Balancing the trade-off between availability and accuracy for neural collaborative filtering models

Frederick Nilsen

22.12.2022

Abstract

The rapid digitalization of online shopping has led to a world-spanning market with both many customers and vendors competing for market shares. In addition, the digitized society has also led to a large amount of information for individual customers and vendors' products. By utilizing this information to its fullest, it is possible to provide a better shopping experience to customers by recommending specific items to each customer. To this end, this project aims to explore some of the most widely used recommendation systems in the context of H&M's publicized customer purchase data. In particular, we are interested in exploring the time complexity for different models and the trade-off between efficiency and accuracy. This is done by implementing a neural collaborative filtering model with matrix factorization, whose baseline model only uses transactional data. Thereafter, an extended model using secondary information about both the customer and the purchased articles can be trained in a similar fashion and the model results compared.

The results show that the choice of the underlying data structures is crucial in providing recommendations quickly. Extending the model does not seem to affect time complexity particularly, although the prediction quality is heavily dependent on the number of samples included in the training. In addition, including more data will negatively impact the run time of the models, demonstrating a clear trade-off between accuracy and time complexity. To address the increased complexity, we propose several measures, such as using more data and combining models with simple aggregations of all data.

Sammendrag

Den raske digitaliseringen av netthandel har ført til et verdensomspennende marked med både mange kunder, og leverandører som konkurrerer om markedsandelene. I tillegg fører også det digitaliserte samfunnet til store mengder informasjon, både for enkelte kunder og leverandørenes produkter. Ved å utnytte denne informasjonen i størst mulig grad, kan man gi en bedre opplevelse til kundene ved å anbefale spesifikke artikler til hver kunde. I forlengelse av dette ønsker prosjektet å utforske de mest brukte anbefalingssystemene i sammenheng med H&Ms offentlig publiserte kundekjøpsdata. Mer konkret er målet å undersøke tidskompleksitet for ulike modeller og dermed avveiningen mellom effektivitet og nøyaktighet. Dette gjennomføres ved å implementere en nevral, kollaborativ filtermodell med matrisefaktorisering, der grunnmodellen kun bruker transaksjonsdata. Deretter trener vi en utvidet modell som bruker sekundær informasjon om kundene og artiklene kjøpt på tilsvarende måte, slik at resultatene kan sammenlignes.

Resultatene viser at valg av de underliggende datastrukturene er avgjørende for å produsere anbefalinger effektivt. Utvidelsen av modellen ser ikke ut til å påvirke tidskompleksiteten i særlig grad, men prediksjonskvaliteten er svært avhengig av antall transaksjoner inkludert i treningen. I tillegg vil mer data påvirke kjøretiden til modellene negativt, noe som viser en tydelig avveining mellom nøyaktighet og effektivitet. For å bøte på den økte kjøretiden foreslås flere tiltak, slik å bruke mer data eller å kombinere modellene med enklere aggregeringer av den fullstendige dataen.

Contents

Figures

Tables

Acronyms

AP acerage precision.

CCS complete cold start.

CF collaborative filtering.

CS cold start.

EDA exploratory data analysis.

H&M Hennes and Mauritz.

ICS incomplete cold start.

KDE kernel density estimation.

MAP mean average precision.

MLP multilayer perceptron.

NCF neural collaborative filtering.

OOM out of memory.

RGB red, green, blue.

SSH secure shell.

Chapter 1

Introduction

A recommender system is a system whose goal is to predict a subset of available items that fit a given user well. "In general, recommendation lists are generated based on user preferences, item features, user/item past interactions" (Zhang et al. 2019). With a more digitized world with services providing their users more options than ever before, a recommender system is pivotal for querying the most relevant options for each user. Some areas where recommender systems are the most prevalent, are for instance recommending movies or videos on a video streaming platform, recommending specific news articles or posts on a news outlet, or recommending articles in online retail.

In particular, larger online stores are challenged with both a large volume of customers from all over the world, but also a large volume of widely different articles for purchase. For instance, Amazon.com reports over 300 million active users worldwide (Quaker 2022). Since the customer base is such a varied set of people, it is not sufficient to merely show the most sold items as recommendations to the different users. Rather, the retailer has to utilize the purchase history of a given customer to distinguish patterns that can be used in a more targeted prediction.

Not only is the customer base diverse, but recommender systems also play a role in scenarios with a diverse set of articles. In normal businesses, the Pareto principle states that most of a business's sales (around 80 %) come from about 20 % of the available items. In contrast to this, we can in e-commerce find a long tail phenomenon, so that niche commodities become a more important part of the business's revenue (Brynjolfsson, Y. Hu et al. 2011). In fact, we can see that recommender systems play a role in the first-order demand-side effects of the long tail (Brynjolfsson, Y. J. Hu et al. 2006). With the effect of more niche items available, the utility of personalized recommendations is also increased.

This project aims to look into such a targeted prediction, and in particular, how to effectively predict articles to registered users of H&M Group. In order to make predictions, we can make use of transactional data (whether or not a specific customer has purchased a specific article), but also specific customer and article data, such as customer age and article product type. Thus, when discussing recommender systems, it will be done in the context of this data for an e-commerce platform.

In addition to providing an accurate prediction of items relevant to each user, the system must also be effective and fast in providing these results. If a system requires too much time before displaying the prediction, the user will most likely exit the service before being presented with the (accurate) results.

For the purposes of this report, the research questions can be summarized as the following bullet points.

- What implementational aspects are important when considering the computational complexity and accuracy of a neural collaborative filtering model?
- How do variations of collaborative filtering models perform to the real-world data?
- What are some possible measures to solve the challenges of traditional collaborative filtering models?

In order to answer the research questions, the report consists of explaining the data used, followed by an exploratory data analysis. Thereafter, relevant background theory as well as the method and model implementations are presented, before the results are shown. What remains is a discussion of the results supporting a final conclusion.

Chapter 2

The dataset

2.1 Authors of the dataset

The dataset is a publicized portion of online sales from H&M Group, also known as *H&M Hennes & Mauritz GBC AB*. The published data was a part of an online competition for the spring of 2022 hosted at the website Kaggle[1].

"H&M Group is a family of brands and businesses with 53 online markets and approximately 4,850 stores" (Ling et al. 2022). Their stores are not limited to H&M but also include brands like Weekday, Monki, and Afound (*Brands - H&M Group* 2022).

2.2 Exploratory data analysis (EDA)

The complete dataset consists of three tables of customer data, article data, and transactional data referencing the customer and article IDs. In addition, images of articles are included in the dataset, whose filenames are the article IDs. We will perform an exploratory data analysis (EDA) for these 3 portions of the dataset.

2.2.1 Customers

Each entry of the customer table has a unique ID, postal code, age, and membership status. The latter includes whether or not they are signed up for *Fashion newsletter* (`FN`), if the account is active, club member status and fashion news frequency. Of these, we can see that `FN` and `Active` are binary variables and that the two others are categorical with 4 and 5 levels respectively,

```
FN [nan  1.]
Active [nan  1.]
club_member_status ['ACTIVE' nan 'PRE-CREATE' 'LEFT CLUB']
fashion_news_frequency ['NONE' 'Regularly' nan 'Monthly' 'None']
```

We note that in reality, `fashion_news_frequency` only has 3 different levels since `'NONE'`, `'None'` and `nan` all indicate the same state.

Furthermore, it is of interest to look into the age distribution for all customers. Here we can find the mean to be 36 years old and the range of ages spans across $[16, 99]$. Looking more into the distributions themselves, we find there to be a bimodal distribution whose modes lay around 20 and 50, as shown in Figure 2.1. The bimodal property of the customer age data can be important when considering models. Attempting to fit with the same model for all customers not considering the age distribution (explicitly or implicitly) may lead to worse predictions.

[1] https://www.kaggle.com/competitions/h-and-m-personalized-fashion-recommendations/

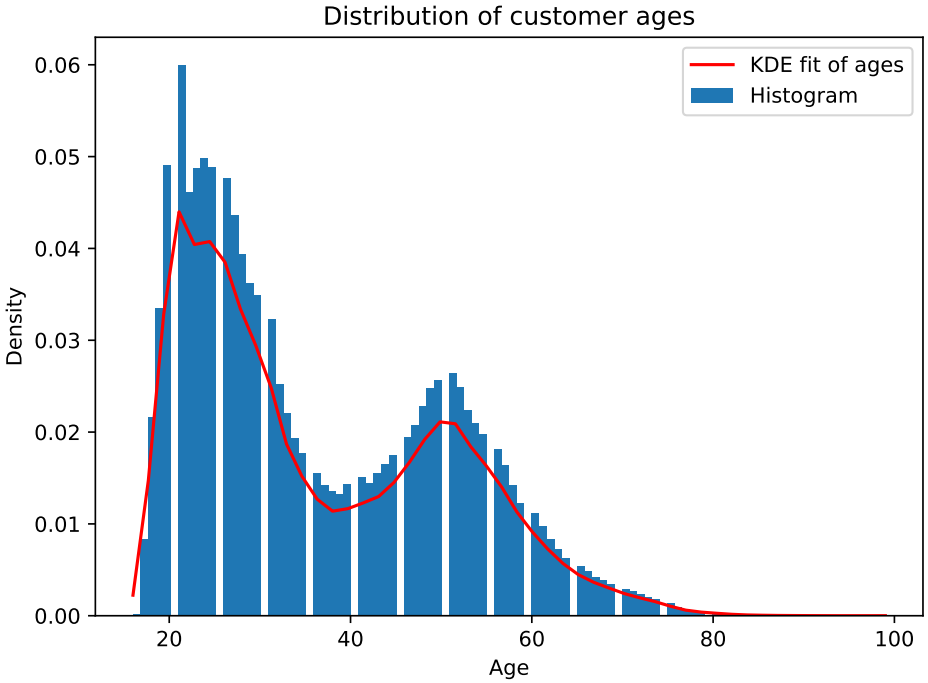


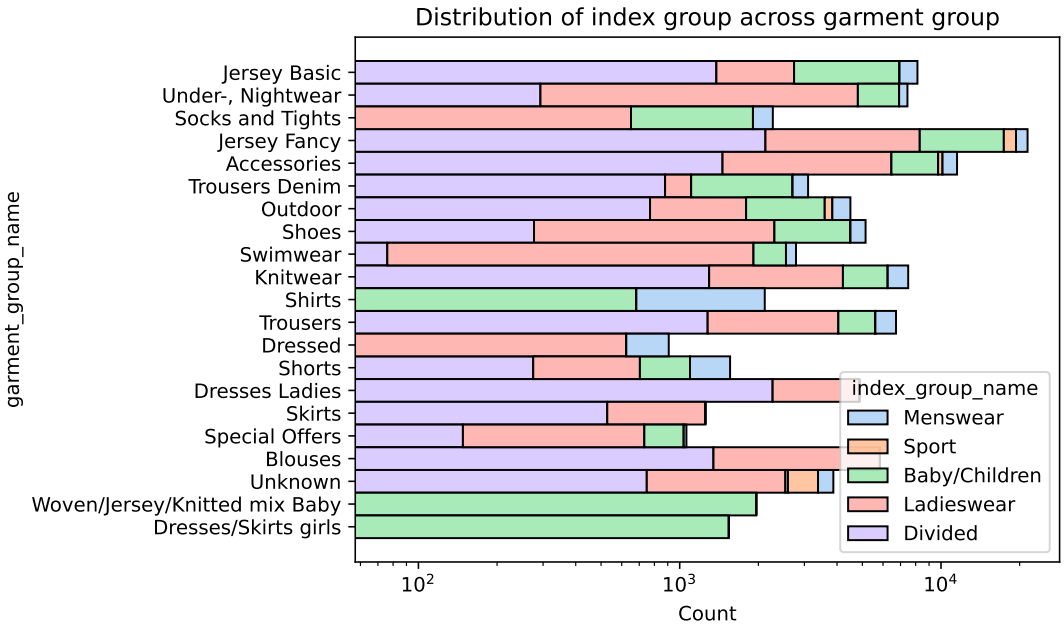
Figure 2.1: Distribution of ages across all customers in the dataset.

2.2.2 Articles and images

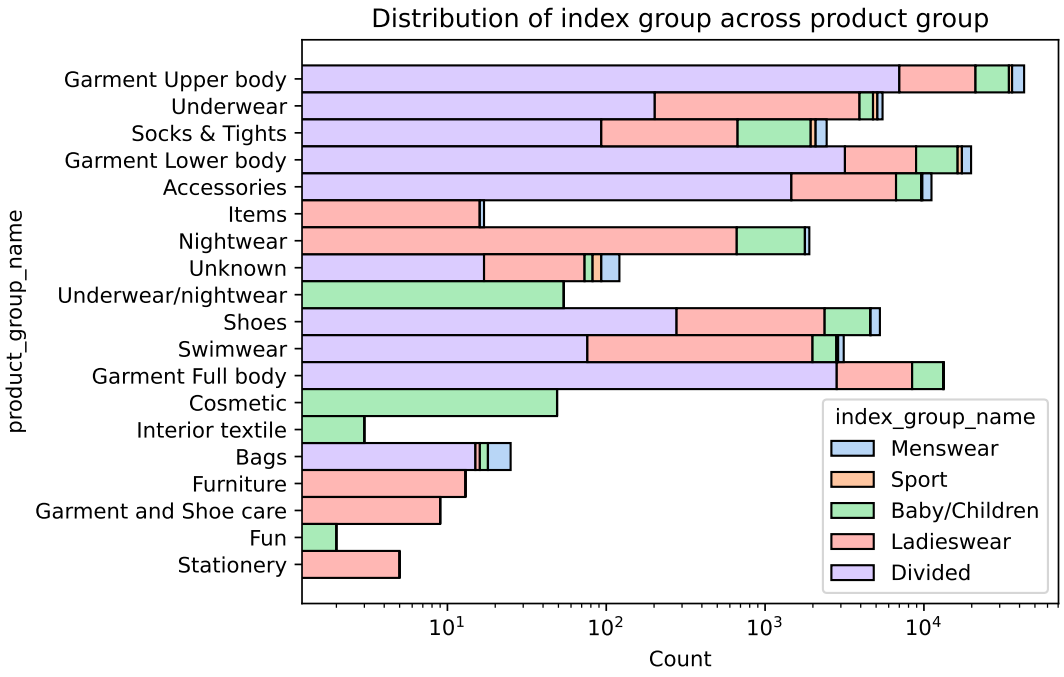
The images are all in 3-channel RGB so that it has the dimensions $(h, w, 3)$, where h and w are the image's height and width. The most common height and width dimension is $1,166 \times 1,750$ pixels, but several articles' images deviate from this measure. More specifically, 89.1 % of all images have this dimension within ± 1 pixel different for either h or w. The rest are widely spread, seeing that there are 2804 different combinations of dimensions where in that $1,008$ of those are unique for one image.

In addition, all images have either a light grey or white background. Note that not all articles have an image. There are a total of $105,100$ articles with an image, i.e. 99.56 % of all articles.

The article data is *denormalized*, meaning that all entries have both a data index and the corresponding data itself, e.g. that every product group has a product group ID. The raw data columns are the product name, type and group, the graphical appearance name, color group, perceived color and perceived master color name, department, index, index group, section, and lastly garment group names. There is also a hierarchical structure for some of these attributes, such as *index group* to *index* and *product group* to *product type*. To gain some insight into what kind of products the data contains, we can plot the index group in relation to both the product group and garment group, as shown in Figure 2.2.



(a) Garment group name



(b) Product group name

Figure 2.2: Product categories

We notice that the most populous garment groups/product groups are in majority from the *divided* index group, meaning articles mainly targeted to teenagers. Another noticeable observation is that the index groups *sport* and *menswear* are relatively sparse and not present in most garment and product groups. Some groups seem to almost entirely consist of *Babies/Children* and *Womenswear*, which is also worth noting before implementing some neural architecture using these data points.

2.2.3 Transactions

The last part of the dataset is a 3.24 GB large set of transactions that have taken place, totaling about 31.7 million transactions. Each transaction contains a time stamp, customer ID, article ID, price and sales channel. The last of which indicates if it is an in-store purchase or online purchase and is not used

	article_id	prod_name	product_type_name
37399	653551001	Benji leather jacket	Jacket
50826	697511001	PE LINDA LEATHER COAT	Coat
79375	797432001	CEMENT leather jacket	Jacket
83012	810872001	PQ AGDA LEATHER DRESS	Dress
89691	839478001	PQ SUSAN LEATHER TUNIC	Dress

Table 2.1: The 5 most expensive articles in the dataset

throughout the project. Upon further analysis, we find that 9,699 customers have not purchased any item whereas the maximum number of articles a single user has bought, is 1,895. The density is shown in Figure 2.3, where the bins are logarithmically scaled. Although the vast majority buys very few articles, the total mean is about 23 articles for each customer, as indicated by the vertical line in the histogram. The distribution shows that a lot of users purchase very few items, which results in a sparse dataset.

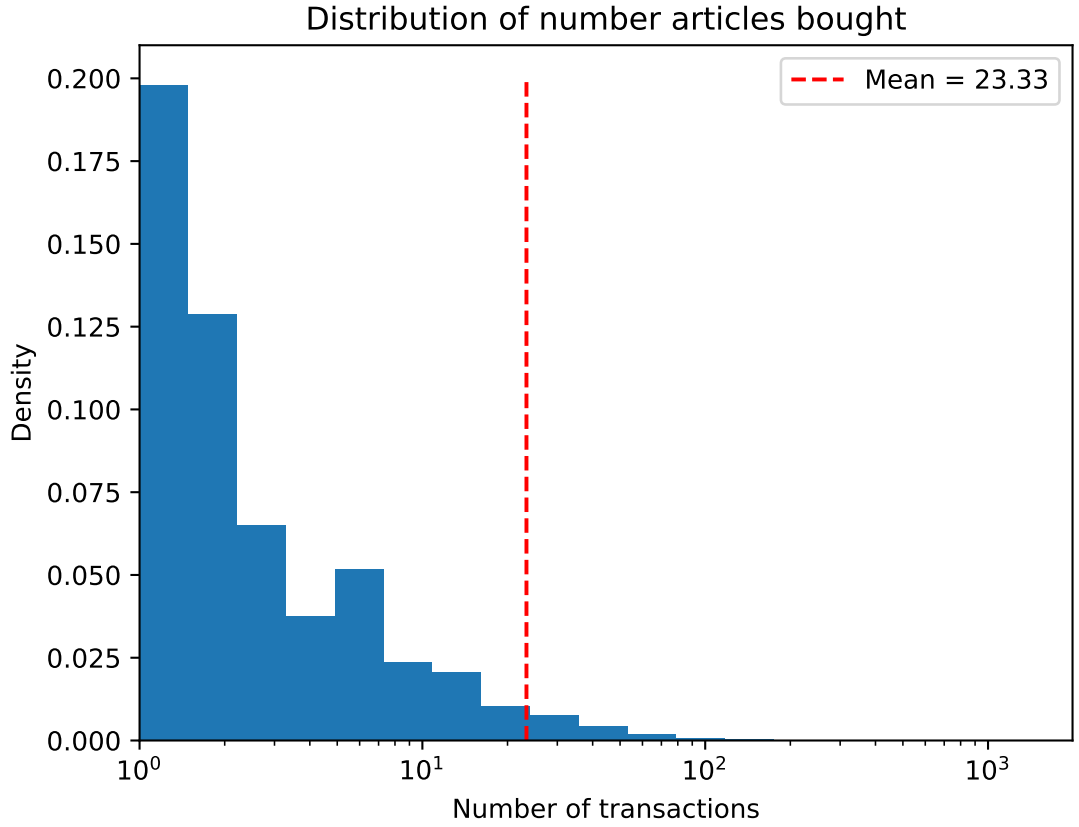


Figure 2.3: Density of number of articles bought for each customer.

Another interesting part of the transactional data is the price. We can find the largest all-time prices, highlighted in Table 2.1, where we can note that all items are relatively similar in that they're made out of leather.

The last characteristic of interest is the time for the transaction. We generate a box and whisker plot to see when the most transactions are occurring. In Figure 2.4 it is apparent that June in both 2019 and 2020 has the largest mean of transactions. The figure also highlights the presence of several outliers, most notably around September-November 2019.

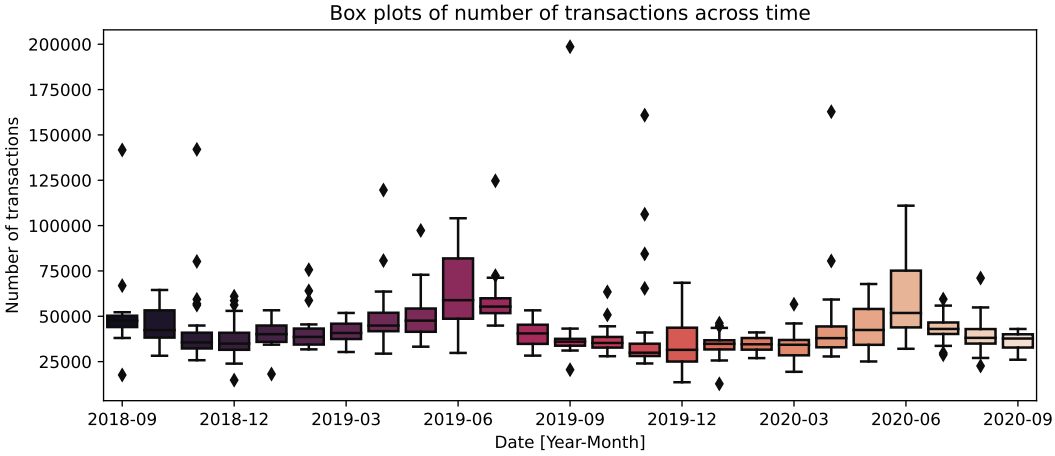


Figure 2.4: Boxplot of the number of transactions over the different months in the dataset.

Chapter 3

Theory

3.1 Recommender systems

In general, any method yielding a set of recommended products, items or articles can be defined as a recommendation system. The most rudimentary system may only aggregate over all purchases and return the n most purchased items of all time, disregarding within-customer variation and the temporal dimension. This can be modified so that the system instead drills down on purchase date, e.g. only considering the k last days of transactions. In general, the process of producing a recommendation can be formalized as follows. We denote U as the set of all users and V set of possible recommendations. Consider also the utility function $r : U \times V \to R$ measuring the usefulness. Thus, the goal is to for all $u \in U$, finding the $v'_u \in V$ such that (Adomavicius and Tuzhilin 2005)

$$\forall u \in U, \quad v'_u = \arg\max_{v \in V} r(u, v) \tag{3.1}$$

It is challenging to define the utility function in (3.1), especially in cases where a user u has not explicitly provided any form of rating to the item v. The three main types of recommender systems tackling this challenge, are content-based systems, collaborative systems and hybrid approaches (Adomavicius and Tuzhilin 2005).

3.1.1 Content-based recommendation

In a content-based system, the extrapolated $\hat{r}(u, \tilde{v})$ for an unrated $\tilde{v}$ will be based on the knowledge of the users and the articles themselves. This is done by introducing an item and user profile, containing significant information to be able to recommend an item. The *item profile* contains identifying features for the item, e.g. the genre in the case of movie recommendation, and each item has its own profile. The *user profile* can contain information about what it previously has rated positively (e.g. which genres it prefers), which can then be compared to the item profile to produce a recommendation.

A favorable property of content-based recommendation is that new and unrated items can also be recommended. Even if an item has no ratings, i.e. $\nexists r(u, v)\ \forall u$, the item v can still be recommended if v's item profile is sufficiently similar to a user profile. Notice however that content-based systems are entirely dependent on a user profile to be able to recommend any items. In addition, the system locks every recommendation to be within the user's defined user profile even though the user might enjoy articles distinct from their historical ratings. Another challenge is to know precisely what parts of the item information should be included in the item profile, which might lead to computationally expensive analyses.

3.1.2 Collaborative filtering

The other main category of recommender systems is called collaborative filtering systems (CF), and share the property of basing a recommendation on other users' ratings rather than only considering the one user u. These systems attempt to first find $U_{(i)} \subseteq U$ where $U_{(i)}$ contains the customers most similar to u by some similarity measure.

When it comes to selecting similarity measures, the Jaccard similarity, cosine similarity, and Pearson correlation can be used. However, we must first establish some notation before following with the definitions of said metrics.

First, let the set V_x contain all items in which user x has given the item a rating and V_{xy} the items where both x and y have rated. Similarly, let U_x denote the set of users whose item x has a rating. Please note that U_x is unrelated to the general subset previously denoted as $U_{(i)}$. In particular,

$$\begin{aligned} V_x &= \{i \in V \ : \ \exists r(x,i)\} \\ V_{xy} &= \{i \in V \ : \ (\exists r(x,i) \wedge \exists r(y,i))\} \\ U_x &= \{j \in U \ : \ \exists r(j,x)\} \end{aligned}$$

Define also $\bar{r}(u) = 1/|V_u| \ \sum_{i \in V_u} r(u,i)$, i.e. the average of all ratings for user u. The aforementioned similarity metrics are thus expressed as

$$s_{\text{Jaccard}}(x,y) = \frac{|V_{xy}|}{|V_x| + |V_y| - |V_{xy}|} \tag{3.2}$$

$$s_{\text{Cosine}}(x,y) = \frac{\sum_{i \in V_{xy}} r(x,i) r(y,i)}{\sqrt{\sum_{i \in V_x} r^2(x,i)} \sqrt{\sum_{i \in V_y} r^2(y,i)}} \tag{3.3}$$

$$s_{\text{Pearson}}(x,y) = \frac{\sum_{i \in V_{xy}} (r(x,i) - \bar{r}(x))(r(y,i) - \bar{r}(y))}{\sqrt{\sum_{i \in V_{xy}} (r(x,i) - \bar{r}(x))^2} \sqrt{\sum_{i \in V_{xy}} (r(y,i) - \bar{r}(y))^2}} \tag{3.4}$$

After finding the subset $U_{(i)}$ of the top N users based on (3.2), (3.3) or (3.4), $\hat{r}(u,\tilde{v})$ can be estimated using the ratings from users in $U_{(i)}$. For instance, if $U_{(i)}$ is the top N users most similar to u, and assuming (without loss of generality) all $u' \in U_{(i)}$ have an explicit rating for item v, the prediction can be computed using some aggregation such as the mean,

$$\hat{r}(u,\tilde{v}) = \frac{1}{N} \sum_{u' \in U_{(i)}} r(u',\tilde{v}). \tag{3.5}$$

One immediate flaw of only computing the average is that some items may be particularly good to recommend, or some users might generally rate items higher than the average user. This can be remedied by introducing a baseline value, consisting of e.g. some deviation measure between u's average ratings and all users' average ratings (b_u), and the mean ratings $\mu_{\tilde{v}}$ defined as

$$\mu_{\tilde{v}} = \frac{1}{|U_{\tilde{v}}|} \sum_{j \in U_{\tilde{v}}} r(j,\tilde{v})$$

An improved prediction to (3.5) becomes

$$\hat{r}(u,\tilde{v}) = \mu_{\tilde{v}} + b_u \frac{1}{N} \sum_{u' \in U_{(i)}} r(u',\tilde{v}). \tag{3.6}$$

The principle of aggregating over existing metrics is denoted as memory-based collaborative filtering. There is also a relatively distinct approach called model-based collaborative filtering, including e.g. Bayesian hierarchical models or deep neural networks, the latter of which we will now cover.

3.1.3 Neural collaborative filtering

By using deep learning models, we can also capture the nontrivial relationships between users and items (Zhang et al. 2019). "If there is an inherent structure that the model can exploit, then deep neural networks ought to be useful" (Zhang et al. 2019).

The extraction of the relationships between users and items is done by matrix factorization. We consider the interaction matrix $\boldsymbol{R}_0 \in \mathbb{R}^{n_u \times n_v}$ for a dataset with n_u users and n_v articles, such that $r_{ij} \in \boldsymbol{R}_0$ indicates user i's interest in item j. We then consider a factorization $\boldsymbol{U}\boldsymbol{V}^T \approx \boldsymbol{R}_0$. In particular, let the user interaction matrix $\boldsymbol{U} \in \mathbb{R}^{n_u \times n_{emb}}$ be a matrix whose n_u rows correspond to each user ID and n_{emb} columns represent the latent factors. Similarly, $\boldsymbol{V} \in \mathbb{R}^{n_v \times n_{emb}}$, has n_v rows corresponding to each item with n_{emb} columns for the items' latent factors. Thus, a score of how much a user would like an item can be approximated as the product of the matrices,

$$\boldsymbol{R} = \boldsymbol{U}\boldsymbol{V}^T,$$

Furthermore, we can introduce biases to customers and articles, contributing similarly to $\mu_{\tilde{v}}$ and b_u in (3.6). The bias weights can capture more variability in customers' tastes and variability in articles with otherwise similar properties. Doing so, we introduce $\boldsymbol{b}_u \in \mathbb{R}^{n_u}$ and $\boldsymbol{b}_v \in \mathbb{R}^{n_v}$, meaning that all users and articles have an additional single latent factor to be learned. The update scheme for a specific user i and article j becomes

$$r_{ij} = \sigma(\boldsymbol{U}\boldsymbol{V}^T + \boldsymbol{b}_u + \boldsymbol{b}_v)_{ij}, \tag{3.7}$$

where $\sigma(x) = e^x/(e^x + 1)$ is the sigmoid function, scaling all scores between 0 to 1. We also define the parameter space $\boldsymbol{\theta}$ for all latent factors. This will be our baseline model.

Machine learning principle for matrix factorization

Although (3.7) shows that we can extract features of users and items with matrix factorization, we still need a method for learning these features. In machine learning, we consider the latent variables $\boldsymbol{\theta}$ of the factorized matrices and biases to be parameters to be optimized. To optimize the parameter space we split our dataset into a subset for training and a subset for validation. The dataset in question will consist of a sequence of known ratings $r(u, v)$ for several users u and items v. We start by initializing all parameters to be $\mathcal{N}(0, 1)$, i.e. standardized normal distribution, and compute $\hat{r}(u, v)$ using (3.7) for the training data. All predictions are compared to its ground truth $(r(u, v))$ and a loss value is then computed using a loss function, which in our case is the binary cross-entropy defined in (3.10). Finally, utilizing the gradient of the loss function and the parameters $(-\nabla\mathcal{L}(\boldsymbol{\theta}))$, we propose new values to the latent field $\boldsymbol{\theta}$ using some optimizer. For this project, we use Adam optimizer with weight decay, defined in Algorithm 4.4.

One other important component is to avoid *overfitting*. A model can lead to overfitting if the latent factors as too specialized to the training subset of the data so that it cannot accurately predict aspects of new data, i.e. data in the validation set. There are a plethora of different regularization techniques that can be used to prevent overfitting, but this report will only cover the techniques used in the NCF implementation, namely reducing the learning rate on a plateau and the use of weight decay.

The first method is to adaptively reduce the learning rate γ once learning stagnates. The learning rate is a crucial hyperparameter in machine learning, and apparent in the optimizer's algorithm referenced previously. Ideally, γ should be as large as possible to contribute to faster training, but not too large as it may lead to model divergence. This tuning is not trivial, "determining a good learning rate becomes more of an art than science for many problems" (Zeiler 2012). In our implementation, we monitor the validation loss (i.e. loss function evaluated using the validation set), and decrement the learning rate by one magnitude (e.g. 10^{-2} to 10^{-3}) if the loss does not decrease sufficiently for several epochs.

The other technique used is weight decay in the optimizer. In the iterative step of updating the weights, the weight decay imposes an L2 penalty on the parameters. An L2-penalty is a scalar (λ) multiplied by the squared L2-norm of the weights, $\lambda\|\theta_i\|_2^2$. The effect of introducing weight decay in your model is a smoothing effect which hopefully results in less overfitting.

3.1.4 The cold start problem

One of the major challenges a recommender system is facing is the *cold start problem* (CS). The cold start is a term used to describe the difficulty of recommending items to users when there is incomplete or no data for the user and/or item. Within the cold start problem, we have complete cold start (CCS), meaning that a user or item has no attributed ratings and the incomplete cold start (ICS) where the interaction matrix is sparse but not completely empty (Wei et al. 2017).

In general, CF models are more susceptible to the cold start problem, although it also negatively affects content-based systems. For the ICS case where an article has no ratings, a CF model will never be able to recommend the particular item, whereas a content-based model can assign a similarity score to other items using pure article data. However, Wei et al. (2017) presents a promising remedy to the CF case, using a deep neural network to build an item profile for new items, and a CF model using the content features for prediction.

Another case of the CS problem can be if a completely new user registers for a service. In such a case, there is no user profile, also making content-based recommendations difficult. Since the user is new and has no ratings, there is no way to obtain the subset of similar users $U_{(i)}$, seeing that $V_x = \emptyset$. Some services solve this initial CCS issue by collecting some fundamental information about the customer

upon registration, such as age, postal code, and in some cases broad interest areas. This is related to demographic filtering already explored in 2003 (Peddy and Armentrout 2003).

3.2 Recommender systems in practice

In larger online businesses, the exact form of recommender system will continually develop and change in line with new empirical results and scientific developments in the area.

One particular example is recent developments for one of the largest recommender systems in production, namely the YouTube recommendation system. M. Chen et al. (2019) made progress using recent improvements to reinforcement learning (RL), which is a subset of machine learning methods involving "agents that take actions in an environment so as to maximize some notion of long term reward" (M. Chen et al. 2019). Another area with more recent developments is to utilize the temporal dimension. Quadrana et al. (2018) categorizes sequence learning models (such as using reinforcement learning or recurrent neural networks), and sequence-aware matrix factorization models as the two subsets of sequence-aware recommender systems. In the latter of the two, Twardowski (2016) obtains good results by aggregating item information in a time-decaying way, i.e.

$$w = \sum_{j=1}^{t} \frac{1}{1+t-j} v^{(j)},$$

where t indicates the temporal sequence length for v.

3.3 Metrics

3.3.1 Mean Average Precision

The mean average precision (MAP) is chosen as the metric when analyzing inference performance. In order to define this metric, we first define the average precision.

We use the definition from Revaud et al. (2019) with some simplified notation. The average precision metric (AP) of some ground truth $r(u,*)$ of length N and predictions $\hat{r}(u,*)$ for a user u, is

$$AP(\hat{r}(u,*), r(u,*)) = \frac{1}{N} \sum_{i=1}^{N} P_i(\hat{r}(u,*), r(u,*)) \Delta \operatorname{re}_i(\hat{r}(u,*), r(u,*)), \tag{3.8}$$

where P_i is the precision at cutoff i and $\Delta \operatorname{re}_i$ is the relevance of the item at rank i, so that it yields 1 if $\hat{r}(u,i)$ is present in $r(u,*)$ and 0 otherwise. They can be expressed as

$$P_k(\hat{R}_{u,*}, R_{u,*}) = \frac{1}{k} \sum_{i=1}^{k} \sum_{j=1}^{N} \mathbb{1}[\hat{r}(u,i) = r(u,j)]$$

$$\Delta \operatorname{re}_k(\hat{r}(u,*), r(u,*)) = \min\left(1, \sum_{j=1}^{N} \mathbb{1}[\hat{r}(u,k) = r(u,j)]\right)$$

By setting N to some $k \leq N$, we denote the average precision with cutoff k as AP@k. Finally, the mean average precision is the mean of (3.8) over all users $U = (u_1, \ldots, u_n)$, i.e.

$$\text{MAP@k} = \frac{1}{|U|} \sum_{u=u_1}^{u_n} \text{AP@}\, k(\hat{r}(u,*), r(u,*)) \tag{3.9}$$

3.3.2 Binary Cross-entropy

In Zhang et al. (2019), the binary cross-entropy is, with somewhat modified notation, defined as

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{(u,v)\in(U,V)} r(u,v) \log \hat{r}(u,v;\boldsymbol{\theta}) + (1 - r(u,v)) \log(1 - \hat{r}(u,v;\boldsymbol{\theta})), \tag{3.10}$$

where $r(u,v)$ is the utility or user u's interest in item v and $\hat{r}(u,v;\boldsymbol{\theta})$ is the predicted utility given the latent space $\boldsymbol{\theta}$.

Chapter 4

Method

In this chapter, we introduce the specific methods used in creating the different models for the neural collaborative recommender system. In short, it consists of generating the dataset, training the parameters, and computing trained predictions (inference).

4.1 Aggregation baselines

Before implementing an actual neural collaborative filtering model, it can be of interest to see how a very simple aggregation process performs in terms of MAP@k. For this purpose, we propose two simple aggregations:

- Recommend the same k items that the most number of unique customers have purchased, to every user.
- For each user, find the index group it has bought the most items. Recommend the k most purchased items within this index group.

For the tests, the same training dataset was used for the other models to get comparable results.

4.2 Generating the dataset

In order to train the model, we need both real samples of the data, and artificially generated negative samples. This section will cover how this is implemented.

We denote the transactional data of n_t transactions as a matrix $\boldsymbol{T} = [\mathcal{C}\ \mathcal{A}]$, where $\mathcal{C} = (c_1, \ldots, c_{n_t})^T$ and $\mathcal{A} = (a_1, \ldots, a_{n_t})^T$ represent customer IDs and article IDs respectively, such that for a given $i \leq n_t$, the collection (c_i, a_i) represents a transaction. Furthermore, let $\mathcal{U}(S)$ denote the uniform distribution across a set S, and the notation $\boldsymbol{t}_{i,*}$ corresponds to the i-th row of $\boldsymbol{T}$. Our goal is, based on all our data, to generate $n_k \leq n_t$ positive labels (i.e. transactions that are in fact part of the dataset), and also μn_k negative labels, consisting of customer-article combinations that have not taken place.

The general sampling algorithm is described in Algorithm 4.1. When it comes to sampling iid from $\mathcal{U}(\mathcal{C})$ and $\mathcal{U}(\mathcal{A})$, we can either use `pandas.DataFrame.sample()` or `numpy.random.choice()`. The efficiency for these methods is covered later in section 5.

Algorithm 4.1 Sample n_k positive and μn_k negative transactions from n_t positive transactions.

$\mathcal{I}_k^+ \leftarrow \{i_1, \ldots, i_{n_k}\} \overset{iid}{\sim} \mathcal{U}(\{1, \ldots, n_t\})$
Positive samples $\leftarrow \boldsymbol{t}_{i \in \mathcal{I}_k^+, *}$
Initialize $\boldsymbol{N}$ as an empty matrix
while Number negative samples $< \left\lfloor \mu n_t + \frac{1}{2} \right\rfloor$ **do**
  $c_i \overset{iid}{\sim} \mathcal{U}(\mathcal{C})$
  $a_i \overset{iid}{\sim} \mathcal{U}(\mathcal{A})$
  if $\nexists k$ s.t. $\boldsymbol{t}_{k,*} = (c_i, a_i)$ **then**
    Add the row (c_i, a_i) to $\boldsymbol{N}$
  end if
end while

We also introduce an alternative approximate negative sampling method. Rather than checking each proposed sample for legitimacy, it removes illegitimate negative samples using a table merge after every sample has been drawn. In the algorithm, we use the notation $[S]$ to denote a *multiset* S, i.e. we allow duplicate values to exist in the collection, and should not be confused with a matrix. We also use the symbol $\rtimes\ltimes$, indicating a full outer join. From relational algebra, a full outer join operation between two tables keeps all tuples in both the left and right relation (Elmasri and Navathe 2015). This approximate sampling method is described in Algorithm 4.2.

Algorithm 4.2 Based on all n_t positive samples, generate an approximate μn_t negative samples

$\boldsymbol{p}_c^- \leftarrow [c_1, \ldots, c_{\mu n_t}] \overset{iid}{\sim} \mathcal{U}(\mathcal{C})$
$\boldsymbol{p}_a^- \leftarrow [a_1, \ldots, a_{\mu n_t}] \overset{iid}{\sim} \mathcal{U}(\mathcal{A})$
Negative proposal sample $\boldsymbol{T}^- = [\boldsymbol{p}_c^- \ \boldsymbol{p}_a^-]$
$\boldsymbol{T}^- \rtimes\ltimes \boldsymbol{T} \rightarrow \boldsymbol{T}^J = [\mathcal{C}^J \ \mathcal{A}^J \ \mathcal{I}^J]$, where $\mathcal{I}^J$ is the indicator column showing where row originates from ("left", "right" or "both")
for Each row $i = 1, \ldots$ **do**
  if $\mathcal{I}_i^J \in \{$`right`,`both`$\}$ **then**
    Assign true label (1) to row i
  else
    Assign false label (0) to row i
  end if
end for

4.3 Baseline model

Initially, we want to make a baseline model of collaborative filtering using a minimum of the available data. The only data we use to train a predictive model is a list of customer-article IDs alongside a label of whether the customer has purchased the article or not.

A high-level explanation of one epoch training will look like Algorithm 4.3.

Algorithm 4.3 High-level overview of one epoch of training the baseline model

Iterate through the data T_B with batch B.
Prediction $\leftarrow$ computed (3.7) $\forall\, [i, j] \in T_B$
Training loss $\leftarrow$ Result of (3.10)
Weights $\leftarrow$ Adam$(\theta, f(\theta) :=$ (3.10)$)$, Hyperparameters $\gamma, \beta_1, \beta_2, \lambda, \epsilon)$

Algorithm 4.4 Adam algorithm with weight decay as described in (Kingma and Ba 2014).

Input: Parameters $\boldsymbol{\theta}$, Loss function $\boldsymbol{f}(\boldsymbol{\theta})$, Learning rate γ, Weight decay λ
Initialize $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$
for $t = 1$ to $\ldots$ **do**
  $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) + \lambda\theta_{t-1}$
  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$
  $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$
  $\theta_t \leftarrow \theta_{t-1} - \gamma \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$
end for
Return $\boldsymbol{\theta}_t$

4.4 Extended neural model

Once a baseline model is up and running, it can be extended by introducing more information about each transaction. The particular way this is used in this project is by generating new embeddings for

side information related to the customer and article in question. The extended model uses in this case the customer's age and the article's index group and garment group which both attain a more broad category for the article type. Specifically, the embeddings are concatenated through a linear layer with sigmoid activation, resulting in the final embedding layers for articles and customers, respectively. The dot product of these is computed, and biases are added before finalizing with a sigmoid function. A high-level overview of this model is illustrated in Figure 4.1. We formalize the model as follows.

Let $\boldsymbol{U}[\mathtt{x}] \in \mathbb{R}^{n_u \times n_{emb}}$ and $\boldsymbol{V}[\mathtt{y}] \in \mathbb{R}^{n_v \times n_{emb}}$ be the embeddings of the partial customer information $\mathtt{x}$ and partial article information $\mathtt{y}$, with latent factors similar to $\boldsymbol{U}$ and $\boldsymbol{V}$ in section 4.2, respectively. Let also $\boldsymbol{w}_u$ and $\boldsymbol{w}_v$ be weight vectors for the linear layer for customer and article embeddings, whose values are added to the latent space $\boldsymbol{\theta}$. Then, the prediction has the form

$$r_{ij} = \sigma\left(\sigma\left(\sum_{\mathtt{x}\in\{\mathtt{ID},\mathtt{Age}\}} \boldsymbol{w}_u \boldsymbol{U}[\mathtt{x}]\right)\sigma\left(\sum_{\mathtt{y}\in\{\mathtt{ID},\mathtt{Index},\mathtt{Garment}\}} \boldsymbol{w}_v \boldsymbol{V}[\mathtt{y}]\right) + \boldsymbol{b}_u + \boldsymbol{b}_v\right) \tag{4.1}$$

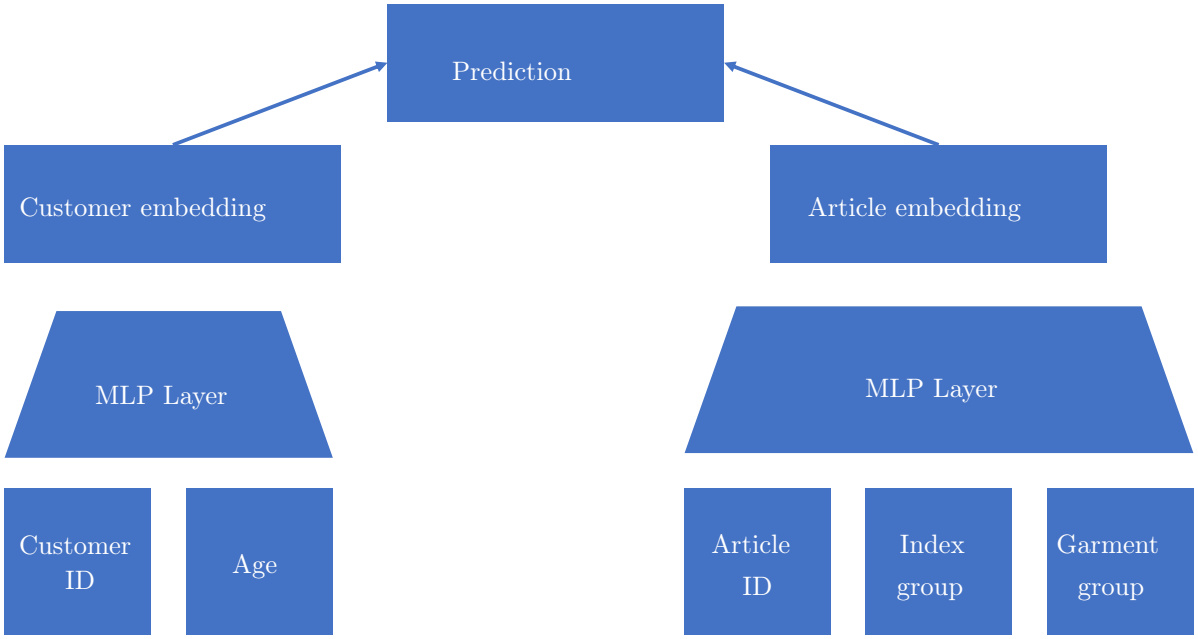


Figure 4.1: Illustration of extended model

4.5 Inference and evaluation

We use MAP@k with cutoff $k = 12$ as a metric to evaluate the overall goodness of a trained model's predictions. Recalling that MAP@12 is the mean of AP@12 for all customers, we need to consider the predicted utility score for each item and each customer in the dataset. Due to the magnitude of customers and articles, this must be implemented by batch-processing a set of customers and freeing memory after a certain threshold has been met, in order to accommodate for less memory usage. The details of this are shown in Algorithm 4.5.

4.6 Technology and development

The models are implemented using the programming language Python 3 with its known libraries pandas[1] for data analysis, NumPy[2] for array management and PyTorch[3] for model building. Among the less

[1] https://pandas.pydata.org/
[2] https://numpy.org/
[3] https://pytorch.org/

Algorithm 4.5 Generate the top k predicted articles for each customer, using a trained model

Initialize empty objects `out` and `preds`
Load trained model state
$C \leftarrow$ all distinct customers $c \in \mathcal{C}$
$A \leftarrow$ all distinct articles $a \in \mathcal{A}$
$A \leftarrow [A_1, \ldots, A_n]^T$, partitioned set of articles, each A_i of specified batch size
for $c \in C$ **do**
  for $A_i \in A$ **do**
    $\texttt{preds}_{c,A_i} \leftarrow$ Result of (3.7)
    if $|\texttt{preds}| \geq M$ **then**
      for Customers $c \in \texttt{preds}$ **do**
        $m \leftarrow \arg\max_i \texttt{preds}_{c,*}$ for the top k elements
        $\texttt{out}_c \leftarrow \texttt{preds}_{c,m}$
      end for
      `preds` $\leftarrow \emptyset$, i.e. remove from memory
    end if
  end for
end for
Return `out`

central modules, we use Matplotlib[4], seaborn[5] and tqdm[6] for visualization, and scikit-learn[7] for encoding labels.

This is developed in a virtual environment using Jupyter Notebooks[8] in Visual Studio Code[9], alongside the Python formatter Black[10]. We also utilize the version control manager git[11] and hosting on Github.com.

For longer runs and computations, we run portions remotely on an external server using the SSH (Secure Shell) protocol and tmux[12]. The server uses 768 GB memory and two 14-core Intel Xeon 2.6 GHz CPUs.

[4] https://matplotlib.org/
[5] https://seaborn.pydata.org/
[6] https://tqdm.github.io/
[7] https://scikit-learn.org/stable/
[8] https://jupyter.org/
[9] https://code.visualstudio.com/
[10] https://github.com/psf/black
[11] https://git-scm.com/
[12] https://github.com/tmux/tmux/wiki

Chapter 5

Results

The results chapter presents the output of the different models and highlights some key numbers in relation to accuracy and time.

5.1 General accuracy

For the baseline model, using a total of 100,000 true transactions and 100,000 negative labels, we reach a MAP@12 of $0.356192 \cdot 10^{-4}$. Similarly, computing the MAP from the trained extended model with 200,000 total samples, we obtain $1.83666 \cdot 10^{-4}$. In short, it seems like the extended model is about one order of magnitude better than the baseline, but both models perform poorly in predicting the items present in the validation set. Remark that this is considering the sampled validation data as ground truth. By instead considering all transactions as the ground truth, we achieve a MAP@12 of $0.108853 \cdot 10^{-3}$ for the baseline case and $0.696622 \cdot 10^{-3}$ for the extended case. Finally, these values can be compared with the simple aggregations, in which the MAP is one to two orders of magnitude larger than the machine learning models.

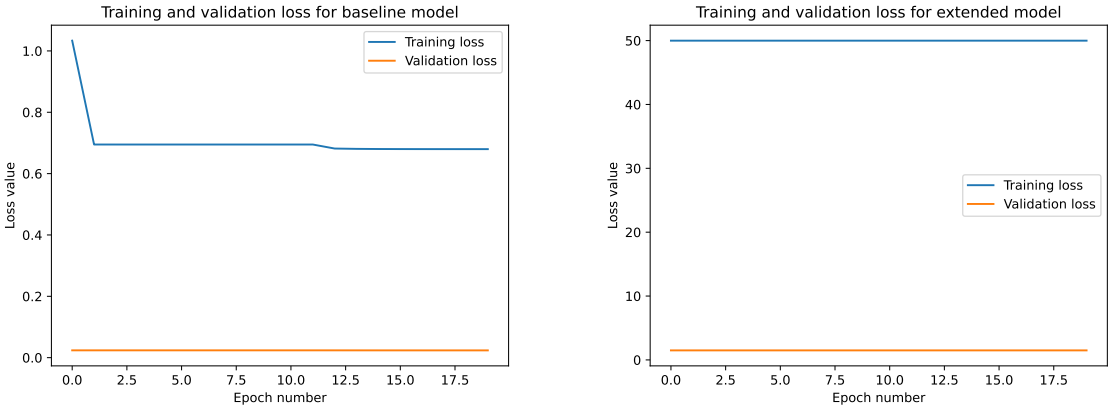
All metrics are summarized in Table 5.1.

We can also have a look at the values of the validation loss for the different models. Considering the models with 200,000 samples, the last validation loss for the baseline is about 0.0237, whereas the extended model has surprisingly 1.493. The training and validation losses are plotted in Figures 5.1a and 5.1b. Although the training versus validation loss is several orders of magnitude in difference, they both seem to attain the same value regardless of epoch number. This seems to apply to both models.

Another metric of interest is the numerical predicted value of the top k items for a user, since this value measures the model's confidence that the user would like the item between 0 and 1. Thus, a value close to 1 means that the model is confident in recommending, and values close to 0.5 can be interpreted as having an equal likelihood of being recommended versus not being recommended. Figure 5.2 shows the average and maximum predicted value of each user's top $k = 12$ items. We see that both the average and the maximum values are uniformly distributed roughly between 0.56 and 0.62, all very close to 0.5.

Model	MAP@12 for all data	MAP@12 for validation data
Top k items	14.52706 e-03	3.755890 e-03
Top k index-based	13.82349 e-03	8.452114 e-03
Baseline model	0.108853 e-03	0.035619 e-03
Extended model	0.696622 e-03	0.183666 e-03

Table 5.1: MAP for the different models.



(a) Baseline **(b)** Extended model

Figure 5.1: Training and validation loss for the baseline and extended models. The blue line indicates the training loss and the orange one indicates the validation loss for both plots.

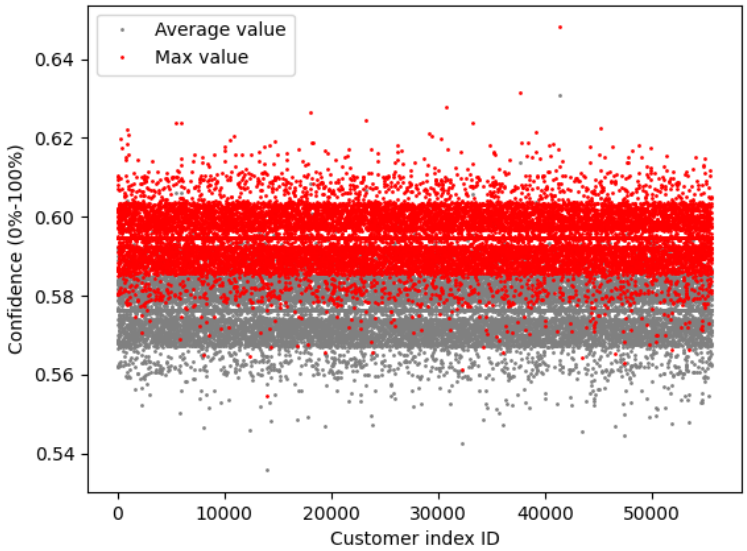


Figure 5.2: Average predicted confidence of recommending each user's top $k = 12$ items (items having the highest confidence). The x-axis is each customer index and y value a confidence between 0 to 1. The red dots show the maximum confidence for each user's top items and the grey ones are the average.

Hyperparameters

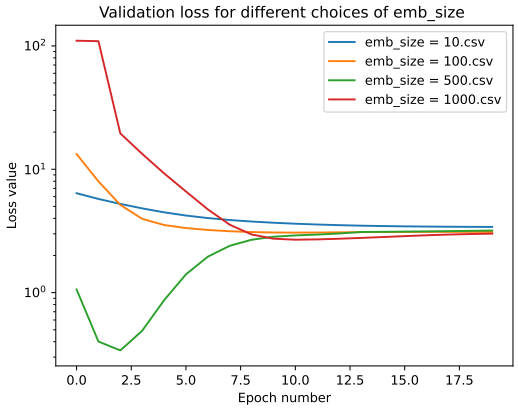
For a test case of only 2000 samples, we try out different hyperparameters. The hyperparameter search is in a sense naïve, since it checks for different choices of one parameter, keeping all other parameters constant at an arbitrary value. A more robust search would check every combination of different choices but would also increase the number of training runs exponentially.

Figure 5.3 shows the change in validation loss over 20 epochs for different choices. Although the losses seem to converge after about 10 epochs, the results show that the embedding size should be around 500, the learn rate around 10^{-2}, and the weight decay should be small, i.e. 10^{-4}. The last result may indicate that we don't need regularization to avoid overfitting.

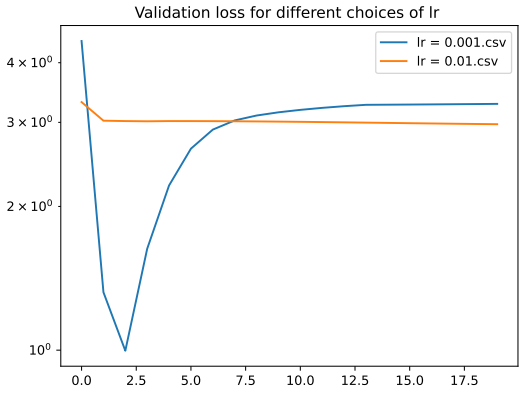
The final parameters used are shown in Table 5.2.

Table 5.2: Hyperparameter settings

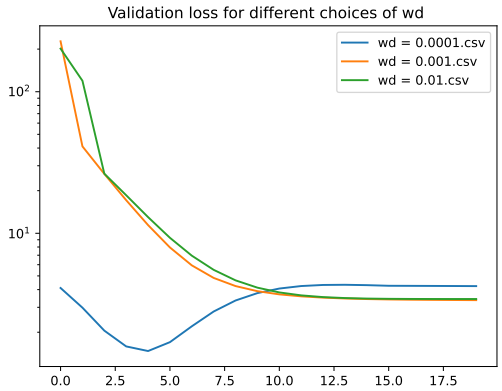
Parameter	Value
Embedding size	500
Learning rate	10^{-2}
Weight decay	10^{-4}
Portion of negatives (μ)	1



(a) Embedding size



(b) Learning rate



(c) Weight decay

Figure 5.3: Validation loss for different settings

5.2 Time complexity

The extended model with 200k samples used 22,816 seconds, or equivalently 6 hours, 20 minutes, and 16 seconds. Surprisingly, the baseline model with the same number of samples seemed to take 35,864 seconds (9 hours, 57 minutes, and 44 seconds). Note however that the runtime is majorly dependent on other processes running and not necessarily an isolated measure of the models' time complexities.

Negative sampling efficiency

In section 4.2, an algorithm for negative sampling is proposed. Figure 5.4 shows how much more efficient it is to use a simpler NumPy object and use `random.choice()` rather than Pandas' `sample()`. Using the latter yields a new data frame that is more complex than an arbitrary n-dimensional array, which attributes to much of the extra computational time.

Furthermore, we test out the alternative negative sampling method (Algorithm 4.2). A proof of concept implementation of this method was run, having $\mu = 1.0$, i.e. generating equally as many negative samples as there were of positive samples, about 31.788 million. Due to some randomly generated samples belonging to actual transactions, the final number of true negative samples was about 31.782 million, and had a runtime of about 170 seconds. We also ran the proof of concept with $\mu = 10$, yielding a runtime of 1046 seconds which also attained the asymptotic property of sampling approximately μn_t negative samples. Note that the memory required for such an operation was to be able to allocate at least a $(n_t + \mu n_t) \times 3$ large dataframe. For the case of $\mu = 10$, 66.399 GB is required by the dataframe object alone. By using modified optimizers for sparse data, we were able to train 5 epochs for the full data with $\mu = 1$, in a simple model without weight decay or bias nodes. This run had an average of $14,309$ seconds per epoch. Scaling this up to 20 epochs would result in about 3 days and 7.5 hours of training, which is not viable.

Inference computations

The MAP computation itself is trivial in terms of time complexity, but there is considerable cost in finding the best predictions since we have to predict the model confidence for each combination of customer and article. For predicting the baseline model with 55,574 unique customers and M in Algorithm 4.5 chosen as 200, $18,496$ seconds elapsed (approximately 5 hours). The extended case uses considerably more time, totaling $86,871$ seconds for the extended model.

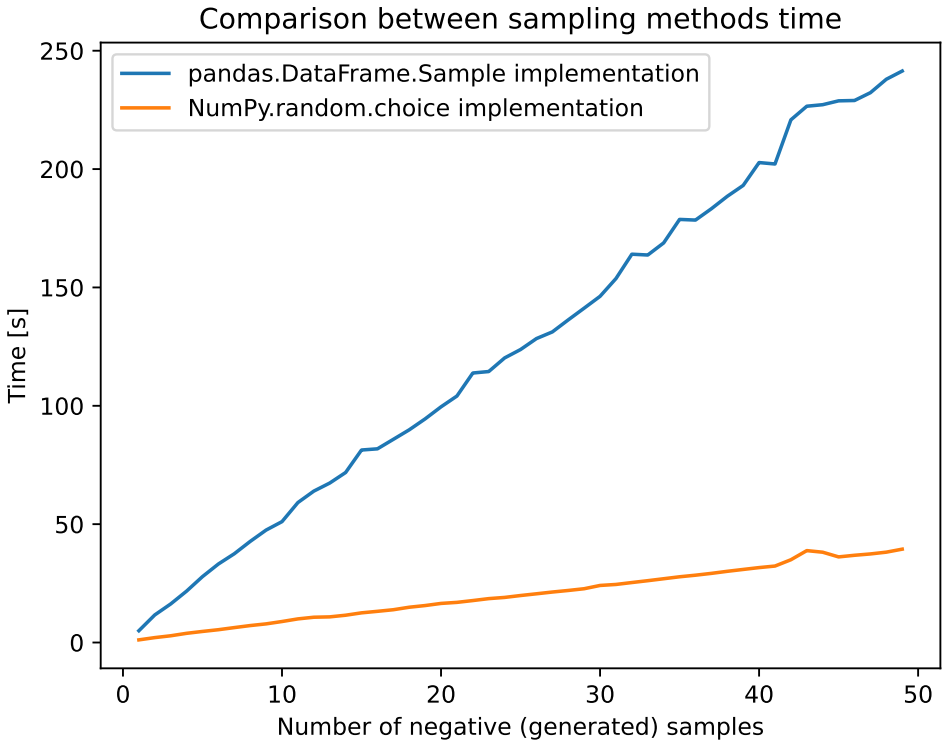


Figure 5.4: Comparing the two methods of sampling negative items from a transaction database of about 13 million items. The blue line shows time spent with Pandas' `sample` method and the orange line shows time spent using NumPy's `random.choice` method.

Chapter 6

Discussion

This section aims to provide more knowledge about the results presented in section 5, and explain some of the shortcomings of the models and their performance.

6.1 Improving speed

One of the major concerns when dealing with a large dataset as in this project is to be computationally efficient and perform all parts of the model pipeline as fast as possible. Although several efforts were made, not all of the methods mentioned in this section have been implemented.

Generating data

A very large factor in terms of time is the generation of the dataset. As described in section 4.2, we generate μn_t negative samples for each positive sample where $1 \leq \mu \leq 10$. Algorithm 4.1 ensures that no randomly sampled negatives are in fact not a part of the true data. This forces the algorithm to scan all n_t positive samples for each proposition of a negative sample, so that in best-case it uses $\mathcal{O}(\mu n_t \cdot n_t) = \mathcal{O}(n_t^2)$ time. This is subpar, considering n_t is about 32 million in the complete dataset, which is the reason behind only training the model on a sample of all transactions. A remedy that was considered was to disregard the check of randomly selected pairs existing in the true data. In practice, this check returns true a significant amount of time, which would contribute to incorrect predictions.

The other proposed algorithm, Algorithm 4.2, improved considerably compared to the other sampler as presented in 5.2. However, since feeding the complete data through the network was unsuccessful, it was not possible to train using this complete data.

A possible alternative approach could be to rather use a so-called heterogeneous collaborative filtering model, which has been shown to outperform state-of-the-art collaborative filtering models (C. Chen et al. 2020). Another important consideration is that the negative samples in the current models provide negative feedback for articles that a customer may have given implicit interest to - such as viewing, adding to cart, and other functionalities that many online stores provide. Using a heterogeneous system can model this heterogeneous feedback, again shown in (C. Chen et al. 2020).

Code infrastructure

Another factor worth noting is the overhead produced by creating Pandas objects. As shown in Figure 5.4, the time cost of sampling alone is really large for pandas objects. These objects are widely used in the model implementation, also outside the sampling process, which might be the source of computational overhead. With more time, it would be possible to restructure the code infrastructure to less comprehensive objects. In extension to this, it is known that there are programming language more efficient both in terms of time and memory compared to Python. For instance, C/C++ has been shown to perform 5-10 times faster than, and uses about half the memory as scripting languages like Python (Prechelt 2000). However, since PyTorch and NumPy both builds on underlying calls to optimized C code (Harris et al. 2022; Paszke et al. 2022), this might not show significant improvement.

Inference

The inference stage is also the root of large computational complexity. Once a model is trained and all latent factors have a trained state, we would like to know what articles provide the best score for each customer, which is explained in Algorithm 4.5. Looking at the algorithm, although there is incorporated batch processing and memory management, the prediction in (3.7) still has to be computed $n_c n_a$ times. The threshold M in Algorithm 4.5 can also be more dynamically set, for instance, based on the current machine's available RAM to allow for larger batches without risking OOM (Out of Memory) errors.

When adding additional features to the model, you also lose a considerable amount of computational efficiency. Although both methods are implemented similarly, there are certain characteristics of the data in the extended model that requires more tuning before inference can be performed, thus increasing time complexity. For technical details pertaining to this, we refer to appendix B.

Parallel computations and distributed systems

Taking advantage of parallelism can work during the sampling and inference part of the pipeline. This would require a larger overhaul of the code to allow for multiprocessing in Python, and was not prioritized.

Similarly to using multiprocessing, big data architecture such as Spark and Vespa would allow for more workers and can be implemented on a distributed cluster rather than to be run in one instance.

6.2 Improving accuracy

Overall, the results in section 5 show that the simple aggregations provide better accuracy than any of the trained models when considering every customer. This result can be explained by the fact that the other proposal models don't perform well in sparse user cases. An area of interest would be to further analyze model accuracy as a function of the sparsity of the current user, in order to empirically determine that this is the cause of the models not performing as well. Alternatively, one could combine the predictions of the most popular items with an NCF model, falling back on the popular items in sparse user cases. One can consider that once the model reaches a lower threshold in its confidence, it rather uses some of the most purchased articles.

Another noteworthy observation is that the overall best model is to recommend the same k most purchased items of all time, regardless of customers' previously purchased items. This can indicate that customers seem to purchase items across several index groups, e.g. both *menswear* and *baby/children*. We will now look into why the models did not deliver good predictions.

The data sampling has a clear consequence in regard to poor predictions. Since we've only used a subset of the data, the validation data contains far fewer true predictions than it would have, had we used the complete data. As a consequence of this, the model can predict an article for a customer which is correct in the full data but not in the sampled data, resulting in a false negative prediction. This is the reason for providing a MAP evaluated using both the full data and the validation data of the sample. Furthermore, we discussed the cold start problem in section 3.1.4. Considering the complete data alone, cold start seems to be a prevalent issue in our data as well. As shown in section 2.2.3, the dataset alone has 9699 registered customers with no transaction history, in which logged-out customers or non-registered customers increase this amount substantially. This issue is magnified by using only 100,000 sampled transactions, or equivalently 0.3 % of the true data.

Loss metric convergence

One apparent trend of the plots in section 5, is that they seem to flatten out at very early epochs, providing almost no improvement in either training loss or validation loss. This should be seen in combination with the final weights of the training process, in which most embeddings evaluate to around 10^{-35} to 10^{-40}. In fact, the maximum value of any of the latent factors in the customer and article matrices, are around 10^{-34}, whereas the biases $\boldsymbol{b}_u$ and $\boldsymbol{b}_v$ have maximum values of order 10^{-1}. It seems like the optimizer has chosen to send the dot product operands to zero, reducing the NCF model to essentially be the sum of the two biases,

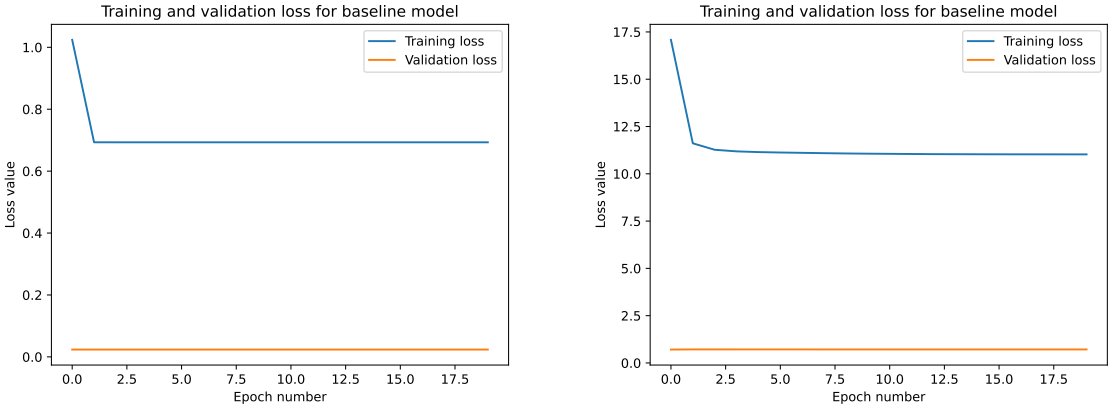
$$r_{ij} \to \sigma(0 + \boldsymbol{b}_u + \boldsymbol{b}_v).$$

There are several potential causes for such behavior. For instance, the usage of weight decay encourages the model to let parameters go toward zero. By using weight decay in situations where it is not necessary, one might get too many model parameters to approximately zero. Another source of this

Parameter	Baseline	Without bias	Without bias and $\gamma = 0$
$\boldsymbol{U}$	1.9103026e-34	1.7407397e-31	5.5402188
$\boldsymbol{V}$	8.624063e-36	6.1274531e-33	5.4648938
$\boldsymbol{b}_U$	0.22759376	-	-
$\boldsymbol{b}_V$	0.383055	-	-

Table 6.1: Maximum values for each of the different parameters of the trained baseline model with 200,000 samples. The first column is the baseline model with bias nodes, the second without bias nodes, and the third with neither bias nodes, nor any weight decay. All models have learning rate regularization.

problem can be the use of bias nodes, in that the model finds it sufficient to only use the biases. To address these two issues, we train two more instances of the baseline model; one without bias nodes but with the same weight decay, and one with neither biases nor weight decay. We find the maximum values for all weights in the parameters of the trained models in Table 6.1. Here, we see that only removing the biases does not seem to help reduce this behavior. Interestingly, removing both biases and weight decay results in significantly higher maximum weights, of order 10^1. Examining the training and validation loss will however yield a similar result as before, shown in Figure 6.1.



(a) Without biases

(b) Without biases or weight decay

Figure 6.1: Training and validation loss for the baseline without biases in an effort to determine the causes of the constant loss curves.

One last factor to consider is the choice of the loss function in combination with the data used to train. Binary cross entropy, as defined in (3.10), penalizes inaccurate predictions by its negative log. By having a balanced dataset of 50 % positive and negative labels, a correct negative classification is rewarded to the same degree as a positive classification. Since the transactional history is very sparse, almost all training cases for a given customer u would return 0, thus encouraging the model to send all parameters to zero. As a result, it might be more appropriate to have fewer negative samples, decrease sparsity by using more samples or use some alternative to negative sampling. However, if this were to influence the predictions negatively, it would argue for using an alternative to negative sampling rather than changing the loss function.

Various measures to increase accuracy

Lastly, we will cover some techniques that can potentially improve the accuracy that has not been tested. One possible remedy would be to introduce pre-trained data. There are several approaches to this, for instance by incorporating it via a linear layer, similar to the additions of data in the extended model in section 4.4. Pre-trained data has been effective in other studies, such as Rajaraman et al. (2018) where using a pre-trained ResNet-50 model out-preformed all other options investigated. In addition, using heterogeneous data as referenced in section 6.1 or temporal data as covered in section 3.2, one might expect improved accuracy. It should also be mentioned that we've only tested one extension of the baseline model in this report, and it's possible that using other combinations of side information about

the customer and/or article, such as image analysis of the article images through some convolutional network, might improve accuracy as well. However, the best remedy would probably be to increase the number of samples in the data.

Chapter 7

Conclusion

This project has had aim to implement a neural collaborative filtering method balancing accuracy and time complexity. We also explored how extensions to the said model would perform on real-world data, in addition to gaining an understanding of the theory behind recommender systems and the principles of the recent developments in model-based collaborative filtering.

The first research question references what aspects of implementing an NCF model has the largest effect on run time and accuracy. The first sampling process alone led to a large overhead in terms of run time, but regardless of which sampling process was used, the bottleneck was feeding large amounts of data through the neural network. To alleviate this, a very small sample of the available data had to be used for training and validation, yielding poor model performance overall.

The second question wishes to determine if extensions to the baseline model influence the accuracy and/or time complexity. Through our findings, we cannot determine any significant changes in time complexity or accuracy, although the latter can be an artifact of limited data samples in the training set.

Finally, the last question points to what measures one can use to improve a model's performance. In general, NCF models with sparse transactional data struggle with the cold start model. Here, we refer to using heterogeneous CF models, distributed systems, or multiprocessing data in section 6.1. We also covered other works showing promising results for incorporating reinforcement learning, time-decaying matrix factorization (in section 3.2), or using pre-trained data (in section 6.2). Lastly, introducing more data or using a hybrid approach combining CF with the most popular articles can also be a remedy for both accuracy and time complexity.

Bibliography

[1] G. Adomavicius and A. Tuzhilin. 'Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions'. In: *IEEE Transactions on Knowledge and Data Engineering* 17.6 (2005), pp. 734–749. DOI: 10.1109/TKDE.2005.99.

[2] *Brands - H&M Group*. https://hmgroup.com/brands/. Accessed: 2022-10-10. 2022.

[3] Erik Brynjolfsson, Yu Hu and Duncan Simester. 'Goodbye pareto principle, hello long tail: The effect of search costs on the concentration of product sales'. In: *Management Science* 57.8 (2011), pp. 1373–1386.

[4] Erik Brynjolfsson, Yu Jeffrey Hu and Michael D Smith. 'From niches to riches: Anatomy of the long tail'. In: *Sloan management review* 47.4 (2006), pp. 67–71.

[5] Chong Chen, Min Zhang, Yongfeng Zhang, Weizhi Ma, Yiqun Liu and Shaoping Ma. 'Efficient Heterogeneous Collaborative Filtering without Negative Sampling for Recommendation'. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.01 (Apr. 2020), pp. 19–26. DOI: 10.1609/aaai.v34i01.5329. URL: https://ojs.aaai.org/index.php/AAAI/article/view/5329.

[6] Minmin Chen, Alex Beutel, Paul Covington, Sagar Jain, Francois Belletti and Ed H. Chi. 'Top-K Off-Policy Correction for a REINFORCE Recommender System'. In: *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*. WSDM '19. Melbourne VIC, Australia: Association for Computing Machinery, 2019, pp. 456–464. ISBN: 9781450359405. DOI: 10.1145/3289600.3290999. URL: https://doi.org/10.1145/3289600.3290999.

[7] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Pearson Education, 2015. ISBN: 9780133971248. URL: https://books.google.no/books?id=tRybCgAAQBAJ.

[8] Charles R. Harris, K. Jarrod Millman, St fan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fern ndez del R o, Mark Wiebe, Pearu Peterson, Pierre G rard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke and Travis E. Oliphant. *NumPy v1.24 Manual*. https://numpy.org/doc/stable/user/whatisnumpy.html. Accessed: 2022-12-19. 2022.

[9] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[10] Carlos García Ling, ElizabethHMGroup, FridaRim, inversion, Jaime Ferrando, Maggie, neuraloverflow and xlsrln. *H&M Personalized Fashion Recommendations*. 2022. URL: https://kaggle.com/competitions/h-and-m-personalized-fashion-recommendations.

[11] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai and Soumith Chintala. *PyTorch 1.13 documentation*. https://pytorch.org/docs/stable/cpp_index.html. Accessed: 2022-12-19. 2022.

[12] Clayton C. Peddy and Derek Armentrout. *Building Solutions with Microsoft Commerce Server 2002*. 1st ed. USA: Microsoft Press, 2003. ISBN: 0735618542.

[13] Lutz Prechelt. 'An empirical comparison of c, c++, java, perl, python, rexx and tcl'. In: *IEEE Computer* 33.10 (2000), pp. 23–29.

[14] Massimo Quadrana, Paolo Cremonesi and Dietmar Jannach. 'Sequence-Aware Recommender Systems'. In: *ACM Comput. Surv.* 51.4 (July 2018). ISSN: 0360-0300. DOI: 10.1145/3190616. URL: https://doi.org/10.1145/3190616.

[15] Daisy Quaker. *Amazon Stats: Growth, sales, and more*. Mar. 2022. URL: https://sell.amazon.com/blog/grow-your-business/amazon-stats-growth-and-sales.

[16] Sivaramakrishnan Rajaraman, Sameer K. Antani, Mahdieh Poostchi, Kamolrat Silamut, Md. A. Hossain, Richard J. Maude, Stefan Jaeger and George R. Thoma. 'Pre-trained convolutional neural networks as feature extractors toward improved malaria parasite detection in thin blood smear images'. In: *PeerJ* 6 (Apr. 2018), e4568. ISSN: 2167-8359. DOI: 10.7717/peerj.4568. URL: https://doi.org/10.7717/peerj.4568.

[17] Jerome Revaud, Jon Almazan, Rafael S. Rezende and Cesar Roberto de Souza. 'Learning With Average Precision: Training Image Retrieval With a Listwise Loss'. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2019.

[18] Bartłomiej Twardowski. 'Modelling Contextual Information in Session-Aware Recommender Systems with Neural Networks'. In: *Proceedings of the 10th ACM Conference on Recommender Systems*. RecSys '16. Boston, Massachusetts, USA: Association for Computing Machinery, 2016, pp. 273–276. ISBN: 9781450340359. DOI: 10.1145/2959100.2959162. URL: https://doi.org/10.1145/2959100.2959162.

[19] Jian Wei, Jianhua He, Kai Chen, Yi Zhou and Zuoyin Tang. 'Collaborative filtering and deep learning based recommendation system for cold start items'. In: *Expert Systems with Applications* 69 (2017), pp. 29–39. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2016.09.040. URL: https://www.sciencedirect.com/science/article/pii/S0957417416305309.

[20] Matthew D. Zeiler. 'ADADELTA: An Adaptive Learning Rate Method'. In: *CoRR* abs/1212.5701 (2012). arXiv: 1212.5701. URL: http://arxiv.org/abs/1212.5701.

[21] Shuai Zhang, Lina Yao, Aixin Sun and Yi Tay. 'Deep Learning Based Recommender System: A Survey and New Perspectives'. In: *ACM Comput. Surv.* 52.1 (Feb. 2019). ISSN: 0360-0300. DOI: 10.1145/3285029. URL: https://doi.org/10.1145/3285029.

Appendix A

Data exploration and analysis

[1]:
```
%reload_ext jupyter_black
```

```
<IPython.core.display.HTML object>
```

We start by importing relevant libraries

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from typing import Union, Tuple, Iterable
from scipy.stats import gaussian_kde
import random, shutil, os, itertools, black, jupyter_black
```

Data loading and preprocessing

[3]:
```
def naive_csv_sampler(
    csv_path: str,
    sample_size: int,
    num_records: int | None = None,
    header: str | None = "infer",
) -> pd.DataFrame:
    """Read samples of rows from csv file

    Args:
        csv_path (str): Path to file including file extensions
        sample_size (int): Number of rows to sample
        num_records (int | NoneType, optional): Total records in file, defaults to␣
↪None. If None, the file will be scanned (costly)
        header (str | NoneType, optional): 'header'-parameter for pandas, defaults␣
↪to 'infer'. Set to None if file has no header.

    Returns:
        pd.DataFrame: Dataframe with sampled entries (and potentially header)
    """
    if num_records is None:
        num_records = newlines_in_csv(csv_path)
    indices_skip = sorted(
        random.sample(range(1, num_records + 1), num_records - sample_size)
    )
    return pd.read_csv(csv_path, skiprows=indices_skip, header=header)
```

```python
def newlines_in_csv(csv_path: str, chunk_size: int = 1024) -> int:
    """Counts number of newlines in csv file without loading entire file to memory.
    The number of newlines is the same as number of rows assuming,
        * EITHER csv has a header and last entry does not end with newline
        * OR csv does not have a header, but last entry ends with newline
        * ALWAYS data does not have any nested newline madness
    Originally from orlp, https://stackoverflow.com/a/64744699

    Args:
        csv_path (str): Path of csv file
        chunk_size (int, optional): How many KB to process at at a time. Defaults␣
 ↪to 1024 = 1 MB.

    Returns:
        int: Number of newlines
    """
    chunk = chunk_size**2
    f = np.memmap(csv_path)
    number_newlines = sum(
        np.sum(f[i : i + chunk] == ord("\n")) for i in range(0, len(f), chunk)
    )
    del f
    return number_newlines
```

[4]:
```python
def copy_img_from_article(df: pd.DataFrame, outpath: str):
    """Helper to copy image files referenced in `df` to other directory"""
    for id in df["article_id"]:
        id0 = "0" + str(id)
        img_path = f"./dataset/images/{id0[:3]}/{id0}.jpg"
        if not os.path.isfile(img_path):
            continue  # ID has no image (happens for some cases)
        out_dir = f"./{outpath}/images/{id0[:3]}/"
        if not os.path.isdir(out_dir):
            os.makedirs(out_dir)
            shutil.copy(img_path, out_dir)
```

[5]:
```python
def load_min_data(filename: str | Iterable):
    """Helper to properly load the mimimal datasets as data frames"""
    dfs = []
    if isinstance(filename, str):
        filename = [filename]
    for fn in filename:
        df = pd.read_csv(fn)
        # All min-datasets have an index column which has to be dropped:
        dfs.append(df.drop(df.columns[0], axis=1))
    return dfs


def clean_customer_data(df: pd.DataFrame) -> pd.DataFrame:
    """Helper to clean the fashion_news_frequnecy column"""
    df.loc[
        ~df["fashion_news_frequency"].isin(["Regularly", "Monthly"]),
        "fashion_news_frequency",
    ] = "None"
    return df
```

Number of rows in complete transactions csv:

[6]:
```
newlines_in_csv("dataset/transactions_train.csv")
```

[6]:
```
31788325
```

Images

We would also like to know how many articles have images. Since each article has at most one image, we simply count number of files in the image directory to compare with number of lines in articles table.

[7]:
```
def find_number_of_images(path: str = "dataset/images") -> int:
    """Scan `path` and return number of files in `path` and all its subdirectories

    Args:
        path (str): Directory to scan over

    Returns:
        int: Number of files
    """
    num_images = 0
    for _, _, files in os.walk(path):
        num_images += len(files)
    return num_images


num_img, num_article = find_number_of_images(), newlines_in_csv("dataset/articles.
↪csv")
print(
    "Number of images:",
    num_img,
    "\nNumber of articles:",
    num_article,
    "\nPercentage of articles with images:",
    num_img / num_article * 100,
)
```

```
Number of images: 105100
Number of articles: 105543
Percentage of articles with images: 99.58026586320268
```

Not all images have the same shape (dimension). Upon scanning the image dimensions, we see that 93 645 articles have the same dimension, within ± 1 pixel on either height or width, making up for 89.1% of all images. The rest are widely spread, since $1,008$ images have a unique dimension:

[8]:
```
# Let's look at iamge dimensions.
import re
from PIL import Image
from tqdm import tqdm

counts = {}
for root, _, files in tqdm(os.walk("dataset/images/")):
    for filename in files:
        image = Image.open(os.path.join(root, filename))
        dimension = "x".join(str(d) for d in image.size)
        image.close()
        if dimension in counts:
            counts[dimension] += 1
        else:
            counts[dimension] = 1
```

```
df_image_dimensions = pd.Series(counts)
df_image_dimensions.sort_values(ascending=False)
```

```
87it [02:05,  1.44s/it]
```

[8]:
```
1166x1750    93377
1167x1750      217
1166x1749       51
1352x1750       21
1371x1750       20
             ...
1130x1749        1
1137x1749        1
1832x1749        1
1687x1749        1
1672x1749        1
Length: 2804, dtype: int64
```

[9]:
```
print(
    f"Number of images with unique image resolution:␣
 ↪{len(df_image_dimensions[df_image_dimensions == 1])}"
)
```

```
Number of images with unique image resolution: 1008
```

Data exploration

[10]:
```
# Load in full data

transactions_full = pd.read_csv("dataset/transactions_train.csv")
customers_full = pd.read_csv("dataset/customers.csv")
articles_full = pd.read_csv("dataset/articles.csv")
```

[11]:
```
# Column types
print("Transactions: ", ", ".join(transactions_full.columns), end="\n---\n")
print("Customers:", ", ".join(customers_full.columns), end="\n---\n")
print("Articles:", ", ".join(articles_full.columns), end="\n---\n")
```

```
Transactions:  t_dat, customer_id, article_id, price, sales_channel_id
---
Customers: customer_id, FN, Active, club_member_status, fashion_news_frequency,
age, postal_code
---
Articles: article_id, product_code, prod_name, product_type_no,
product_type_name, product_group_name, graphical_appearance_no,
graphical_appearance_name, colour_group_code, colour_group_name,
perceived_colour_value_id, perceived_colour_value_name,
perceived_colour_master_id, perceived_colour_master_name, department_no,
department_name, index_code, index_name, index_group_no, index_group_name,
section_no, section_name, garment_group_no, garment_group_name, detail_desc
---
```

Customer data

Let's start by looking at what values do the categorical columns take:

[12]:
```
for col in ["FN", "Active", "club_member_status", "fashion_news_frequency"]:
    print(col, customers_full[col].unique())
```

```
FN [nan  1.]
Active [nan  1.]
club_member_status ['ACTIVE' nan 'PRE-CREATE' 'LEFT CLUB']
fashion_news_frequency ['NONE' 'Regularly' nan 'Monthly' 'None']
```

This is the motivation for cleaning `fashion_news_frequency` in the helper function at the start. We would also like to know if there is any invalid data in the customers.

[13]:
```
num_customers = customers_full.size
res = {"Column": [], "Number NaN": [], "Portion of entries NaN": []}
for col in customers_full.columns:
    num_na = customers_full[pd.isna(customers_full[col])].size
    res["Column"].append(col)
    res["Number NaN"].append(num_na)
    res["Portion of entries NaN"].append(round(num_na / num_customers, 4))
print(num_customers)
pd.DataFrame(res)
```

```
9603860
```

[13]:

	Column	Number NaN	Portion of entries NaN
0	customer_id	0	0.0000
1	FN	6265350	0.6524
2	Active	6353032	0.6615
3	club_member_status	42434	0.0044
4	fashion_news_frequency	112063	0.0117
5	age	111027	0.0116
6	postal_code	0	0.0000

We see that about 1.16% of ages are NaN. Thus, it should be safe to omit these entries, whereas for *FN* and *Active*, over half of the entries are NaN. This is since NaN here represents the negative in a binary variable, where 1 indicates `True` and NaN indicates `False`.

Furthermore, we look into the age of the customers.

[14]:
```
# Variation of ages in ages
customers_full.describe()["age"][1:]
```

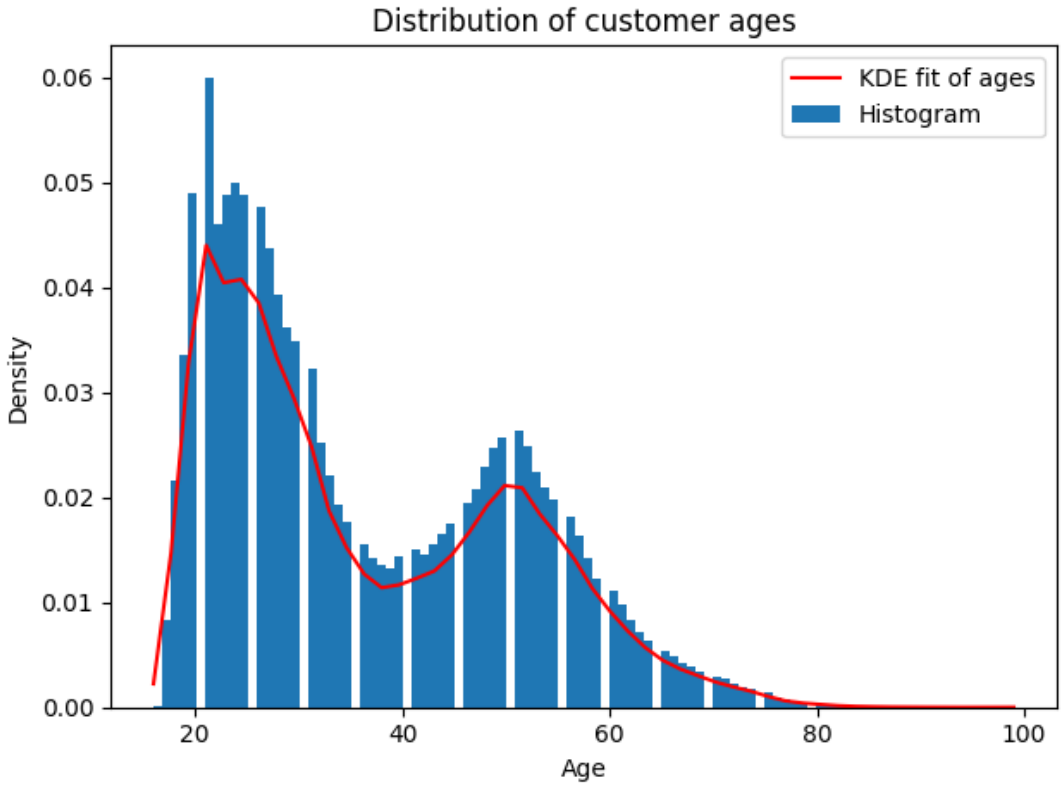
[14]:
```
mean    36.386965
std     14.313628
min     16.000000
25%     24.000000
50%     32.000000
75%     49.000000
max     99.000000
Name: age, dtype: float64
```

The total mean of ages are 36, but is the distribution of ages unimodal?

[15]:
```
samples = customers_full["age"].dropna()
h, e = np.histogram(samples, bins=100, density=True)
x = np.linspace(e.min(), e.max())

kde_fit = gaussian_kde(samples)
plt.plot(x, kde_fit.pdf(x), "r", label="KDE fit of ages")
plt.xlabel("Age")
plt.ylabel("Density")
plt.bar(e[:-1], h, width=np.diff(e), ec=None, align="edge", label="Histogram")
```

```
plt.legend()
plt.title("Distribution of customer ages")
plt.tight_layout()
plt.savefig("figures/distr_ages.pdf")
plt.show()
```



We see that the ages have a multimodal distribution, where the first peak is for the early 20's and the next peak is around 50.

Exploring the article data

From the previous output showing the columns in the datasets, we notice that all variables are indexed with a specific ID/group number;

Index	Value
product_code	prod_name
product_type_no	product_type_name
	product_group_name
graphical_appearance_no	graphical_appearance_name
colour_group_code	colour_group_name
perceived_colour_value_id	perceived_colour_value_name
perceived_colour_master_id	perceived_colour_master_name
department_no	department_name
index_code	index_name
index_group_no	index_group_name
section_no	section_name
garment_group_no	garment_group_name

There are also some clear hierarchies within these column types

[16]:
```
articles_full.groupby(["index_group_name", "index_name"]).count()["article_id"]
```

[16]:
```
index_group_name  index_name
Baby/Children     Baby Sizes 50-98                   8875
                  Children Accessories, Swimwear     4615
                  Children Sizes 134-170             9214
                  Children Sizes 92-140             12007
Divided           Divided                           15149
Ladieswear        Ladies Accessories                 6961
                  Ladieswear                        26001
                  Lingeries/Tights                   6775
Menswear          Menswear                          12553
Sport             Sport                              3392
Name: article_id, dtype: int64
```

[17]:
```
articles_full.groupby(["product_group_name", "product_type_name"]).
↪count()["article_id"]
```

[17]:
```
product_group_name   product_type_name
Accessories          Accessories set         7
                     Alice band              6
                     Baby Bib                3
                     Bag                  1280
                     Beanie                 56
                                           ...
Underwear            Underwear corset        7
                     Underwear set          47
Underwear/nightwear  Sleep Bag               6
                     Sleeping sack          48
Unknown              Unknown               121
Name: article_id, Length: 132, dtype: int64
```

[18]:
```
# Exploring the article data
for col in (
    "department_name",
    "index_name",
    "index_group_name",
    "section_name",
    "garment_group_name",
):
    print(articles_full.groupby(col).count()["article_id"].
↪sort_values(ascending=False))
```

```
department_name
Jersey                              4604
Knitwear                            3503
Trouser                             2655
Blouse                              2362
Dress                               2087
                                    ...
Accessories Other                      1
Kids Boy License                       1
Jersey inactive from S.6               1
Woven bottoms inactive from S.7        1
Shirt Extended inactive from s1        1
Name: article_id, Length: 250, dtype: int64
index_name
Ladieswear                         26001
```

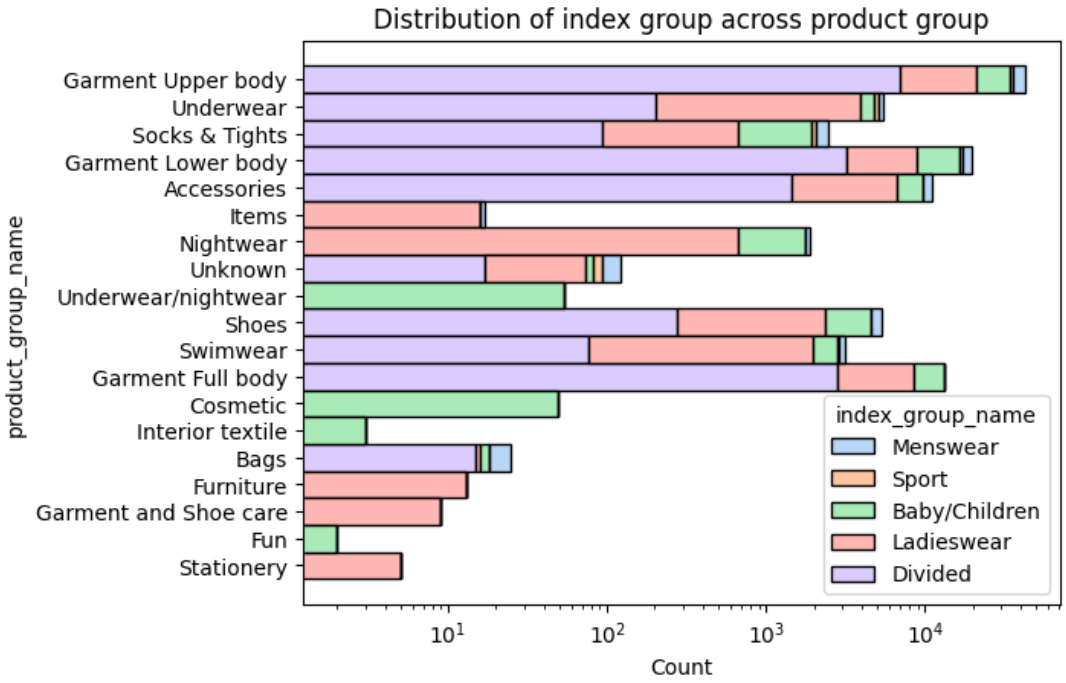
```
Divided                           15149
Menswear                          12553
Children Sizes 92-140             12007
Children Sizes 134-170             9214
Baby Sizes 50-98                   8875
Ladies Accessories                 6961
Lingeries/Tights                   6775
Children Accessories, Swimwear     4615
Sport                              3392
Name: article_id, dtype: int64
index_group_name
Ladieswear       39737
Baby/Children    34711
Divided          15149
Menswear         12553
Sport             3392
Name: article_id, dtype: int64
section_name
Womens Everyday Collection        7295
Divided Collection                7124
Baby Essentials & Complements     4932
Kids Girl                         4469
Young Girl                        3899
Womens Lingerie                   3598
Girls Underwear & Basics          3490
Womens Tailoring                  3376
Kids Boy                          3328
Womens Small accessories          3270
Womens Casual                     2725
Kids Outerwear                    2665
Womens Trend                      2622
Divided Projects                  2364
Young Boy                         2352
H&M+                              2337
Men Underwear                     2322
Mama                              2266
Kids & Baby Shoes                 2142
Boys Underwear & Basics           2034
Womens Shoes                      2026
Ladies H&M Sport                  1894
Womens Swimwear, beachwear        1839
Contemporary Smart                1778
Baby Girl                         1760
Divided Accessories               1732
Kids Accessories, Swimwear & D    1731
Divided Basics                    1723
Baby Boy                          1717
Womens Big accessories            1665
Womens Everyday Basics            1581
Womens Nightwear, Socks & Tigh    1566
Contemporary Casual               1560
Contemporary Street               1490
Men Suits & Tailoring             1428
Men Accessories                   1337
Womens Premium                    1270
Ladies Denim                      1101
Divided Selected                   991
Men H&M Sport                      872
```

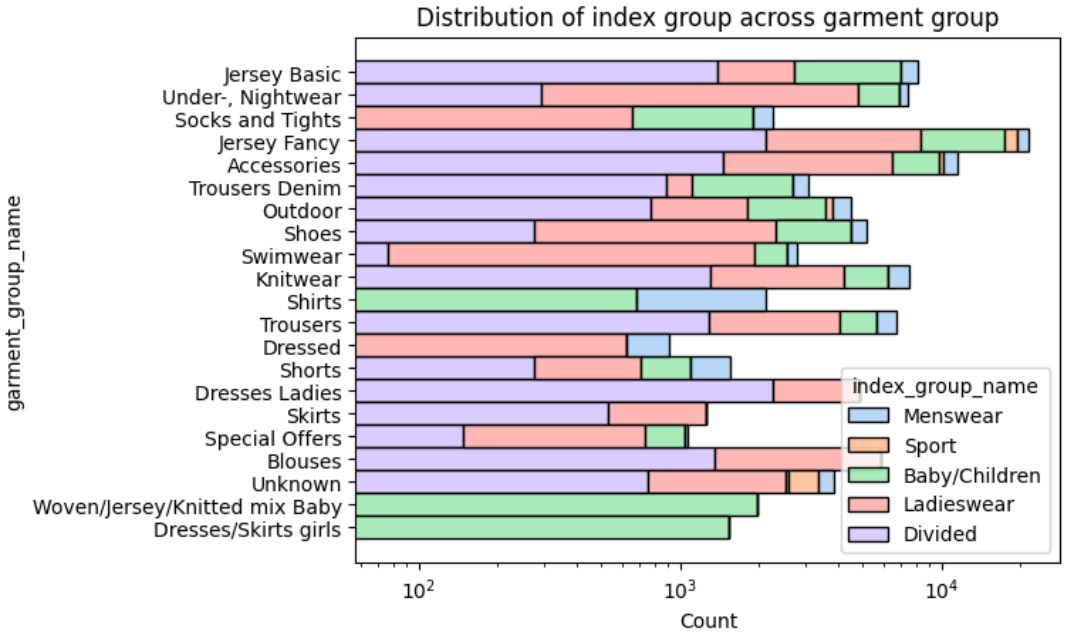
```
Womens Jackets                     829
Special Collections                682
Men Shoes                          645
Mens Outerwear                     629
Kids Sports                        626
Collaborations                     559
Denim Men                          521
Men Edition                        330
Men Project                        298
Divided Asia keys                  280
Kids Local Relevance               192
Men Other 2                        190
Divided Complements Other           35
EQ Divided                          26
Men Other                           25
Ladies Other                         4
Name: article_id, dtype: int64
garment_group_name
Jersey Fancy                     21445
Accessories                      11519
Jersey Basic                      8126
Knitwear                          7490
Under-, Nightwear                 7441
Trousers                          6727
Blouses                           5838
Shoes                             5145
Dresses Ladies                    4874
Outdoor                           4501
Unknown                           3873
Trousers Denim                    3100
Swimwear                          2787
Socks and Tights                  2272
Shirts                            2116
Woven/Jersey/Knitted mix Baby     1965
Shorts                            1559
Dresses/Skirts girls              1541
Skirts                            1254
Special Offers                    1061
Dressed                            908
Name: article_id, dtype: int64
```

[19]:
```python
# Let's see how the indicides are spread across
def barplot_articles_on_index(df: pd.DataFrame, y: str):
    import seaborn

    plt.xscale("log")
    plt.title(f"Distribution of index group across {' '.join(y.split('_')[:-1])}")
    seaborn.histplot(
        data=df,
        y=y,
        hue="index_group_name",
        multiple="stack",
        palette="pastel",
        hue_order=["Menswear", "Sport", "Baby/Children", "Ladieswear", "Divided"],
    )
    plt.savefig(f"figures/seaborn_{y}.pdf", bbox_inches="tight")
    plt.show()
```

```
barplot_articles_on_index(articles_full, "product_group_name")
barplot_articles_on_index(articles_full, "garment_group_name")
```





Some general oberservations from the bar plots:

Most garment and product groups consists mainly of *Divided* index group, followed by a substatial portion *Ladieswear*. Furhtermore, very few items are assigned as *Sport*, some garment and product groups only contains products for *Baby/Children*, and a couple of garment groups contains only *Ladieswear*.

One last observation is how the corresponding garment groups, index groups and other numbers are spread.

[20]:
```
article_id_cols = filter(
    lambda n: n.endswith("_code") or n.endswith("_no") or n.endswith("_id"),
    articles_full.columns,
)
for col in article_id_cols:
    print(col, end="\t")
    print(np.sort(articles_full[col].unique()))
```

```
article_id      [108775015 108775044 108775051 ... 956217002 957375001
959461001]
product_code    [108775 110065 111565 ... 956217 957375 959461]
product_type_no [ -1  49  57  59  60  66  67  68  69  70  71  72  73  74  75  76
77  78
  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96
  98 100 102 144 155 156 161 196 230 231 245 252 253 254 255 256 257 258
 259 260 261 262 263 264 265 267 268 270 271 272 273 274 275 276 277 283
 284 285 286 287 288 289 291 293 295 296 297 298 299 300 302 303 304 305
 306 307 308 326 349 351 366 389 464 465 467 468 470 475 476 477 483 489
 491 492 493 494 495 496 498 499 503 504 508 509 510 511 512 514 515 521
 523 525 529 532 761 762]
graphical_appearance_no [     -1 1010001 1010002 1010003 1010004 1010005 1010006
1010007 1010008
 1010009 1010010 1010011 1010012 1010013 1010014 1010015 1010016 1010017
 1010018 1010019 1010020 1010021 1010022 1010023 1010024 1010025 1010026
 1010027 1010028 1010029]
colour_group_code       [-1  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 17 19
20 21 22 23 30 31
 32 33 40 41 42 43 50 51 52 53 60 61 62 63 70 71 72 73 80 81 82 83 90 91
 92 93]
perceived_colour_value_id       [-1  1  2  3  4  5  6  7]
perceived_colour_master_id      [-1  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 18 19 20]
department_no   [1201 1202 1212 1222 1241 1244 1310 1313 1322 1334 1336 1338
1339 1343
 1344 1352 1410 1414 1422 1444 1447 1510 1515 1522 1543 1545 1547 1610
 1612 1616 1620 1626 1636 1640 1641 1642 1643 1644 1645 1646 1647 1648
 1649 1652 1660 1666 1670 1676 1686 1710 1717 1722 1723 1727 1743 1744
 1745 1747 1752 1772 1773 1774 1778 1779 1909 1912 1919 1920 1926 1929
 1939 1941 1948 1949 2031 2032 2033 2034 2035 2080 2930 2932 2950 3000
 3030 3040 3080 3090 3209 3409 3419 3420 3439 3509 3510 3519 3527 3528
 3529 3608 3610 3611 3629 3630 3705 3708 3709 3710 3828 3929 3931 3932
 3936 3937 3938 3939 3941 3943 3944 3945 3946 3948 4020 4030 4090 4091
 4092 4210 4211 4212 4213 4214 4215 4216 4220 4221 4222 4224 4225 4242
 4310 4311 4312 4313 4314 4317 4320 4321 4322 4323 4327 4342 4343 4344
 4345 4920 5131 5231 5252 5262 5283 5454 5555 5626 5627 5631 5641 5656
 5658 5672 5673 5679 5683 5686 5687 5690 5731 5741 5767 5777 5783 5787
 5828 5831 5832 5833 5848 5858 5868 5878 5882 5883 5884 5888 5950 5952
 5956 5957 5958 5959 5960 5961 5962 5963 5999 6280 6281 6283 6512 6515
 6525 6526 6541 6545 6546 6550 6555 6557 6561 6563 6564 6565 7020 7050
 7188 7388 7389 7510 7520 7530 7613 7616 7617 7618 7648 7655 7656 7657
 7658 7659 7668 7757 7812 7814 7819 7848 7852 7854 7857 7912 7917 7920
 7921 7922 7930 7931 7932 7952 7956 7987 7988 7989 8030 8090 8310 8316
 8394 8396 8397 8398 8558 8559 8560 8563 8564 8615 8616 8617 8713 8716
 8717 8718 8748 8755 8756 8757 8758 8768 8812 8815 8852 8888 8917 8956
 9020 9984 9985 9986 9989]
index_code      ['A' 'B' 'C' 'D' 'F' 'G' 'H' 'I' 'J' 'S']
```

```
index_group_no  [ 1  2  3  4 26]
section_no      [ 2  4  5  6  8 11 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31
 40 41 42 43 44 45 46 47 48 49 50 51 52 53 55 56 57 58 60 61 62 64 65 66
 70 71 72 76 77 79 80 82 97]
garment_group_no        [1001 1002 1003 1005 1006 1007 1008 1009 1010 1011 1012
1013 1014 1016
 1017 1018 1019 1020 1021 1023 1025]
```

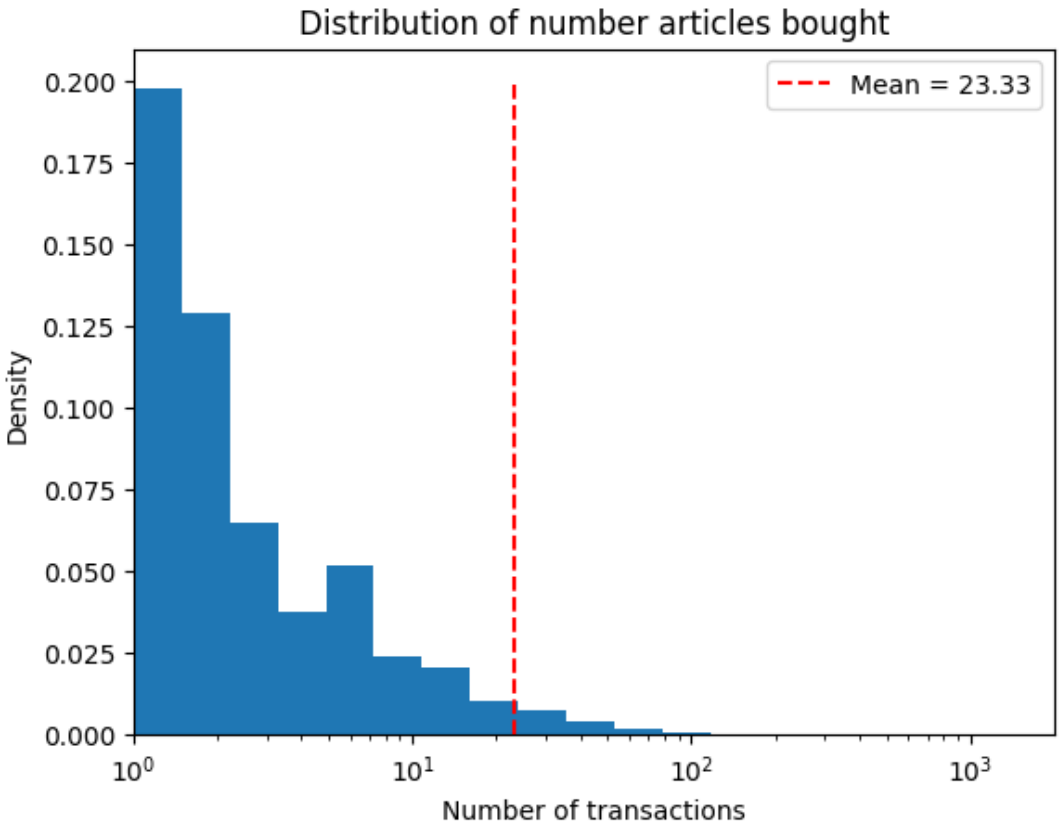
Some of the most interesting findings here, is that a lot of attributes, such as graphical apperance and color has a -1, which most likely is used for articles missing information regarding this propterty. We also notice that some of the data starts with 1 as its first ID whereas others start with 1001. This is useful knowledge when constructing the infrastructure of data loading to the model.

Transactional data

We can look at the number of articles bought for each customer.

[21]:
```
articles_per_customer = transactions_full.groupby("customer_id")["article_id"].
↪count()
logbins = np.geomspace(
    articles_per_customer.values.min(), articles_per_customer.values.max(), 20
)
```

[22]:
```
plt.hist(articles_per_customer.values, bins=logbins, density=True)
plt.vlines(
    x=articles_per_customer.mean(),
    ymin=0,
    ymax=0.2,
    colors="red",
    linestyles="dashed",
    label=f"Mean = {round(articles_per_customer.mean(),2)}",
)
plt.title("Distribution of number articles bought")
plt.ylabel("Density")
plt.xlabel("Number of transactions")
plt.xlim(left=1)
plt.xscale("log")
plt.legend()
plt.savefig("figures/density_articles_bought.pdf")
plt.show()
```

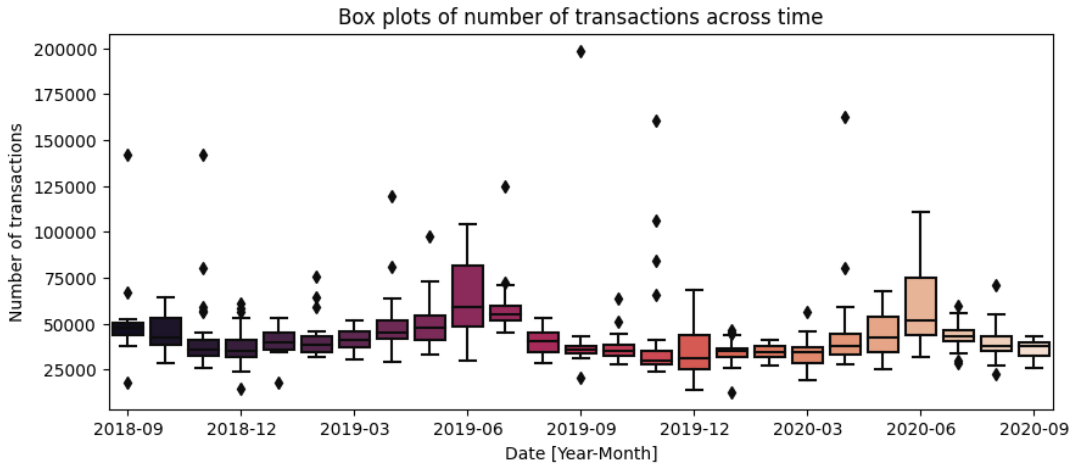


Here, we look into number of transactions over time

[23]:
```
# frequency over time
transactions_full["t_dat"] = pd.to_datetime(transactions_full["t_dat"])
import seaborn as sns

trans_gr_month = transactions_full.groupby("t_dat").size().rename("no_transactions")
trans_gr_month = trans_gr_month.reset_index()
trans_gr_month["month_year"] = trans_gr_month["t_dat"].dt.to_period("M")


fig, ax = plt.subplots(figsize=(10, 4))
ax = sns.boxplot(
    x="month_year", y="no_transactions", data=trans_gr_month, palette="rocket"
)
# plt.xticks(rotation=90)
plt.locator_params(axis="x", nbins=10)
ax.set_xlabel("Date [Year-Month]")
ax.set_ylabel("Number of transactions")
plt.title("Box plots of number of transactions across time")
plt.savefig("figures/boxplot_transactions.pdf")
plt.show()
```



There are also some customers who haven't purchased anything in the dataset.

[24]:
```
customers_full[~customers_full["customer_id"].
↪isin(transactions_full["customer_id"])]
```

[24]:

	customer_id	FN	Active	club_member_status	fashion_news_frequency	age	postal_code
131	00058ecf091cea1bba9d800cabac6ed1ae284202cdab68...	NaN	NaN	ACTIVE	NONE	21.0	1e40bebdc7cbfcb0f0ee995f5e0db13428430874a75c4b...
297	000df4d2084d142416b8165bdd249bab8fea2393447aed...	1.0	1.0	ACTIVE	Regularly	22.0	2c29ae653a9282cce4151bd87643c907644e09541abc28...
544	00193ff7f374dbcfecfa7fead0488e454be4918bec1ebd...	NaN	NaN	ACTIVE	NONE	28.0	0e78861d5ef1caf30cd0c4935cff27657c069c1f2fd2ba...
671	001f00e8c1eba437ff0dbad26a9a3d49e47cbf05fff02a...	1.0	1.0	ACTIVE	Regularly	27.0	d61345e04372ca317ac03dc220889f26a99b4eb20fbf60...
809	002648d8f3b288531b24860f4a68a31d029ec5a0495c04...	1.0	1.0	ACTIVE	Regularly	41.0	4812c1daacee854264500f67aef71d1e713dca9b4755d7...
...	...	...	...	...	...	...	...
1371440	ffe5801cb2a5b51d4d068322d7f8082e995f427a6f22a6...	NaN	NaN	ACTIVE	NONE	55.0	5b8b142d4bb9e28c0d2fc8f87ca22337469f21e7eeb428...
1371554	ffeb3ca867aba57a312fe9d28d67dd46ef2240fe92a94c...	1.0	1.0	ACTIVE	Regularly	21.0	666ea034757d9ac0873fa8f213d7ea359312936e004e99...
1371739	fff456fa60aac9174456c2f36ede5e0f25429a16c88a34...	NaN	NaN	PRE-CREATE	NONE	40.0	
1371872	fffa8d3cea26d4f5186472b923629b35fa28051f258030...	1.0	1.0	ACTIVE	Regularly	21.0	
1371961	ffff01710b4f0d558ff62d7dc00f0641065b37e840bb43...	1.0	1.0	ACTIVE	Regularly	18.0	

```
1371739  efb648f93b15fd00f8a0067791bd47655f6577902b21fc...
1371872  c5641c33d0f01f66e69356fdc8ccb3012b14b1f51a90d0...
1371961  fac16a7bf6ad863c32009166cf834aab744af6caaee552...

[9699 rows x 7 columns]
```

Since the transactions contain information about the price, we can also look into what articles correspond to the highest purchase prices in the dataset.

[25]:
```
groupbyobj = transactions_full.groupby("article_id").agg({"price": list}).
 ↪reset_index()
most_expensive_idx = (
    groupbyobj["price"].apply(np.max).sort_values(ascending=False).head().index
)
most_expensive_idx
```

[25]:
```
Int64Index([50768, 89491, 37363, 79226, 82848], dtype='int64')
```

[26]:
```
groupbyobj.iloc[most_expensive_idx]["price"].apply(
    max
)  # Get the highest price of artices
```

[26]:
```
50768    0.591525
89491    0.506780
37363    0.506780
79226    0.506780
82848    0.506780
Name: price, dtype: float64
```

[27]:
```
# Product names for the most expensive items
articles_full[
    articles_full["article_id"].isin(groupbyobj.
 ↪iloc[most_expensive_idx]["article_id"])
][["article_id", "prod_name", "product_type_name"]]
```

[27]:

	article_id	prod_name	product_type_name
37399	653551001	Benji leather jacket	Jacket
50826	697511001	PE LINDA LEATHER COAT	Coat
79375	797432001	CEMENT leather jacket	Jacket
83012	810872001	PQ AGDA LEATHER DRESS	Dress
89691	839478001	PQ SUSAN LEATHER TUNIC	Dress

Appendix B

Neural collaborative filtering implementation

Neural Collaborative Filtering

This section contains the implementation of the NCF models.
We start by imporing necessary libraries

[1]:
```
%reload_ext jupyter_black
```

```
<IPython.core.display.HTML object>
```

[2]:
```
# >>>>> IMPORTS
import os
import torch
import importlib
import functools
import logging
import pickle
import numpy as np
import pandas as pd
from torch.utils.data import DataLoader, Dataset
from typing import Tuple, Any, Iterable, Union
from sklearn.preprocessing import LabelEncoder
from tqdm import tqdm
from datetime import datetime
from dataclasses import dataclass
from collections import defaultdict
import utils.metrics as metric
import matplotlib
import matplotlib.pyplot as plt

importlib.reload(metric)  # To caputre changes in metrics module
matplotlib.use("Agg")  # Backend for SSH runs


# <<<<< IMPORTS
```

Pre-processing

We start by implementing necessary helper functions used in data cleaning and pre-processing stage of the pipeline.

[3]:
```python
# >>>>> PREPROCESSING
def load_min_data(filename: Union[str, Iterable]):
    """Helper to load minimzed datasets with only 2000 samples, for testing"""
    dfs = []
    if isinstance(filename, str):
        filename = [filename]
    for fn in filename:
        df = pd.read_csv(fn)
        # All min-datasets have an index column which has to be dropped:
        dfs.append(df.drop(df.columns[0], axis=1))
    return dfs


def clean_customer_data(df: pd.DataFrame):
    """Helper to remove NAs from Age and use of consistent labeling"""
    df = df.dropna(subset=["age"])
    df.loc[
        ~df["fashion_news_frequency"].isin(["Regularly", "Monthly"]),
        "fashion_news_frequency",
    ] = "None"
    return df


def clean_article_data(df):
    """Helper that maps index group number `26' to 0 for easier
    label encoding"""
    df.loc[df["index_group_no"] == 26, "index_group_no"] = 0
    return df


# <<<<< PREPROCESSING
```

Data loading

Now that pre-processing methods are defined, we can establish the main class containing the datasets. The main class, `Data_HM`, contains all necessary info regarding positive and negative samples, DataLoaders and training/validation sets.

Below is a child class inherited from `Data_HM`. This specifically is used for generating an equivalent instance with a modified negative sampling algorithm with considerably lower run time.

[4]:
```python
# >>>>> DATA LOADING


class Data_HM(Dataset):
    """This is the general HM Dataset class whose children are train-dataset
    and validation-dataset

    Args:
        Dataset: Abstract Dataset class from pyTorch
    """

    def __init__(
        self,
        total_cases: int,
        portion_negatives: float,
        df_transactions: pd.DataFrame,
        batch_size: int,
        train_portion: Union[float, None] = None,
```

```
        test_portion: Union[float, None] = None,
        transform: Any = None,
        target_transform: Any = None,
    ) -> None:
        super().__init__()
        if train_portion is None:
            if test_portion is None:
                raise ValueError(
                    "Both train portion and test portion cannot be None."
                    )
            self.train_portion = 1 - test_portion
        self.batch_size = batch_size
        self.df_id, self.le_cust, self.le_art = self.generate_dataset(
            total_cases, portion_negatives, df_transactions
        )
        self.train_portion = train_portion
        self.train, self.val = self.split_dataset()
        self.transform, self.target_transform = transform, target_transform

    def generate_dataset(
        self, total_cases: int, portion_negatives: float,
        df_transactions: pd.DataFrame
    ) -> Tuple[pd.DataFrame, LabelEncoder, LabelEncoder]:
        """Produce DataFrames for positive labels and generated negatives

        Args:
            total_cases (int): Total number of transactions
            portion_negatives (float): The portion of the `total_cases` that
                                       should be negative. Balanced 0/1 when 0.5
            df_transactions (pd.DataFrame): Transactions to pull
                                       samples/generate samples from

        Returns:
            Tuple[pd.DataFrame, LabelEncoder, LabelEncoder]:
                * DataFrame on form (customer, article, label) where
                    the IDs are label-encoded,
                * Customer and article LabelEncoder objects
        """
        assert (
            0 <= portion_negatives <= 1
        ), r"portion negatives must be a float between 0%=0.0 and 100%=1.0!"
        if total_cases is None:
            # Defaults to using all cases
            total_cases = len(df_transactions)
        n_positive = round(total_cases * (1 - portion_negatives))
        n_negative = total_cases - n_positive

        df_positive = (
            df_transactions
            .sample(n=n_positive)
            .reset_index(drop=True)
            )
        df_positive = df_positive[["customer_id", "article_id"]]
        df_positive["label"] = 1

        # Sampling negative labels:
        #   We select a random combination of `customer_id`, `article_id`,
```

```
        #   and ensure that this is not a true transaction.
        #   Then we make a 2-column dataframe on same form as `df_positive`

        df_np = df_transactions[["customer_id", "article_id"]].to_numpy()
        neg_np = np.empty((n_negative, df_np.shape[1]), dtype="<U64")
        for i in range(n_negative):
            legit = False
            while not legit:
                sample = [
                    np.random.choice(df_np[:, col]) for col in
                    range(df_np.shape[1])
                ]
                # Checking if random sample actually exists:
                legit = not (
                    (df_np[:, 0] == sample[0]) & (df_np[:, 1] == sample[1])
                ).any()
            neg_np[i, :] = sample
        neg_np = np.column_stack((neg_np, [0] * neg_np.shape[0]))
        df_negative = pd.DataFrame(neg_np, columns=df_positive.columns)
        # Return a shuffled concatenation of the two dataframes
        full_data = (
            pd.concat((df_positive, df_negative))
            .sample(frac=1)
            .reset_index(drop=True)
        )

        # Make label encodings of the IDs
        le_cust = LabelEncoder()
        le_art = LabelEncoder()
        le_cust.fit(full_data["customer_id"])
        le_art.fit(full_data["article_id"])
        cust_encode = le_cust.transform(full_data["customer_id"])
        art_encode = le_art.transform(full_data["article_id"])
        return (
            pd.DataFrame(
                data={
                    "customer_id": cust_encode,
                    "article_id": art_encode,
                    "label": full_data["label"].astype(np.uint8),
                }
            ),
            le_cust,
            le_art,
        )

    def __len__(self):
        return len(self.df_id.index)

    def __getitem__(self, idx):
        row, label = self.df_id.iloc[idx, :-1].values, self.df_id.iloc[idx, -1]
        label = int(label)  # Stored as str initially
        if self.transform:
            row = self.transform(row)
        if self.target_transform:
            label = self.target_transform(label)
        return row, label
```

```
    def split_dataset(self):
        """Split full data to train and validation Subset-objects

        Returns:
            Tuple[Subset, Subset]: Train and validation subsets
        """
        length = len(self)
        train_size = int(length * self.train_portion)
        valid_size = length - train_size
        train, val = torch.utils.data.random_split(
            self, [train_size, valid_size]
            )
        return train, val

    def get_data_from_subset(self, subset: torch.utils.data.Subset):
        """Retrieve data from Subset object directly"""
        return subset.dataset.df_id.iloc[subset.indices]

    def get_DataLoader(self, trainDL: bool = True):
        """Retrieve DataLoader object for either train or validation set.

        Args:
            trainDL (bool, optional): Flag to determine which set to load.
            Defaults to True.

        Returns:
            DataLoader: Torch DataLoader
        """
        subset = self.train if trainDL else self.val
        return DataLoader(dataset=subset, batch_size=self.batch_size)


class Data_HM_Complete(Data_HM):
    def __init__(
        self,
        total_cases: int,
        portion_negatives: float,
        df_transactions: pd.DataFrame,
        batch_size: int,
        train_portion: Union[float, None] = None,
        test_portion: Union[float, None] = None,
        transform: Any = None,
        target_transform: Any = None,
        random_seed: int = None,
    ) -> None:
        self.random_seed = random_seed
        super().__init__(
            total_cases,
            portion_negatives,
            df_transactions,
            batch_size,
            train_portion,
            test_portion,
            transform,
            target_transform,
        )
```

```
    def generate_dataset(
        self, total_cases: int, portion_negatives: float,
        df_transactions: pd.DataFrame
    ) -> Tuple[pd.DataFrame, LabelEncoder, LabelEncoder]:
        logging.debug("Entered child `generate_dataset` method (expected)")
        # NB: Overrides parent method but returns the same info ...
        if self.random_seed is not None:
            np.random.seed(self.random_seed)
            logging.debug(f"Seed set to {self.random_seed}")
        df_transactions = df_transactions[
            ["customer_id", "article_id"]
        ]  # This object has the same behavior as df_positive
        customers = df_transactions["customer_id"].unique()
        articles = df_transactions["article_id"].unique()
        n_positive = round(total_cases * (1 - portion_negatives))
        n_negative = total_cases - n_positive  # mu inferred implicitly

        logging.debug(f"Sampling {n_positive} positives from dataframe")
        # Lazy evalution: if we have to sample 99% of the data,
        # we just take everything
        if n_positive < 0.99 * len(df_transactions):
            positive_samples = df_transactions.sample(
                n=n_positive, replace=False
                )
        else:
            positive_samples = df_transactions

        logging.debug(f"Generating {n_negative} random negative samples")
        samples = pd.DataFrame(
            {
                "customer_id": np.random.choice(
                    customers, size=n_negative, replace=True
                ),
                "article_id": np.random.choice(
                    articles, size=n_negative, replace=True
                ),
            }
        )
        logging.debug("Merging samples in order to remove false negatives")
        samples = pd.merge(
            samples,
            df_transactions,
            on=["customer_id", "article_id"],
            how="outer",
            indicator=True,
        )
        logging.debug("Indexing out false negatives")
        samples = samples.loc[samples["_merge"] == "left_only"].drop(
            "_merge", axis=1
            )

        positive_samples["label"] = 1
        samples["label"] = 0

        logging.debug(
            "Added labels. Now: concat of positive and negative plus a shuffle\
                of all data"
```

```
        )
        samples = (
            pd.concat((positive_samples, samples)).sample(frac=1).reset_index(
                drop=True)
        )

        customers_in_sample = samples["customer_id"].unique()
        articles_in_sample = samples["article_id"].unique()

        le_cust = LabelEncoder()
        le_art = LabelEncoder()
        logging.debug("Fitting label encoders")
        le_cust.fit(customers_in_sample)
        le_art.fit(articles_in_sample)

        # Doing as much as possible in-place here:
        logging.debug("Transforming IDs to label-encoded IDs")
        samples["customer_id"] = le_cust.transform(samples["customer_id"])
        samples["article_id"] = le_art.transform(samples["article_id"])
        samples["label"] = samples["label"].astype(
            np.uint8
        )  # Just to reproduce parent method...
        logging.info("Done generating data!")
        return (
            samples,
            le_cust,
            le_art,
        )


# <<<<< DATA LOADING
```

Model specifications

In this portion, we define the models used for training. `HM_model` is the baseline class which only uses transactional data. The extended model is called `HM_Extended`, and uses attributes from customers and articles to try to improve the prediction.

We also define the dataclass `Hyperparameters`, working as a container of all possible hyperparameters to change.

[ ]:
```
# >>>>> MODEL DEFINITINON AND TRAINING


class HM_model(torch.nn.Module):
    """Baseline HM model"""

    def __init__(
        self,
        num_customer,
        num_articles,
        embedding_size,
        bias_nodes: bool = True,
        sparse: bool = False,
    ):
        super(HM_model, self).__init__()
        self.customer_embed = torch.nn.Embedding(
            num_embeddings=num_customer, embedding_dim=embedding_size,
            sparse=sparse
```

```
        )
        self.art_embed = torch.nn.Embedding(
            num_embeddings=num_articles, embedding_dim=embedding_size,
            sparse=sparse
        )
        if not bias_nodes:  # Default is that we DO have biases
            # They're added lienarly so this should give no effect
            self.customer_bias = lambda row: 0
            self.article_bias = lambda row: 0
        else:
            self.customer_bias = torch.nn.Embedding(
                num_customer, 1, sparse=sparse)
            self.article_bias = torch.nn.Embedding(
                num_articles, 1, sparse=sparse)

    def forward(self, customer_row, article_row):
        """The forward pass used in model training (matrix factorization)

        Args:
            customer_row (Tensor): Tensor of (batch of) customer row(s)
            article_row (Tensor): Tensor of (batch of) article row(s)

        Returns:
            Tensor: Activation of U@V^T + B_u + B_v
        """
        customer_embed = self.customer_embed(customer_row)

        art_embed = self.art_embed(article_row)
        # dot_prod_old = torch.sum(torch.mul(customer_embed, art_embed), 1)
        dot_prod = (customer_embed * art_embed).sum(dim=1, keepdim=True)
        # Add bias nodes to model:
        dot_prod = (
            dot_prod + self.customer_bias(customer_row) +
            self.article_bias(article_row)
        )
        return torch.sigmoid(dot_prod)


class HM_Extended(HM_model):
    """Model class extending the information used in learning process"""

    def __init__(
        self,
        num_customer,
        num_articles,
        num_age,
        num_idxgroup,
        num_garmentgroup,
        embedding_size,
        bias_nodes: bool = True,
        sparse: bool = False,
    ):
        super().__init__(
            num_customer, num_articles, embedding_size, bias_nodes
            )
        del self.art_embed  # For model loading compatibility
        self.article_embed = torch.nn.Embedding(
```

```
            num_articles, embedding_size, sparse=sparse
        )
        self.age_embed = torch.nn.Embedding(num_age, embedding_size)
        self.indexgroup_embed = torch.nn.Embedding(
            num_idxgroup, embedding_size
            )
        self.garmentgroup_embed = torch.nn.Embedding(
            num_garmentgroup, embedding_size
            )

        self.article_MLP = torch.nn.Linear(embedding_size * 3, embedding_size)
        self.customer_MLP = torch.nn.Linear(embedding_size * 2, embedding_size)

    def article_transform(self, article, garment_group, index_group):
        """Concatenates each article embedding through its linear layer,
        activation is sigmoid"""
        embeds = (
            self.article_embed(article),
            self.indexgroup_embed(index_group),
            self.garmentgroup_embed(garment_group),
        )
        final_embedding = self.article_MLP(torch.cat(embeds, 1))
        return torch.sigmoid(final_embedding)

    def customer_transform(self, customer, age):
        """Concatenates each customer embedding through its linear layer,
        activation is sigmoid"""
        embeds = (self.customer_embed(customer), self.age_embed(age))
        final_embedding = self.customer_MLP(torch.cat(embeds, 1))
        return torch.sigmoid(final_embedding)

    def forward(self, row):
        """Forward pass for extended model, overrides parent method"""
        customer, article, age, garment_group, index_group = [
            row[:, i] for i in range(5)
        ]
        # Manipulate IDs to be zero-indexed, thus we don't need label encoders.
        age[age < 0] = 36  # Cast NaNs to average of all customers, 36
        garment_group = garment_group - 1001
        index_group = index_group - 1
        age = age - 1
        customer_matrix = self.customer_transform(customer, age)
        article_matrix = self.article_transform(
            article, garment_group, index_group
            )
        biases = self.customer_bias(customer), self.article_bias(article)
        x = (customer_matrix * article_matrix).sum(1, keepdim=True)
        x = x + biases[0] + biases[1]
        return torch.sigmoid(x)


# Class method to HM_Extended
def _extend_row_data(
    self: HM_Extended,
    customer_rows: Iterable[str],
    article_rows: Iterable[str]
) -> pd.DataFrame:
```

```
    """Adds the given customer/article rows to dataset-object (not in-place)

    Args:
        self (Data_HM): Main dataset object
        customer_rows (Iterable[str]): List of column names of customer info
        article_rows (Iterable[str]): List of column names of article info

    Returns:
        pd.DataFrame: Modified self.df_id with the additional info
    """
    customer_rows = ["customer_id"] + customer_rows
    article_rows = ["article_id"] + article_rows

    # Find original customer and article IDs present in dataset
    df_decoded = self.df_id.copy()
    df_decoded["article_id"] = self.le_art.inverse_transform(
        df_decoded["article_id"]
    )
    df_decoded["customer_id"] = self.le_cust.inverse_transform(
        df_decoded["customer_id"]
    )
    enc_customers, enc_articles = load_kaggle_assets(
        to_download=["customers.csv", "articles.csv"]
    )
    enc_customers = enc_customers[
        enc_customers["customer_id"].isin(df_decoded["customer_id"])
    ]
    enc_articles = enc_articles[
        enc_articles["article_id"].isin(df_decoded["article_id"])
    ]

    enc_customers["customer_id"] = self.le_cust.transform(
        enc_customers["customer_id"]
    )
    enc_articles["article_id"] = self.le_art.transform(
        enc_articles["article_id"]
    )
    df_ext = self.df_id.merge(enc_customers[customer_rows]).merge(
        enc_articles[article_rows]
    )
    # Ensure that last column is the label
    ordered_columns = df_ext.columns[df_ext.columns != "label"].append(
        pd.Index(["label"])
    )
    logging.debug(
        f"df_id has been extended with\
            {', '.join(customer_rows[1:]+article_rows[1:])}"
    )
    return df_ext[ordered_columns]


@dataclass
class Hyperparameters:
    lr_rate: float = 1e-3
    weight_decay: str = 1e-4
    epochs: int = 20
    validation_frequency: int = 1
```

```
    optimizer: Any = torch.optim.Adam
    lossfnc: Any = torch.nn.BCELoss
    embedding_size: int = 500
    bias_nodes: bool = True
    save_loss: Union[bool, str] = True
    verbose: bool = False
    min_lr: float = 0.0  # For LR scheduler

    # These have no use if dataset is loaded from file
    dataset_cases: int = 2000
    dataset_portion_negatives: float = 0.9
    dataset_train_portion: float = 0.7
    dataset_batch_size: int = 5
    dataset_full: bool = False
```

Model training

Below is the implementations for training the models

[ ]:
```
def train_one_epoch(
    model: HM_model,
    data: Data_HM,
    epoch_num: int,
    optimizer,
    loss,
    verbose: bool = False,
    baseline: bool = True,
):
    """Trains a single epoch of Data_HM (or child) model

    Args:
        model (HM_model): HM_model class or child to HM_model.
        data (Data_HM): Dataset class, either Data_HM or a child class
        epoch_num (int): Which epoch this run is
        optimizer (_type_): Optimizer type from main call, usually Adam
        loss (_type_): Loss function, usually BCE
        verbose (bool, optional): Verbose flag, deprecated. Defaults to False.
        baseline (bool, optional): Baseline flag (training a little different
                                   for extended moel). Defaults to True.

    Returns:
        float: Total loss for current epoch
    """
    epoch_loss = 0
    device = "cuda" if torch.cuda.is_available() else "cpu"
    for item in data.get_DataLoader(trainDL=True):
        item = tuple(t.to(device) for t in item)
        row, label = item
        if not baseline:
            row = row.int()  # For ext. model
        optimizer.zero_grad()
        if not baseline:
            pred = model(row)
        else:
            pred = model(row[:, 0], row[:, 1])
        loss_value = loss(pred.view(-1), torch.FloatTensor(
            label.tolist()
        ).to(device))
```

```
        loss_value.backward()
        optimizer.step()
        epoch_loss += loss_value
    if verbose:
        logging.info(f"\t| Training loss for epoch\
            {epoch_num+1}: {epoch_loss}")
    return epoch_loss


def train(model, data, params, baseline: bool = True, plot_loss: bool = False):
    """Main training function for models

    Args:
        model (HM_model | HM_Extended): Model class to train
        data (Data_HM | Data_HM_Complete): Dataset class to use
        params (Hyperparameters): Hyperparameter container
                (and also some other settings)
        baseline (bool, optional): Baseline flag. Defaults to True.
        plot_loss (bool, optional): Flag to produce loss plots to disk or not.
                                    Defaults to False.

    Returns:
        float: Validation loss for the last (computed) epoch
    """
    device = "cuda" if torch.cuda.is_available() else "cpu"
    # Uses binary cross entropy at the moment.
    loss_metric = params.lossfnc().to(device)
    if params.optimizer == "SparseAdam":
        # Does not have weight decay parameter
        optimizer = torch.optim.SparseAdam(
            model.parameters(), lr=params.lr_rate
        )
    else:
        optimizer = params.optimizer(
            model.parameters(), lr=params.lr_rate,
            weight_decay=params.weight_decay
        )

    # Adjust lr once model stops improving using scheduler
    lr_scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
        optimizer, min_lr=params.min_lr, verbose=params.verbose
    )

    save_loss = params.save_loss
    if save_loss:
        train_losses = []
        valid_losses = []
        # `settings` just contains all hyperparameter info, to be able to
        # distinguish which models produce the loss plots when
        # running multiple at the same time
        settings = ",".join([str(v) for v in params.__dict__.values()])

    for epoch in tqdm(range(params.epochs)):
        model.train()
        dataloader_train = data.get_DataLoader(trainDL=True)
        dataloader_valid = data.get_DataLoader(trainDL=False)
        epoch_loss = train_one_epoch(
```

```
            model,
            data,
            epoch,
            optimizer,
            loss_metric,
            params.verbose,
            baseline,
        )
        if not epoch % params.validation_frequency:
            # Validate step
            model.eval()
            valid_loss = 0.0
            for item in dataloader_valid:
                item = tuple(t.to(device) for t in item)
                row, label = item
                if not baseline:
                    row = row.int()
                    pred = model(row)
                else:
                    pred = model(row[:, 0], row[:, 1])
                loss = loss_metric(
                    pred.view(-1), torch.FloatTensor(label.tolist()).to(device)
                )
                valid_loss = loss.item() * row.size(0)

            lr_scheduler.step(valid_loss)  # Update lr scheduler
            if params.verbose:
                logging.info(f"Provisory results for epoch {epoch+1}:")
                logging.info(
                    f"Loss for training set\t\
                        {epoch_loss.tolist() / len(dataloader_train)}"
                )
                logging.info(
                    f"Loss for validation set\t\
                        {valid_loss / len(dataloader_valid)}"
                )
            if save_loss:
                train_losses.append(
                    epoch_loss.tolist() / len(dataloader_train)
                )
                valid_losses.append(
                    valid_loss / len(dataloader_valid)
                )
    if save_loss:
        fn_append = save_loss if isinstance(save_loss, str) else ""
        save_dir = os.path.join("results", datetime.today().strftime(
            "%Y.%m.%d.%H.%M"
            ))
        if not os.path.isdir(save_dir):
            os.makedirs(save_dir)
        filename = (
            os.path.join(save_dir, f"losses_{fn_append}.csv")
            .replace("<", "")
            .replace(">", "")
        )
        np.savetxt(
            filename,
```

```
            np.transpose([train_losses, valid_losses]),
            delimiter=",",
        )
        logging.debug(f"Saved losses for this run to {filename}")
        if plot_loss:
            plot_loss_results(
                filename,
                save=f"{'b' if baseline else 'e'}_{settings}.pdf",
                baseline=baseline,
            )
            # # If you want to remove the csv files of the losses,
            # # you can call `os.remove(filename)` here
            # os.remove(filename)
    return valid_loss / len(dataloader_valid)


def load_dataset_and_train(
    use_min_dataset: bool = False,
    persisted_dataset_path: Union[str, None] = None,
    save_model: Union[str, bool] = False,
    hyperparams=Hyperparameters(),
    transactions_path: str = "dataset/transactions_train.csv",
    baseline: bool = True,
):
    """This model loads the data files, database object and
    runs the train method. Will also save trained model"""

    # Load data
    if use_min_dataset:
        df_c, df_a, df_t = load_min_data(
            [
                f"dataset_sample/{n}_min.csv"
                for n in ("customer", "articles", "transactions")
            ]
        )
    elif hyperparams.dataset_full:
        logging.debug("Creating full dataset object")
        df_t = pd.read_pickle(transactions_path)
        dataset_params = {
            "total_cases": hyperparams.dataset_cases,
            "portion_negatives": hyperparams.dataset_portion_negatives,
            "df_transactions": df_t,
            "train_portion": hyperparams.dataset_train_portion,
            "batch_size": hyperparams.dataset_batch_size,
        }
        data = Data_HM_Complete(**dataset_params)
        logging.debug("Full dataset object done!")
    elif persisted_dataset_path is None:
        df_c = pd.read_csv("dataset/customers.csv")
        # Articles IDs all start with 0 which disappears if cast to a number
        df_a = pd.read_csv("dataset/articles.csv", dtype={"article_id": str})
        df_t = pd.read_csv(transactions_path, dtype={"article_id": str})
        df_c = clean_customer_data(df_c)
        df_a = clean_article_data(df_a)

        dataset_params = {
            "total_cases": hyperparams.dataset_cases,
```

```
            "portion_negatives": hyperparams.dataset_portion_negatives,
            "df_transactions": df_t,
            "train_portion": hyperparams.dataset_train_portion,
            "batch_size": hyperparams.dataset_batch_size,
        }
        data = Data_HM(**dataset_params)
        save_dataset_obj(
            data,
            "object_storage/dataset-" +
            f"{datetime.today().strftime('%Y.%m.%d.%H.%M')}.pckl",
        )
    else:
        data = read_dataset_obj(persisted_dataset_path)
        logging.debug("Read dataset successfully")

    model = load_model(
        baseline, data, hyperparams.embedding_size, hyperparams.bias_nodes
    )
    logging.debug("Created model sucessfully")
    last_valid_loss = train(model, data, hyperparams, baseline, plot_loss=True)
    if save_model:
        if isinstance(save_model, bool):
            save_model = datetime.today().strftime("%Y.%m.%d.%H.%M") + ".pth"
        if not os.path.isdir("models"):
            os.mkdir("models")
        torch.save(model.state_dict(), os.path.join("models", save_model))
    return last_valid_loss


def get_k_most_purchased(
    k: int,
    transactions_path: Union[str, pd.DataFrame],
    pass_as_df: bool = False,
    by_index_group: Union[bool, int] = False,
    bulk_index_group=None,
) -> np.ndarray:
    if isinstance(transactions_path, str):
        if transactions_path.endswith("pckl"):
            df = pd.read_pickle(transactions_path)
        else:
            df = pd.read_csv(transactions_path, dtype={"article_id": str})
    else:
        df = transactions_path
    if by_index_group:
        logging.debug(f"Accessing topk based on index group {by_index_group}")
        _, df_articles = load_kaggle_assets(["customers.csv", "articles.csv"])
        if bulk_index_group is not None:
            out = []
            for index_group in bulk_index_group:
                df_i = df[df["index_group_no"] == index_group]
                out.append(
                    df_i.groupby("article_id")
                    .agg("customer_id")
                    .count()
                    .sort_values(ascending=False)
                    .head(k)
                    .index.values
```

```
                )
            return out

        df_articles[df_articles["index_group_no"] == by_index_group]
        df_articles = df_articles[["article_id", "index_group_no"]]
        df = df.merge(df_articles, on="article_id")

    return (
        df.groupby("article_id")
        .agg("customer_id")
        .count()
        .sort_values(ascending=False)
        .head(k)
        .index.values
    )


# <<<<< MODEL DEFINITION AND TRAINING
```

Utility functions

Below are a couple important utility functions.

- `time_pd_vs_np` computes the runtime from the sampling methods available, used to show that the pandas method is really slow
- `plot_negative_sampling` is also related to the same result, producing the plots
- `save_dataset_obj` is technically a general method to serialize any object. If the path in argument does not exist, it will create it (safer)
- `read_dataset_obj` similarly just reads a pickled object
- `load_kaggle_assets` downloads tables from the competition site, loads to DataFrames and removes the files from disk.
- `load_from_gdrive` downloads a file based on Google drive URL/ID
- `load_model` will retrieve the correct object based on what model is passed as argument.
- `plot_loss_results` shows the training and/or validation loss for a training result and stores to disk
- `compare_hyperparameter_results` produces similar plots specialized to list of results for different hyperparameters

A few of these are also not used at the moment:

- `heuristic_embedding_size` is a result from FastAI indicating optimal choice for embedding size. Not actually used currently
- `explore_hyperparameters` does a *naïve* hyperparameter search, finding the optimal value by tuning one variable at a time, keeping the rest constant.
- `alternative_hyperparam_exploration` checks *every* combination of variable choices, but due to runtime is not used
- `split_test_set_file` is a method to make a 70-30 split in dataset

[ ]:
```
# >>>>> UTILITIES


def time_pd_vs_np(n_negative, df) -> Tuple[float, float]:
    """Compute time it takes to sample n_negative negative transactions

    Args:
        n_negative (int): Number of negative samples
        df (pd.DataFrame): Dataframe to sample from, requires columns
        'customer_id' and 'article_id'
```

```
    Returns:
        Tuple[float, float]: Time taken using Pandas objects (first value) and
        NumPy objects (second value)
    """
    import time

    start_pd = time.time()
    num_written = 0
    tmpStr = "customer_id,article_id\n"
    while num_written < n_negative:
        # Choose random customer and article
        selection = np.array(
            [
                df["customer_id"].sample().values,
                df["article_id"].sample().values,
            ]
        ).flatten()
        if not (
            (df["customer_id"] == selection[0]) & (
                df["article_id"] == selection[1]
                )).any():
            tmpStr += f"{selection[0]}, {selection[1]}\n"
            num_written += 1
    with open("tmp.csv", "w") as f:
        f.write(tmpStr)
    _ = pd.read_csv("tmp.csv")
    os.remove("tmp.csv")
    time_pd = time.time() - start_pd

    # Numpy method
    start_np = time.time()
    df_np = df[["customer_id", "article_id"]].to_numpy()
    neg_np = np.empty((n_negative, df_np.shape[1]), dtype="<U64")
    for i in range(n_negative):
        legit = False
        while not legit:
            sample = [np.random.choice(
                df_np[:, col]
            ) for col in range(df_np.shape[1])]
            legit = not (
                (df_np[:, 0] == sample[0]) & (df_np[:, 1] == sample[1])
            ).any()
        neg_np[i, :] = sample
    time_np = time.time() - start_np

    return time_pd, time_np


def plot_negative_sampling(
    start: int,
    stop: int,
    step: int = 1,
    filename: Union[str, None] = None,
    persist_data: bool = True,
    cont_from_checkpoint: bool = True,
) -> None:
    """Plot the outputs of `time_pd_vs_np` for different ranges of n_negative
```

```
    Args:
        start (int): Range of n_negative (inclusive)
        stop (int): Range of n_negative (exclusive)
        step (int, optional): Step in range of n_negative. Defaults to 1.
        filename (str | None, optional): Plot output file name, if None,
                                         does not save file. Defaults to None.
        persist_data (bool, optional): Serialization option to store each
                                           iterate's result. Defaults to True.
        cont_from_checkpoint (bool, optional): Reads previous runs and doesn't
                                 recompute if done before. Defaults to True.
    """
    import matplotlib.pyplot as plt
    from tqdm import tqdm
    import pickle

    xax = list(range(start, stop, step))

    if cont_from_checkpoint:
        with open("plotData.pckl", "rb") as f:
            plot_values = pickle.load(f)

        # Add empty list for keys not covered by checkpoint:
        computed = set([x_i for x_i in plot_values.keys()])
        to_add = set(xax) - computed
        for elem in to_add:
            plot_values[elem] = []

        # Skip those already computed
        xax = [x for x in xax if x not in computed]

    else:
        plot_values = {x_i: [] for x_i in xax}

    for n_negative in tqdm(xax):
        time_pd, time_np = time_pd_vs_np(n_negative)
        plot_values[n_negative].extend([time_pd, time_np])

        if persist_data:
            with open("plotData.pckl", "wb") as f:
                pickle.dump(plot_values, f)

    plt.plot(
        plot_values.keys(),
        plot_values.values(),
        label=[
            "pandas.DataFrame.sample implementation",
            "NumPy.random.choice implementation",
        ],
    )
    plt.legend()
    plt.xlabel("Number of negative (generated) samples")
    plt.ylabel("Time [s]")
    plt.title("Comparison between sampling methods time")
    if filename is not None:
        plt.savefig(f"{filename}.pdf")
    plt.show()
```

```python
def save_dataset_obj(data: HM_model, dst: str) -> None:

    if not os.path.isdir(os.path.dirname(dst)):
        os.makedirs(os.path.dirname(dst))
    with open(dst, "wb") as f:
        pickle.dump(data, f)


def read_dataset_obj(src: str) -> Any:

    with open(src, "rb") as f:
        data = pickle.load(f)
    return data


def load_kaggle_assets(
    to_download: Union[Iterable[str], None] = None
) -> Iterable[pd.DataFrame]:
    """Downloads data from Kaggle competition, loads to Dataframes and deletes
    original files. Made due to low disk quota on server

    Args:
        to_download (list[str] | None, optional): List of files to download,
        if None will download all csv files. Defaults to None.

    Returns:
        list[pd.DataFrame]: Dataframes from downloaded files
    """
    logging.debug("Loading assets from Kaggle to Markov ...")
    from zipfile import ZipFile
    from kaggle.api.kaggle_api_extended import KaggleApi

    api = KaggleApi()
    api.authenticate()
    if to_download is None:
        to_download = [
            "customers.csv", "transactions_train.csv", "articles.csv"
        ]
    pd_objects = []

    for i, file in enumerate(to_download):
        logging.debug(f"Downlodaing file {file}")
        api.competition_download_file(
            competition="h-and-m-personalized-fashion-recommendations",
            file_name=file
        )

        zf = ZipFile(f"{file}.zip")
        zf.extractall()
        zf.close()

        # Customers-file does not require dtype specification, the others do
        if i == 0:
            df = pd.read_csv(file)
        else:
```

```
            df = pd.read_csv(file, dtype={"article_id": str})
        pd_objects.append(df)
        # Remove files created from download now that it's loaded to memory
        os.remove(file)
        os.remove(f"{file}.zip")
        logging.debug(f"Sucessfully loaded and removed {file}")
    return pd_objects


def load_from_gdrive(gd_id: str, outpath: str = "tmp_model.pth") -> None:
    """Assumes you want to download a trained model from google drive (big!)"""
    import gdown

    gdown.download(id=gd_id, output=outpath)
    assert os.path.isfile(outpath), "Unable to fine downloaded file!"
    logging.debug(
        "Sucessfully downloaded pth file. Make sure to remove after loading" +
        " to memory"
    )
    return outpath


def load_model(
    baseline: bool,
    data: Data_HM,
    emb_sz: int = 500,
    bias: bool = True,
    sparse: bool = False,
) -> object:
    """Based on baseline flag, retrieves model and ensures
    column types are correct.

    Args:
        baseline (bool): Baseline flag
        data (Data_HM): Original Dataset object (NB self.df_id can be
        modified when passed).
        emb_sz (int, optional): Embedding size, must match size in `data`.
        Defaults to 500.
        bias (bool, optional): Bias flags (include bias nodes or not).
        Defaults to True.
        sparse (bool, optional): Sparse property to model embeddings.
        Defaults to False

    Returns:
        object: HM_model or HM_Extended
    """
    device = "cuda" if torch.cuda.is_available() else "cpu"
    n_cust, n_art, *_ = data.df_id.nunique()
    if baseline:
        model = HM_model(
            n_cust, n_art, embedding_size=emb_sz, bias_nodes=bias,
            sparse=sparse
        ).to(device)
    else:
        additional_columns = {"age", "garment_group_no", "index_group_no"}
        if additional_columns == additional_columns.intersection(
            set(data.df_id.columns)
```

```
        ):
            logging.debug(
                "Dataset object is already extended, no need to download anew"
            )
        else:
            logging.debug("Transforming df_id to work for extended model")
            data.df_id = _extend_row_data(
                data, ["age"], ["garment_group_no", "index_group_no"]
            )
        # age/idxgroup/ggroup have to be the max instead of nunique
        n_age, n_idxgroup, n_garmentgroup = (
            data.df_id.max()[2:5].values.astype(int)
        )

        model = HM_Extended(
            num_customer=n_cust,
            num_articles=n_art,
            num_age=n_age,
            num_idxgroup=n_idxgroup,
            num_garmentgroup=n_garmentgroup,
            embedding_size=emb_sz,
            bias_nodes=bias,
        ).to(device)
    return model


def plot_loss_results(
    lossfile,
    which: str = "all",
    plot_title: bool = True,
    save: Union[str, None] = None,
    baseline: bool = True,
):
    """Generate plot of training and/or validation loss across trained epochs

    Args:
        lossfile (str): Path to csv file containing training
            and validation losses
        which (str, optional): Which plot to make (training, validation, all).
            Defaults to "all".
        plot_title (bool, optional): Title flag. Won't use any titles if False.
            Defaults to True.
        save (Union[str, None], optional): Destination path for plot.
            Won't save to disk of None. Defaults to None.
        baseline (bool, optional): Baseline flag
            (assumes model=='extended' if False). Defaults to True.
    """
    res = pd.read_csv(lossfile, header=None)
    plt.figure()
    if which != "validation":
        plt.plot(res[0], label="Training loss")
    if which != "training":
        plt.plot(res[1], label="Validation loss")
    plt.xlabel("Epoch number")
    plt.ylabel("Loss value")
    plt.legend()
    title = "Training and validation" if which == "all" else which.capitalize()
```

```
    title += f" loss for {'baseline' if baseline else 'extended'} model"
    if plot_title:
        plt.title(title)
    if save is not None:
        plt.savefig(save)
    else:
        plt.show()


def compare_hyperparameter_results(data: Iterable[str]):
    """Helper to plot the training loss for each run specified in `data`"""
    plt_calls = {}
    for results_fn in data:
        plt.xlabel("Epoch number")
        plt.ylabel("Loss value")
        res = pd.read_csv(results_fn, header=None)
        setting = results_fn[
            results_fn.find("_") + 1:
        ]  # Retrieves filename more or less
        variable = setting.split("=")[0]
        # Here we just iterate through Hyperparam. choices and
        # store the function call for later use
        if variable in plt_calls:
            plt_calls[variable].append(
                functools.partial(plt.plot, res[0], label=setting)
            )
        else:
            plt_calls[variable] = [
                functools.partial(plt.plot, res[0], label=setting)
            ]

    # Iterate through previously established function calls and plots
    for var, cmd_list in plt_calls.items():
        for cmd in cmd_list:
            cmd()  # plt.plot(...) for specific variable
        plt.legend()
        plt.yscale("log")
        plt.title(f"Training loss loss for different choices of {var}")
        plt.savefig(f"hyperparams_train_{var}.pdf")
        plt.show()


def explore_hyperparameters():
    """Runs model for different choices of weight decay,
    embedding size and learn rate"""
    weight_decays = (1e-4, 1e-3, 1e-2, 1e-1)
    embedding_size = (10, 100, 500, 1000, int(1e4))
    lr_rates = (1e-5, 1e-4, 1e-3, 1e-2)
    print("Testing EMBEDDING SIZE")
    for emb_size in embedding_size:
        testing_param = f"{emb_size = }"
        print(testing_param)
        hparams = Hyperparameters(
            embedding_size=emb_size,
            save_loss=testing_param
        )
        load_dataset_and_train(
```

```
            persisted_dataset_path="object_storage/HM_data.pckl",
            hyperparams=hparams
        )
    print("Testing WEIGHT DECAYS")
    for wd in weight_decays:
        testing_param = f"{wd = }"
        print(testing_param)
        hparams = Hyperparameters(weight_decay=wd, save_loss=testing_param)
        load_dataset_and_train(
            persisted_dataset_path="object_storage/HM_data.pckl",
            hyperparams=hparams
        )
    for lr in lr_rates:
        testing_param = f"{lr = }"
        print(testing_param)
        hparams = Hyperparameters(lr_rate=lr, save_loss=testing_param)
        load_dataset_and_train(
            persisted_dataset_path="object_storage/HM_data.pckl",
            hyperparams=hparams
        )


# Functions below are currently not in use, but can be practical


def heuristic_embedding_size(cardinality):
    # https://github.com/fastai/fastai/blob/master/fastai/tabular/model.py#L12
    return min(600, round(1.6 * cardinality**0.56))


def alternative_hyperparam_exploration(
    wds: Iterable,
    embszs: Iterable,
    lrs: Iterable,
    dataset_path=None,
    baseline: bool = True,
) -> dict:
    """Checks each combination of choices and prints out the one
    with best validation loss

    Args:
        wds (Iterable): List of weight decay
        embszs (Iterable): List of embedding sizes
        lrs (Iterable): List of learning rates
        dataset_path (str|None, optional): Path to dataset instance,
            if None will generate a new each time. Defaults to None.
        baseline (bool, optional): Flag if model to test is the
            baseline or extended. Defaults to True.

    Returns:
        dict: dictionary of parameter choices corresponding to best model
    """
    best_model = {}
    best_loss = np.inf
    for emb_size in embszs:
        for wd in wds:
            for lr in lrs:
```

```
                params = {
                    "embedding_size": emb_size,
                    "epochs": 20,
                    "save_loss": True,
                    "weight_decay": wd,
                    "lr_rate": lr,
                    "verbose": True,
                    "validation_frequency": 1,
                }
                logging.debug(
                    "Training 20 epochs of " +
                    f"{'baseline' if baseline else 'extended'} model." +
                    f"Settings: {lr = }, {wd = }, {emb_size = }"
                )
                loss = load_dataset_and_train(
                    persisted_dataset_path=dataset_path,
                    hyperparams=Hyperparameters(**params),
                    baseline=baseline,
                    save_model=True,
                )
                logging.debug(f"Last epoch loss: {loss}")
                if loss < best_loss:
                    best_model = params
                    best_loss = loss
                    logging.debug(f"Found better model:\n{best_model}\n--")
    return best_model


def split_test_set_file():
    """Method to make a 70-30 split in dataset, not in use"""
    full_set = pd.read_csv(
        "dataset/transactions_train.csv", dtype={"article_id": str}
    )
    num_train = int(len(full_set) * 0.7)
    num_test = len(full_set) - num_train
    test_idx = np.random.randint(0, len(full_set), size=num_test)
    full_set.iloc[test_idx].to_csv("dataset/tr_test.csv")
    full_set.drop(test_idx).to_csv("dataset/tr_train.csv")


# <<<<< UTILITIES
```

Inference

Below we implement the methods for computing the MAP@12 metric.

[ ]:
```
# >>>>> INFERENCE


def predictions_aggregator(data: Data_HM, by_index: bool = True):
    """Get k top predictions for customer only based on
    what index group it has the most purchases in"""
    if not by_index:
        df = data.df_id[data.df_id["label"] == 1].drop("label", axis=1)
        top_predictions = get_k_most_purchased(12, df).tolist()
        return {
            customer: [top_predictions] for customer in df[
                "customer_id"
```

```
            ].unique()
            }

    df = _extend_row_data(data, [], ["index_group_no"])  # df_id
    train = data.get_data_from_subset(data.train)  # train set of df_id
    df = df.merge(train)
    df = df[df["label"] == 1].drop("label", axis=1)
    logging.debug(f"DF has now rows {df.columns} and is of length {len(df)}.")
    all_index_groups = df["index_group_no"].unique()
    all_top_predictions = get_k_most_purchased(
        k=12,
        transactions_path=df,
        pass_as_df=True,  # Note that this is currently unsued
        by_index_group=True,
        bulk_index_group=all_index_groups,
    )
    top_predictions = {
        index_group: value
        for index_group, value in zip(all_index_groups, all_top_predictions)
    }
    logging.debug(top_predictions)
    best_preds = {}
    group = (
        df
        .groupby("customer_id")
        .agg({"index_group_no": list})
        .reset_index()
    )
    group["index_group_no"] = group["index_group_no"].apply(
        lambda lst: max(set(lst), key=lst.count)
    )
    for idx, row in group.iterrows():
        best_preds[row.customer_id] = [
            top_predictions[row.index_group_no].tolist()
        ]

    return best_preds


@torch.inference_mode()
def topk_for_all_customers(
    model_path,
    test_data,
    k: int = 12,
    baseline: bool = False,
    n_customer_threshold: int = 100,
    remove_pth_file_after_load: bool = False,
) -> dict:
    """Make dictionary on form {customer: [art_1, ..., art_k]}
    for k highest predicted articles, for each customer and article
    in the *validation set*"""

    def _update_out_dict(all_preds: defaultdict):
        """Helper to add predictions to out_dict"""
        for customer_dict in all_preds:
            k_i = min(k, len(all_preds[customer_dict]))
            keys_i = np.array(list(all_preds[customer_dict].keys()))
```

```
            values_i = list(all_preds[customer_dict].values())
            top_ind = np.argpartition(values_i, -k_i)[-k_i:]
            out_dict[customer_dict] = [keys_i[top_ind].tolist()]

    def iterate_through_extended_model(all_preds: defaultdict):
        """Internal method if predictions are not for baseline"""
        logging.debug("Entered extended iterating method")
        customer_count = 0
        for customer_row in tqdm(valid_customers):
            customer_i = customer_row[0].item()  # Don't include age data
            customer_count += 1
            for article_rows in valid_articles:
                bz = article_rows.shape[0]
                customer_rows = customer_row.expand(bz, 2)
                row = torch.concat(
                    (
                        customer_rows[:, 0].reshape((bz, 1)),
                        article_rows[:, 0].reshape((bz, 1)),
                        customer_rows[:, 1].reshape((bz, 1)),
                        article_rows[:, 1:].reshape((bz, 2)),
                    ),
                    dim=1,
                )
                pred = model(row).view(-1)
                for i, pred_i in enumerate(pred):
                    article_i = row[i, 1].item()
                    all_preds[customer_i][article_i] = pred_i
            if customer_count % n_customer_threshold == 0:
                # Clears all predictions again to save memory
                logging.debug("(new) Sending batch to out dict...")
                _update_out_dict(all_preds)
                # Free memory and re-initiate defaultdict
                del all_preds
                all_preds = defaultdict(dict)

    # Initialize model and dataset
    all_preds = defaultdict(
        dict
    )  # Essentially a 2D-dict, see `collections.defaultdict`

    device = "cuda" if torch.cuda.is_available() else "cpu"
    model = load_model(baseline, test_data, emb_sz=500)
    model.load_state_dict(torch.load(
        model_path, map_location=torch.device(device)
    ))
    if remove_pth_file_after_load:
        logging.info(f"Removing pth-file {model_path}")
        os.remove(model_path)
    model.eval()

    # Access validation data
    data = test_data.get_data_from_subset(test_data.val).to_numpy()
    if not baseline:
        _, customer_idx = np.unique(data[:, 0], return_index=True)
        valid_customers = data[customer_idx, :]
        valid_customers = valid_customers[:, [0, 2]]  # customer id and age
        logging.debug(f"Customers with age component,\n{valid_customers}")
```

```
        _, article_idx = np.unique(data[:, 1], return_index=True)
        valid_articles = data[article_idx, :]
        valid_articles = valid_articles[:, [1, 3, 4]]  # article data
    else:
        valid_customers = np.unique(data[:, 0])
        valid_articles = np.unique(data[:, 1])
    valid_customers, valid_articles = torch.IntTensor(valid_customers).to(
        device
    ), torch.IntTensor(valid_articles).to(device)
    from math import ceil

    # Create batches of articles similar to the model batch size
    valid_articles = np.array_split(
        valid_articles, ceil(valid_articles.shape[0] / test_data.batch_size)
    )
    out_dict = dict()
    logging.info("Computing predictions for customers ...")
    if not baseline:
        iterate_through_extended_model(
            all_preds
        )  # model,valid_customers,test_data,n_customer_threshold,all_preds)
    else:
        logging.debug("Entered iteration through baseline data")
        customer_count = 0
        for customer in tqdm(valid_customers):
            customer_count += 1
            for article in valid_articles:
                # For customer 3, we pass model([3, ..., 3], [a1, ..., an])
                customer_exp = customer.expand(article.shape[0])
                pred = model(customer_exp, article).view(-1)
                for i, pred_i in enumerate(pred):
                    all_preds[customer.item()][article[i]] = pred_i
            if customer_count % n_customer_threshold == 0:
                # Clears all predictions again to save memory
                logging.debug("Sending batch to out dict...")
                _update_out_dict(all_preds)
                # Free memory and re-initiate defaultdict
                del all_preds
                all_preds = defaultdict(dict)

    # Call this once more for the last n<n_customer_threshold customers
    _update_out_dict(all_preds)
    return out_dict


@torch.inference_mode()
def compute_map(
    best_preds: dict,
    test_data: Data_HM,
    k: int = 12,
    use_all_data_as_ground_truth: bool = False,
) -> pd.DataFrame:
    """Compute average precision @k for predicted entries
    `out_dict` from previous function

    Args:
        best_preds (dict): Output dictionary from
```

```
            topk_from_all_customers function
        test_data (Data_HM): Dataset object
        k (int, optional): Cutoff. Defaults to 12.
        use_all_data_as_ground_truth (bool, optional): Bool flag, if False
            will only consider validation set. Defaults to False.

    Returns:
        pd.DataFrame: Average precision for all customers such that its
        .mean() is the MAP@k value
    """
    if use_all_data_as_ground_truth:
        logging.debug("Returning best predicted values back to true IDs ...")
        decoded_preds = dict()
        for key, v in best_preds.items():
            unenc_k = test_data.le_cust.inverse_transform([key])[0]
            unenc_v = test_data.le_art.inverse_transform(v[0])
            decoded_preds[unenc_k] = [unenc_v]
        best_preds = decoded_preds  # Overwrites best_preds

        # Ground truth from entire database:
        logging.debug("Reading ground truth ...")
        ground_truth = pd.read_pickle("object_storage/transactions.pckl")
        # # Alternative method if csv is stored to disk.
        # ground_truth = pd.read_csv(
        #     "dataset/transactions_train.csv",
        #     dtype={"article_id": str},
        #     usecols=["customer_id", "article_id"],
        # )

        ground_truth = ground_truth[
            ground_truth["customer_id"].isin(list(best_preds.keys()))
        ]
        logging.debug("Grouping ground truth by customer")
        ground_truth = (
            ground_truth
            .groupby("customer_id")
            .agg({"article_id": list})
            .reset_index()
        )
        logging.debug(f"Head of ground_truth\n{ground_truth.head()}")
    else:
        logging.debug("Loading only data in validation set")
        # This only checks the true values of the (training and) validation set
        ground_truth = (
            test_data.df_id[test_data.df_id["label"] == 1]
            .groupby("customer_id")
            .agg({"article_id": list})
            .reset_index()
        )
    # Load model predictions to dataframe
    logging.debug("Converting best_preds to dataframe")
    preds = pd.DataFrame.from_dict(best_preds, orient="index").reset_index()
    preds.columns = ["customer_id", "est_article_id"]
    logging.debug(f"Head of preds\n{preds.head()}")

    # Remove duplicates (i.e. where U bought V several times)
    ground_truth["article_id"] = ground_truth["article_id"].apply(
```

```
        lambda c: list(set(c))
    )
    logging.debug("Merging ground truth with preds")
    merged = ground_truth.merge(preds)
    # Just having a look making sure everything looks good with the new thing
    logging.debug(f"Columns of merged DF: {merged.columns}")
    logging.debug(
        f"Merged has length {len(merged)} and preds had {len(preds)}."
    )

    logging.debug("Computing MAP ...")
    # Compute average precision here
    average_precision = np.zeros(merged.shape[0])
    for row_num, row in tqdm(merged.iterrows()):
        ap = 0
        truth = np.array(row.article_id)
        preds = np.array(row.est_article_id)
        for i in range(1, k + 1):
            if preds[i - 1] in truth:
                ap = ap + len(np.intersect1d(preds[:i], truth)) / i
        average_precision[row_num] = ap / min(
            k, len(row.est_article_id), len(row.article_id)
        )

    # Returns average precision for each customer instead of taking the mean,
    # which gives us MAP@k
    return average_precision
```

Main method

Finally, we make some wrappers for easier use, and define different actions that may be of interest. We requrie every call to include an `-action` flag, referencing different functions of interest. These are wrapped with a custom decorator ensuring to log any exception to the log file.

[ ]:
```
def try_except_action(action):
    """Customer that encapsulates (void) function in try-except block"""

    def wrapper():
        action_name = action.__name__
        try:
            action()
        except Exception as e:
            logging.exception(f"Exception in {action_name}!\n{e}")
        else:
            logging.deug(
                f"Successfully done with action {action_name}. Exiting ..."
                )
        finally:
            return None

    return wrapper


@try_except_action
def action_hyperparams():
    logging.debug("Starting param exploration for extended model")

    alternative_hyperparam_exploration(
```

```
        wds=(1e-4, 1e-3, 1e-1),
        embszs=(500, 100, 1000),
        lrs=(1e-2, 1e-3, 1e-4),
        dataset_path="object_storage/dataset-2022.11.26.12.04.pckl",
        baseline=False,
    )
    logging.debug("Starting hyperparameter exploration of baseline")
    alternative_hyperparam_exploration(
        wds=(1e-4, 1e-3, 1e-1),
        embszs=(500, 100, 1000),
        lrs=(1e-2, 1e-3, 1e-4),
        dataset_path="object_storage/dataset-2022.11.26.12.04.pckl",
        baseline=True,
    )


@try_except_action
def action_map():
    logging.debug("Computing MAP for model")
    data = read_dataset_obj(
        "object_storage/dataset-2022.11.26.12.04.pckl"
    )  # 200k samples
    # # Computes MAP for aggregation predictions first.
    # First: aggregation not index group-based
    res_simple_agg_full = compute_map(
        predictions_aggregator(data, by_index=False), data, 12, True
    )
    res_simple_agg_valid = compute_map(
        predictions_aggregator(data, by_index=False), data, 12, False
    )
    logging.info(
        f"With all data: {res_simple_agg_full.mean()}, with validation data:" +
        f" {res_simple_agg_valid.mean()}."
    )

    res = compute_map(predictions_aggregator(data), data, 12, True)
    logging.info(f"With all data: {res.mean()}")
    res = compute_map(predictions_aggregator(data), data, 12, False)
    logging.info(f"With validation only: {res.mean()}")

    for name, model in zip(
        ["baseline", "extended"],
        [
            "1L8VmsQ3dRedgheUIAREvLBuHOPFxwTG8",
            "1-t_jh7ajCUZRSm2EwUK4cLqAcZ2-Q6zK"
        ],
    ):
        baseline_bool = name == "baseline"
        logging.debug(f"Downloading {name} model from drive...")
        mod_weights_path = load_from_gdrive(gd_id=model)
        from time import perf_counter

        time_start = perf_counter()
        predictions_dict = topk_for_all_customers(
            mod_weights_path,
            test_data=data,
            n_customer_threshold=200,
```

```
            remove_pth_file_after_load=True,
            baseline=baseline_bool,
        )
        logging.info(
            "Time spent finding the top k for " +
            f"{'baseline' if baseline_bool else 'extended'} model: " +
            f"{perf_counter() - time_start} seconds."
        )
        # Seemed to be OS-related issues to storing this, hence the try block
        try:
            save_dataset_obj(
                predictions_dict, "object_storage/preds-dec18.pckl"
            )
        except Exception as e:
            logging.exception(
                f"Unable to store predictions dict for some reason!\n{e}"
            )
            logging.exception("Using fallback which is to paste all data:")
            logging.debug(f"\n{predictions_dict}")
        logging.debug("Finished making predictions, now computing MAP")
        average_precision = compute_map(
            predictions_dict, data, use_all_data_as_ground_truth=False
        )
        logging.info(f"Computed MAP: {average_precision.mean()}")
        average_precision = compute_map(
            predictions_dict, data, use_all_data_as_ground_truth=True
        )
        logging.info(
            f"Now with full data: Computed MAP: {average_precision.mean()}"
        )


@try_except_action
def action_savemodel():
    logging.debug("Training model to save")
    hyperparams = Hyperparameters(
        lr_rate=1e-2,
        weight_decay=1e-4,
        embedding_size=500,
        save_loss="baseline200k",
        verbose=True,
    )
    logging.debug(f"With params {hyperparams.__dict__}")
    load_dataset_and_train(
        persisted_dataset_path="object_storage/dataset-2022.11.26.12.04.pckl",
        save_model=True,
        hyperparams=hyperparams,
        baseline=True,
    )


@try_except_action
def action_fulldata():
    # Baseline with full data - a couple of different settings
    load_dataset_and_train(
        transactions_path="object_storage/transactions.pckl",
        hyperparams=Hyperparameters(
```

```
            lr_rate=1e-2,
            min_lr=1e-3,
            epochs=10,
            weight_decay=0,
            dataset_cases=1_000_000,  # 2 * 31_788_324,
            dataset_portion_negatives=0.5,
            dataset_train_portion=0.7,
            dataset_batch_size=128,
            dataset_full=True,
            bias_nodes=False,
            save_loss="NO_BIAS",
        ),
    )
    load_dataset_and_train(
        transactions_path="object_storage/transactions.pckl",
        hyperparams=Hyperparameters(
            lr_rate=1e-2,
            min_lr=1e-3,
            epochs=10,
            weight_decay=0,
            dataset_cases=1_000_000,  # 2 * 31_788_324,
            dataset_portion_negatives=0.5,
            dataset_train_portion=0.7,
            dataset_batch_size=128,
            dataset_full=True,
            bias_nodes=True,
            save_loss="WITH_BIAS",
        ),
    )
    load_dataset_and_train(
        transactions_path="object_storage/transactions.pckl",
        hyperparams=Hyperparameters(
            lr_rate=1e-4,
            min_lr=1e-3,
            epochs=10,
            weight_decay=0,
            dataset_cases=1_000_000,  # 2 * 31_788_324,
            dataset_portion_negatives=0.5,
            dataset_train_portion=0.7,
            dataset_batch_size=128,
            dataset_full=True,
            bias_nodes=False,
        ),
    )
    load_dataset_and_train(
        transactions_path="object_storage/transactions.pckl",
        hyperparams=Hyperparameters(
            lr_rate=1e-5,
            min_lr=1e-3,
            epochs=10,
            weight_decay=0,
            dataset_cases=1_000_000,  # 2 * 31_788_324,
            dataset_portion_negatives=0.5,
            dataset_train_portion=0.7,
            dataset_batch_size=128,
            dataset_full=True,
            bias_nodes=False,
```

```
        ),
    )


@try_except_action
def action_baseline():
    logging.debug("Starting training of baseline - with some weight analysis")
    df0 = pd.read_pickle("object_storage/transactions.pckl")
    # Current data loads full dataset
    data = Data_HM_Complete(
        total_cases=2 * 31_788_324,
        portion_negatives=0.5,
        df_transactions=df0,
        batch_size=128,
        train_portion=0.7,
    )
    # Baseline model with \gamma = 0.01, \lambda = 0, n_emb = 500,
    # no bias nodes, embeddings sparse, sparseAdam optimizer
    model = load_model(
        baseline=True, data=data, emb_sz=500, bias=False, sparse=True
    )
    hyperparams = Hyperparameters(
        lr_rate=0.01,
        epochs=5,
        optimizer="SparseAdam",
        lossfnc=torch.nn.MSELoss,
        weight_decay=0,
        embedding_size=500,
        save_loss="sparseAdam",
        bias_nodes=False,
        verbose=True,
        dataset_full=True,
        min_lr=0.01,  # So that LR don't decrement dynamically
    )
    logging.debug(
        f"Starting training of model with parameters {hyperparams.__dict__}"
        )
    last_validation_loss = train(
        model, data, hyperparams, baseline=True, plot_loss=True
    )
    logging.debug(
        "Model done with training (and loss plots stored to disk). " +
        f"Last validation loss {last_validation_loss}"
    )
    # Prints info on the weights to log
    for parameter in model.named_parameters():
        max_value = parameter[1].detach().numpy().max()
        logging.info(f"Parameter {parameter[0]} has max value {max_value}")

    del model  # Ensuring the weights aren't part of new model
    model = load_model(baseline=True, data=data, emb_sz=500, bias=False)
    hyperparams = Hyperparameters(
        lr_rate=1e-2,
        weight_decay=0,
        embedding_size=500,
        save_loss=True,
        verbose=True
```

```
    )
    logging.debug(
        f"Starting training of model with parameters {hyperparams.__dict__}"
        )
    last_validation_loss = train(
        model, data, hyperparams, baseline=True, plot_loss=True
    )
    logging.debug(
        "Second model done with training (and loss plots stored to disk)"
        )

    # Prints info on the weights to log
    for parameter in model.named_parameters():
        max_value = parameter[1].detach().numpy().max()
        logging.info(f"Parameter {parameter[0]} has max value {max_value}")

    logging.debug("Finished action sucessfully. Exiting ...")


@try_except_action
def action_predvalues():
    """We load the model and predictions, and re-evaluate the model
    for the best predicitons to AFAP find the model's confidence"""
    data = read_dataset_obj("object_storage/dataset-2022.11.26.12.04.pckl")
    device = "cuda" if torch.cuda.is_available() else "cpu"
    baseline = True
    model = load_model(baseline=baseline, data=data, emb_sz=500, bias=True)
    mod_weights_path = load_from_gdrive(
        gd_id="1L8VmsQ3dRedgheUIAREvLBuH0PFxwTG8"
        )
    model.load_state_dict(
        torch.load(mod_weights_path, map_location=torch.device(device))
    )
    os.remove(mod_weights_path)
    with open("object_storage/preds-dec6.pckl", "rb") as f:
        best_predictions = pickle.load(f)
    prediction_values = {}
    logging.debug("All stuff has been loaded sucessfully.")

    model.eval()
    for customer_id, article_ids in best_predictions.items():
        article_tensor = torch.IntTensor(article_ids[0])  # Flatten list
        customer_tensor = torch.IntTensor([customer_id]).expand(
            article_tensor.shape[0]
        )
        prediction = model(customer_tensor, article_tensor)
        # Make 12-length list
        prediction_values[customer_id] = prediction.detach().numpy()
    logging.debug(
        "Done with finding predictions. Creating plot of confidence."
        )

    averages = []
    maxes = []
    for predictions in prediction_values.values():
        averages.append(predictions.mean())
        maxes.append(predictions.max())
```

```
    plt.plot(averages, "o", color="grey", markersize=1, label="Average value")
    plt.plot(maxes, "ro", markersize=1, label="Max value")
    plt.legend()
    plt.xlabel("Customer index ID")
    plt.ylabel("Confidence (0%-100%)")
    save_path = "average_prediction_confidence.png"
    plt.savefig(save_path)
    logging.debug(f"Saved plot to {save_path}")


if __name__ == "__main__":

    logging.basicConfig(
        filename="ssh_run.log",
        # encoding="utf-8", # Not present in Python 3.8 so commented out
        level=logging.DEBUG, # Change this to not get all debug messages
        format="%(asctime)s %(levelname)s %(message)s",
    )
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument("-action", help="What type of action to run...")
    args = parser.parse_args()
    action = args.action
    logging.debug(f"4-shh-compatible called with action flag: {action}")
    possible_actions = (
        "(hyperparams, map, savemodel, baseline, fulldata, predvalues)"
    )

    if action == "hyperparams":
        action_hyperparams()
    elif action == "map":
        action_map()
    elif action == "savemodel":
        action_savemodel()
    elif action == "baseline":
        action_baseline()
    elif action == "fulldata":
        action_fulldata()
    elif action == "predvalues":
        action_predvalues()
    elif action is None:
        print(f"Please provide -action flag when running {possible_actions}")
    else:
        print(f"Action not recognized. Possible actions: {possible_actions}")
```