ISAE-SUPAERO

Master's Degree Thesis

Engineering Scalable Recommender Systems: From Model Selection to E-Commerce Implementation

Supervisors

Valentin GORCE

Thomas TAYLOR

Candidate

Soël MEGDOUD

SEPTEMBER 2024

Summary

This report outlines the work conducted during my final internship at Headmind Partners. The primary focus of the project was to design and optimize recommendation pipelines for large-scale inference, with a specific application to e-commerce.

The report provides a comprehensive overview of the development process, including a detailed literature review of both traditional and state-of-the-art recommendation systems like DLRM and Wide&Deep models.

Two main pipeline architectures are explored:

- A personalized recommendation pipeline that recommends products to customers based on their preferences and historical interactions.
- An item-similarity pipeline designed to recommend similar products during the browsing experience on e-commerce platforms.

These architectures have demonstrated their effectiveness in handling an extensive dataset, enabling them to deliver recommendations in near real-time. They follow a multi-stage approach (Retrieval, Scoring, and Ordering) and adhere to MLOps best practices, such as feature management using feature stores and inline feature processing.

Cutting-edge tools, such as Nvidia's Merlin along NVTabular and Triton Inference Server, are employed to enhance performance.

In addition, the report presents a methodology for generating labeled datasets from CRM (Customer Relationship Management software) data, which can be used to train scoring models offline.

Through experimentation, DLRM from Meta achieved the best performance, while simpler models such as Random Forest showed competitive results. To further validate the system's performance in real-world conditions, a demonstrator simulating an e-commerce website was developed.

This report details the methodologies employed, the challenges faced, the solutions implemented and the results obtained.

Table of Contents

List of Tables VII

List of Figures VIII

1 Introduction 1

1.1 About the firm . . . 1

1.2 About the team . . . 1

1.3 Context . . . 1

1.3.1 Objectives . . . 1

1.3.2 H&M CRM dataset . . . 2

2 Recommender System Models 3

2.1 Recommender systems taxonomy . . . 3

2.2 Matrix Factorization . . . 4

2.3 Factorization Machines . . . 5

2.4 DLRM . . . 6

2.5 Wide & Deep Model . . . 8

2.6 Two-Tower Model . . . 9

2.7 Other algorithms . . . 10

2.8 Performance Comparison . . . 10

2.8.1 Evaluation Metrics . . . 11

2.8.2 Results . . . 12

3 Recommendation and development framework 13

3.1 Engineering challenges . . . 13

3.2 Recommendation framework . . . 13

3.2.1 Merlin Models . . . 13

3.2.2 Nvtabular . . . 14

3.2.3 Triton Inference Server . . . 14

3.2.4 Feast Feature Store . . . 14

3.2.5 FAISS: Facebook AI Similarity Search . . . 15

3.3 Development environment . . . 15
3.3.1 Setting up with Docker and JupyterLab . . . 15
3.3.2 Deployment with Triton Inference Server . . . 16

4 Personalised recommendation pipeline **17**
4.1 Multi-stages pipeline architecture . . . 17
4.2 Retrieval . . . 17
4.2.1 The Two-Tower trick for fast retrieval . . . 17
4.2.2 Features fetching and pre-processing . . . 19
4.3 Scoring and ordering . . . 19
4.4 Pipeline overview . . . 20

5 Item similarity pipeline **21**
5.1 Pipeline overview . . . 21
5.2 Item embedding with Fashion CLIP . . . 21
5.2.1 FashionCLIP . . . 21
5.2.2 Item embeddings . . . 22

6 Feature engineering **23**
6.1 Feature engineering . . . 23
6.1.1 Customer features . . . 23
6.1.2 Item features . . . 23
6.2 Prepossessing . . . 24
6.3 Creation of a labeled dataset . . . 25

7 Results **29**
7.1 Benchmark of scoring models . . . 29
7.1.1 Metrics . . . 29
7.1.2 Results . . . 30
7.2 Features importance analysis . . . 32
7.3 Inference times . . . 33

8 Demonstrator **35**
8.1 Objective . . . 35
8.2 Demonstrator architecture . . . 35
8.2.1 Frontend (Streamlit) . . . 36
8.2.2 Backend (FastAPI) . . . 36
8.2.3 Triton Inference Server (Docker) . . . 36
8.3 Screenshots . . . 36

9 Conclusion 39
9.1 Key Learnings and Outcomes 39
9.2 Critiques and Areas for Improvement 39

Bibliography 41

List of Tables

7.1 Performance of scoring models 30

List of Figures

2.1 Models taxonomy . . . 3
2.2 Matrix Factorization problem formulation . . . 4
2.3 Factorization Machines problem formulation . . . 5
2.4 DLRM architecture . . . 7
2.5 Wide & Deep model architecture . . . 8
2.6 Two-Tower model architecture . . . 9
2.7 Algorithm ranking based on Borda count at cutoff length 10 . . . 12

3.1 Benchmark of librairies handling dataframe . . . 14
3.2 Docker container architecture . . . 16

4.1 Multi-stages recommendation pipeline . . . 17
4.2 Retrieval module with Faiss similarity search . . . 18
4.3 Features fetching and processing pipelines . . . 19
4.4 Scoring pipeline . . . 20
4.5 Personalised recommendation pipeline overview . . . 20

5.1 Item similarity pipeline . . . 21
5.2 Item embedding with FashionCLIP . . . 22

6.1 In-line features processing . . . 24
6.2 Purchase history splitting for data labelling . . . 25
6.3 Recency of customers in the generated dataset . . . 28
6.4 Number of purchases per item in the last 30 days in the generated dataset . . . 28

7.1 Random Forest feature importances . . . 32
7.2 Triton inference times . . . 33

8.1 Demonstrator architecture . . . 35
8.2 Login page screenshot . . . 37
8.3 Home page screenshot . . . 37

8.4 Product page screenshot 38
8.5 Item similarity recommendations in product page screenshot 38

Chapter 1

Introduction

1.1 About the firm

Headmind Partners is a consulting firm gathering 1500 associates split into 3 business units, Digital, Cybersecurity and AI. The digital BU is the most important with around 1000 people. Then this is the cybersecurity BU with around 450 people and finally the AI BU with approximately 30 people. Headmind Partners offices are located only in two countries, France and Belgium, and the firm represents around 130M€ of turnover.

1.2 About the team

I joined the AI team composed of 30 permanent AI consultants. The AI BU is built around 3 departments which are: NLP, Computer Vision and AIOps. I was integrated into the AIOps team but the teams are not independent, and I have worked with experts in computer vision and NLP as well. The internship was organized as follows: I started my internship on the 2nd of April and I spent the first 4 months of my internship working full time on my research topic, then I joined an AIOps team at an energy customer's to work part-time on RAG solutions more closely aligned with NLP.

1.3 Context

1.3.1 Objectives

Within the AIOps department, I was granted the autonomy to select a project that would enhance the firm's expertise in Know Your Customer (KYC) practices. KYC encompasses a broad spectrum of customer knowledge, including marketing, sales, and customer service.

I opted to focus on the development of recommender systems. The project had several objectives:

- Develop a recommender system using state-of-the-art algorithms applicable to various e-commerce scenarios.

- Master different frameworks and tools to construct highly optimized inference pipelines adhering to AIOps best practices.

- Create a dynamic demonstrator utilizing these inference pipelines to simulate a realistic use case.

1.3.2 H&M CRM dataset

To achieve these goals, I employed a dataset from the H&M Personalized Fashion Recommendations Kaggle competition. [1]

This dataset is derived from H&M's Customer Relationship Management (CRM) system. A CRM is a software collecting, organizing, and managing data from multiple sources related to a company's business activities, typically encompassing sales, marketing, and customer service.

The H&M dataset is one of the most extensive and realistic datasets available for e-commerce use cases, comprising 35 GB of data across four different databases:

- **Customers:** Basic information such as name, age, gender, location, and occupation for 1.4 million H&M customers.
- **Articles:** Product information including title, category, and color for over 500,000 H&M products.
- **Images:** Photos of each product in the articles dataset.
- **Transactions:** Detailed records of past transactions per customer, including products bought and prices paid.

This rich datasets is a goldmine for developing business-related solutions such as predictive analytics, customer segmentation, personalized marketing, churn prediction, or recommender systems. It allowed me to create production-like pipelines to handle huge datasets and provide operationally effective recommendations.

For H&M, I developed two distinct pipelines:

- **Personalized Items Recommendation:** Tailored recommendations based on individual customer history.
- **Item Similarity Recommendation:** Recommendations based on the similarity between items.

Chapter 2

Recommender System Models

2.1 Recommender systems taxonomy

Recommender systems encompass many different types of algorithm. It is useful to distinguish between the different categories. Historically, recommender systems have been classified according to the types of data they use. There are 2 main categories of data:

1. **Product descriptions**: Product descriptions features and attributes Examples: product tags, descriptions, categories etc.

2. **Customer-product interactions**: Information on customer-product interactions Examples: interactions type (purchase, add to basket, click), date, location etc.

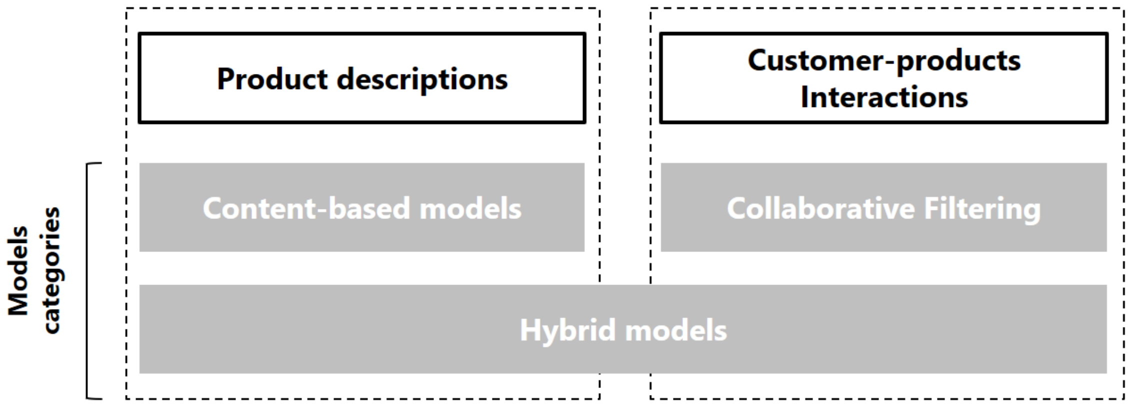


Figure 2.1: Models taxonomy

Models based exclusively on product features are called content-based and models based exclusively on customer-product interactions are called collaborative filtering. Initially, content-based systems were implemented, using KNN algorithms on product features (such as the 'similar products' tab on Amazon).

In the 2010s, Collaborative filtering stole the spotlight when the Matrix Factorization algorithm won a Kaggle competition organised by Netflix, which offered $1m to anyone who could build a recommendation system significantly superior to Netflix's own. [2]

Then, with Deep Learning, many so-called hybrid algorithms were democratised, incorporating more contextual data.

Let's review some popular recommendation algorithms.

2.2 Matrix Factorization

Matrix Factorization [3] is the most popular algorithm of Collaborative Filtering. It aims to predict user preferences by modelling customer-product interactions in a matrix and decomposing the matrix into two lower-dimensional matrices, capturing the latent factors that influence user-item interactions.

Problem Formulation

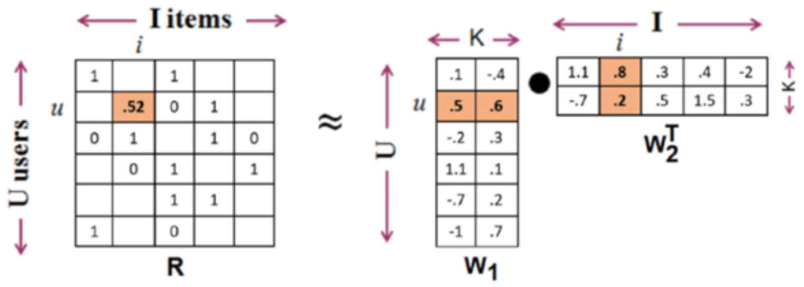


Figure 2.2: Matrix Factorization problem formulation

Given a user-item interaction matrix R of size $m \times n$, where m is the number of users and n is the number of items, the goal is to factorize R into two matrices: P and Q. P is an $m \times k$ matrix representing the user-specific latent factors, and Q is a $n \times k$ matrix representing the item-specific latent factors.The predicted interaction between user u and item i is given by the dot product of the corresponding vectors in P and Q:

$$\hat{R}_{ui} = P_u \cdot Q_i^T$$

where P_u is the u-th row of matrix P, and Q_i is the i-th row of matrix Q.

Objective Function

The Matrix Factorization approach aims to minimize the difference between the observed interactions and the predicted interactions. This can be formalized using the following loss function:

$$\min_{P,Q} \sum_{(u,i)\in\mathcal{K}} \left(R_{ui} - P_u \cdot Q_i^T\right)^2 + \lambda \left(\|P_u\|^2 + \|Q_i\|^2\right)$$

where $\mathcal{K}$ represents the set of user-item pairs for which interactions are known, and λ is a regularization parameter to prevent overfitting.

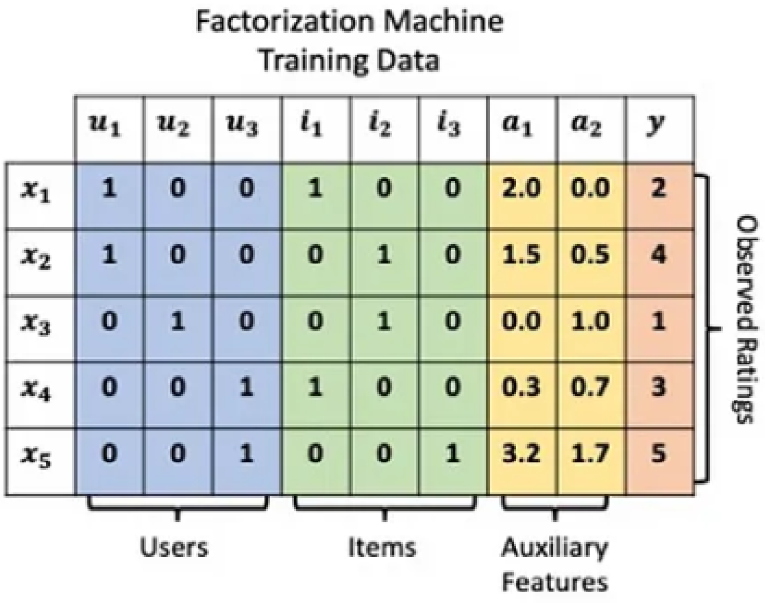
The optimization of the objective function is typically performed using methods like Alternating Least Squares (ALS). ALS alternates between fixing P and solving for Q, and vice versa.

Factorization is particularly effective in capturing the underlying structure of user-item interactions. It can handle large, sparse datasets and has been successfully applied in various domains. Moreover, extensions of MF, such as Non-negative Matrix Factorization (NMF) and SVD++, allow incorporating additional information like implicit feedback and side information. [4]

2.3 Factorization Machines

Factorization Machines (FM) [5] are supervised learning algorithms aiming to grasp higher-order interactions between variables. Their main advantage over Matrix Factorization is that it is possible to add features other than product-customer interactions (such as product features or contextual information). They are particularly useful in scenarios where data is sparse, such as in recommendation systems and click-through rate (CTR) prediction.

Problem Formulation


Factorization Machine Training Data

	u_1	u_2	u_3	i_1	i_2	i_3	a_1	a_2	y
x_1	1	0	0	1	0	0	2.0	0.0	2
x_2	1	0	0	0	1	0	1.5	0.5	4
x_3	0	1	0	0	1	0	0.0	1.0	1
x_4	0	0	1	1	0	0	0.3	0.7	3
x_5	0	0	1	0	0	1	3.2	1.7	5

Observed Ratings
Users
Items
Auxiliary Features

Figure 2.3: Factorization Machines problem formulation

Factorization Machines consider the recommendation problem as a regression problem with very sparse features. Given an input vector $\mathbf{x} \in \mathbb{R}^n$, where n is the number of features, the prediction $\hat{y}$ is given by:

$$\hat{y}(\mathbf{x}) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \langle \mathbf{v}_i, \mathbf{v}_j \rangle x_i x_j$$

where:

- w_0 is the global bias,
- w_i is the weight of the i-th feature,

- $\mathbf{v}_i \in \mathbb{R}^k$ is the k-dimensional latent vector associated with the i-th feature,
- $\langle \mathbf{v}_i, \mathbf{v}_j \rangle$ denotes the dot product between vectors $\mathbf{v}_i$ and $\mathbf{v}_j$,
- x_i and x_j are the values of the i-th and j-th features, respectively.

This formulation captures not only the linear interactions but also the pairwise feature interactions, making FM capable of modeling complex relationships within the data.

Objective Function

The Factorization Machines algorithm seeks to minimize the prediction error by optimizing the parameters w_0, w_i, and $\mathbf{v}_i$. This can be expressed using a regularized loss function:

$$\min_{w_0,\mathbf{w},\mathbf{V}} \sum_{(y,\mathbf{x})\in\mathcal{D}} (y - \hat{y}(\mathbf{x}))^2 + \lambda_w \sum_{i=1}^{n} w_i^2 + \lambda_v \sum_{i=1}^{n} \|\mathbf{v}_i\|^2$$

where:

- $\mathcal{D}$ represents the dataset consisting of pairs $(y, \mathbf{x})$,
- λ_w and λ_v are regularization parameters to prevent overfitting.

The ability of Factorization Machines to handle sparse datasets efficiently makes them particularly powerful in recommendation systems. They have been extended to more complex models such as neural Factorization Machines (NFM).

However, one problem to keep in mind is that a labelled dataset is necessary, which is not always the case in recommendation.

2.4 DLRM

The Deep Learning Recommendation Model (DLRM) [6] is a state-of-the-art deep learning-based architecture designed by Meta for personalized recommendation systems. DLRM is optimized for handling sparse and dense features in large-scale recommendation tasks, particularly in social media and e-commerce environments. This algorithm uses the flexibility of deep learning algorithms and reproduces the concept of pairwise interaction of Factorization Machines.

Problem Formulation

The DLRM separates the input features into two categories:

- **Sparse Features**: These are categorical variables, such as user IDs, item IDs, and other categorical attributes that can take on a large number of distinct values.
- **Dense Features**: These are continuous numerical variables, such as item price, or other measurable quantities.

DLRM effectively combines these features to produce a prediction score that estimates the probability of user interaction with an item.

Model Architecture

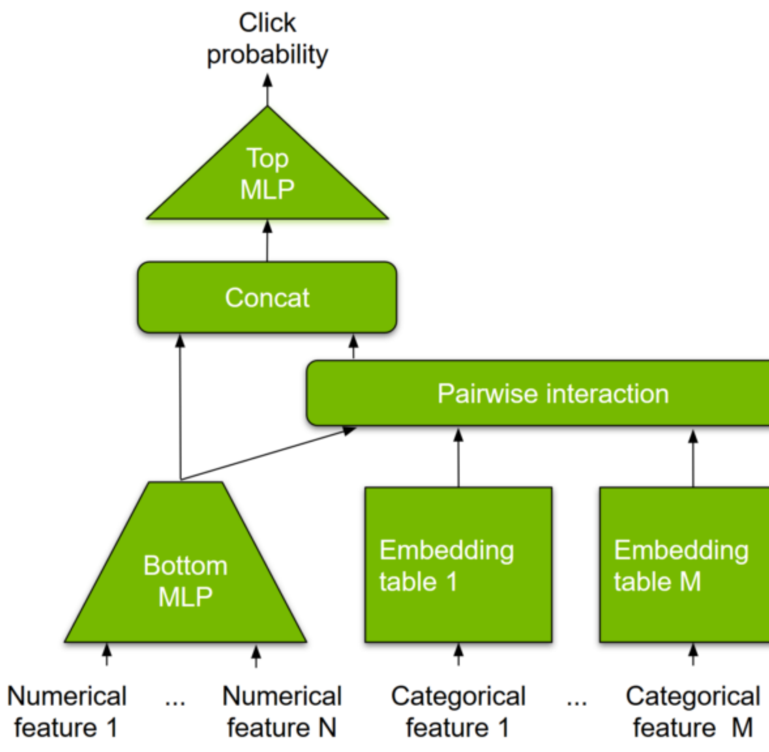


Figure 2.4: DLRM architecture

The architecture of DLRM consists of three main components:

- **Embedding Layers**:
  - **Sparse Feature Embedding**: Categorical features are first transformed into dense vector representations using embedding layers. Each unique category (e.g., a specific user or item ID) is mapped to a fixed-size embedding vector, generalizing the concept of latent factors used in Matrix Factorization.
  - **Dense Feature Processing**: Dense features are passed through a series of fully connected layers (Bottom MLP) to generate dense representations of the same size.
- **Interaction Layer**:
  - **Dot-Product Interaction**: The core of DLRM is its interaction layer, where the dense and sparse feature embeddings interact, following the intuition for handling sparse data provided in Factorization Machines. The embeddings for sparse features are combined with dense features using a dot-product operation, which captures pairwise interactions between different features.
  - **Concatenation**: The outputs of the interaction layer are concatenated with the dense features and passed through the subsequent layers of the model.
- **Top MLP**: The concatenated features are fed into a series of fully connected layers (the top MLP), and fed into a sigmoid function to give a probability.

The DLRM is typically trained using a binary cross-entropy loss function, suitable for binary classification tasks like click-through rate prediction or conversion prediction.

One of the significant strengths of DLRM is its ability to scale effectively to handle massive datasets full of sparse features. Meta uses model parallelism to mitigate the memory bottleneck produced by the embeddings.

2.5 Wide & Deep Model

The Wide & Deep Learning model [7] is a recommendation algorithm designed by Google to click-through-rate prediction. It was initially developed for app recommendations in Google Play. The goal is to achieve both memorization of historical patterns and generalization to unseen instances. To do so, like the DLRM model, the Wide&Deep model also calculates interactions between features but more explicitly.

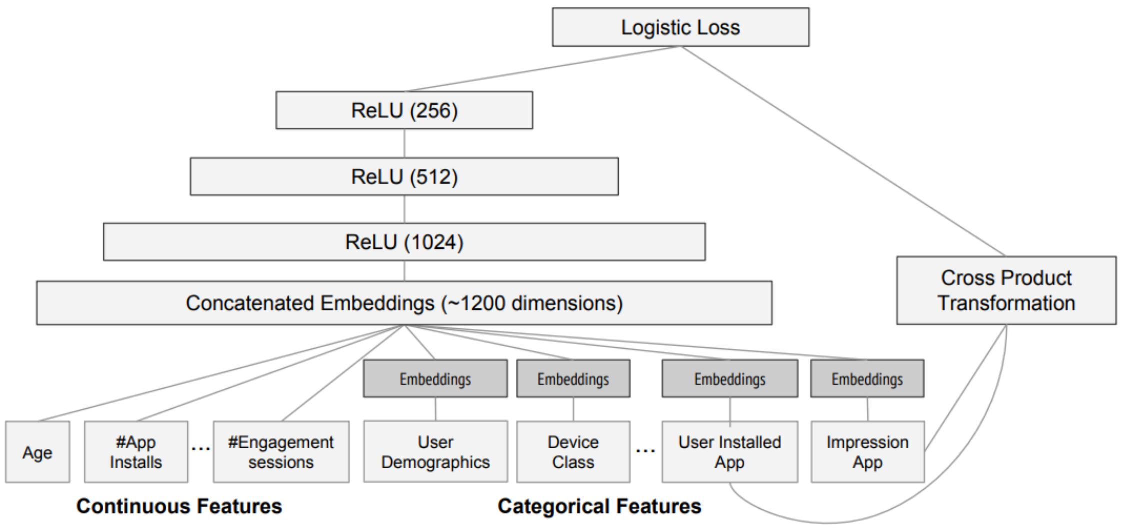


Figure 2.5: Wide & Deep model architecture

Model Architecture

Wide Component

The Wide component is a generalized linear model, formulated as:

$$y = w^T x + b$$

where x is the feature vector composed of raw and transformed features, w represents the learned weights, and b is the bias term.

The key feature of the Wide component is its ability to capture specific, manually-defined interactions between features through cross-product transformations. For example, a cross-product transformation for binary features (e.g., **“AND(gender=female, language=en)”)** is 1 if and only if the constituent features (**“gender=female”** and **“language=en”) are all 1, and 0 otherwise.**

Such interactions are particularly useful for learning fixed, memorized patterns within the data, where certain combinations of features are important for prediction.

Deep Component

Like the DLRM model, each categorical feature is first mapped into a dense, low-dimensional embedding vector. These embedding vectors are then processed through several fully-connected layers.

Final Prediction

The wide component and deep component are then fed to one common logistic loss function for joint training. For a logistic regression problem, the prediction is given by:

$$P(Y = 1|x) = \sigma \left(w_{\text{wide}}^T [x, \phi(x)] + w_{\text{deep}}^T a^{(l_f)} + b \right)$$

where Y is the binary class label, $\sigma(\cdot)$ is the sigmoid function, and x represents the original input features. The term $\phi(x)$ refers to the cross-product transformations of the features, and b is the bias term. The vector w_{wide} contains the weights from the wide component, while w_{deep} are the weights applied to the final activations $a^{(l_f)}$ from the deep component.

2.6 Two-Tower Model

The Two-Tower model [8] is a deep learning-based architecture commonly used in recommendation systems, particularly for large-scale retrieval tasks. It is designed to efficiently model interactions between users and items by encoding them separately in two "towers" or neural networks, which are then combined to generate a similarity score.

Problem Formulation

The Two-Tower model match users with items by learning separate embeddings for users and items in two separate neural networks.

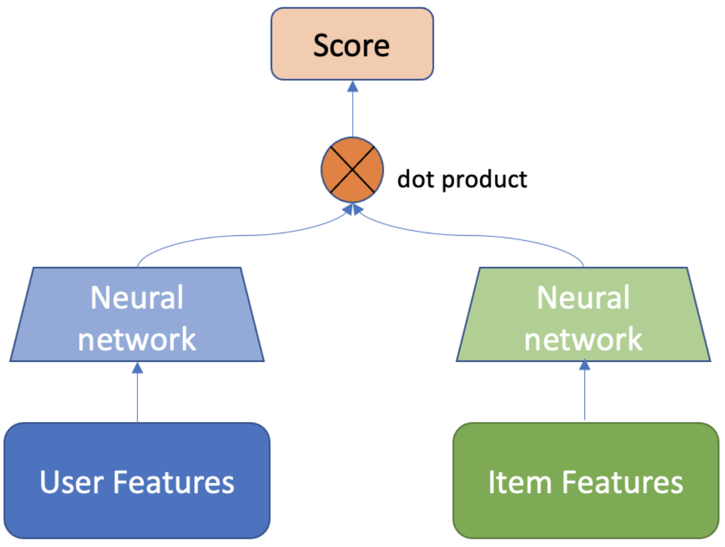


Figure 2.6: Two-Tower model architecture

Given a user u and an item i, the model consists of two towers:

- **User Tower**: Encodes user's features (e.g., demographic information, interaction history) into a dense embedding vector $\mathbf{e}_u$.
- **Item Tower**: Encodes item's features (e.g., item metadata, content information) into a dense embedding vector $\mathbf{e}_i$.

The user and item embeddings are then combined to compute a similarity score, typically using the dot product:

$$\hat{y}(u, i) = \mathbf{e}_u \cdot \mathbf{e}_i$$

This score $\hat{y}(u, i)$ reflects the likelihood that the user u will interact with the item i.
Each tower is trained to produce embeddings in a shared latent space where similar users and items are close together. The overall architecture is highly scalable, items embeddings can be computed in advance and the dot product operation is computationally efficient.

The Two-Tower model is trained by optimizing a loss function that encourages the model to bring similar users and items closer together in the embedding space while pushing dissimilar pairs apart. A commonly used loss function is the contrastive loss, often in the form of a variant of the cross-entropy loss for binary classification.

The Two-Tower model is very popular for large-scale retrieval tasks due to its scalability and efficiency. It is particularly useful in scenarios where the recommendation system must handle millions of users and items. However, this algorithm is not very well suited to accurately recommending the best articles.

2.7 Other algorithms

These algorithms represent only a proportion of all the algorithms used in recommender systems, but they are among the most popular.

There are other categories of algorithms known as session-based or sequenced-based. These algorithms focus on the task of predicting a user's next action (purchase, click), based on a sequence of recent interactions. These algorithms are particularly used on social networks to offer content that dynamically adapts to the user's session. However, as part of my intership, I focus on classic recommendation systems, with an emphasis on operational performance.

2.8 Performance comparison

Over the last few years, there have been many efforts to try and advance the state of the art of recommendation systems (notably pushed by Meta, Google and Netflix). However, there is very little neutral comparison of these different algorithms.

Most studies show that there is no algorithm that systematically wins. The results vary greatly depending on the datasets and metrics used. It turns out that models developed several years ago often outperform the most recent neural models.

In Vito Walter Anelli et al. study [9], several algorithms have been tested across 3 diffent datasets :

- Non-personalized baseline: Popularity-based recommendation (MostPop).
- Neighborhood-based and simple graph-based models: UserKNN, ItemKNN, RP3.
- Linear models: SLIM, EASE.
- Matrix factorization models: BPRMF, MF2020, iALS.
- Neural models: NeuMF, MultiVAE.

However, I did not find a study evaluating the latest Deep Learning models like Wide&Deep and DLRM.

In this study, a Borda countranked voting scheme has been implemented to aggregate observed ranking for 3 different datasets on nDCG and recall metrics.

2.8.1 Evaluation Metrics

nDCG

nDCG (Normalized Discounted Cumulative Gain) is metric commonly used for retrieval tasks. nDCG evaluates the ranking quality of the recommendations by comparing the relevance of items in a ranked list to an ideal ranking.

$$DCG_N = \sum_{i=1}^{N} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

where rel_i represents the relevance score of the item at position i, and N is the cutoff length (e.g., top 10). To obtain the normalized DCG (nDCG), we divide DCG by the Ideal DCG (IDCG), which corresponds to the best possible ordering of items:

$$nDCG_N = \frac{DCG_N}{IDCG_N}$$

nDCG provides a measure of how well relevant items are ranked, giving higher scores to algorithms that rank relevant items earlier in the recommendation list.

It is important to understand that it is necessary to know the real relevance of the items to calculate the nDCG. In the datasets used (e.g. movie rating dataset), users explicitly give scores to items, which serves as a basis for calculating the nDCG. However, in the majority of cases, there is no information on the rating of items by users, which can make this metric inaccessible.

The recommendation task becomes a scoring task since the aim becomes to predict the relevance of items for users.

Recall

Recall is a metric that evaluates the recommendation system's ability to retrieve relevant items within the top N recommended results. It is defined as the ratio between the number of relevant items found in the top N recommendations and the total number of relevant items:

$$Recall_N = \frac{|RelevantItems \cap TopNRecommendedItems|}{|RelevantItems|}$$

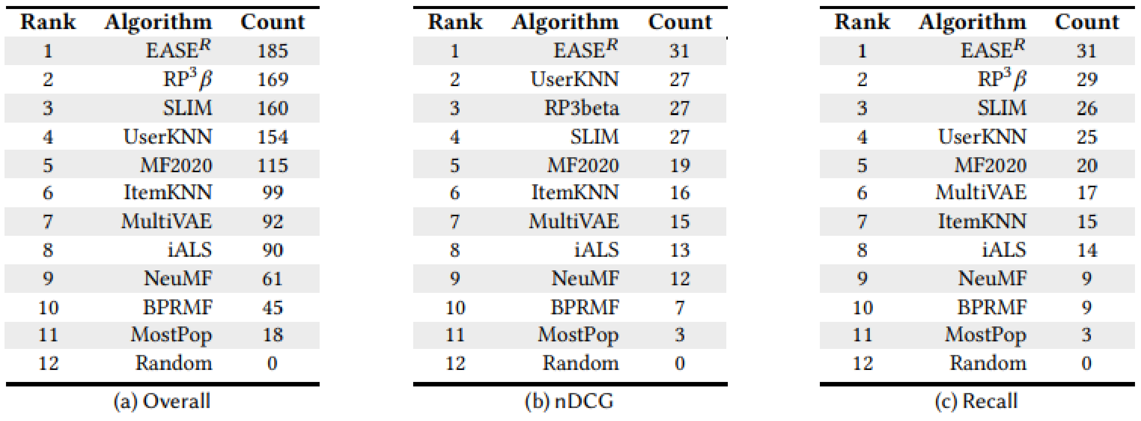
This metric is particularly useful in scenarios where we aim to maximize the number of relevant items retrieved in a limited set of recommendations.

Borda Count

The Borda Count is a ranked voting method used to aggregate multiple rankings into a single ranking. Each algorithm is assigned a score based on its position in the ranking across multiple datasets. These points are summed across all datasets, and the algorithms are then ranked based on their total score.

This voting scheme helps to provide a more robust evaluation by combining the performance of algorithms across multiple datasets, offering an aggregated view of their effectiveness.

2.8.2 Results



Rank	Algorithm	Count
1	EASE^R	185
2	RP$^3\beta$	169
3	SLIM	160
4	UserKNN	154
5	MF2020	115
6	ItemKNN	99
7	MultiVAE	92
8	iALS	90
9	NeuMF	61
10	BPRMF	45
11	MostPop	18
12	Random	0

(a) Overall

Rank	Algorithm	Count
1	EASE^R	31
2	UserKNN	27
3	RP3beta	27
4	SLIM	27
5	MF2020	19
6	ItemKNN	16
7	MultiVAE	15
8	iALS	13
9	NeuMF	12
10	BPRMF	7
11	MostPop	3
12	Random	0

(b) nDCG

Rank	Algorithm	Count
1	EASE^R	31
2	RP$^3\beta$	29
3	SLIM	26
4	UserKNN	25
5	MF2020	20
6	MultiVAE	17
7	ItemKNN	15
8	iALS	14
9	NeuMF	9
10	BPRMF	9
11	MostPop	3
12	Random	0

(c) Recall

Figure 2.7: Algorithm ranking based on Borda count at cutoff length 10

This study shows that linear regression and nearest-neighbour models perform consistently well, whereas Deep Learning models rank poorly.

However, recent models seem to be widely used in industry. Some engineering reasons can explain this:

- **Deep Learning models are more hybrid**, taking more diverse data types in input whereas Matrix Factorization model, for example, only consider user-item interactions.
- **Cold start users and items**: It is difficult to deal with new users or items. Factorization Machine and Matrix Factorization models require to be re-trained when new users or items are registered to add new rows or columns to the model.

For example, Netflix never used the million-dollar code from its competition. In a blog post following the competition, the Netflix engineering team said that the increase in accuracy on the winning improvements "did not seem to justify the engineering effort needed to bring them into a production environment".

Chapter 3

Recommendation and development framework

3.1 Engineering challenges

When it comes to training and deploying large-scale recommender systems, there are several key challenges, including:

- **Huge Datasets:** Commercial recommender systems often require processing terabytes of data, where ETL (Extract, Transform, Load) and preprocessing steps can take much longer than model training.
- **Extensive Repeated Experimentation:** The entire process of data ETL, feature engineering, training, and evaluation must be repeated multiple times across various model architectures, necessitating significant computational resources and time.
- **Huge Embedding Tables:** Handling categorical variables, such as user and item IDs, can require large embedding tables that are memory bandwidth-constrained requiring model parallelism.
- **Real-time Inference:** Serving high-throughput, low-latency inferences for thousands of user-item pairs in real-time for online engines.

I wanted to use the best operational tools to develop effective recommendation pipelines using industry best practice and leveraging Headmind's Nvidia H100 GPUs. To achieve this, I used the Nvidia Merlin framework. These open-sources librairies are not only leading references in recommendation systems and operational performance, but also synergise perfectly with Nvidia GPUs, using CUDA to accelerate computation time.

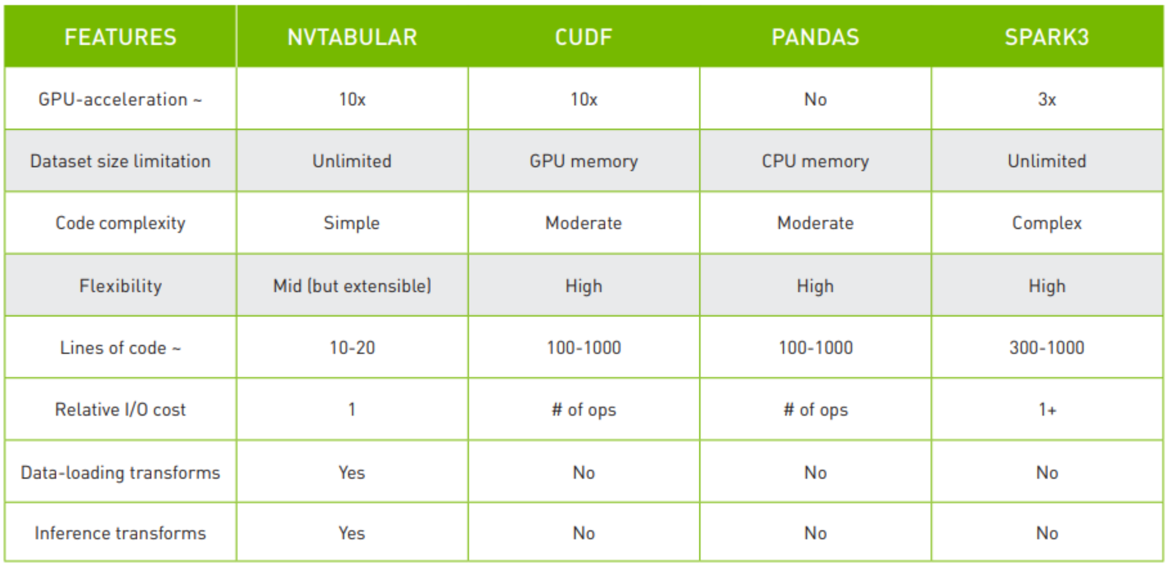
3.2 Recommendation framework

3.2.1 Merlin Models

Nvidia Merlin Models is designed specifically for building high-performance recommendation systems. It includes a collection of state-of-the-art recommender system implementations, like Wide&Deep or DLRM, optimised for GPUs. For very large scale datasets (Terrabytes), there is also HugeCTR which is a C++ training framework that supports multi-GPU and multi-node training, with model-parallel and data-parallel scaling.

3.2.2 Nvtabular

Nvtabular is a GPU-accelerated library for data preprocessing and feature engineering, which is a crucial step in the recommendation pipeline. It handles large-scale datasets efficiently and prepares them for model training. Nvtabular enables efficient data transformation operations such as filtering, encoding, feature engineering, and normalization, thereby preparing the data for training recommendation models. Nvidia claims that it can achieve up to 10X speedup compared to optimized CPU-based approaches.



FEATURES	NVTABULAR	CUDF	PANDAS	SPARK3
GPU-acceleration ~	10x	10x	No	3x
Dataset size limitation	Unlimited	GPU memory	CPU memory	Unlimited
Code complexity	Simple	Moderate	Moderate	Complex
Flexibility	Mid (but extensible)	High	High	High
Lines of code ~	10-20	100-1000	100-1000	300-1000
Relative I/O cost	1	# of ops	# of ops	1+
Data-loading transforms	Yes	No	No	No
Inference transforms	Yes	No	No	No

Figure 3.1: Benchmark of librairies handling dataframe

3.2.3 Triton Inference Server

Triton Inference Server is a machine learning model deployment platform that allows serving inference models at scale efficiently. The key advantages of Triton include:

- **Multi-Framework Support:** Compatible with various machine learning frameworks such as TensorFlow, PyTorch, ONNX, and TensorRT.
- **Performance Optimization:** Utilizes GPU capabilities to deliver low-latency, high-throughput inferences through dynamic batching, concurrent execution, batch inference etc.
- **Cloud agnostic:** Running pipelines in Triton allows to be independent from cloud providers and easily deploy inference infrastructures on any cloud.

3.2.4 Feast Feature Store

Feast (Feature Store) is an open-source feature store that plays a critical role in managing features in both a context of development and production. It serves as a centralized hub for managing, sharing, and serving machine learning features to models. Integrating Feast into the pipeline offers several key benefits:

- **Centralized Feature Management:** Feast allows for the systematic and consistent management of features across development and production environments, reducing the risk of data leakage and inconsistencies.
- **Real-Time Feature Serving:** In operational contexts where inference times are important, Feast supports low-latency retrieval of features, enabling the model to respond to user interactions in near-real time.
- **Scalability:** Feast is designed to scale, allowing it to handle vast amounts of feature data.

Incorporating Feast into the recommendation pipeline provide a robust, scalable, and consistent feature management solution.

3.2.5 FAISS: Facebook AI Similarity Search

FAISS (Facebook AI Similarity Search) is an open-source library developed by Meta, designed for efficient large-scale similarity search. It is particularly suited for retrieval tasks, where the goal is to compare embeddings to identify similar items within large datasets. Integrating FAISS into the pipeline offers several key advantages:

- **Fast and Efficient Search:** FAISS is optimized for performing similarity searches on high-dimensional vector sets. It significantly accelerates the retrieval process while maintaining high accuracy.
- **Handling Large Databases:** FAISS is built to manage extremely large volumes of data, containing millions or even billions of vectors leveraging GPU partitioning.

FAISS fits well for comparing user embedding with item embeddings in the Two-Tower model. By integrating FAISS into the recommendation pipeline, it is possible to significantly enhance the system's retrieval performance.

3.3 Development environment

3.3.1 Setting up with Docker and JupyterLab

To facilitate the development process, I used Docker to pull a NVIDIA Merlin TensorFlow container, which includes several key components to simplify the installation like (NVTabular, Merlin and Triton Inference Server).

I used a JupyterLab server within the container to access the development environment. Using Docker volumes, I mounted my data into the container, ensuring data persistence across different sessions. Deploying JupyterLab within the Docker container serves several important purposes in addition to providing all the dependencies :

- **Reproducibility and Isolation**: By running JupyterLab inside a Docker container, we ensure that the development environment is isolated and reproducible. The container can be moved or transferred to the cloud easily.

- **Collaboration**: JupyterLab can be accessed via a web browser allowing multiple users access making it easy to share the development environment with other developers.

In summary, deploying JupyterLab within the Docker container offers a flexible, reproducible, and collaborative environment. It proved to be an effective setup for development.

3.3.2 Deployment with Triton Inference Server

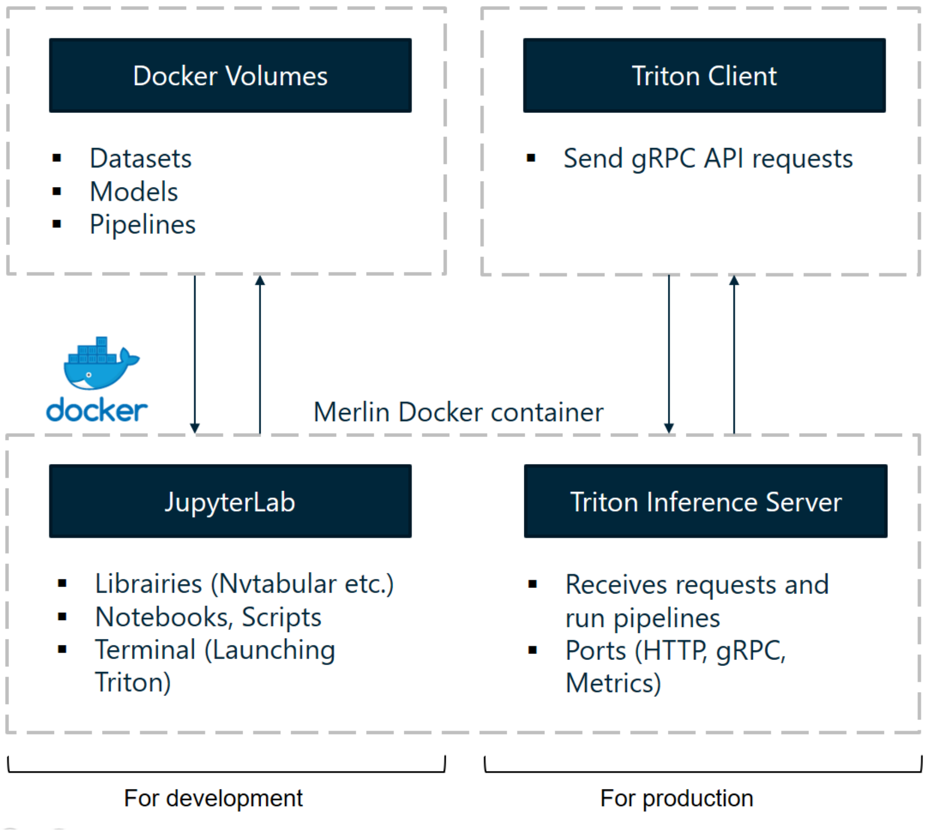


Figure 3.2: Docker container architecture

The recommendation pipelines developed using the Merlin framework can be converted in Triton Inference Server files. To run these pipelines, I used a Triton Inference Server within the Docker container opened with carefully chosen ports to the outside of the container. This setup allows to send gRPC requests to the inference server from outside the container and receive the responses.

The advantage of having the inference server inside the container is that it simplifies the deployment in production. Using Triton, the entire pipeline is cloud-agnostic and can be deployed easily on any cloud platform. Additionally, having the inference server in a container means that it can be managed as an independent microservice and optimized, for example with Kubernetes, in a production and load-balancing context.

Chapter 4

Personalised recommendation pipeline

4.1 Multi-stages pipeline architecture

A personalised recommendation pipeline consists of returning a list of items relevant to a given user.

However, as with the H&M dataset, there can be a huge number of potential products. In order to speed up inference time, in-line recommendation pipelines are often made up of several stages, including a rapid retrieval stage to reduce the number of candidates, followed by a more precise scoring stage to determine the best products to recommend. The figure below shows the common architecture of a multi-stage recommendation pipeline.

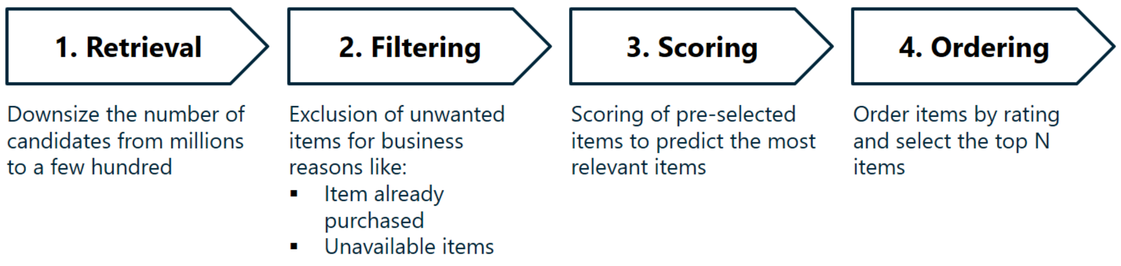

1. Retrieval
Downsize the number of candidates from millions to a few hundred
2. Filtering
Exclusion of unwanted items for business reasons like:
Item already purchased
Unavailable items
3. Scoring
Scoring of pre-selected items to predict the most relevant items
4. Ordering
Order items by rating and select the top N items

Figure 4.1: Multi-stages recommendation pipeline

4.2 Retrieval

4.2.1 The Two-Tower trick for fast retrieval

For the retrieval part, I choose the widely used Two-Tower model. The aim is to get a sub-selection of items as fast as possible. The advantage of the Two-Tower model is that item embeddings resulting from the Item-Tower can be computed off-line. There is no need to to recalculate embeddings for all items for each inference. The only thing left to do in-line is to compute the user's embedding by passing his features into the Query-Tower (or User-Tower) and to compute the dot product with all the pre-compute item embeddings.

However, the process can still be improved. By registering the items' embeddings in a more clever way, we can use approximate vector search algorithms that directly return N vectors among those most similar to the user's embedding vector in a very short amount of time. This can be done by FAISS. [10]

Faiss Algorithm Overview

- **Index Building:** The item embeddings dataset is first partitioned into clusters using k-means clustering for example. Within each cluster, vectors are further compressed using Product Quantization, which reduces the dimensionality of vectors by splitting them into sub-vectors and quantizing each sub-vector separately.

- **Querying:** During the search, the query vector (user embedding) is first matched against clusters to identify the nearest clusters. Only the vectors in these clusters are considered for the search, significantly reducing the number of distance calculations. FAISS then retrieves the N vectors from the selected clusters with the smallest distances to the query vector.

The trade-off between accuracy and speed is controlled by parameters such as the number of clusters and the level of quantization. By adjusting these parameters, FAISS can achieve near-exact results with a fraction of the computational cost.

This algorithmic approach allows FAISS to efficiently search through the user embeddings dataset making it ideal to quickly returns the best output from the Two-Tower model.

Retrieval module with Two-Tower and Faiss

This figure details the stages of the retrieval module with Faiss during inference. It takes as input the processed features of the user requested in the query and returns a list of candidate item IDs.

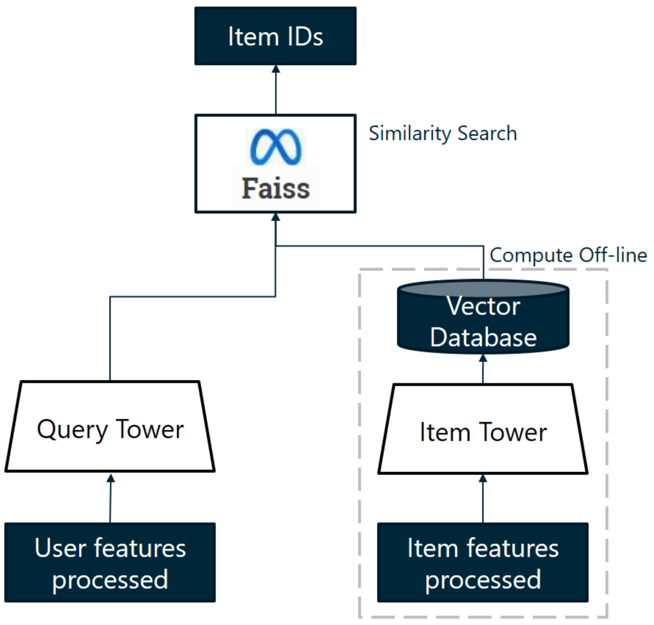


Figure 4.2: Retrieval module with Faiss similarity search

4.2.2 Features fetching and pre-processing

The aim of a recommendation system is to give it a user id and have it return a list of item ids. So, using the requested user id, we need to retrieve the user features from a database to give it to the Two-Tower model. I used Feast feature store for this purpose.

I saved the raw data (customer and product features) in the Feast feature registry and during inference, Feast takes care of retrieving the features associated with the user id. As this data is raw, it needs to be processed so that it can be given as input to the Query-Tower of the Two-Tower model. Most customer and product features are categorical and need to be associated with integers. This process is detailed later in the section 6.2.

In short, this pre-processing stage is used to transform the raw attributes of the dataset into processed numerical data similar to the data used to train the models. To do this, I used Nvtabular, which allows me to save workflows of pre-processing operations during development and deploy them in production. This data retrieval and processing pipeline is also applied to the candidate items from the similarity search.

Ultimately, the aim of the retrieval pipeline is to have processed the user features and those of the candidate items to give them to the scoring algorithm. The figure below illustrates the retrieval pipeline as a whole.

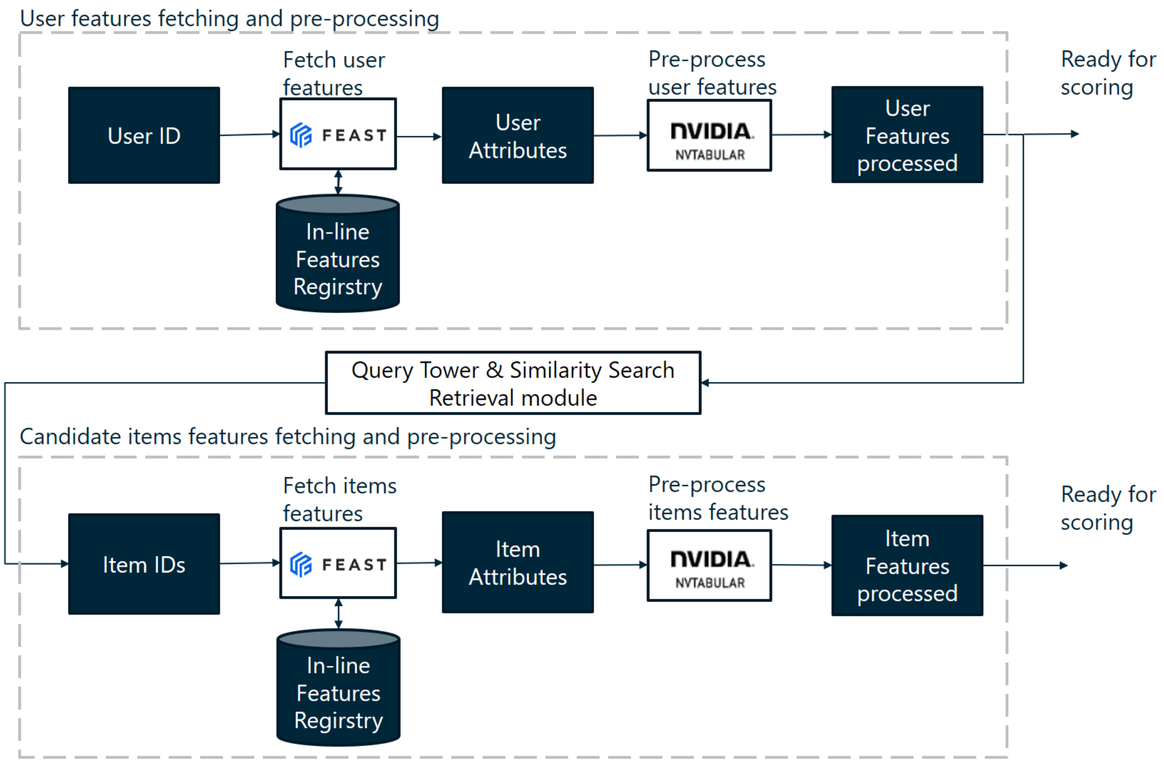

User features fetching and pre-processing
User ID
Fetch user features
FEAST
In-line Features Regirstry
User Attributes
Pre-process user features
NVIDIA NVTABULAR
User Features processed
Ready for scoring
Query Tower & Similarity Search Retrieval module
Candidate items features fetching and pre-processing
Item IDs
Fetch items features
FEAST
In-line Features Regirstry
Item Attributes
Pre-process items features
NVIDIA NVTABULAR
Item Features processed
Ready for scoring

Figure 4.3: Features fetching and processing pipelines

4.3 Scoring and ordering

Once the features have been processed and retrieved from the user and all the candidate items, all we have to do is concatenate them and give them as input to our scoring model. This finer-grained scoring model is often a supervised model such as DRLM or Wide&Deep. This model gives scores for each item, so all we have to do is choose the N-best items, order them and return their ids, and that's it, the personalised recommendation pipeline is complete.

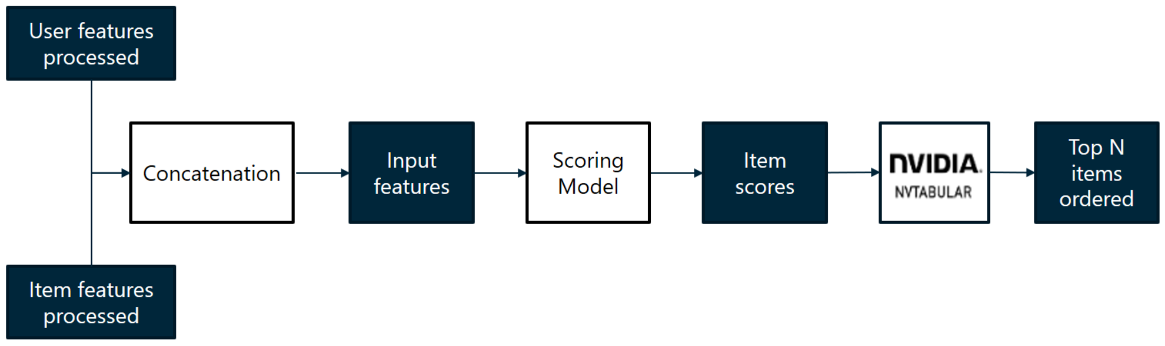


Figure 4.4: Scoring pipeline

4.4 Pipeline overview

By combining the retrieval and scoring pipelines, we can get an overall picture of the personalised recommendation pipeline. The following figure is an overview of the pipeline and its main components.

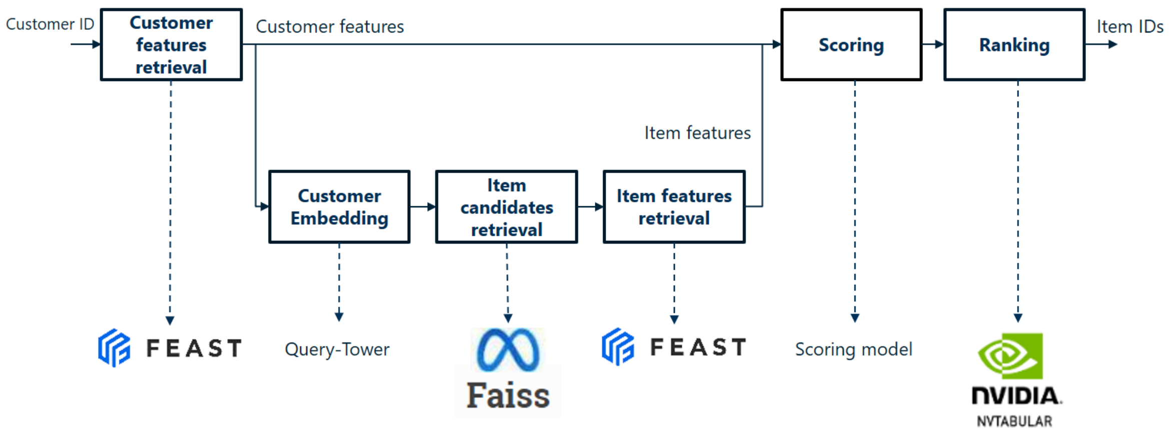


Figure 4.5: Personalised recommendation pipeline overview

Chapter 5

Item similarity pipeline

5.1 Pipeline overview

The aim of an item similarity pipeline is to suggest products that are similar to the product consulted by the user. This option is very popular on e-commerce websites, making it easier for users to browse products they like. This pipeline is much simpler than the previous one. The aim is to return a list of items that are similar to a given one, which is an content-based recommender system.

To do this, we simply retrieve the item's embedding and compare it with the embeddings of other items, with Faiss for example, and return a list of similar items. The overall structure of the pipeline is detailed in this image.

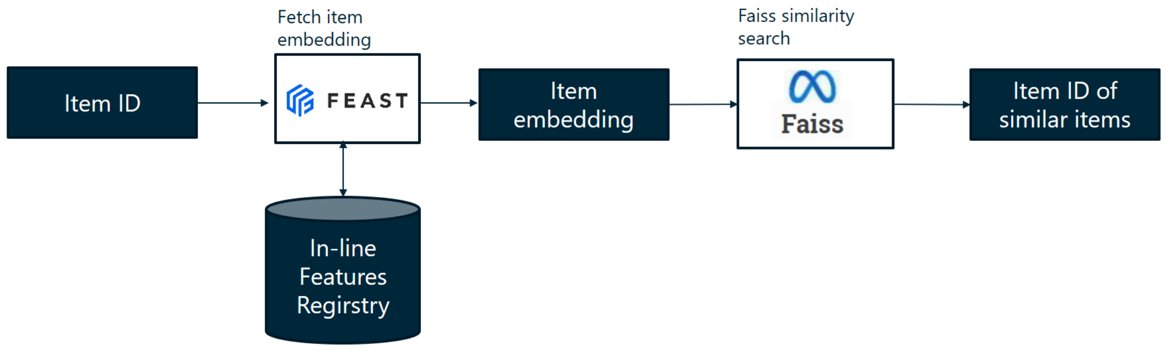


Figure 5.1: Item similarity pipeline

5.2 Item embedding with Fashion CLIP

To achieve a good level of similarity between the articles recommended, it is important to have precise enough embeddings. We can therefore use an embedding model different from the Two-Tower model, which better grasp product features.

5.2.1 FashionCLIP

I chose to use Fashion CLIP to create the fine-grained embeddings. FashionCLIP is a domain-specific adaptation of the CLIP model, fine-tuned to produce general product representations for fashion-related items. Leveraging the pre-trained `ViT-B/32` architecture released by OpenAI, FashionCLIP was fine-tuned on a large fashion dataset to enhance zero-shot performance across various benchmarks.

A Vision Transformer (ViT-B/32) serves as the image encoder, while a masked self-attention Transformer functions as the text encoder. These components are trained jointly via a contrastive loss to maximize the similarity between image and text pairs in the fashion domain. The input data consists of standard product images with white backgrounds and accompanying text descriptions, typically containing a combination of highlights (e.g., "stripes", "long sleeves") and concise product descriptions (e.g., "80s styled t-shirt"). [11]

5.2.2 Item embeddings

The H&M dataset contains all the product features as well as photos of the products on a white background which is perfect for Fasion-CLIP. However, I decided not to use the visual features to generate the embeddings and only concatenate the product characteristics and embed them using the text encoder. The H&M product descriptions are sufficiently detailed and standardised to generate high quality embeddings. The items are therefore embedded in vectors of dimension 512.

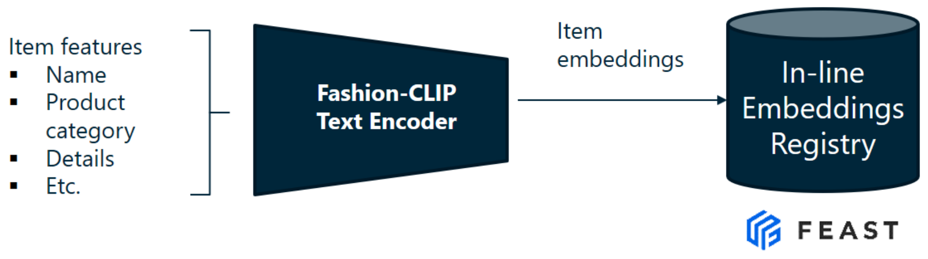


Figure 5.2: Item embedding with FashionCLIP

Chapter 6

Feature engineering

6.1 Feature engineering

There are two categories of features: **item features** and **customer features**.

Among the customer features, the H&M dataset provides a lot of features, including attributes such as `age`, `postal_code`, and `club_member_status`.

Similarly, among item features there are features such as `product_code`, `product_name`, and `product_type`.

In addition to these predefined features, I introduced several new features to enrich the dataset:

6.1.1 Customer features

For customers: I included the classic RFM (Recency, Frequency, Monetary) features commonly used in marketing segmentation:

- `Recency`: Time since the customer's last purchase.
- `Frequency`: The average frequency of purchases made by the customer.
- `Amount`: The average amount spent by the customer on purchases.

Moreover, I added features to capture personal preferences and buying habits, such as:

- `popular_product_type`: The product type most frequently purchased by the customer.
- `last_product_type`: The type of product bought in the customer's most recent transaction.

These features were further extended to other product features.

6.1.2 Item features

For items: I have build features that reflect their popularity:

- `count_Nd_purchased`: The number of times a product was purchased in the last N days.

- `Time_Weighted_Purchased`: The number of times the product was purchased, weighted by the purchase date, giving more importance to recent transactions.

6.2 Prepossessing

Most of these features are categorical encoded with strings, it is necessary to convert them to numbers so that they can be interpreted by the models. To do this I used Nvtabular and applied classical prepossessing step for recommender systems.

- Categorical features have been categorized in integers and categories have been saved to allow the same prepocessing during inference.
- Numerical features have been normalized to floats to keep an order relationship.

These processing operations are necessary for model training, but with Nvtabular, these pipelines can be saved to apply the same processing steps to the raw features from the feature store during inference.

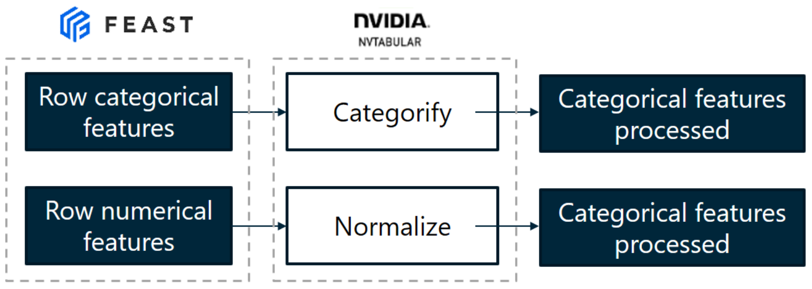


Figure 6.1: In-line features processing

For some categorical features, there may be many possible categories (especially for product descriptions). Then, it is possible to set frequency thresholds to categorise categories that appear too infrequently into a dedicated category. However, this parameter had no effect on my results.

6.3 Creation of a labeled dataset

The majority of algorithms used for scoring are supervised algorithms, which means you need to have labeled dataset. H&M's data comes from their CRM and is therefore the customer's purchase history. In order to train the scoring models, we need examples of negative customer-product interactions which means products that customers have not purchased. Unfortunately, we don't have these information. In the case of Web Analytics and CLick-Through-Rate Prediction for example, we may collect information about users through the website and know that they have seen a product without clicking on it.

A popular approach in the literature to generate a labelled dataset is to use random sampling. Traditional approach typically begins by discarding "cold-start" customers who have very few interactions. Then, the customer base is divided into folds for cross-validation. For each test fold, half of the events from customer profiles are randomly removed to construct the test set. The remaining half, along with other folds, is used to train the models.

However, some benchmarks [12] highlighted that there is a risk that some interactions in the training set may contain implicit signals from the test period. This is particularly problematic because the model can inadvertently learn from future information, inflating its performance metrics unrealistically.

To create a training dataset, I chose to chronologically split the dataset. For the positive examples, I selected the last 20 days of purchase history, with all purchases made during this period serving as our positive interactions. All prior data beyond this 20-day period was used as training data. This chronological split is crucial to avoid any information from the future leaking into the training set.

All features such as `recency` or `count_Nd_purchased` have been calculated from the split date to avoid biasing the dataset. By isolating the target period, we ensure that the model is evaluated as if it were being used in a real-world scenario at a given point in time.

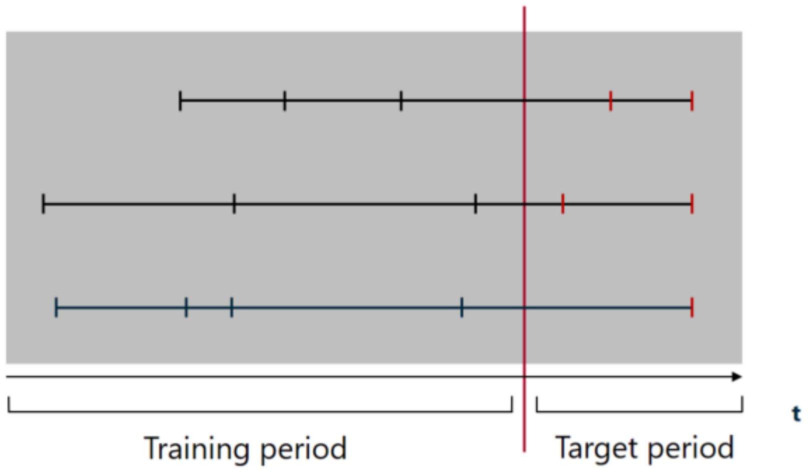


Figure 6.2: Purchase history splitting for data labelling

Since there are no explicit negative interactions (i.e., products viewed but not purchased), a sampling method was employed to generate them. Negative examples were generated by sampling random customers and items with a distribution indexed on recent customers and very popular products to avoid relying solely on popularity. Without choosing a distribution of this kind, it was far too easy for the models to identify false interactions, as many of them referred to customers who had not bought in a long time or to old-fashioned items. This approach also ensures an equal balance of positive and negative samples to help the model to focus on the products purchased (which reflects the user's real interests). The figure below details my process for building the dataset.

Algorithm 1 Procedure for generating a test set with negative sampling

- $\mathcal{D}$: Dataset of customer-product interactions over time.
- T: Time window (last 20 days) for labelling.
- n_{pos} and n_{neg}: Number of positive and negatives interactions in the testset.

Step 1: Positive Sample Extraction

- Extract all customer-product pairs (c, p) where a transaction occurred in the last T days.
- Label these pairs as positive: $y = 1$ and store these pairs as $\mathcal{P}_{pos}$.

Step 2: Negative Sample Generation

- While $n_{neg} < n_{pos}$
  - Randomly sample products by batch, using a distribution indexed on their number of purchases weighted by their recency (to target recently popular items).
  - Randomly sample customers by batch, using a distribution indexed on their recent activity (to focus on active buyers).
  - Ensure that the sampled customer-product pairs do not exist in $\mathcal{P}_{pos}$.
  - Label these customer-product pairs as negative: $y = 0$ and store these pairs as $\mathcal{P}_{neg}$.

Step 3: Final dataset

- Combine both sets: $\mathcal{T} = \mathcal{P}_{pos} \cup \mathcal{P}_{neg}$ and shuffle it to ensure a random order of positive and negative samples.
- Merge customer and product features from external datasets to each observation.

In this way, the dataset has a strong distribution of regular customers and recently purchased items. It is an attempt to force the model to find underlying relationships about customer tastes and not just recommend items because they are popular.

Here is the distribution of customer recency in the generated dataset and the number of purchases per item made in the 30 days preceding the target period.

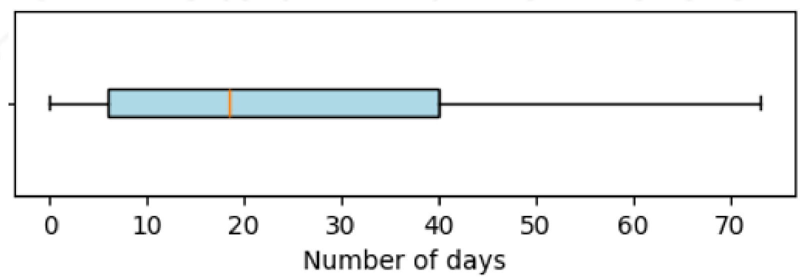


Figure 6.3: Recency of customers in the generated dataset

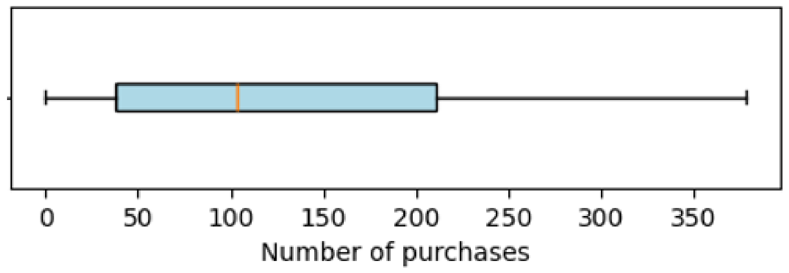


Figure 6.4: Number of purchases per item in the last 30 days in the generated dataset

Finally, 50% of customers in this training dataset made purchases between 7 and 40 days prior to the labelling period and about 75% of the items in this dataset were purchased more than 50 times in the last month before the labelling period.

Chapter 7

Results

7.1 Benchmark of scoring models

7.1.1 Metrics

As item ratings or relevancy are not provided in CRM data, it is unfortunately not possible to calculate the nDCG. In the context, I chose to evaluate models on 3 metrics: AUC (Area Under the Curve), Precision, and Recall.

Precision and Recall

Precision and Recall are standard metrics in classification tasks. *Precision* measures the proportion of true positives (TP, this means recommended items that were actually purchased during the target period)) out of all predicted positives ($TP + FP$), making it important for minimizing false positives:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall, on the other hand, captures the proportion of true positives out of all actual positives ($TP + FN$), which is crucial for ensuring that relevant items are not missed:

$$\text{Recall} = \frac{TP}{TP + FN}$$

In recommendation systems, where we can only suggest a limited number of items, Precision is preferred over Recall as high precision ensures that the recommended items are relevant.

However, precision and recall alone do not fully capture the performance of models designed for *scoring* tasks, where the goal is not just to classify items but to rank them in order of relevance. This is where AUC becomes particularly useful.

AUC (Area Under the ROC Curve)

AUC is a metric that evaluates a model's ability to discriminate between positive and negative classes across all possible classification thresholds. The AUC is derived from the Receiver Operating Characteristic (ROC) curve, which plots the True Positive

Rate (TPR) against the False Positive Rate (FPR) at different thresholds. The True Positive Rate is equivalent to Recall:

$$\text{TPR} = \frac{TP}{TP + FN}$$

and the False Positive Rate is defined as:

$$\text{FPR} = \frac{FP}{FP + TN}$$

The AUC score is computed as the area under the ROC curve, which ranges from 0 to 1, with a higher value indicating better ranking ability.

The advantage of the AUC for comparing scoring models

Unlike precision and recall, which depend on a fixed threshold, AUC provides a more comprehensive measure of a model's ability to separate classes across all thresholds. This is critical in recommendation systems, where the goal is to rank items by relevance scores rather than classifying them as relevant or not. A model with a high AUC ranks relevant items higher in the list, even if it misclassifies some individual cases.

Thus, AUC is often a chosen metric to select scoring-based models as it captures the model's ability to prioritize relevant items.

7.1.2 Results

With the generated dataset, I have been able to test different scoring models. I have chosen Random Forest as my baseline because it has been the best-performing model in the H&M Kaggle competition. These models are fast and well suited to categorical data. I also tried a gradient boosting algorithm.

Regarding Deep Learning models, I tried Meta's DLRM and Google's Wide&Deep, which are considered to be the state of the art. I also tested DCN, which is an older variant of both. For comparison, I'm also testing a simple Multi Layer Perceptron which does not calculate feature interactions.

The results of these models on the testset for AUC, Precision and Recall are displayed in the table below.

Model	AUC	Precision	Recall
Random Forest	0.8050	**0.8326**	0.7488
Gradient Boosting	0.7981	0.8226	0.7440
DLRM	**0.8794**	0.8236	0.7100
Wide&Deep	0.8485	0.8104	0.6416
DCN	0.8286	0.7569	0.6814
MLP	0.8195	0.7072	**0.7543**

Table 7.1: Performance of scoring models

An important result is that **DLRM outperforms the other models in AUC, which makes it better for scoring**. It explains my choice in my personalised recommendation pipeline. Unsurprisingly, tree-based models perform very well in terms of both precision and recall, which supports the results of the Kaggle competition. However, DLRM and Wide&Deep perform very close to Random Forest in precision.

There is a big gap between old DCN and new DLRM and Wide&Deep models for all metrics. The simple MLP model has poorer precision than the other models meaning that calculating interactions between features really improves performance.

7.2 Features importance analysis

Tree models such as Random Forest allow to display the feature importances used for prediction. Here are the feature importances obtained.

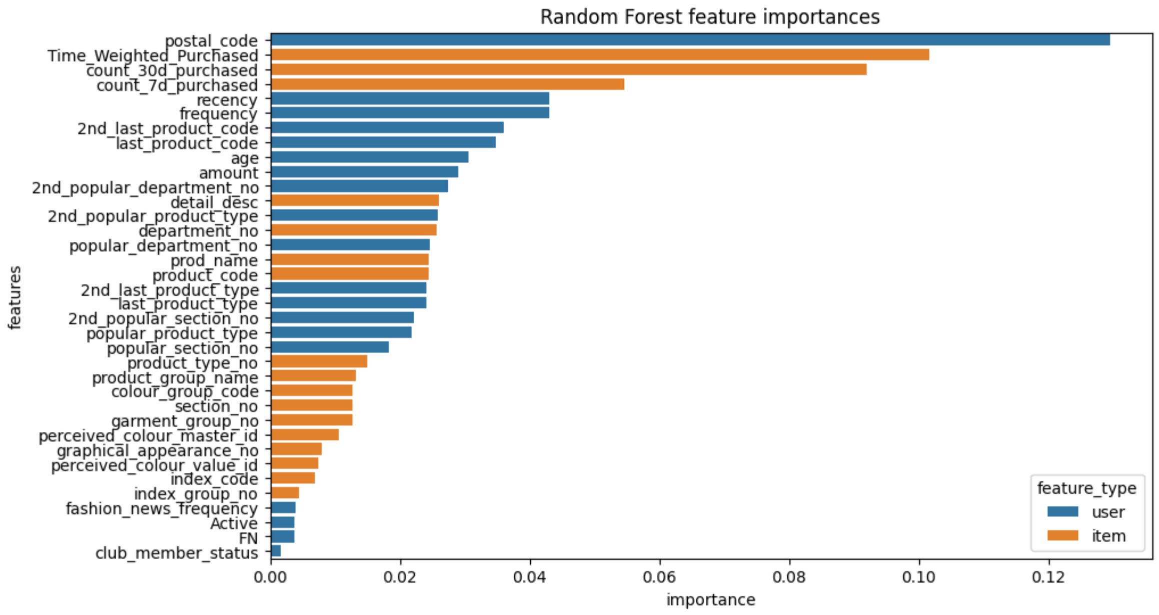


Figure 7.1: Random Forest feature importances

- The first surprising insight is the postal code being the most important feature. The winners of the Kaggle competition also highlighted this point. The most probable explanation is that this CRM data comes from physical stores. As these stores have different stocks or products on display, the model learned which items were unlikely to be available in certain geographical areas.

- Then, the 3 most important features for prediction are product popularity-based features. This seems consistent with the fashion industry. People like trendy and popular products and then these products go out of fashion. It may also be a bias due to the fact that physical customers see more the products displayed in stores.

- Recency and Frequency from RFM features are the most important user features to make predictions, regular and compulsive shoppers can buy trendy products more easily.

- Surprisingly, information about last purchases are even more important than features describing their most frequent purchase.

- The least useful features include the product attributes, particularly those related to appearance or material.

People are more likely to buy clothes because they are very popular than because of their features.

7.3 Inference times

I have implemented my pipeline with a DLRM for the scoring, according to its performance. Now, it is important to verify the operational performance of the pipeline. Lots of effort have been made to make it as fast as possible using retrieval, similarity search and so on.

The inference time is about 1.4s for the customized recommendation pipeline and 1s on average for asynchronous inference. The inference time is about 0.3s for the item-similarity pipeline. This is still quite long for in-line inference. Using Triton logs, we can look at the detail of inference times

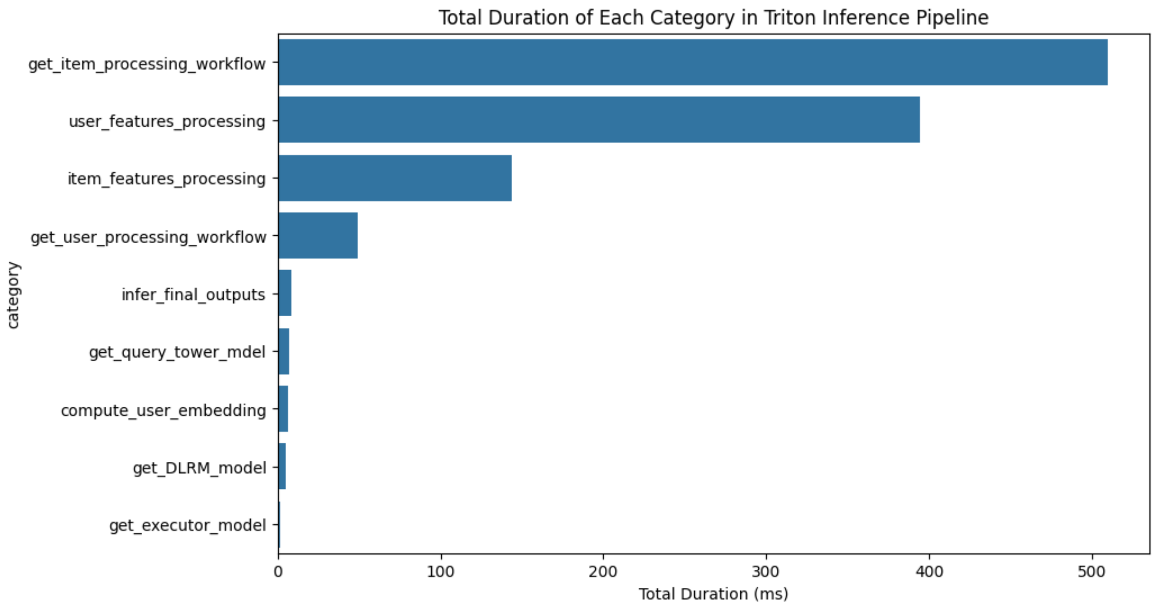


Figure 7.2: Triton inference times

As illustrated in the figure, the most time-consuming part of the pipeline is getting the Nvtabular preprocessing workflows and processing the raw data from the feature store. These steps alone account for approximately 1.2 seconds, or 80% of the total inference time, which is significant. On the other hand, retrieving the models (Query-Tower and DLRM) is extremely fast, as they are preloaded into RAM when the server starts. The inference times for these models are also very low.

A potential optimization could be to store the preprocessed data directly in the feature store, rather than storing raw data and processing them during inference. This change could drastically reduce the total inference time, potentially saving as much as 1 second.

However, this approach raises a question: **why store raw data and process it in-line during inference?**

The answer depends on the use case. Real-time feature transformation allows for greater flexibility and operational scalability. For example, if new customers register or new products are introduced, their row data can be directly integrated into the feature store, and the pipeline will continue to function seamlessly. This pipeline is plug-and-play on the CRM system. This is particularly usefull for use cases requiring recommendations for new customers or new products without reprocessing the entire dataset. Moreover, in-line processing simplifies development, as the database remains static and only grows in size. This allows for easy creation of new Nvtabular workflows and models to update the production pipeline without requiring changes to the underlying data infrastructure.

In contrast, storing pre-processed data would necessitate running the feature transformation pipeline before every data update and loading the processed data into the feature store, complicating maintenance.

In summary, there is a trade-off between faster inference times and operational complexity. For use cases where recommendations are generated offline (e.g., weekly product recommendations), sacrificing some inference speed for a simpler, more maintainable system could be a more practical choice.

Chapter 8

Demonstrator

8.1 Objective

Assessing the quality of a recommendation system on a testset is a good, but it is also appreciated when you can see it in live. In this way, I designed a dynamic demonstrator so that I could try it the recommender system as if my pipelines were running in production. The aim of the demonstrator is to reproduce the pages of an e-commerce site like H&M's, where you can log in with your account, receive personalised product recommendations and browse by having similar products recommended to you.

8.2 Demonstrator architecture

The architecture of the demonstrator is composed of three main components: a frontend built using Streamlit, a backend based on FastAPI, and a recommendation system which is the Triton Inference Server in a container.

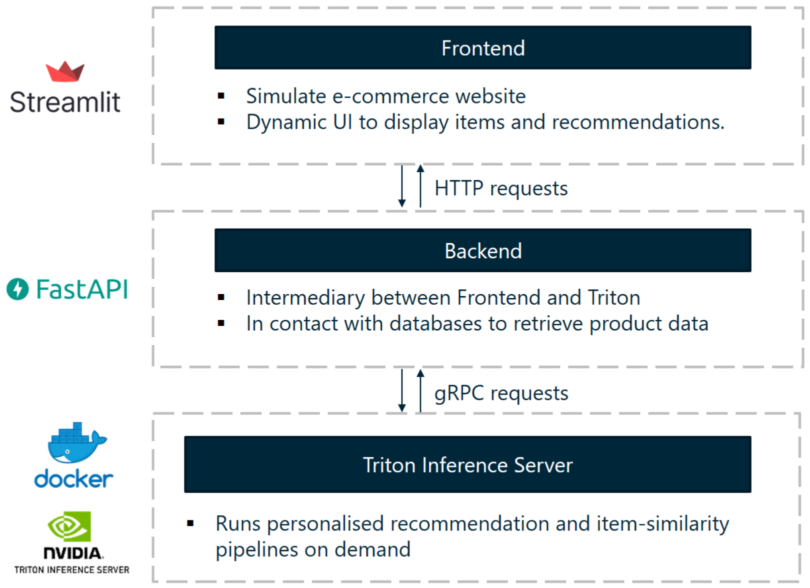

Streamlit
Frontend
Simulate e-commerce website
Dynamic UI to display items and recommendations.
HTTP requests
FastAPI
Backend
Intermediary between Frontend and Triton
In contact with databases to retrieve product data
gRPC requests
docker
NVIDIA
TRITON INFERENCE SERVER
Triton Inference Server
Runs personalised recommendation and item-similarity pipelines on demand

Figure 8.1: Demonstrator architecture

This architecture ensures that product recommendations are provided in real-time, while maintaining flexibility and ease of integration between the different components. The following section describes each component and their interactions.

8.2.1 Frontend (Streamlit)

The frontend simulates the interface of a typical e-commerce platform, where users can browse and search for products. The key functionalities of this module include:

- Simulating an interactive e-commerce environment
- Displaying personalized product recommendations generated by the recommendation system.
- Sending HTTP requests to the backend (FastAPI) to retrieve recommendation results based on user interactions and preferences.

Streamlit serves as the primary user interface for this system, allowing users to interact with the platform as if they were shopping on a real e-commerce website.

8.2.2 Backend (FastAPI)

The FastAPI backend acts as the technical backbone of the system. It is a bridge between the frontend and the Triton Inference Server, handling requests and communication across the system. The backend module is responsible for:

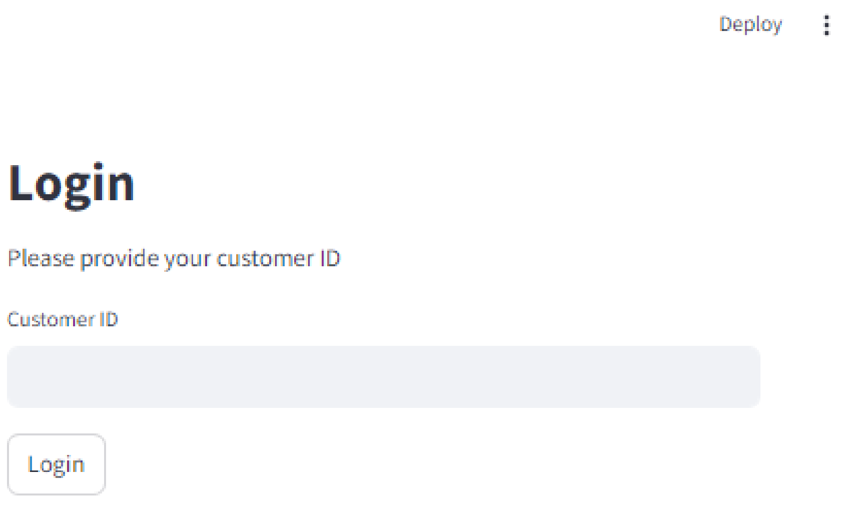
- Receiving HTTP requests from the frontend and transforming them into gRPC requests to query the Triton Inference Server.
- Retrieving product images and descriptions in the database for display in the frontend.

8.2.3 Triton Inference Server (Docker)

The recommendation system is powered by Triton Inference Server, which is hosted in a Docker container. This setup enables scalable and high-performance inference serving. The Triton Inference Server is responsible for hosting and running two recommendation pipelines, the personalised recommendation pipeline and the item-similarity pipeline.

8.3 Screenshots

Here are some screenshots from the demonstrator to better visualize the solution.

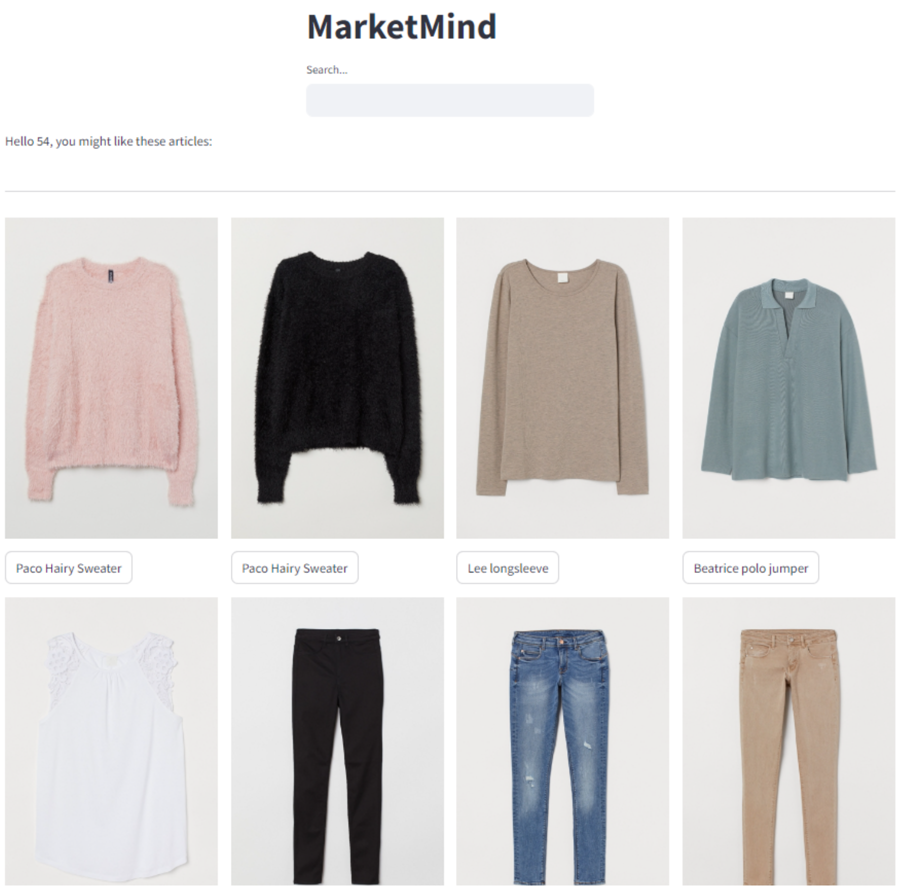

Deploy

Login

Please provide your customer ID

Customer ID

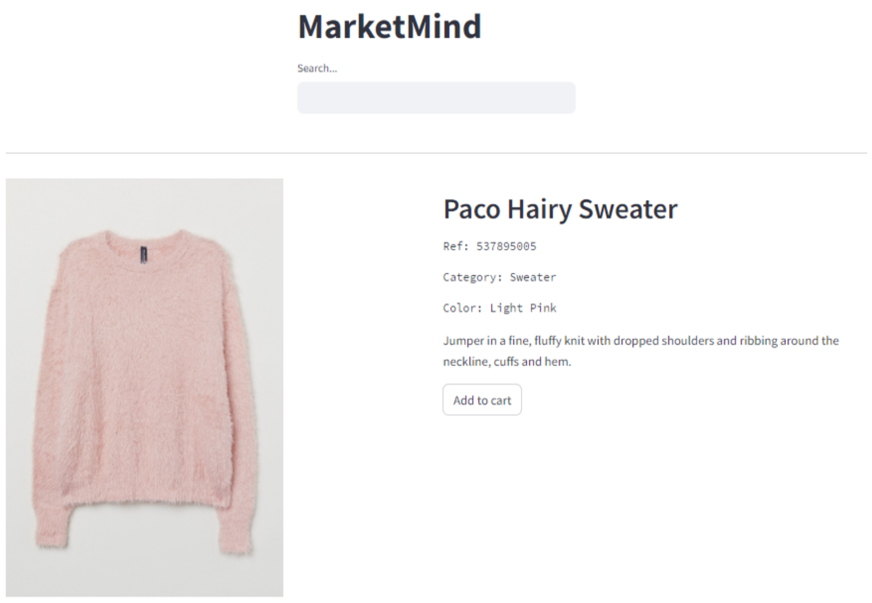
Figure 8.2: Login page screenshot


MarketMind

Search...

Hello 54, you might like these articles:

Figure 8.3: Home page screenshot



Paco Hairy Sweater

Ref: 537895005

Category: Sweater

Color: Light Pink

Jumper in a fine, fluffy knit with dropped shoulders and ribbing around the neckline, cuffs and hem.

Figure 8.4: Product page screenshot

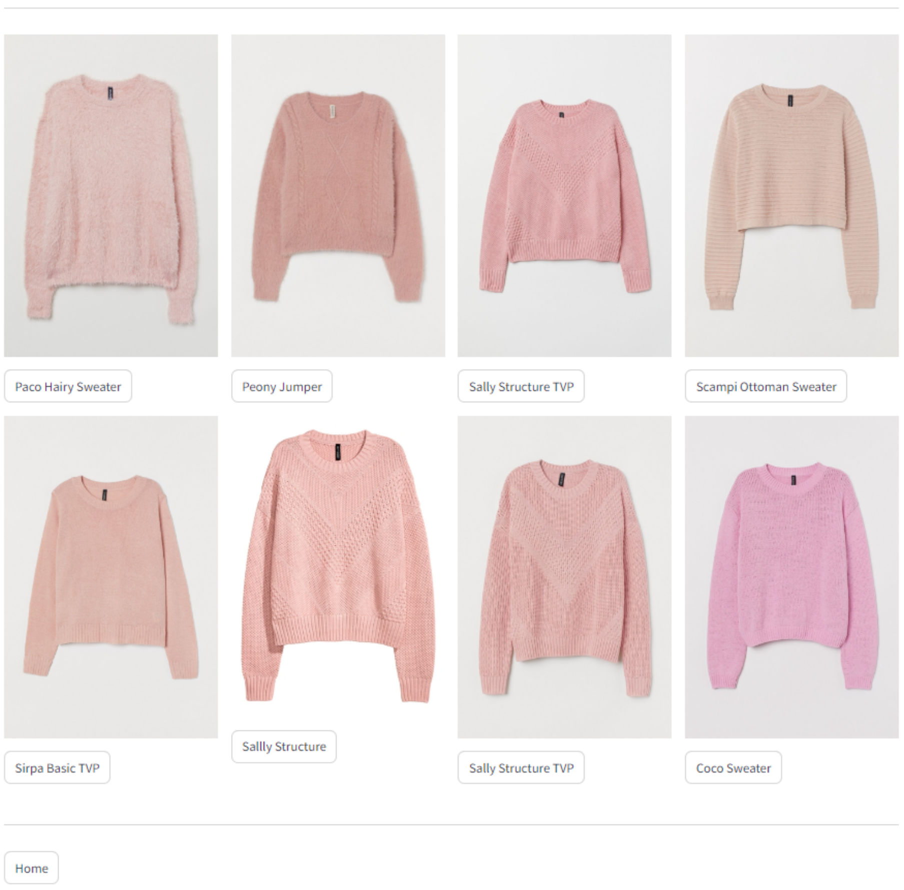


Figure 8.5: Item similarity recommendations in product page screenshot

Chapter 9

Conclusion

9.1 Key Learnings and Outcomes

The architectures developed have proven to be effective for handling a large-scale dataset and enabling near real-time inference. Incorporating a retrieval step is essential when dealing with a large number of items, as it prevents inference times from becoming prohibitively long.

A procedure for generating a labeled dataset using negative sampling from CRM data has been implemented. This method chronologically splits the dataset at an arbitrary cut-off date, creating a training period and a target period. By excluding any data from the target period during training, this approach minimizes potential data leakage and provides a more realistic simulation of model performance in a production environment at a given time.

The sampling process also focuses on popular items and frequent customers, to avoid relying solely on popularity and ensures an equal balance of positive and negative samples to help the model to focus on the products purchased (which reflects the user's real interests).

Leveraging state-of-the-art models through the Nvidia Merlin framework allowed for a comprehensive comparison with more traditional models, confirming their performance. The DLRM emerged as the best-performing scoring model in terms of AUC on the H&M dataset. The Wide & Deep model displayed similarly strong performance. It is worth noting, however, that simpler tree-based models like Random Forest also achieved competitive Precision scores with less complexity.

In the e-commerce fashion sector, the most impactful features turned out to be related to product popularity, as customers tend to purchase popular items more frequently. For customer features, the frequency of purchases is more usefeull than specific buying habits. Regular customers are more likely to make repeat purchases, making this information crucial for accurate recommendations.

9.2 Critiques and Areas for Improvement

Optimizations, such as storing processed data in the feature store, could drastically reduce the inference time. However, taking feature processing out of the pipeline makes maintenance and addition of new items and customers much more complex. There is a trade-off between inference speed and operational complexity.

While the recommendation system was tested offline on the generated dataset and online using the demonstrator, nothing compares to real-world testing in a production environment. In actual e-commerce businesses, new models are only validated after extensive A/B testing in production. For example, Google's Wide & Deep model was proven successful because it increased app acquisition rates in the Google Play Store.

AUC is a helpful metric for selecting scoring models, but it is not the ultimate measure. Ideally, we would have also calculated nDCG, which better reflects the ranking quality of recommendations or Mean Average Precision (MAP) to assess the average rank of the last purchased item among the recommended products. Another interesting benchmark would have been to generate recommendations for the 1 million customers in the dataset and submit the results to the Kaggle H&M competition leaderboard. However, with an average inference time of 1 second per recommendation, this would have required more than 11 days of computation on a single H100 GPU.

The winners of the Kaggle competition used optimized GPU-based decision trees, with CatBoost-GPU performing 30 times faster than LightGBM-CPU inference. They also divided customers into 28 groups and ran inferences simultaneously across multiple GCP servers, utilizing more than 300GB of GPU RAM. These optimizations highlight the importance of infrastructure in handling large-scale inference efficiently.

Bibliography

[1] Carlos García Ling. *HM Personalized Fashion Recommendations.* 2022. URL: https://kaggle.com/competitions/h-and-m-personalized-fashion-recommendations (cit. on p. 2).

[2] Netflix. *Netflix 1MCompetition.* 2006. URL: www.netflixprize.com (cit. on p. 3).

[3] Yehuda Koren, Robert Bell, and Chris Volinsky. «Matrix Factorization Techniques for Recommender Systems». In: *Computer* 42.8 (2009), pp. 30–37. DOI: 10.1109/MC.2009.263 (cit. on p. 4).

[4] Yu-Xiong Wang and Yu-Jin Zhang. «Nonnegative Matrix Factorization: A Comprehensive Review». In: *IEEE Transactions on Knowledge and Data Engineering* 25.6 (2013), pp. 1336–1353. DOI: 10.1109/TKDE.2012.51 (cit. on p. 5).

[5] Steffen Rendle. «Factorization Machines». In: *2010 IEEE International Conference on Data Mining.* 2010, pp. 995–1000. DOI: 10.1109/ICDM.2010.127 (cit. on p. 5).

[6] Maxim Naumov et al. *Deep Learning Recommendation Model for Personalization and Recommendation Systems.* 2019. arXiv: 1906.00091 [cs.IR]. URL: https://arxiv.org/abs/1906.00091 (cit. on p. 6).

[7] Heng-Tze Cheng et al. *Wide Deep Learning for Recommender Systems.* 2016. arXiv: 1606.07792 [cs.LG]. URL: https://arxiv.org/abs/1606.07792 (cit. on p. 8).

[8] Xinyang Yi, Ji Yang, Lichan Hong, Derek Zhiyuan Cheng, Lukasz Heldt, Aditee Kumthekar, Zhe Zhao, Li Wei, and Ed Chi. «Sampling-bias-corrected neural modeling for large corpus item recommendations». In: *Proceedings of the 13th ACM Conference on Recommender Systems.* RecSys '19. Copenhagen, Denmark: Association for Computing Machinery, 2019, pp. 269–277. ISBN: 9781450362436. DOI: 10.1145/3298689.3346996. URL: https://doi.org/10.1145/3298689.3346996 (cit. on p. 9).

[9] Vito Walter Anelli, Alejandro Bellogín, Tommaso Di Noia, Dietmar Jannach, and Claudio Pomo. «Top-N Recommendation Algorithms: A Quest for the State-of-the-Art». In: *Proceedings of the 30th ACM Conference on User Modeling, Adaptation and Personalization.* UMAP '22. ACM, July 2022. DOI: 10.1145/3503252.3531292. URL: http://dx.doi.org/10.1145/3503252.3531292 (cit. on p. 11).

[10] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. *The Faiss library.* 2024. arXiv: 2401.08281 [cs.LG]. URL: https://arxiv.org/abs/2401.08281 (cit. on p. 18).

[11] Patrick John Chia, Giuseppe Attanasio, Federico Bianchi, Silvia Terragni, Ana Rita Magalhães, Diogo Goncalves, Ciro Greco, and Jacopo Tagliabue. «Contrastive language and vision learning of general fashion concepts». In: *Scientific Reports* 12.1 (Nov. 2022). ISSN: 2045-2322. DOI: `10.1038/s41598-022-23052-9`. URL: `https://doi.org/10.1038/s41598-022-23052-9` (cit. on p. 22).

[12] Dimitris Paraschakis, Bengt J. Nilsson, and John Holländer. «Comparative Evaluation of Top-N Recommenders in e-Commerce: An Industrial Perspective». In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. 2015. DOI: `10.1109/ICMLA.2015.183` (cit. on p. 25).