our input data, which means articles of the first customer then articles of second customers etc.… We concatenate this list as a column to our input dataframe so we can assign those scores to the corresponding pairs. Then we only keep the best 12 scores for each customer.

As a first attempt I only made prediction for customers from the training data, for the other customers I used the 12 most popular items as recommendation like we did in the first submission.

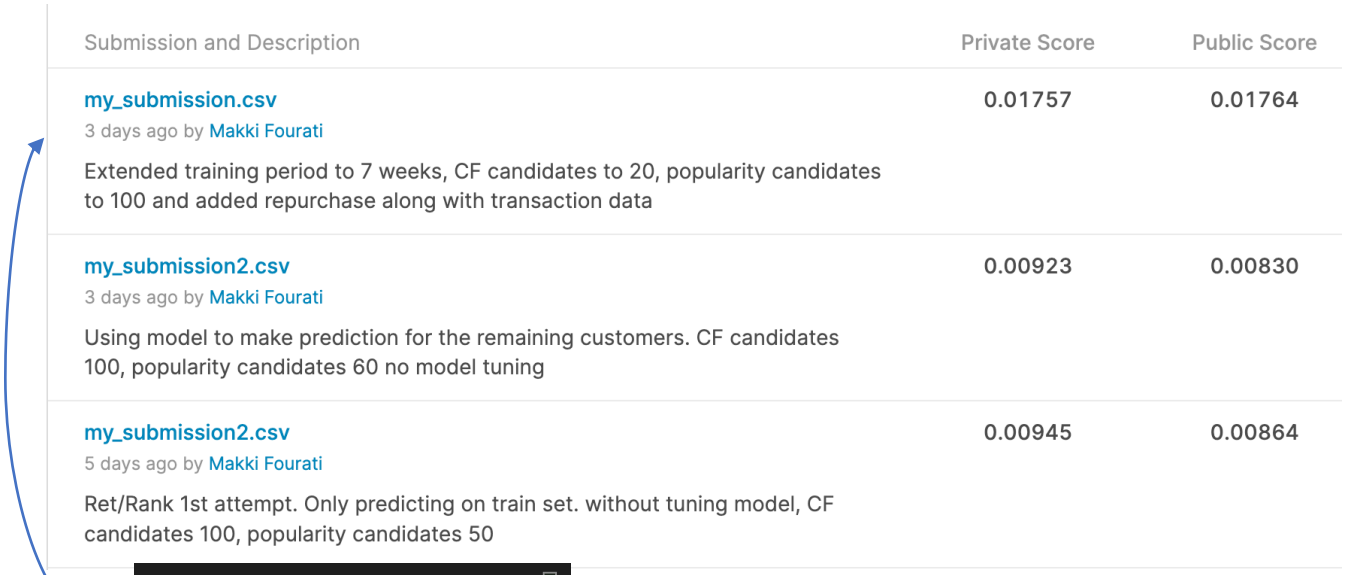
Doing this didn’t improve the score at all, it even decreased a little. So, I moved straight away to making prediction for all the customers using our new ranking model.
However, this is not a straightforward task. We need to generate candidates using popularity for all the remaining customers who are, at least, three times as much in number. A corresponding dataframe would not fit into a 25GB memory.

To cope with this issue, we need to do the candidate generation and prediction by batches. We start by taking a list of the remaining customers to which we add the columns of the other candidate generation methods set to ‘0’. Then we choose the number of batches. Too high and it will be too slow having so many loops doing heavy tasks, too low and it can crash the memory. I went with 8 which was a sweet spot.

For each batch we will take a portion of the customers, make a cross product with the N_MOST_POPULAR items and their popularity score, join article and customer metadata, pass the resulting dataframe to the model to make prediction and finally take the 12 best recommendation per customer as we did before.

The order of the columns must be the same as in X_train since lightgbm does not recognize the features names.

Here are the most important submissions I made for the retrieval ranking part. They are listed from most recent to oldest



Submission and Description	Private Score	Public Score
my_submission.csv 3 days ago by Makki Fourati Extended training period to 7 weeks, CF candidates to 20, popularity candidates to 100 and added repurchase along with transaction data	0.01757	0.01764
my_submission2.csv 3 days ago by Makki Fourati Using model to make prediction for the remaining customers. CF candidates 100, popularity candidates 60 no model tuning	0.00923	0.00830
my_submission2.csv 5 days ago by Makki Fourati Ret/Rank 1st attempt. Only predicting on train set. without tuning model, CF candidates 100, popularity candidates 50	0.00945	0.00864

```
model = lightgbm.LGBMRanker(
    objective='lambdarank',
    learning_rate=0.001,
    n_estimators=300,
    num_leaves=40,
    n_jobs=-1,
)
```

We can see here the parameters of the model that was trained for the last submission.

It is worth noticing the important increase in private score between the last submission and all the others.

Note about validation: It goes without saying that validation is a crucial step in training a model, which is not an exception here in recommendation. Normally, one should have considered the last week of transactions as validation set and created a local validation function with MAP@K to validate the model. However, for time constraints reason and because this project is not just about getting the highest score possible in the competition, I trained the model on all the training data and tested on the Kaggle test set.

Also since the competition was already over when I was making my last submissions, the private score was available when submitting, which gives a good idea on how the model really performed.

IV Category recommendation

As a second part of this project, I decided to look at the problem from a different perspective, outside the scope of the competition. Instead of recommending specific products for every customer, we want to recommend a certain category to a certain set of customers. A

category can be anything from department, section, product type... I chose to work with the latter in this project.
This would be useful mainly in scenarios like when the fashion company is making a discount on a certain selection of products during a certain period. Before the launch of this event, we want to inform and target in advance (promotion emails for ex) the customers that might be interested, without bothering the ones that are not concerned.
This problem was addressed using two different methods, which are going to be discussed in the following sections.

Remark: Building a recommender system to achieve this goal (for example selecting from the list of recommended products the most prominent category) is not ideal as it is not going to give accurate results and will be too much computational power for a problem we can solve using more efficient methods.

1 - 1st attempt: Multi class category classification

The first idea was to translate this problem into a multi class classification problem, i.e., we take a set of customers along with their purchase history and personal data and try to predict the category of their next purchased item. Our target variable here is therefore a category of product among a set of N categories.
The purchase history is encoded into N columns, each one representing a category and having as values the number of times a customer has purchased an item of that type.

File: *Category__Rec.ipynb* [9]
We start off by loading the purchase history data from transactions.csv and extract all purchases made past a certain date. Here I chose it to be 22nd of January 2020 which is 8 months before the last date of transactions. The motivation behind that was to capture any seasonality in the customers behavior since we are working on a fashion problem. We also want enough data to feed our purchase history features.

We then add the articles data by loading articles.csv, and then group all items by category, which I chose it to be the product_type_name variable. We can see the result of the grouping in the table on the right:

	product_type_name	count
25	Clothing mist	4
110	Toy	5
20	Bumbag	16
109	Towel	20
36	Eyeglasses	25
...	...	...
107	Top	1583408
104	T-shirt	2203750
99	Sweater	2783274
32	Dress	3238428
111	Trousers	4217017

130 rows × 2 columns

We can see that the distribution is highly skewed and that we have 130 different categories which is not ideal. Therefore, we proceed to remove purchases of categories that appear less than a certain number of times in order to reduce class imbalance and have enough data for each class. I chose this number to be 20’000 which gives us 62 most purchased categories to work with.

From these transactions we group customers and compute how many purchases each one of them made.
Then we remove customers with a purchase_count less then 25 to avoid sparse data, leaving us with 365899 customers. We also subtract 1 from the purchase count (number_purchases), since we don’t want to include the last purchase as it is our target.

Now we can start building our final dataframe. From the transactions we create a copy and keep only one row for each customer

The next step is to prepare our target variable and encoded purchase history. The goal is to have for each customer the N-1 first purchased item encoded as features and the single most recent purchase as target variable.

To do that we start by iterating through customer_id grouped transaction data and for each customer sort transactions by date, take the most recent one and add the corresponding row to a separate dataframe called target, while discarding it from the original dataframe. Then we store the id of the customer in a dictionary as a key and in its corresponding value, we store a dictionary having as keys the different types of products that the customer bought and as values, the corresponding number of times each type was purchased.

Now in our final dataframe, we initialize a new column with 0’s for each one of the 62 types of products and then populate them using the dictionary that we just created.

Now that we have our data ready, we can apply machine learning models to make the predictions

	F1-score on train	F1-score on test
KNN	0.16	0.088
Neural Net	0.2	0.16
LightGBM	0.23	0.17

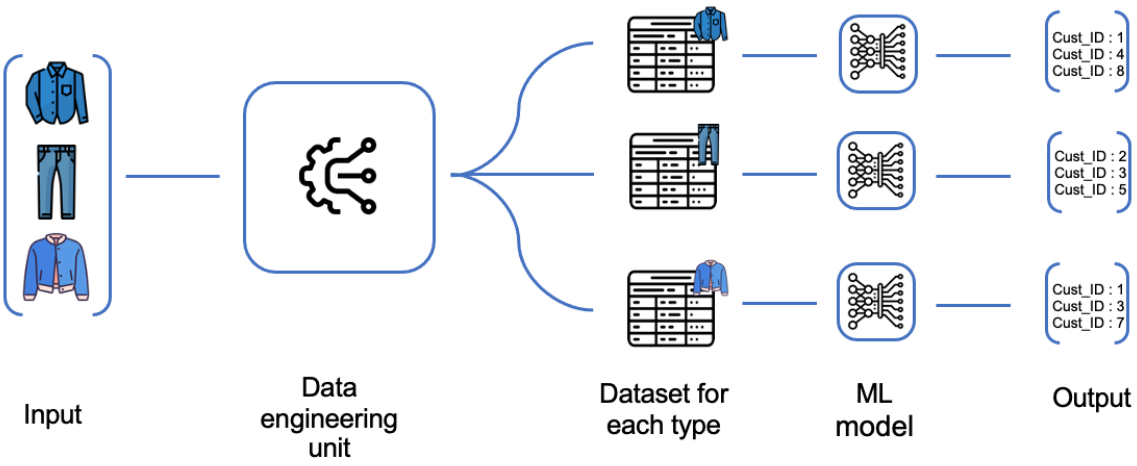
Unfortunately, despite a lot of model selection and the complexity of certain models, I couldn't achieve an F1 score higher than 0.2. This led me to thinking that the problem might come from the data and how the problem was conceptualized in the first place. One of the reasons that I think might be responsible for this bad performance is the fact that the last purchased item does not come from the same period of the year (or month) from one customer to another. This can have as effect to confuse the model since the type of products is highly affected by time in fashion. Including time representing variables might not solve this issue as it is not commonplace to capture all the customer behavior froma time perspective. Reducing the data to a restricted period of time would probably alleviate this ambiguity but it will in return make the purchase history features poorly populated and almost obsolete.

The other point is that when we are predicting, we are only outputting one category per customer, the one with the highest score the model computed. The category of interest that is concerned by the discount might be the second or third best option, which means that we will be missing a lot of key customers.

Having this in mind, I came up with another strategy that aims at overcoming these two issues and addresses the problem in a more efficient way.

2 - Binary classification for a set of categories

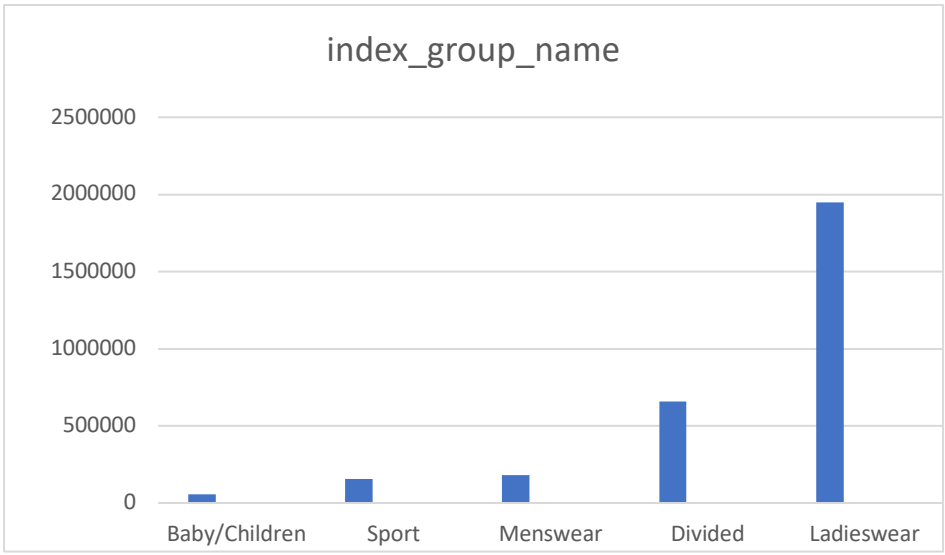
File: *Category__Rec_2.ipynb* [9]
In this section, we are going to use a more elaborate approach to achieve
the same general goal that we wanted to achieve in the previous section. The idea is to build a system that takes as input a list of categories of products (issued by a certain department eventually) and outputs for each one of these categories a list of customers that are most likely going to be interested in the promotion of the category in question, and this through a binary classification machine learning model.



The overall system works for any given list and the data engineering part works on all the different elements of the list at once up till the creation of the processed datasets. The ML part must be done individually for every category of product since data varies along these categories and different analysis, models and techniques should be applied in each case to get accurate results.

2.1 - Data engineering

We start by taking only the last 2-3 months of transactions, to which we join not only product types data but also another feature called index_group_name. This feature essentially represents the department to which belongs the product, something that could be useful for our model. The figure below shows the distribution of this feature among the transactions.



This time we will construct our target differently. We want to predict for all the users whether they are going to buy a product that falls under category X during a fixed period which is the last 2 weeks of the data (9/9/2020 to 22/9/2020).
Hence, our features will be constructed from the transaction data of the several weeks preceding the period mentioned above. This way we have on one hand consistency along the timeline and on the other the model will mimic the expected behavior which is predicting for the short-term future.
In this regard, we split our purchase history data into two parts:

- The last two weeks: used to construct our target variable
- The remaining first weeks (≈ 2 months): used for creating the features

Next, from these two datasets, we filter out the products that don't belong to any of the categories within our input list. In addition to that we create a dataset with unique customer ids on top of which we will construct our final dataset.

Then we proceed with three different encodings done through a function that takes as input the unique customer ids dataset, a transaction dataset, the aggregate function for grouping and the column we want to encode. Here is a breakdown of the encodings:

1) **Encoding the target variables**: we go to the filtered last two weeks of transactions and OneHotEncode the product_type_name column and then group by customers and take the max. This way we have N encoded columns, one for each category, populated with 1's and 0's respectively for customers that will or will not purchase the ith category during the next two weeks.

2) **Encoding category count:** here we are basically encoding from the first weeks of transactions the purchase history of each customer for every one of the categories. In other words, for every customer, we will have the number of purchased items along each category. Therefore, the process is similar, but we are working with a different transaction period and instead of taking the max after grouping, we take the sum.

3) **Encoding index_group_name:** we essentially want to have a feature for every index group that represent how many items from that group were purchased. But a very important point that differs from the other two encodings is that **we must use unfiltered transaction data**.

This made a humongous jump in performance in the ML model, which makes sense in a way because the index group is a more general feature than product type, so by encoding filtered transactions we are not taking into account other categories of products (other than the ones on the list) that belong to the same index group and the final number will be misleading since it might not represent well the customer.

Right after this, here is what our data looks like:

customer_id	target_Bra	target_Jacket	target_Trousers	Bra	Jacket	Trousers	Baby/Children	Divided	Ladieswear	Menswear	Sport
d7c16d184c5ba...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	4.0	0.0	0.0
a622349a3ca01...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	8.0	4.0	0.0	0.0
a635584ffb83b6...	0.0	0.0	0.0	0.0	0.0	3.0	0.0	0.0	10.0	0.0	4.0
3bedca29c9fb41...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.0	0.0	0.0
e6f7c934cd4aeb...	0.0	0.0	0.0	0.0	0.0	2.0	0.0	0.0	6.0	0.0	1.0

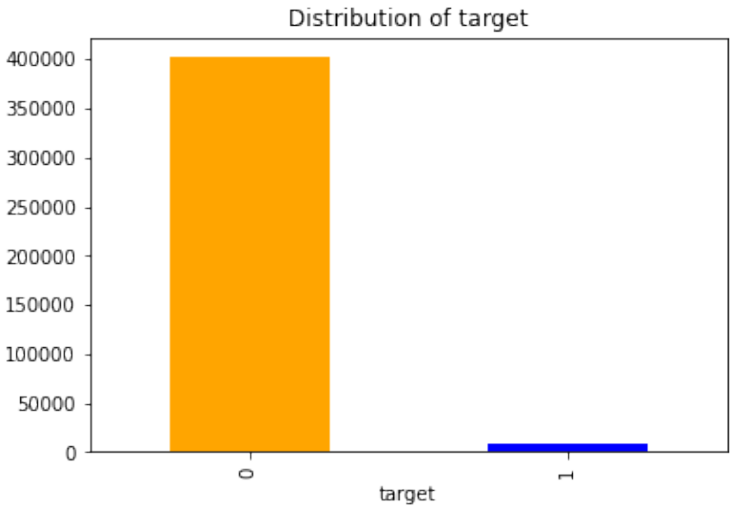
We also join customer related data and do our usual cleaning and preprocessing as in the previous parts of the project.

Now comes the last part of the data engineering unit, which is creating N different dataframes, one for each category. Each one will have the corresponding target variable and corresponding category count. The rest of the data is the same. We store these dataframes in a dictionary as values with the name of the category as key.
As mentioned at the beginning of this section, from now on each dataset will be treated individually.

In my notebook I chose one of the categories to work with as an example. My choice landed on the category ‘Jacket’, which was kind of a middle ground since I didn’t want to work with one that was very present in the data neither with one that was rarely purchased, and this in order to have more or less representative results.

2.2 - Model

From the dictionary of dataframes we get the one corresponding to the Jacket category. First thing I did is to look at the distribution of the target variable.



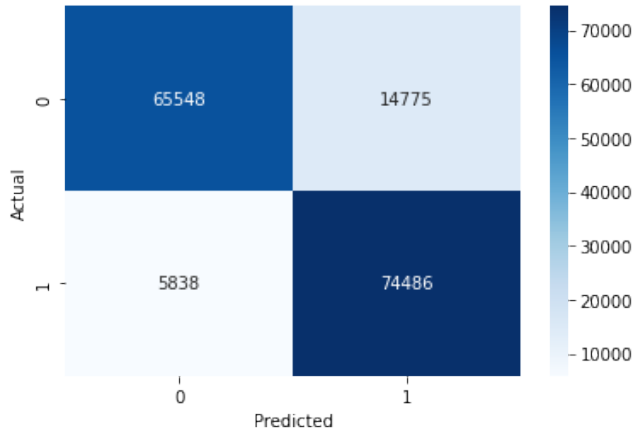
As we can see there is a considerate imbalance in our data that can affect our model's performance. To cope with this problem, I used the library SMOTE (Synthetic Minority Over-sampling Technique) which over samples the minority class data using nearest neighbors. As opposed to random sampling, this method will generally lead to better results, especially with a big imbalance. It is worth mentioning that data should be standardized first since this technique uses k nearest neighbors to generate samples.

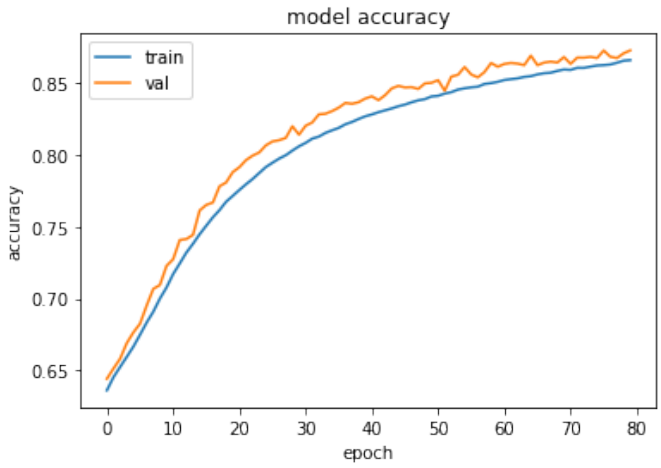
The best performing model is a Neural Network with the following architecture:

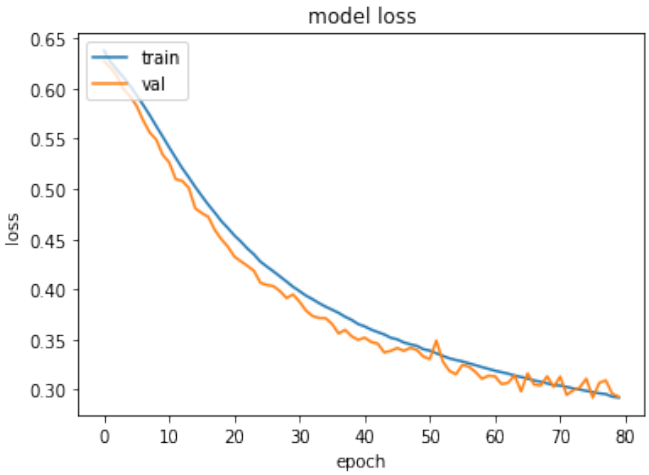
```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 dense (Dense)                (None, 1000)              13000

 dropout (Dropout)            (None, 1000)              0

 dense_1 (Dense)              (None, 2000)              2002000

 dense_2 (Dense)              (None, 2000)              4002000

 dense_3 (Dense)              (None, 1)                 2001

=================================================================
Total params: 6,019,001
Trainable params: 6,019,001
Non-trainable params: 0
_________________________________________________________________
```

Here are the results of the model

```
***RESULTS ON TRAIN SET***
f1_score:  0.8906547591655446
Precision:  0.8514070016603046
Recall:  0.9336958280940701
--
***RESULTS ON TEST SET***
f1_score:  0.8784503346404458
Precision:  0.8344741824537032
Recall:  0.9273193566057467
```







We can see that recall is slightly higher than precision and F1 score, which is due to the fact that I chose the probability threshold to be 0.45 instead of 0.5. The motivation behind that is that in this scenario we would rather have false positives than false negatives, which means that we prefer sending discount emails to people that were not necessarily going to buy, then missing out on the ones that were. Moreover, despite the complexity of the model, over fitting was not an issue.

V Conclusion

Recommendation system is a complex and very interesting topic in machine learning. The complexity makes it such a unique and challenging problem as there exists endless ways to go about it and the number of parameters and variables to play with doesn't come any close to what we usually see in machine learning problems. It is worth mentioning though that this competition is on the harder side of the spectrum and working on it is probably not the best way to discover the topic. Nonetheless I really enjoyed this competition with its challenges and the learning opportunity it gave me.

It was also interesting to see the problem from another perspective outside the scope of the competition and having to define my own guidelines and goals while applying what I already know to get insightful results.

Challenges:

The first and biggest challenge throughout the project was to get the right information and learning material about this advanced topic.

Working with such big dataframes was also another challenge, which made me learn a few new methods and optimization practices to be able to overcome memory and computational issues.

VI References

[1] Z. Ahmad, «medium.com,» [En ligne]. Available: https://medium.com/analytics-vidhya/recommender-systems-explicit-feedback-implicit-feedback-and-hybrid-feedback-ddd1b2cdb3b#:~:text=One%20type%20of%20feedback%20is,after%20observing%20the%20users'%20behavior..

[2] H. Group, «Kaggle,» [En ligne]. Available: https://www.kaggle.com/competitions/h-and-m-personalized-fashion-recommendations/overview/description.

[3] h-and-m-personalized-fashion-recommendations, «Kaggle,» [En ligne]. Available: https://www.kaggle.com/competitions/h-and-m-personalized-fashion-recommendations/discussion/307001.

[4] g. c. training, «Coursera,» [En ligne]. Available: https://www.coursera.org/learn/recommendation-models-gcp.

[5] L. p. Inc, «udemy,» [En ligne]. Available: https://www.udemy.com/course/recommender-systems/.

[6] Apple, «apple.github.io,» [En ligne]. Available: https://apple.github.io/turicreate/docs/api/turicreate.toolkits.recommender.html.

[7] Wikipedia, «Wikipedia/,» [En ligne]. Available: https://en.wikipedia.org/wiki/Matrix_factorization_(recommender_systems).

[8] Google, «developers.google.com,» [En ligne]. Available: https://developers.google.com/machine-learning/recommendation/collaborative/matrix.

[9] M. Fourati, «Github,» [En ligne]. Available: https://github.com/MakkiFourati/RecSys-Thesis.