FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Semantic Product Search
Student:	Bc. Patrik Schweika
Supervisor:	doc. Ing. Pavel Kordík, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2024/2025

Instructions

Survey the field of modern neural networks used for Information Retrieval task. Focus on robust transformer based architectures and pretrained backbone models. Design and implement a benchmarking system, that will evaluate performance of various methods and approaches for semantic product search on a dataset e.g. [1]. Use metrics such as F-score, MAP and nDCG to compare the performance of the models. Take into account memory and speed requirements of models, not only their performance. Measure these additional requirements with model size and inference time. Assess potential biases of the models and provide interpretable results. Recommend the best possible architectures for semantic product search under different time/resources constraints.

[1] https://paperswithcode.com/paper/shopping-queries-dataset-a-large-scale-esci

Electronically approved by Ing. Magda Friedjungová, Ph.D. on 4 January 2024 in Prague.

Master's thesis

SEMANTIC PRODUCT SEARCH

Bc. Patrik Schweika

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: doc. Ing. Pavel Kordík, Ph.D.
May 9, 2024

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Schweika Patrik. *Semantic Product Search.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

List of code listings

Primarily I would like to thank my supervisor doc. Ing. Pavel Kordík, Ph.D. especially for his valuable insights, ideas, and advice during the work on this thesis. Additionally, I want to thank Rodrigo Augusto da Silva Alves, Ph.D. for creating the interaction dataset. Finally, my thanks go to my family, friends, and colleagues at work for their emotional support during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 9, 2024

Abstract

In this thesis, we deal with the topic of semantic product search, with a focus on pre-trained backbone models. We leverage recommendation system techniques to improve the accuracy of the information retrieval. We have designed and implemented various models for three different tasks: semantic re-ranking of products, search via interactions, and personalized semantic search. In the final task, we combined recommendation and semantic search techniques. Experiments confirmed that the combination of these techniques leads to better accuracy. Mainly the use of semantic search for cold-start cases.

Keywords information retrieval, learning to rank, recommender systems, transformer, semantic search

Abstrakt

V této práci se zabýváme tématem semantického vyhledávaní produktu, se zaměřením na před trénované modely transformerů. Používáme také techniky doporučovacích systeml pro zlepšení přesnosti vyhledávání. V práci jsme navrhli a implementovali různé modely pro tři úkoly: semantické řazení produktů, vyhledávaní podle interakcí a personalizované semantické vyhledávání. V posledním úkolu jsme spojili techniky doporučovacích systémů a semantického vyhledávání. Experimenty potvrdili, že spojení těchto technik vede na lepší přesnost. Především použití semantického vyhledávání pro případy s malým počtem interakcí.

Klíčová slova získávání informací, learning to rank, doporučovací systémy, transformer, semantické vyhledávání

Abbreviations

IR	Information Retrieval
LTR	Learning to Rank
MLP	Multi-Layer Perceptron
BERT	Bidirectional Encoder Representations from Transformers
RS	Recommendation Systems
CB	Collaborative Filtering
CBF	Content-Based Filtering
TF-IDF	Term frequency-inverse document frequency
ML	Machine learning
MRR	Mean Reciprocal Rank
DCG	Discounted Cumulative Gain
nDCG	Normalized Discounted Cumulative Gain

Introduction

Semantic Product Search has become an increasingly relevant topic today, especially in the era of e-commerce and digital marketplaces. As the amount of online product and user information rises, the need for simpler and more capable search methods becomes vital. Large language models (LLMs) like BERT, GPT, and others, transformed how we can process and understand natural language, making them good candidates for enabling semantic search capabilities. These models are capable of understanding the nuances of human language, enabling them to go beyond keyword matching to grasp the contextual meanings behind search queries. This can lead to more meaningful search results, improving user experience by helping consumers find products that meet their needs and preferences, without creating complex queries based on keywords and complex filters. Additionally, using these advanced models could help companies increase conversion rates by effectively offering personalized products for customers.

My motivation for choosing the topic is that personalized semantic search in e-commerce is still quite an unexplored field. I want to tackle it and see whether we could combine semantic search methods with recommendation system techniques in order to improve the quality of search results.

In the thesis, we first go through the literature on information retrieval and recommendation systems. Then we design the tasks we want to tackle in the thesis. Subsequently, we will design and implement models for each task. The models will be compared in terms of accuracy, memory requirements, and inference time. Lastly, we will evaluate the models on test data and discuss the results from the experiments.

The thesis will be structured into 5 main chapters and a final discussion. In the next chapter, we will outline the objectives of the thesis. Then we will do a literature review on information retrieval and related topics. After that, we will describe the dataset and design tasks we want to perform in our thesis. There will be a chapter for each task. Each of this chapter will include a dataset description, design and implementation of the models, and an experiment where we use the implemented models for prediction on test data. Lastly, we will discuss the results of the experiments and recommend the best models for each task.

Objectives

The goal of this thesis is to design and implement models for various tasks of information retrieval. Ranging from ranking of products based on their relevancy to a query to personalized semantic search. Models will be compared for each task in terms of accuracy, memory size, and inference time. Finally, we want to test the hypothesis that we can achieve the best results by combining interaction and semantic search techniques.

The first objective of the thesis is to do a literature review on information retrieval and recommendation systems and related topics such as learning to rank, the basics of text embeddings, and recommender Systems. Lastly, we want to review the state-of-the-art methods for semantic search with a focus on pre-trained transformer-based architectures.

The second and third objectives are to design and implement experiments with a benchmarking system that will allow us to measure the accuracy, memory, and inference time of used models. For each experiment, we will design and implement appropriate models. We will have three tasks. The first one will be focused on a semantic re-ranking of products for a given query. The second one is based on relevant product retrieval solely from past user interactions. In the third one, we will use product attributes and user interactions to test the hypothesis that semantic search can help recommender systems with the retrieval of relevant products.

The fourth objective of the thesis is to compare the implemented models in terms of their accuracy and performance metrics. After each experiment, we will discuss the outcomes and highlight the best models under different memory/time constraints.

The last objective is the overall discussion of the experiments. The goal is to recommend the best methods for each task based on their measured accuracy, memory requirements, and inference time.

Chapter 1
Literature review

In this chapter, we will research methods for Semantic Product Search, which is an *Information Retrieval* Task. Firstly we will define an Information Retrieval task and how it relates to Learning to Rank. Then we will take a look, at how text can be embedded into vectors. Then, we will research Recommendation Systems, as we want to know whether interactions can help with semantic search. Lastly, we will take a look at state-of-the-art methods for Information Retrieval with a focus on pre-trained neural networks.

1.1 Information retrieval

Information Retrieval (IR) plays a critical role in everyday life by powering various essential applications like internet searches, question-answering systems, personal assistants, chatbots, and digital libraries. The main goal of IR is to find information relevant to a user's query. Usually, more than one records are retrieved, and they are typically ordered based on how closely they match the user's query in terms of similarity. Initially, the field of IR mainly utilized traditional text retrieval systems that operated by matching terms found in both documents and queries. However, these systems based on term matching face several issues, including polysemy (multiple meanings for a single word), synonymy (different words with similar meanings), and lexical gaps (missing words), which can reduce their effectiveness. [1]

Information retrieval involves a system that manages a collection of documents. When a user submits a query, the system identifies and retrieves documents from the collection that contain the query words. It then ranks these documents and displays the highest-ranked documents, such as the top 1,000, to the user. [2]

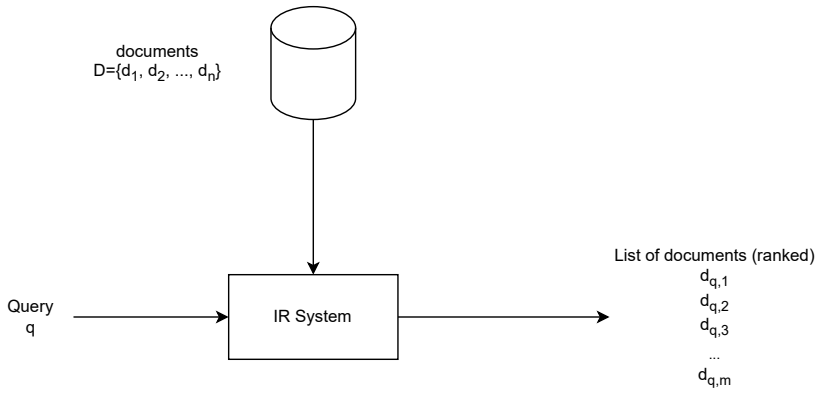


Figure 1.1 Information retrieval system

1.1.1 Learning to rank

Information retrieval involves a critical task where ranking is essential. The system holds a collection of documents and, upon receiving a user's query, retrieves documents that include the query words. It then ranks these documents and displays the highest-ranked ones to the user. The ranking primarily depends on how relevant the documents are to the query. Training data usually consists of queries with assigned lists of items with given relevancy ordering. [2]

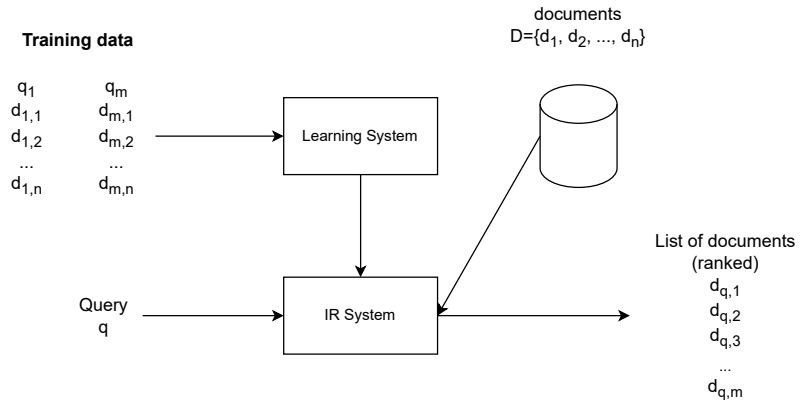


Figure 1.2 Learning to rank system

According to [2] there are three ways how to approach a learning-to-rank task: **Pointwise** approach converts the ranking task into either a regression or classification problem for individual items. In this method, the model processes one query and one item at a time, predicting the relevance score of the item or assigning it to a specific class.

Pairwise tackles the issue by classifying pairs of items, determining which item in the pair is more relevant than the other. This binary comparison allows for the ranking of items from most to least relevant.

Listwise treats the entire list of items as a single learning sample. It simultaneously predicts the scores for all items in response to a specific query, rather than evaluating items individually or in pairs.

1.1.2 Models for Information Retrieval

Here we will go through what types of models are typically used for IR. The following models are gathered from the book [3].

Vector Space models (VSM) models represent documents and queries as vectors in a high-dimensional space. Similarity between documents and queries is measured typically using cosine similarity. Use: Ideal for basic search tasks and when computational resources are limited.

Boolean models are based on Boolean logic, these models retrieve documents that satisfy a logical expression of keywords. Useful for simple, keyword-based queries but lack the nuance of semantic understanding.

Probabilistic models such as BM25, use probabilistic frameworks to estimate the likelihood that a document is relevant to a query. Effective for ranking documents by relevance and are widely used in search engines.

Latent Semantic Indexing (LSI) uses singular value decomposition (SVD) to reduce the dimensionality of the term-document matrix, capturing latent semantic relationships. Good for capturing semantic similarities but can be computationally intensive.

Learning to Rank (LTR) Models use machine learning techniques to learn how to rank documents based on features extracted from the query and documents. Effective for complex ranking tasks in large-scale search engines.

Graph-Based models These models, such as PageRank, use the link structure of the web (or any document set) to determine the importance of a document. Useful for web search where the interlinking between pages is significant.

Neural Network models: More recent developments in IR have incorporated deep learning. Models like word embeddings (e.g., word2vec), and neural networks have been used to understand complex query/document relationships beyond simple keyword matching.

1.1.2.1 BM25

The BM25 is a ranking algorithm developed in the 1990s by Robertson and Walker. And currently has multiple variants. The purpose of BM25 is to judge the relevance of a document to a given query and rank the documents from the corpus based on these relevancy scores. It is based on the TF-IDF, but it expands on its shortcomings. [4]

The algorithm considers the frequency of query tokens in the documents, the length of the document, and the average document length in the entire corpus. Apart from classical TF-IDF, the BM25 incorporates document length normalization to address the impact of document length on relevance. Since longer documents tend to have more occurŕences of a term, which leads to bias. Normalization counteracts this by dividing the term frequency by the document's length. And also **Query Term Saturation** also included a term saturation function to mitigate the impact of very high term frequencies. This function reduces the effect of high-term frequencies on relevance scoring. Since if one term appears in a document many times it is less informative than other terms. [5]

The algorithm requires tuning parameters b and k_1 to control the impact of document length normalization and term frequency. Commonly used values are $b = 0.75$ and $k_1 = 1.2$.

1.1.3 Metrics

Metrics used for model evaluation in IR and LTR according to [2] are:

Precision measures the accuracy of the retrieved results. It is the proportion of retrieved documents that are relevant to the query. A higher precision score indicates that a higher proportion of the documents retrieved are relevant. High precision is important in situations where the cost of retrieving an irrelevant document is high.

$$\text{precision} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{retrieved documents}\}|}$$

Recall measures the completeness of the retrieved results. It is the proportion of relevant documents that are successfully retrieved. A higher recall score indicates that the system retrieved a larger portion of all relevant documents available. High recall is crucial in scenarios where it is important not to miss any relevant document.

$$\text{recall} = \frac{|\{\text{relevant documents}\} \cap \{\text{retrieved documents}\}|}{|\{\text{relevant documents}\}|}$$

F1-score is the harmonic mean of precision and recall.

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

CG (Cumulative Gain) is the sum of the relevance values of all results in a search query or recommendation list. It is one of the most simple metrics and in practice, it is rarely used. Since it does not differentiate the relevance of items based on their positions in the result list. This is a significant drawback because items at the top of the list are generally more important than those further down.

$$\text{CG}_p = \sum_{i=1}^{p} rel_i$$

where p is number of documents and rel_i is relavancy score of i-th document.

DCG (Discounted Cumulative Gain) builds upon cumulative gain by introducing a discounting factor that reduces the relevance value of items based on their position in the list. The relevance score of each item is divided by a logarithmic function of its rank position. This means that the relevance value of items at the top of the list is reduced less than those lower down. DCG is more effective than CG at reflecting the decreased importance of relevant items that appear lower in a search result or recommendation list.

$$\text{DCG}_p = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)}$$

nDCG (Normalized Discounted Cumulative Gain) is an improvement of DCG that normalizes the score, allowing for a comparison between queries or recommendation lists of different sizes or different levels of item relevance. This normalization allows for the comparison of nDCG scores across different datasets. Making it a robust metric for evaluating and comparing the performance of ranking algorithms in various contexts.

$$\text{nDCG}_p = \frac{\text{DCG}_p}{\text{IDCG}_p},$$

where **IDCG** is ideal discounted cumulative gain:

$$\text{IDCG}_p = \sum_{i=1}^{|\text{REL}_p|} \frac{rel_i}{\log_2(i+1)}$$

In summary, CG, DCG, and nDCG offer increasing levels of sophistication for measuring the effectiveness of ranking systems. While CG offers a basic measure of total relevance, DCG and nDCG provide more nuanced assessments that take into account the position of items in the ranking, with nDCG offering a standardized way to compare different datasets.

MRR (Mean Reciprocal Rank) measures the average position of the first relevant item across all queries. It doesn't take into account any other relevant items after the first one. It is usually used in tasks where the goal is to retrieve one item. This metric measures how far down is the relevant item in the predicted items list on average.

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{rank}_i}$$

where Q is a set of queries and rank_i is a position of the first relevant item to the query q_i. If it is on the first position then the reciprocal rank for that query is one. If there is no relevant item, then $rank_i = \infty$ and the fraction results in zero.

1.2 Text representation

Machine learning models require numerical input to be trained and make predictions. In this section, we will review what are the methods for text embedding into numerical vectors. Firstly we will go through text tokenization. Then we will review embedding methods. We are starting from the simple TF-IDF embedding, which is an important statistical measure of word occurrence in a corpus of documents. To neural embeddings, which are better at grasping the semantic context of a sentence.

1.2.1 Tokenization

Text tokenization is the first step in the text embedding process. We have to break down the text into **tokens**, which are small parts of the text (characters, words, subwords, sentences). Each of these tokens is then given a unique number (index). Tokens extracted from a given set of documents are called **vocabulary**. We will discuss the various methods for text tokenization. Some methods simply divide the text into words, which are then used as tokens. However, this approach does not take into account the contextual meaning of the words, since we do not know their position in a sentence. Other methods can concatenate neighboring words to better capture the context of the word in a sentence.

- **Word unigram** splits the text into individual words. Punctuation marks are usually separated from the words. For sentence `"Hello world!"` the following tokens would be extracted `["Hello", "world", "!"]`.
- **Word n-gram** splits the text into subsequent N words. This is rather useful when we want to preserve some context already in the tokens, so we don't have to use LSTM or GRU for short-term memory. For example, if we choose $n = 3$ (tri-gram) for text `"Samsung Galaxy S20 red case"`. Then the following tokens will be created `["Samsung Galaxy S20", "Galaxy S20 red", "S20 red case"]`.
- **Character n-gram** breaks down the text into N subsequent characters. It is useful when we want to have more fine-grained tokens. However, this method creates a much larger vocabulary than word n-grams. If we use tri-grams the sentence `"Hello world!"` would get broken down into tokens `["Hel", "ell", "llo", "lo ", "o w", " wo", "wor", "orl", "rld", "ld!"]`.
- **Subword** tokenization breaks down words into smaller units. BERT's WordPiece and GPT's Byte Pair Encoding (BPE) are popular subword tokenization algorithms.
- **Sentence** tokenization breaks down the text into sentence tokens. It is useful when we want to capture the context of a longer text.

1.2.2 TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) is a statistical measure used to determine the mathematical significance of words in documents. [6] It takes into account how often a word appears in a document compared to the whole corpus of documents.

TF (Term Frequency) is a ratio of how many times a token appears in a document to all the tokens in a document. For example if we have a document with text `He ate a red apple`. The term frequencies for words would be `[0.2, 0.2, 0.2, 0.2, 0.2]`.

$$\mathrm{tf}(x, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

where $tf(x, d)$ is a count of token x in document d.

IDF (Inverse Document Frequency) for a token x is a logarithm of the ratio of the total number of documents in a corpus to the number of documents that x appeared in. So IDF is higher for tokens that do not appear in many documents in the corpus. This helps to shift focus from common occurring words like a, in, on, at, etc. To more meaningful words, that only appear in a smaller amount of documents. This helps us to extract keywords from a document, that are unique to it.

$$\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

where D is a corpus of all documents and N is the total number of documents in the corpus.

The overall TF-IDF is then calculated by the multiplication of TF by IDF. The highest values of TF-IDF are reached when token x appears many times in document d, but does not appear in other documents from corpus D.

$$\mathrm{tfidf}(t, d, D) = \mathrm{tf}(t, d) \cdot \mathrm{idf}(t, D)$$

For example, if we have documents:

1. `Josh is driving a red car`
2. `He is eating a red apple`
3. `He came home from a work`

Table 1.1 Example of term frequencies

	d_1	d_2	d_3
Josh	1	0	0
is	1	1	0
driving	1	0	0
a	1	1	1
red	1	1	0
car	1	0	0
he	0	1	1
eating	0	1	0
apple	0	1	0
came	0	0	1
home	0	0	1
from	0	0	1
work	0	0	1

Table 1.2 Example of TF-IDF for d_1

	idf	tfidf
Josh	0	0
is	0	0
driving	0	0
a	0	0
red	0	0
car	0	0

1.2.3 Embedding

There are many ways, how the text can be embedded into a vector space. Vectors can then be used for training a model. We will go from simple embeddings, that mainly capture the word occurrences in a document. To more complex embeddings, which can capture the semantic meaning of words, so that words with similar meanings are close to each other in the vector space. And lastly, we will look at embeddings that can capture the contextual meaning of the words or sentences.

Simple embedding techniques only count occurrences of a word in a document. And do not capture any semantic or contextual meaning of these words.

- **One-hot encoding**: each token in the vocabulary has a vector assigned to it. It has 1 on the position of a token in a vocabulary and 0 in other places. So document embedded through one hot encoding would result in a vector that has 1's for every token in the document.
- **Bag of words**: It is similar to one-hot encoding, with the exception, that it counts how many times a token appeared in a document. So if one token appeared 5 times in the document. The resulting token would have a value of 5 on the position of the token.
- **TF-IDF**: The embedding process is similar to one-hot encoding, but with a difference that the value for a token is calculated with tfidf.

Semantic embeddings can capture a semantic meaning of words in a vector space. One famous example is Word2Vec which maps words King and Queen close to each other. And if we add vector Woman to King we get a Queen.

- Word2Vec (Google): uses a neural network model to learn word association from a large corpus of text. It comes in two variants: Continuous Bag of Words and Skip-gram. Word2Vec can capture semantic and syntactic word relations.
- Glove (Stanford):
- FastText (Facebook):

Contextual embeddings unlike the previous methods are context-dependent. This means the same word can have different embedding based on its usage and place in a sentence.

- ELMo: Uses a deep, bidirectional LSTM model to create the embeddings.
- BERT (Google): A transformer-based model that generates bidirectional representations by pre-training on a large corpus using tasks liked Masked LM and Next Sentence Prediction. BERT embeddings are context-sensitive and have significantly improved performance on a range of NLP (Natural Language Processing) tasks.

1.3 Recommender systems

Recommender systems have transformed how users discover products, content, and services by filtering large information spaces to provide personalized recommendations. They are crucial in areas such as e-commerce, online streaming, and social networking. Recommender systems fall into three categories: **collaborative filtering**, **content-based filtering**, and hybrid methods.

- Collaborative Filtering (CF): This method makes recommendations based on past interactions recorded between users and items. The assumption is that those who agreed in the past will agree in the future about item preferences [7].

- Content-Based Filtering (CBF): Recommendations are provided by analyzing the content of items and the profile of a user's interests. This technique uses item features to recommend other items similar in nature [8].
- Hybrid approaches: These methods combine collaborative and content-based filtering to improve recommendation quality and overcome specific limitations associated with each approach [9].

1.3.1 Collaborative filtering

Collaborative filtering is a method used to build personalized recommendations by learning from the past behaviors of a group of users. Unlike content-based filtering, CF relies on user-item interactions without requiring knowledge of the item itself.

According to [7] CF is the most common technique when it comes to building recommendation systems. Since it can recommend content to users based on their preferences and the preferences of similar users. Most popular websites like Amazon, YouTube & Netflix already use collaborative filtering methods to recommend content to their users.

Collaborative filtering recommends content to users based on interactions of similar users. So for example if user A watches similar movies to user B. And then user B watches movie Y that A hasn't seen. Then it can be displayed in the recommended movies list for user A.

There are many techniques how to determine, which users are similar to each other. Most of the techniques use so-called **rating matrix** or interaction matrix. These terms are often used interchangeably. A rating matrix is a 2D matrix typically with users represented as rows of the matrix and items as the columns of the matrix. Where i, j value of the interaction matrix says how much user i interacted with item j.

Table 1.3 Example of rating matrix

	i_1	i_2	i_3	i_4
u_1	5	5	0	4
u_2	1	2	0	4
u_3	0	1	3	0
u_4	1	3	1	0
u_5	1	4	0	5

These interactions are typically collected from implicit or explicit feedback from the users. Explicit feedback as the name suggests is the feedback that the user intentionally provides for a product. This can be for example: rating a product, writing a review, or filling out a survey. On the other hand, implicit feedback is gathered based on the actions or behaviors of users in the system. Users are typically not aware that they provided this type of feedback to the system. Implicit feedback for example is: clicking on an item in the shop, purchasing an item, time spent watching a movie, or even mouse movements can be considered as explicit feedback.

To use this feedback in the rating matrix it has to be converted to numerical values. For explicit feedback, it is usually rather simple when scoring is involved. For example, if the user i rated item j with 5 stars. The i, j-th value of the rating matrix will be 5. We will put 0 for items that the user didn't interact with. If we have implicit feedback we need to assign numerical importance to each action. Then we can sum these actions to get the interaction value. For example, we could encode the implicit actions like this:

- Clicking on Item = 1
- Purchasing an item = 100
- Watching a movie = time spent watching a movie * multiplier

On internet websites, the implicit feedback is much more relevant than the explicit feedback. Since it doesn't require any intentional action from users.

1.3.1.1 Techniques

To build a recommender system based on collaborative filtering there are multiple steps involved to it. The first step in collaborative filtering is to determine, which users are similar to one another. The second step is to predict ratings of items, which the user didn't interact with based on the interactions of similar users. The last step is to measure the accuracy of the predicted ratings. [7]

The metric for measuring accuracy depends a lot on the requirements for the task. If the ordering of the items does not matter and we want the predicted ratings to be the most accurate. We could use RMSE (Root mean squared error) between the predicted rating and the true rating of the user. But if we would more care about the ordering of the recommended items to a user instead of the predicted ratings. We could use NDCG or MRR, which places heavy emphasis on the order of predicted items. Techniques that predict relevant items higher in the list will have much better accuracy in this case.

However, one thing to keep in mind is that collaborative filtering will not take into account any additional attributes of users like age, country, or gender. Or attributes of the items like category, release year, color, etc. So recommender systems often use a pipeline of ML techniques to recommend content to its users and collaborative filtering is just one part of the whole pipeline.

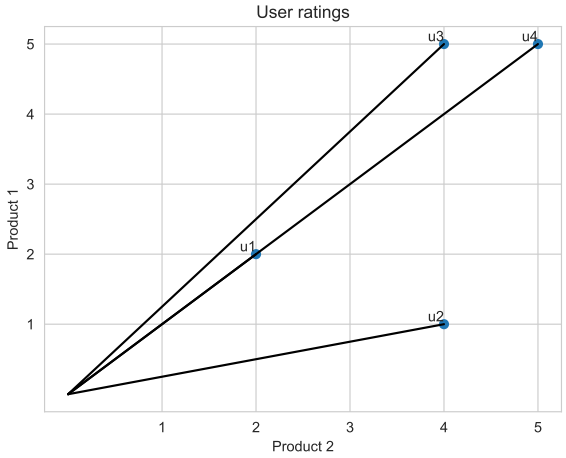
Collaborative filtering has several pitfalls. One of them is the cold start problem. This means that when a new user is added to the system he will not have many relevant similar users. Other techniques are usually used to recommend content to new users. Until they have gathered enough interactions, another problem with CF is scalability. As the number of users and items grows, memory-based CF becomes computationally expensive and less practical for real-time recommendations.

Memory-based algorithms use statistical methods to predict interactions solely from interactions in the dataset. These methods do not use any trained model. They only use the raw interaction data to make predictions.

We find rating R that would user U give to item I in two steps:

1. Find similar users to U
2. Calculate average ratings of similar users for I

To find similar users a metric has to be used that will measure how distant are two users. For example, euclidean distance can be used between two user ratings. However euclidean distance is not ideal as it would make users that have higher ratings on average to be more similar. Putting less emphasis on which particular items they rated. Cosine similarity is a better metric to measure similarity between users in terms of items they interacted with.



User	Product 1	Product 2
User 1	2	2
User 2	1	4
User 3	5	4
User 4	5	5

Figure 1.3 User rating example containing 2 products and 4 users.

For example, if we take a look at Figure 1.3 we can see that users u_3 and u_4 would be very close to each other in terms of Euclidean, distance. Since they only differ by one in the rating of Product 2. However, if we look at users u_1 and u_4 we can see that their lines are overlapping each other. That means they have the same angle. That means that they have the "same" scoring but with a different mean. If we would subtract the mean rating of each user, u_1 and u_4 would be identical. Their cosine distance is equal to zero. So u_1 and u_4 could be more similar users, but u_1 just rates items lower on average, since he could be a critique. While u_4 rates everything very highly.

K-nearest neighbors are the most common model that is used for this task. As it does not involve any fitting of the model and the predictions are solely made from the training data. Cosine distance is then used typically to find n most similar neighbors. The predicted rating for item i by user u can then be calculated with the following formula:

$$R_{i,u} = \frac{1}{n} \sum_{k=1}^{n} R_{i,k}$$

Where we take the average rating of n-neighbors. However, this formula has a flaw that it does not take the distance of the users into account. It is better to use similarity/distance to the neighbors in the formula and calculate a weighted average from that. The improved formula is:

$$R_{i,u} = \sum_{k=1}^{n} \frac{R_{i,k} \cdot S_k}{S_k}$$

Where S_k is a similarity of user k to user u.

Model based family of algorithms uses dimensionality reduction of training ranking matrix to get latent features of the space.

In the ranking matrix, there are two dimensions: 1) a number of users, and 2) a number of items. And the matrix is very sparse most of the time. We can then use matrix factorization methods or autoencoders to perform the dimensionality reduction.

Matrix factorization is a process of splitting large matrix into a product of smaller ones. A matrix $A \in \mathbb{R}^{m,n}$ can be made of matrices $X \in \mathbb{R}^{m,p}$ and $Y \in \mathbb{R}^{p,n}$. Matrix A is then retrieved by formula $A = X \cdot Y$. Where p is a shared dimension of the submatrices and is usually defined as a hyperparameter in the matrix factorization algorithms.

The reduced matrices X and Y then represent the users and items individually. Since X would have m rows for m users and matrix Y would have n columns for n items.

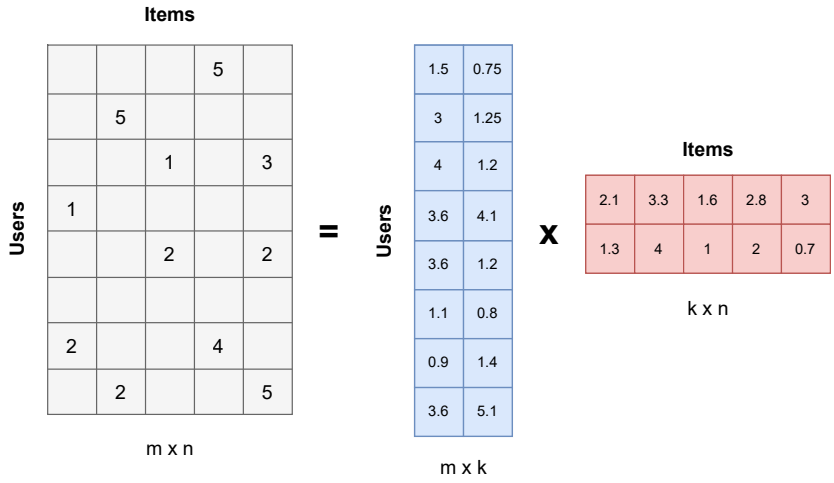


Figure 1.4 Matrix factorization example

Some of the most popular algorithms for matrix factorization are SVD (Singular Value Decomposition) and ALS (Alternating Least Squares). Or we can use typical dimensionality reduction methods like PCA (Principal Component Analysis), t-SNE (t-Distributed Stochastic Neighbor Embedding), or LDA (Linear Discriminant Analysis). We could also use a combination of multiple methods. For example, we can reduce 256 dimensions to 50 by PCA and then use t-SNE to reduce 50 dimensions to 10. This can have better results since PCA can only remove linear dependence between variables while t-SNE can also remove non-linear dependencies. [7]

1.3.2 Content-based filtering

Content-based filtering involves recommending items similar to those a user has previously liked, focusing solely on the properties of the items themselves. Unlike collaborative filtering, CBF does not rely on data from other users, making it particularly useful in situations with sparse user interaction data. The main cornerstone of CB filtering is user profiles, which are created based on their previous item interactions. This profile is built from item attributes that they interacted with. Item attributes can be title, genre, colors, price, image, audio, etc. The recommender system then recommends similar items based on his user profile. For example, if the user was showing interest in horror movies the system could recommend him a lot more horror movies, than movies of other genres. [8]

One advantage of content-based filtering over collaborative filtering is that it focuses on personalized users instead of on other similar users. And so it is also independent of other users' interactions. This will make the recommendation much more personalized. The pitfall of CB filtering is the cold start problem. We cannot build an accurate user profile when the user does not have enough interactions. The most popular items can then be recommended instead of the personalized item recommendations.

1.3.2.1 Techniques

Content-based filtering is performed in multiple steps. In the first step item features are embedded into vectors. Feature extraction techniques for different types of data are for example:

- **Textual Data**: Techniques such as TF-IDF (Term Frequency-Inverse Document Frequency) or word embeddings (like Word2Vec and GloVe) are used to convert text into numeric vectors.
- **Image Data**: Convolutional Neural Networks (CNNs) can extract features from images.

- **Audio Data**: Spectral features, Mel Frequency Cepstral Coefficients (MFCCs), and other audio feature extraction techniques can be used for music or podcast recommendations.

Then, we build user and item profiles by aggregating the extracted features. This is typically done with sum-product or by averaging them. It is a good idea to have the features in unit length, so they have the same weight when aggregating them. User profiles are then created by aggregating the item profiles. Which encodes which kind of items the user interacts with.

Finally, items are recommended based on the similarity measures between user and item profiles. The similarity measures are:

- **Euclidean Distance**: Measures the straight-line distance between two points in Euclidean space.
- **Cosine Similarity**: Measures the cosine of the angle between two vectors, often used for text data.
- **Pearson Correlation**: Measures the linear correlation between two variables.

1.4 State-of-the-art methods

In this section, we will go through various state-of-the-art methods that are currently being used in IR. Since the IR is quite broad we will mainly point out neural network/transformer-based methods, which are the main focus of this thesis.

Recently, there have been major breakthroughs in Natural Language Processing (NLP) due to the greater accessibility of extensive labeled datasets and improved computational capabilities. These developments have enabled researchers to use deep learning approaches for a variety of applications. Such methods have improved traditional text retrieval systems and helped overcome the constraints associated with term-based retrieval methods. Nonetheless, these advanced techniques demand significant data and computational power. Consequently, researchers are continually crafting more sophisticated deep learning algorithms to satisfy these requirements and enhance outcomes in NLP tasks. [10]

1.4.1 Word-embedding

Word embeddings, which are dense, continuous representations that encapsulate semantic meanings of words, have become increasingly popular in IR systems for representing documents and queries. A variety of methods have been introduced to integrate word embeddings with other techniques. These methods range from aggregation techniques [11], bilingual word embeddings [12], to context representation in natural language generation [13]. The coming of neural networks enabled the mapping of documents and queries into continuous spaces, utilizing similarity metrics for ranking similarity of documents [14]. By leveraging these varied methodologies, IR systems can significantly enhance their functionality and effectiveness through the strategic use of word embeddings. [10]

1.4.2 Indexing techniques

Indexing schemes play a crucial role in IR tasks, as they structure and make the retrieval of large-scale documents faster. Dense retrieval methods, which utilize dense vector representations for both queries and documents, typically rely on Approximate Nearest Neighbor (ANN) techniques to enhance vector searches in internet search [15], [16]. Current approaches to dense retrieval often separate the processes of learning representations and building indexes, leading to several drawbacks. Specifically, the indexing process cannot leverage supervised learning, and the independently developed representations and indexes may not align perfectly, thereby diminishing

retrieval efficacy. Some research has explored the simultaneous training of indexes and encoders within the contexts of recommendation and image retrieval [17], [18]. [10]

1.4.3 Transformers

There are two types of transformer architectures used for comparing the semantic similarity of two texts:

- **Bi-encoders**: Encode documents and queries separately, then measure the similarity of embeddings. It is Fast and scalable, but less precise in capturing complex query-document relationships. [19]
- **Cross-encoders**: Encodes the query and document together, offering a deeper understanding of their relationship. More accurate but computationally more intensive. The output of a cross-encoder is a similarity score and not an embedding. [19]

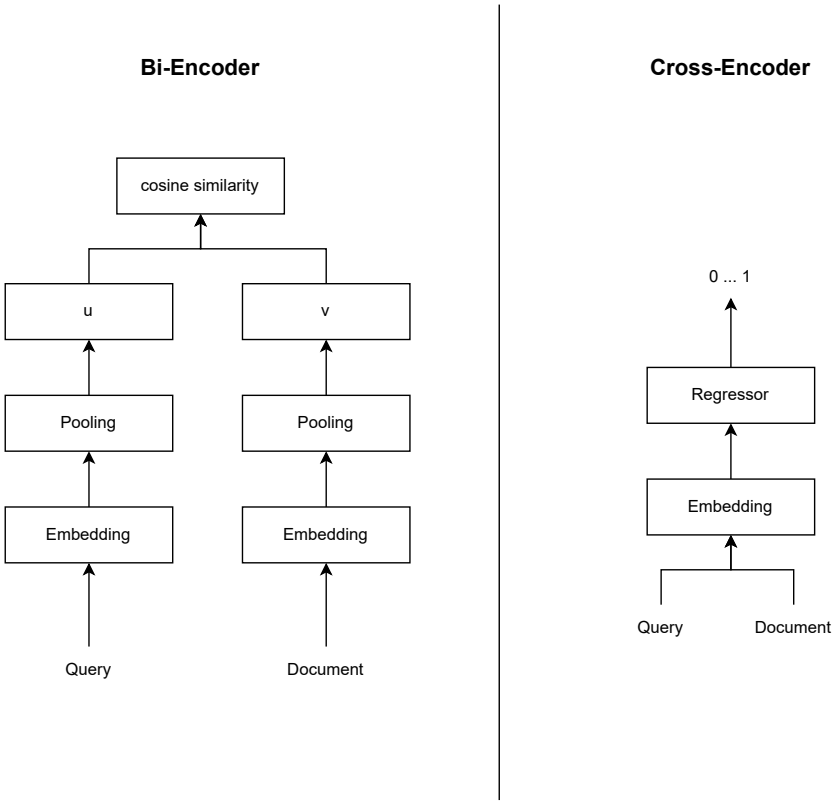


Figure 1.5 Scheme of transformer architectures

1.4.4 Pre-trained transformer models

Transformers have revolutionized the field of natural language processing and have been successfully applied to various tasks, such as text ranking. Unlike LTR methods that often rely on manually created features, transformer-based models use deep learning to develop complex representations of texts and generate rankings. This shift has enabled the creation of more sophisticated text ranking models that are better at understanding semantic context, thereby enhancing search engine effectiveness and user experiences. [10]

1.4.4.1 BERT

BERT (Bidirectional Encoder Representations from Transformers) is a groundbreaking approach in the field of natural language processing (NLP) that revolutionized the way machines understand human language. Developed by researchers at Google AI Language in 2018, BERT is a deep learning model based on the Transformer architecture, which utilizes attention mechanisms to process a wide range of language data. [19]

BERT is pre-trained on a large corpus of text in an unsupervised manner using two innovative strategies: Masked Language Modeling (MLM) and Next Sentence Prediction (NSP). In MLM, random words in a sentence are masked, and the model is trained to predict these words based on their context. NSP involves training the model to predict whether two given sentences are logically sequential or not.

1.4.4.2 GPT

The Generative Pre-trained Transformer (GPT) model represents a significant advancement in the field of natural language processing (NLP). Developed by OpenAI, GPT utilizes the transformer architecture, which is based on self-attention mechanisms that efficiently process sequences of data. The model is distinct for its training approach, which involves pretraining on a large corpus of text followed by fine-tuning on specific tasks. [20]

The backbone of GPT is the transformer model, which differs from previous sequence-based models by using self-attention mechanisms. These mechanisms allow the model to weigh the importance of different words within a sentence, irrespective of their positional distance from each other, leading to a more nuanced understanding of the text. [21]

GPT is characterized by its two-stage training process. In the pretraining phase, the model is trained on a diverse dataset to learn a broad understanding of language patterns and contexts. This phase is unsupervised, involving tasks like predicting the next word in a sentence. During fine-tuning, GPT is specifically trained on a smaller, task-specific dataset, allowing it to adapt its general language abilities to particular applications. [22]

1.4.4.3 Mixedbread

Mixedbread released a new promising open-source embedding model `mxbai-embed-large-v1` in March 2024. The model is also available on Hugging Face. This model enhances document search capabilities through Retrieval-Augmented Generation (RAG), which it turns documents into searchable formats in vector databases. It achieves state-of-the-art performance on an MTEB dataset. It even outperforms proprietary models like OpenAI `text-embedding-3-large` and Cohere `embed-english-v3.0`. [23]

1.4.5 Recommendation techniques for IR

In practice, real-world search engines might incorporate hundreds or more features. For systems that have extensive user bases, features based on user behavior, such as query frequency or the frequency of link clicks in different contexts, are crucial indicators of relevance. These features are thoroughly integrated into learning-to-rank algorithms. [10]

Traditional CF employs information filtering techniques to recommend items based on historical interactions. CF algorithms typically achieve superior predictive performance when there are numerous interactions between users and items. However, in scenarios with insufficient user data, CF tends to underperform due to its inability to effectively recommend items that lack prior interactions. These challenges are commonly referred to as the cold-start, sparsity, and scalability issues [24]. To address these problems, various strategies have been devised, including content-based filtering, which utilizes categorical information about items. [25]

In paper [26] they introduced a new hybrid recommender system model called SemCF, which utilizes both rating data of users and semantic information about items. It involves selecting semantic neighbors (based on item features) and satisfaction-based neighbors (based on user ratings), and combining these into a unified list of neighbors for better recommendation accuracy. Using the unified list, SemCF predicts ratings for unrated items and generates recommendations.

The authors in paper [27] propose novel semantic-enhanced Neural Collaborative Filtering (NCF) models that solve issue with traditional NCF, which is unable to capture the dynamic user preference over time. They interagrated a deep learning with ontology-like modeling. This integration aims to improve the accuracy of movie recommendations by using two main approaches:

1. **Building a Semantic Knowledge Base for Movies**: Leveraging ontology to create structured, meaningful representations of movie data.
2. **User Behavior Analytic Model**: Developing models that use user session data to predict user preferences, incorporating these predictions into the Neural Collaborative Filtering for better recommendation accuracy.

Chapter 2
Design

In the thesis, we will work on three tasks. The first one will be semantic re-ranking where we will want to rank products for a query based on their semantic relevancy. The second one will be the retrieval of relevant documents based solely on user interactions with products without any additional product information. Lastly, we will combine both of these methods and see whether we can leverage having both product information and user interactions. We will use multiple approaches for every task and compare their performance, memory requirements, and inference time.

2.1 Dataset

We will use the Amazon ESCI dataset [28], which was part of KDDCup 2022. The dataset contains multilingual (English, Spanish, and Japanese) queries and products. In this thesis, we will focus only on the English part of the dataset. We will further analyze and describe the dataset in the upcoming chapters. The competition in KDDCup 2022 [29] had 3 challenges to compete in:

1. **Learning to rank**: Given a user query and a list of matched products, the goal of this task is to rank the products, so that the relevant products are ranked above the non-relevant ones.
2. **ESCI product classification**: Given a query and products, the goal of this task is to classify each product into one of the ESCI categories.
3. **Substitute classification**: Given a query and products, the goal of this task is to classify whether a product is *Substitute* or not.

For tasks involving interactions, we use an interaction dataset that was artificially created from the ESCI dataset by Rodrigo Augusto da Silva Alves, Ph.D. This dataset also served for student competition in NI-ADM course.

2.2 Semantic re-ranking

In this part, we will focus on the first task of the KDDCup competition. The goal will be to rank products for a given query based on their relevancy. Products will have attributes like title, description, brand, and color. Each query-product pair will have an assigned relevancy score ranging from 0 to 1.

To complete this task we will use models that predict the similarity of two texts. As a baseline model, we will use TF-IDF embedding where similarity will be computed as a cosine similarity

between two embeddings. We will compare the TF-IDF with pre-trained sentence transformer and cross-encoder models. The performance of the models will be measured by the nDCG score which places heavy emphasis on the correct order of items based on their relevancy score.

2.3 Interaction similarity search

In the second task, we will use an interaction dataset that contains only user ID, query, and item ID. The goal will be to predict 100 relevant items for the user-query pairs in the test data, based on the interactions in the training set. In this task, we will use recommender systems techniques to leverage the interactions in the training set. Performance in this task will be measured by MRR score. Which will indicate the mean position of the relevant item in the 100 predicted items by the model.

2.4 Personalized semantic search

The last task will be the same as the second one. But with a change that this time we will also have product information from the ESCI dataset such as product title, description, color, brand, etc. We expect to achieve a higher MRR score than in the second task, given that we have more information to use. We will use the recommendation techniques from the previous task but with a combination of semantic search techniques.

2.5 Framework and library selection

We did not choose to use search frameworks like Elasticsearch or Apache Solr. Since we want to experiment with custom models for semantic search. It is easier to implement those by hand instead of integrating them into a complex framework. We will implement the models in Python language with the use of appropriate libraries in the form of Jupyter notebooks. Which are suitable for carrying out experiments and data visualization.

To work with data, we will use `pandas` library [30], which is a well-established library for handling structured data. For mathematical operations involving vectors and matrices, `numPy` [31] will be our library of choice due to its efficiency and wide usage. For visualizing the data, the `matplotlib` [32] package will be used, as it provides a comprehensive range of customizable plots. Additionally, for various machine learning models and statistical methods, we will use `sklearn` [33], a versatile library that includes several algorithms and tools for model building and evaluation. For working with sparse matrices there is a library called `scipy` [34], which will be used for modeling interactions. Lastly, for tasks involving neural networks and tensor operations, we will utilize `torch` [35], known for its flexibility and powerful computing capabilities in deep learning frameworks.

Displayed code snippets in the thesis are often simplified from the real implementation. To take up less space in the document and to make the code easier to understand for a reader. In cases where applicable, the models are fitted in batches, and similarly, the prediction process of these models operates in a batch-oriented manner.

2.6 Testing environment

The experiments within this thesis will be run on a dedicated local machine with the following hardware specifications:

- SSD: Samsung 970 EVO PLUS 2TB
- RAM: 32GB DDR5 5600MHz CL40
- CPU: Intel I7-12700K (Alder Lake)
- GPU: NVIDIA RTX 3080 (10GB VRAM)

Chapter 3

Semantic re-ranking

In this chapter, we will describe the Amazon ESCI Dataset. And use it for semantic re-ranking task. The goal of the task is to correctly rank products based on their relevancy to a query. The main accuracy metric for this task is nDCG. Then we will choose and implement appropriate models for this task, from the literature. Lastly, we will predict the relevancy scores for the test data with implemented models and compare them in terms of accuracy, inference time, and memory size.

3.1 ESCI dataset

The ESCI Dataset [28] was released by Amazon as part of the KDDCup 2022 challenge to improve the semantic matching of queries and products. For each query, the dataset has a list of up to 40 product results. With assigned ESCI relevancy rankings (Exact, Substitute, Complement, Irrelevant), which indicates the relevancy relationship between the query and the product. The dataset contains queries in multiple languages: English, Japanese, and Spanish. We will focus only on the English language.

There are 2 versions of the dataset: small and large version. The small version of the data set contains 48,300 unique queries and 1,118,011 rows corresponding each to a ¡query, item¿ judgment. The larger version of the dataset contains 130 652 unique queries and 2 621 738 judgments. The smaller version also contains queries that are considered easier to match products with. We will use the large version which contains all the queries and products.

The dataset consists of two tables. The first table consists of products. Which contains all the product attributes. These products are then referenced by product IDs from the second table which contains query-product relevancy scores.

3.1.1 Products

The product table contains information about specific products. However, it does not contain any user interactions such as comments, ratings, purchases, etc. But we can find here the product ID, title, description, brand, color, and bullet points (information about the product listed in points). The product ID is an ASIN, which stands for Amazon Standard Identification Number. We can use this ID to find the products in the Amazon store.

The mandatory fields are:

- `product_id` - ID of the product.
- `product_title` - Title of the product.

- `product_locale` - Locale of the product: `us`, `es`, `jp`.

Optional fields:

- `product_description` - Description of the product.
- `product_bullet_point` - Contains characteristics of the products in bullet points.
- `product_brand` - Brand of the product.
- `product_color` - Color of the product.

Table 3.1 Summary of the products

Language	# Products	Avg. queries
English (US)	1,215,854	1.5
Spanish (ES)	260,011	1.37
Japanese (JP)	339,059	1.32
Overall	1,814,924	1.45

Table 3.1 presents a summary of the products. The largest number of products, totaling 1,215,854, is in English, with each item appearing an average of 1.5 times per query. Following English, Japanese comes next in terms of quantity, with 339,059 products and an average of 1.32 queries per item. Lastly, Spanish comprises 260,011 products, with an average of 1.37 queries per item.

3.1.2 Queries

The query table consists of query and product pairs with their ESCI relevance judgments. The queries are completely anonymized as it does not contain user IDs or timestamps when they were entered into the system.

The meaning of each ESCI category is:

- **Exact** - Exact match for the query.
- **Substitute** - Substitute match for what the query was looking for. For Eg. searching for an iPhone mobile could recommend an iPhone charger.
- **Complement** - Complements the product that the query was looking for. Eg. searching for running shoes could recommend jogging pants.
- **Irrelevant** - Irrelevant product for the query.

The ESCI categories are then encoded into numerical counterparts with the following encoding:

Table 3.2 ESCI numerical encoding

ESCI	Encoding
E	1
S	0.1
C	0.01
I	0

The mandatory fields are:

- `example_id` - ID of the example.
- `query` - Contents of the query.
- `query_id` - ID of the query.
- `product_id` - ID of the product. Can be used for merging with the products table.
- `product_locale` - Local of the product.
- `esci_label` - ESCI scores.
- `small_version` - Indicates whether the example belongs to the smaller dataset.
- `large_version` - Indicates whether the example belongs to the larger dataset.
- `split` - Indicating to which split this pair belongs to: `train` or `test`.

Table 3.3 Summary of the queries

Language	# Queries	# Judgments	Avg. depth
English (US)	97 345	1 818 825	18.68
Spanish (ES)	15 180	356 410	23.48
Japanese (JP)	18 127	446 056	24.61
Overall	130 652	2 621 288	20.06

The summary for queries is displayed in table 3.3. In the summary for each language, we outline a number of queries, and number of query-product relevancy judgments, and the average query depth (the average number of products per query). The most amount of unique queries is for the English language sitting at 97,345 with a total of 1,818,825 judgments. The average depth of a query (mean number of products for query) sits at 18.68. The Japanese language follows the English language in terms of quantity. The number of unique queries is 18,127 number of judgments is 446,056 and the average depth is 24.61. And lastly, the Spanish language contains 15,180 unique queries, 446,056 judgments and the average depth of a query is 23.48.

3.1.3 Enhanced scraped dataset

We have found an enhanced version of the shopping queries dataset [36]. This dataset contains additional metadata about the products. It was scraped directly from the Amazon website. It contains additional product metadata:

- Categories - list of categories of the product. Where on the first place on the list is the main category of the product.
- Price - the price of the product.
- Ratings - number of ratings given to the product.
- Stars - overall rating of the product.
- Image - URL to a main image of the product
- Reviews - anonymized user reviews with the textual content of the review, data, and an assigned rating to the product. It does not contain any username or user ID.

This enhanced dataset gives us additional information about the products. Which could be used to improve the performance of semantic search. It also enables us to visualize additional properties of the products. This dataset contains 91.5% percent of products from the original dataset. So if we would use it we would not have the additional product metadata about all of the products. But the coverage is still quite high.

3.2 Dataset exploration

In this section, we will do an exploration of the dataset, which will help us to better understand its structure. And also it will aid us in choosing appropriate parameters for the models.

Table 3.4 Example of products from ESCI dataset

Title	Category	Brand
Molton Brown Jasmine Shower Gel, 10 Fl Oz	Beauty & Personal Care	Molton Brown
PlayStation 4 Slim 1TB Console (Renewed)	Video Games	Sony
Kung Fu: The Complete Series Collection	Movies & TV	WHV
Organic Lemon Regular, 1 Each	Grocery & Gourmet Food	Produce
Adidas Galaxy 4 Shoe - Mens Running 14	Clothing, Shoes & Jewelry	Adidas

Table 3.4 contains a selection of product examples from the Amazon ESCI dataset. Each entry includes the product title, the category to which the product belongs, and the brand associated with the product. Examples include items from everyday consumer electronics to personal care products.

Table 3.5 Example of queries

Query	Product title	ESCI
toothbrush cover	4 Pack Toothbrush Travel Case	E
mickey mouse pajamas	Disney Boys' Toddler 3-Piece Pajama Set, Mickey Mouse	E
p40 pro huawei	Huawei P40 Lite 5G Dual-SIM 128GB ROM	S
amd ryzen heatsink	AMD CPU Fan Bracket Base	C
us made robot vacuum	CleanView Swivel Upright Bagless Vacuum Carpet Cleaner	I

Table 3.5 displays a sample of search queries and corresponding product titles from the dataset. The table also includes the ESCI label for each query-product pair, indicating the type of search interaction: Exact (E), Substitute (S), Complementary (C), or Irrelevant (I).

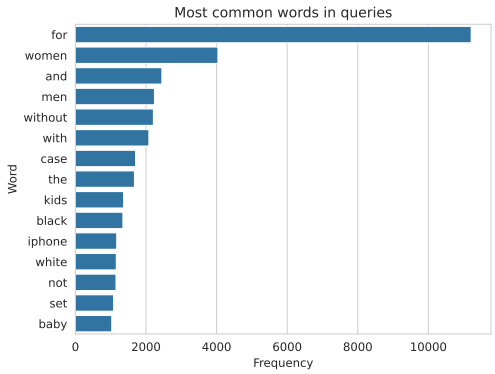


Figure 3.1 Most common words in queries

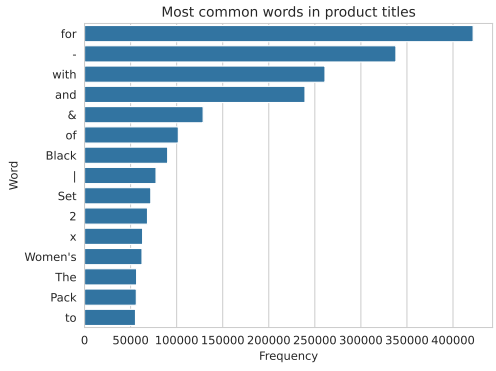


Figure 3.2 Most common words in product titles

We captured the most common words in queries in Figure 3.1. The words for queries are quite common English words. We also have to take into account that these words are biased towards the Amazon Eshop. So it appears logical that "iphone" is high on the list. One interesting thing is that "word" for appeared quite a lot of times. This could be caused by the fact that people

are searching for products ”for someone”. For example: ”t-shirt for kids”, ”parfume for women”, etc.

The most common words in product titles in Figure 3.1 are a bit messier than the ones in product titles. That is caused by the fact that product titles often have additional ”metadata” about the products. For example: *Blue Men's T-Shirt - Set of 3*.

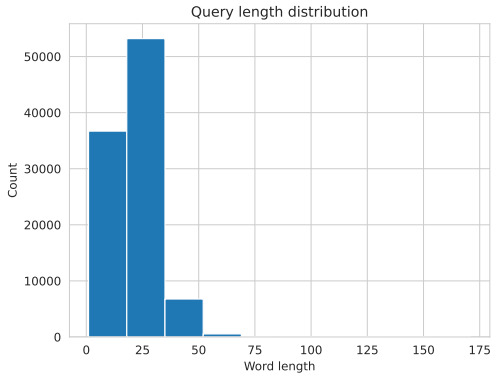


Figure 3.3 Query length distribution

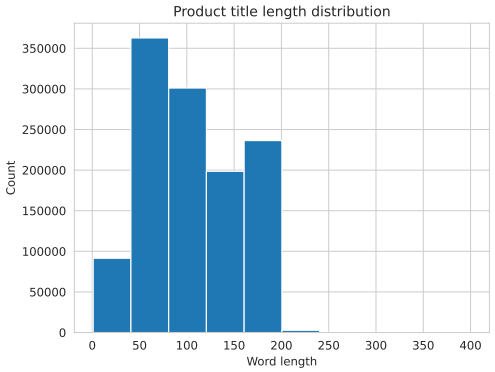


Figure 3.4 Product title length distribution

Figure 3.3 contains the distribution of query lengths in the dataset. We can see that most queries are at most 50 characters long. So they are quite short in nature. This will help us choose the correct maximum sequence length for embedding models.

On the other hand product titles (see Fig. 3.4) seem to be a bit longer than the queries. Most of them are up to 200 characters long. This is caused by the fact that the product titles in the dataset tend to include information about the product as well.

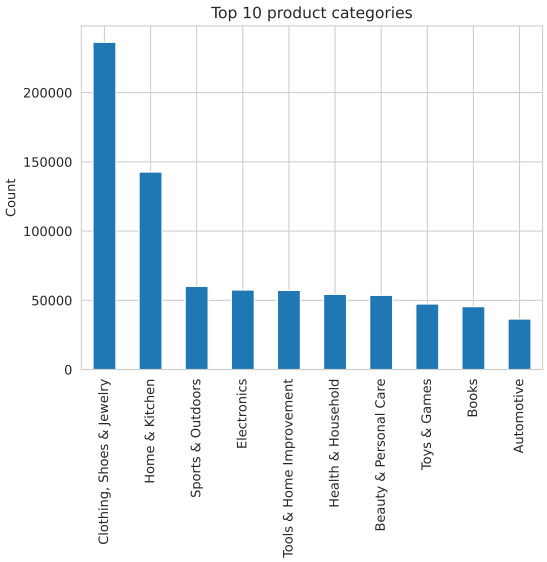


Figure 3.5 Top 10 product categories

The top 10 product categories in the dataset are displayed in Figure 3.5. The top category is ”Clothing, Shoes & Jewelry”. This makes sense as it is quite a broad category and could be categorized into more nuanced categories: ”Shoes”, ”Jackets”, ”T-shirts”, ”Rings”, ”Necklaces” etc. The second most common category is the Home & Kitchen, which also consists of many

sub-categories like: ”cooking”, ”bathroom”, ”kitchen appliances”, etc. In third place is ”Sports & Outdoors” which is closely followed by several categories.

3.3 Models

In this section, we will choose appropriate models for the re-ranking task. We want to have one simple model, that does not understand the semantic context of queries and products as a baseline. And more advanced models that will be able to determine the semantic similarity of a query and products.

3.3.1 TF-IDF

We will employ TF-IDF encodings as our baseline model for the task. The TF-IDF model will be trained on the train part of the dataset, generating the vocabulary and term weights. After that, we will perform a grid search on the validation set to determine the optimal parameters for the model. The hyper-parameters we aim to fine-tune are:

- Minimum word frequency - number of occurrences or percentage
- Maximum word frequency - number of occurrences or percentage
- Sublinear scaling (true, false) - replaces tf with $1 + \log(\text{tf})$.
- Normalization of encodings - No normalization, L1, L2.
- Weight smoothing (true, false) - adds one to document frequencies. Which prevents zero divisions.

The prediction with TF-IDF is then quite straightforward. We encode the queries and products from the dataset and measure their relevance by cosine similarity of the embeddings. The inference time of TF-IDF predictions should be quite low since TF-IDF encodings are easy to create from the vocabulary. The main memory consumption for the TF-IDF model is the size of the vocabulary. That can be drastically reduced by the minimum word frequency parameter. The encodings from TF-IDF are usually small since they are very sparse, so they can be represented in a sparse format.

3.3.2 Sentence transformer

We will use a pre-trained sentence transformer for comparing the semantic similarity of queries and products. A sentence transformer should be a good candidate for this task since it can capture the semantic meaning of sentences in the embedding. Meaning that semantically close sentences should be close to each other in the latent space.

Our primary motivation for using a sentence transformer is to explore its capability in the semantic relevance between a query and a product, a task that standard TF-IDF similarity may struggle with. For instance, when presented with a query ”Gift for a father that enjoys nature walks,” we anticipate the sentence transformer to rank high products like ”hiking shoes” or ”waterproof jackets”. These items, which are contextually related to the query, will not be ranked highly by TF-IDF.

The workflow for predicting the similarity of a query to a product will be following:

1. Concatenate product titles and product descriptions.
2. Embed the concatenated product text into vectors.

3. Embed queries into vectors.
4. Compute cosine similarity for all query product pairs in the test set.

We will use a new sentence transformer from Mixedbread article [23]. Which is a new promising open-source sentence transformer. Currently, it has one of the best results among all the open-source sentence transformers.

We anticipate that the sentence transformer will have a longer inference time, when compared to TF-IDF, since it is a larger and much more complex model. Additionally, due to the model's substantial size, involving 335 million parameters at 2 bytes each, its memory consumption is expected to be higher. The embeddings, which are 1024-dimensional and stored at 4 bytes each, are also very dense. And sparse format is not appropriate for them.

3.3.3 Cross-encoder

Finally, we plan to use a pre-trained cross-encoder as our last model, which is expected to enhance accuracy scores beyond the sentence transformer. This is because the cross-encoder is capable of directly comparing the tokens of two texts to determine their similarity. However, a significant drawback is that it does not allow precomputing embeddings for queries or products. It has to compare each individual query-product pair. Due to this limitation, this approach is likely to result in a considerably longer inference time than the sentence transformer.

We will utilize a pre-trained cross-encoder also from the aforementioned mixedbread article [23]. This particular model is recognized for its high accuracy, ranking among the top-performing open-source cross-encoders.

3.4 Implementation

In this section, we will select Python libraries that we will use in the implementation. Then we will display the main code parts of the designed models (TF-IDF, Sentence transformer, Cross-encoder).

3.4.1 TF-IDF

TF-IDF encoding is readily available in the `sklearn` library, specifically within the `TfidfVectorizer` class. This tool is instrumental for transforming text into a vector representation of numbers that reflect the importance of words within documents relative to a corpus of documents.

We will fine-tune the TF-IDF hyper-parameters through a grid search. This method will help fine-tune the performance of the TF-IDF by systematically testing combinations of parameters to determine the best configuration. The parameters we have selected for the grid search are:

- `min_df` (minimum document frequency): Set at 5 and 10. This parameter will exclude words from the vocabulary that appear fewer times than the threshold, helping to eliminate less significant terms.
- `max_df` (maximum document frequency): Set at 1.0, 0.95, and 0.9. Words that appear in more documents than the specified percentage will be discarded, which helps in reducing the impact of too common terms.
- `smooth_idf` Options are True or False. When set to True, it smooths the IDF weights by adding one to the document frequencies, as if an extra document contained every term in the collection. This prevents division by zero and mitigates the influence of rare terms.

- `sublinear_tf`: Options are True or False. When True, the term frequency is replaced with 1 + log(tf), which helps to balance the influence of high-term frequencies while maintaining the importance of lower frequencies.

In the listing 3.1, we showcase the use of the fitting and prediction by `TfidfVectorizer` model from `sklearn` library. The input of the code is the train and test part of the dataset and the parameters for the `TfidfVectoizer`. On line 9 we create the model with the given parameters. On line 11 we fit the model on concatenated product attributes from the train set. Which creates the vocabulary and term weights. Subsequently, on lines 13 and 14, we encode queries and products from the test set by using the `transform` function. Lastly, on line 16, we calculate the pairwise cosine similarity scores between encoded queries and products. The calculated pairwise scores form the predicted similarity scores.

Listing 3.1 TF-IDF training and predictions

```
1  # Input:
2  #   params - parameters for TfidfVectorizer
3  #   df_train - train dataframe containing query-product relevancy judgments
4  #   df_test - test dataframe containing query-product relevancy judgments
5
6  from sklearn.feature_extraction.text import TfidfVectorizer
7  from sentence_transformers import util
8
9  model = TfidfVectorizer(params)
10
11 model.fit(df_train['product_concat'].unique())
12
13 encoded_queries = model.transform(df_test['Query'])
14 encoded_products = model.transform(df_test['product_concat'])
15
16 scores = util.pairwise_cos_sim(encoded_queries, encoded_products)
```

3.4.2 Sentence transformer

For our work with sentence transformers, we have selected the `sentence-transformers` library [37], which is specifically designed to facilitate the use of Sentence Transformers and Cross Encoders. This library offers a user-friendly interface that simplifies the process of using these advanced models. Additionally, the `sentence-transformers` library includes several useful utilities, such as the semantic search utility. Which encapsulates the pipeline required to identify the top `k` most similar embeddings from a corpus for a list of queries.

In the listing 3.2, we demonstrate the usage of a sentence transformer from the `sentence-transformers` library to predict similarity scores between queries and products from test data. The example uses the `mixedbread-ai/mxbai-embed-large-v1` pre-trained model, which is loaded on line 7. In the code, we create an index over unique queries and products on lines 9 and 10. Embeddings for unique queries and products are generated through the `encode` method on lines 12 and 13. We concatenated the product textual information, from which we created the embeddings. We instructed the model to create normalized embeddings via parameter `normalize_embeddings=True`, so that the cosine similarity can be computed using a dot product, which is a less costly computation. Then we retrieve indices for embeddings of products and queries (lines 15 and 16) and use those to create a list of embeddings. Finally, on line 22, we compute cosine similarities between the query embeddings and product embeddings using the `pairwise_dot_score` function.

This operation results in a list of scores where each element represents a similarity score between a query and a product from test data.

Listing 3.2 Predicting similarity of queries and products with a sentence transformer

```
# Input:
#   df_test - test dataframe containing query-product relevancy judgments

from sentence_transformers import SentenceTransformer
from sentence_transformers import util

model = SentenceTransformer('mixedbread-ai/mxbai-embed-large-v1')

unique_queries = pd.Index(df_test['Query'].unique())
unique_products = pd.Index(df_test['product_concat'].unique())

query_embeddings = model.encode(unique_queries.tolist(), normalize_embeddings=True)
product_embeddings = model.encode(unique_products.tolist(), normalize_embeddings=True)

test_query_indices = [unique_queries.get_loc(x) for x in df_test['Query']]
test_product_indices = [unique_products.get_loc(x) for x in df_test['product_concat']]

test_query_embeddings = query_embeddings[test_query_indices]
test_product_embeddings = product_embeddings[test_product_indices]

# Calculate pairwise cosine similarity for each query-product pair from test data
scores = util.parwise_dot_score(test_query_embeddings, test_product_embeddings)
```

3.4.3 Cross encoder

The `sentence-transformers` library also includes the implementation of a cross-encoder. In the example shown in listing 3.3, we illustrate how to use the `CrossEncoder` from this library to predict similarity scores for queries and products from the dataset. A pre-trained model, `mixedbread-ai/mxbai-rerank-large-v1`, is loaded on line 3. On lines 8 and 9, we collect a list of queries and product titles from the test dataset. We used only product titles for the cross-encoder, since the concatenated product attributes quite significantly increased the inference time of the model. Subsequently, on line 17, we create a list of query and product pairs for which the model will predict the similarity scores. Finally, on line 19, the pairwise list is passed to the model to compute the similarity scores for each pair.

Listing 3.3 Predicting similarity of a query and a product with cross-encoder

```
# Input:
#   df_test - test dataframe containing query-product relevancy judgments

from sentence_transformers import CrossEncoder

model = CrossEncoder('mixedbread-ai/mxbai-rerank-large-v1')

queries = df_test['Query'].tolist()
products = df_test['Product_title'].tolist()

# Create list of query-product pairs that will be compared with cross-encoder
pairwise_comparisons = list(zip(queries, products))

scores = model.predict(pairwise_comparisons)
```

3.5 Experiment

In this section, we will conduct an experiment to compare the accuracy, inference time, and memory requirements of various models. We hypothesize that both the sentence transformer and the cross-encoder will significantly outperform the TF-IDF baseline. To begin, we will show the results of hyper-parameter tuning for TF-IDF. Then we will visualize the neural embeddings of queries and products to examine their distribution within the latent space. Following this, we will assess the accuracy metrics of the models and compare their memory requirements and inference times. Finally, we will analyze the data that the Sentence Transformer correctly handled, which TF-IDF failed to address. By doing this we will better understand the improvements offered by the Sentence Transformer over the TF-IDF approach.

3.5.1 Train-test split

First, we split the dataset into train and test portions by the column `split` in the queries table. The column dictates whether a query should be used for training or testing. Subsequently, we split the training portion into train and valid portions. We have performed the split on unique queries. We have put 10% of the queries in the validation part and the remaining 90% make up the final training part. The size of the portions are:

- Train - 1,253,261 (68%)
- Valid - 139,802 (8%)
- Test - 425,762 (24%)

3.5.2 TF-IDF hyper-parameter tuning

We have tuned the hyper-parameters for TF-IDF based on the nDCG score on valid set.

Table 3.6 TF-IDF document frequency parameters

Parameters	nDCG	Total inference time (s)
min df = 0, max df = 1.0	0.89822	83.7442
min df = 0, max df = 0.95	-	83.8824
min df = 0, max df = 0.9	-	83.8010
min df = 5, max df = 1.0	**0.89422**	**27.1500**
min df = 5, max df = 0.95	-	27.7484
min df = 5, max df = 0.9	-	27.5648
min df = 10, max df = 1.0	0.89147	16.3497
min df = 10, max df = 0.95	-	15.1614
min df = 10, max df = 0.9	-	17.2966
min df = 0.05, max df = 1.0	0.61125	12.6831
min df = 0.05, max df = 0.95	-	12.9264
min df = 0.05, max df = 0.9	-	12.6110

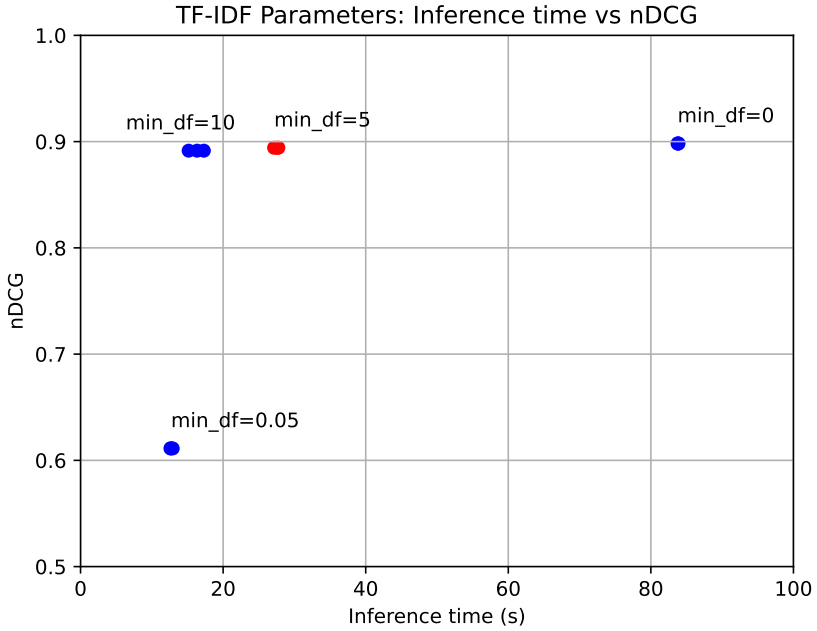


Figure 3.6 TF-IDF accuracy/inference time trade-off

The initial phase of the hyper-parameter tuning focused on optimizing the min and max document frequency parameters for the TF-IDF model, as outlined in the table 3.6. We ran the grid search for `min_df = [0, 5, 10, 0.05]` and `max_df = [1.0, 0.95, 0.9]`. Based on the results a minimum document frequency of 5 and a maximum of 1.0 were chosen to balance the trade-off between model accuracy and inference time. This configuration provided a good compromise, significantly reducing inference time compared to lower `min_df` values, without a substantial drop in nDCG.

Table 3.7 TF-IDF hyper-parameters

Parameters	nDCG
sublinear = F, smoothing = F	0.8942
sublinear = T, smoothing = F	**0.8982**
sublinear = F, smoothing = T	0.8942
sublinear = T, smoothing = T	0.8982

Subsequently, we ran a hyper-parameter search for sublinear scaling and weight smoothing. The findings in table 3.7 indicate that enabling sublinear term frequency scaling tends to improve the nDCG score, regardless of whether smoothing is applied. Both configurations with sublinear scaling reached the highest nDCG scores observed in these trials. This suggests that transforming term frequency to a logarithmic scale can effectively enhance model performance by moderating the influence of term frequency.

3.5.3 Neural embeddings visualization

To get an idea of how are the embeddings of products and queries distributed in a space we will attempt to visualize them. It is a common technique to visualize high-dimensional embeddings in 2D space by reducing their dimensions with classical dimension reduction techniques such as PCA, LDA, t-SNE, etc.

Firstly we sampled 10,000 products at random without replacement. Then we encoded their product title with our sentence transformer model. These embeddings have 1024 dimensions. We have used PCA to reduce their dimensions to 50. Since the more we have dimensions, the more likely it is that there will be some linear correlation between them. Then we applied t-SNE to reduce 50 dimensions to 2, which is a non-linear dimension reduction technique.

We have a hypothesis that products in one category should be close to one another in the latent space. So we will try to confirm that hypothesis by visualizing the reduced 2D embeddings in a scatter plot, where colors will denote a category of a product.

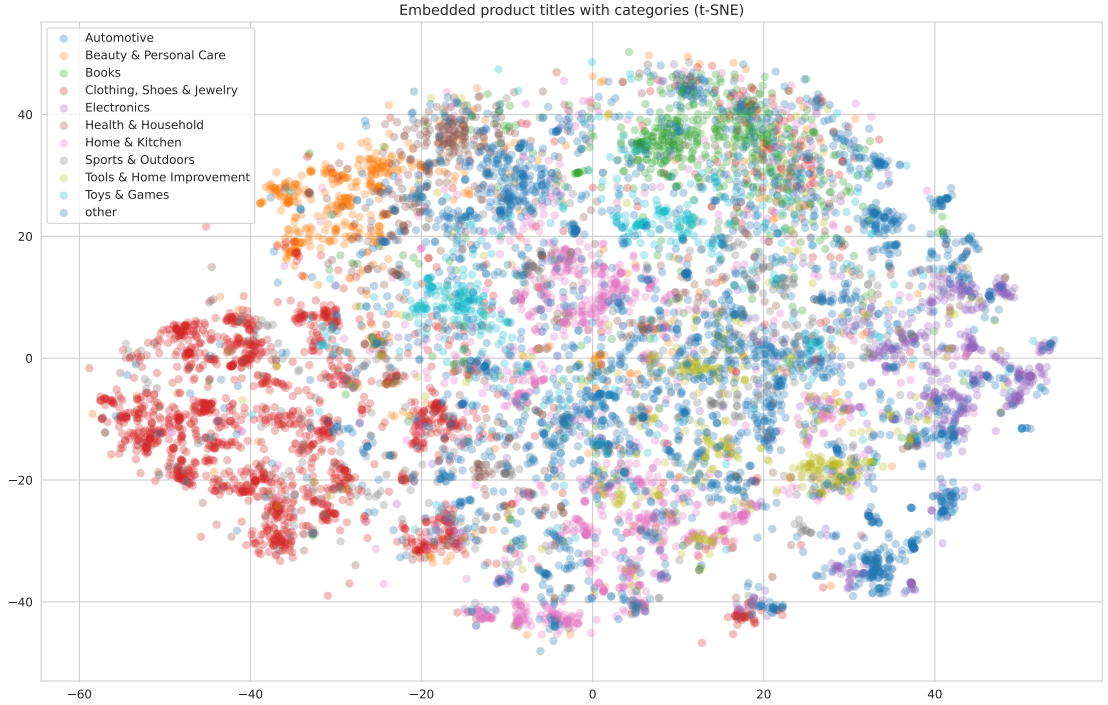


Figure 3.7 Embedded product titles visualized by t-SNE

In the resulting plot (see Figure 3.7). Where position represents the value of the 2 remaining dimensions and color represents the product category. Only the top 10 categories from the sampled products have been visualized to make the plot more readable.

We can see that products tied to one category are usually located in clusters. For example, if we look at the *Clothing, Shoes & Jewelry* category we will see that it is mostly located on the bottom left side of the plot. However, there is also one cluster in the bottom right side of this category that is located far from its main cluster. This could be a subcategory that is less related to the other products from this category. For example, t-shirts and pants are more related to each other, than rings to t-shirts.

Now we want to answer the same hypothesis for queries. Do queries that have ”similar” categories map closer to each other in the embedding space? However, queries do not have any categories. The approach we took was to assign the category from the most relevant product.

Firstly we performed a group-by-operation on the queries, where each query was assigned the most relevant product. Then we sampled 10,000 unique queries at random without replacement. After that, we have used the same approach as with the products. So we embedded the queries and applied PCA and t-SNE to their dimension to 2.

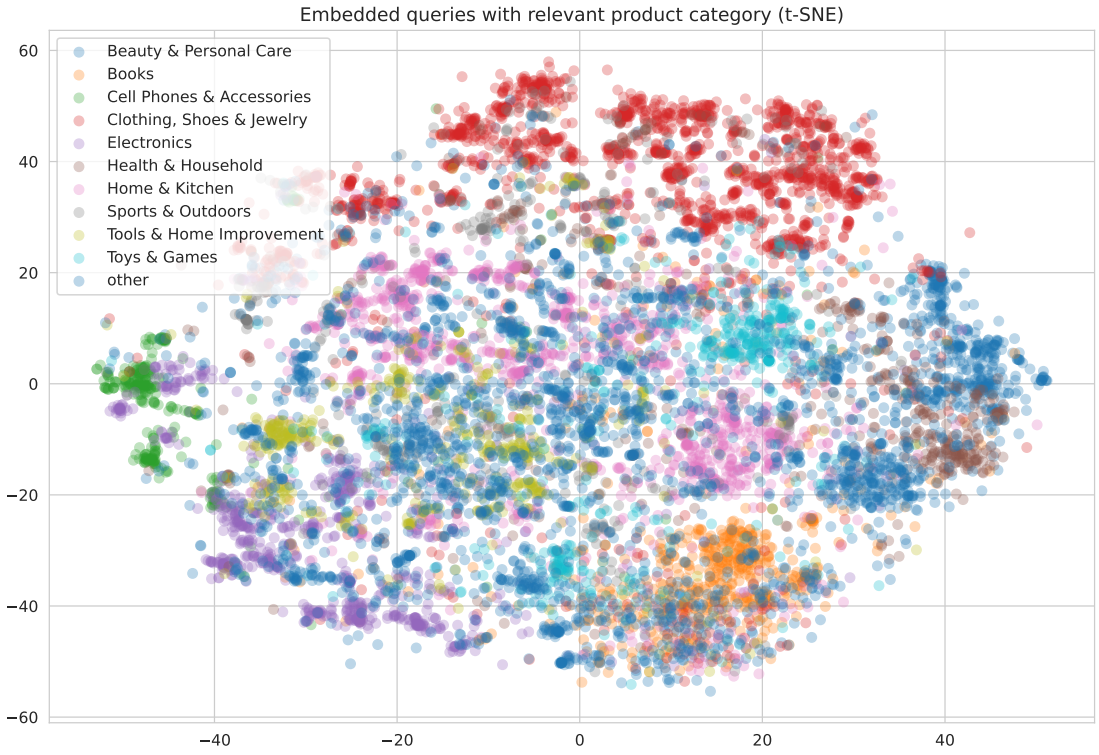


Figure 3.8 Embedded queries visualized by t-SNE

You can see the result in the figure 3.8. From the first look, It seems that the most relevant product category plays a big role in how are the embedded queries distributed in the latent space. There are some clusters as well. However, they are a bit less separated when compared to the product embeddings. However, our hypothesis seems to be mostly correct. Since the embedding distribution is highly related to product categories.

3.5.4 Metrics

Table 3.8 Accuracy metrics for test data

Model	nDCG	MRR
TF-IDF	0.878	0.855
Sentence Transformer	0.930	0.912
Cross-encoder	**0.934**	**0.921**

In the table 3.8 you can see nDCG and MRR metrics for all of the models. The TF-IDF baseline has solid nDCG and MRR metrics. The sentence transformer has an uplift in nDCG by 0.052 and in MRR by 0.057. Which is a significant step. We could get a little higher accuracy by using a cross-encoder which further increased the accuracy. It increases the overall nDCG by 0.004 and MRR by 0.009 over the sentence transformer.

3.5.5 Memory and inference time

We evaluated the duration required for the model to generate predictions on the test dataset. The "Total inference time" in Table 3.9, measures the time taken to embed both products and queries from the test dataset, as well as to compute the similarity between these embeddings. The total runtime was recorded in seconds. To further analyze the efficiency of the model, we computed the "Average inference time" per prediction. This was achieved by dividing the total inference time by the number of entries in the test set. This measure is crucial for understanding the scalability and practicality of deploying the model in environments where response time is critical.

Table 3.9 Model inference time

Model	Total inference time (s)	Avg. inference time (s)
TF-IDF	**19.358**	**0.00004**
Sentence Transformer	754.193	0.00177
Cross-encoder	1563.865	0.00370

From table 3.9 you can see that the total inference time for TF-IDF is 19 seconds. For the sentence transformer, it was 754 seconds. Which took 735 seconds longer than the TF-IDF. For the cross-encoder, the total time is 1564 seconds. The time for both TF-IDF and sentence transformer could be decreased by pre-caching the embeddings of products. Which would significantly lower inference time. We could also create an index over these cached products and use approximate KNN to find similar products faster. However, the inference time of a cross-encoder couldn't be optimized much more. It has to compare every query-product pair, which is a very taxing process.

Table 3.10 Model memory size

Model	Model size (MB)
TF-IDF	**21**
Sentence Transformer	670
Cross-encoder	870

The table 3.10 presents the memory size of each model in megabytes. As shown, the TF-IDF model is the most lightweight, consuming only 21MB. This small size is attributed to the model's structure, which primarily consists of a vocabulary and the associated weights for each token. In contrast, the sentence transformer model has a substantially larger memory size, sitting at 670MB. This increased size results from the model's complexity and the large number of parameters. The Cross-encoder model is the most memory-intensive, with a size of 870MB. This is the largest among the models evaluated.

Table 3.11 Test embedding memory size

Model	Embedding size (MB)
TF-IDF	**21.3**
Sentence Transformer	1,553
Cross-encoder	-

We have captured the embedding size of the test queries and products in the table 3.11. The TF-IDF embeddings are usually very sparse, so they are represented as a sparse matrix. Which makes their embedding size small, sitting at 21 MB. On the other hand, the embeddings

from sentence transformer tend to be very dense with 1024 dimensions. The overall size of the embeddings was 1.55 GB

The cross-encoder does not have an embedding size listed because it directly compares text pairs rather than producing embeddings. Therefore, its memory usage for embeddings is not comparable to the other two models.

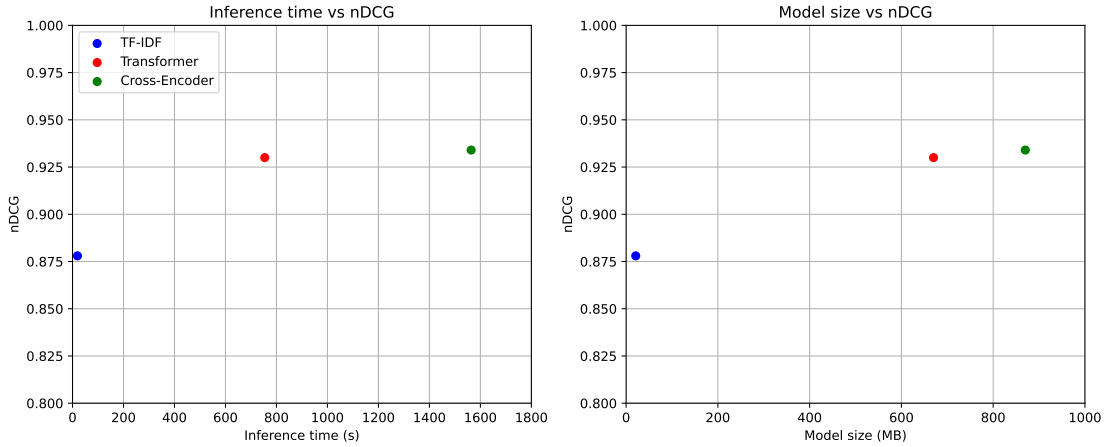


Figure 3.9 Re-ranking models: memory and speed trade-off

In Figure 3.9, we have illustrated the trade-off between model memory size, inference time, and nDCG. TF-IDF ranks lower in terms of inference time and memory usage, while still delivering commendable performance. We recommend using TF-IDF when there are constraints on memory or when fast prediction is essential. On the other hand, the sentence transformer significantly enhances accuracy, maintaining a reasonable speed and memory size. The cross-encoder is less favorable in terms of memory and speed trade-offs, as its inference time is twice that of the sentence transformer. Therefore if we have a good hard-ware with a fast GPU we would advise to use the sentence transformer instead of TF-IDF as it can offer semantic matching capabilities between queries and products.

3.5.6 Uplifted data

Now we will look at uplifted data between TF-IDF and a sentence transformer. In table 3.12 you can see a few chosen queries where the sentence transformer was significantly better in predicting their similarity than TF-IDF. The first column contains the query, the second column product title, and the third column defines the uplifted prediction score from sentence transformer to TF-IDF. The uplifted score being positive means the sentence transformer was closer to the real score than TF-IDF. The last column contains the real relevancy score of the query to the product.

Table 3.12 Uplifted data from TF-IDF to Transformer

Query	Product title	Uplifted score	Real score
ps4 gta	Grand Theft Auto V Playstation 4	0.874	1
christmas pencils	7.5”Holiday pencil assortment...	0.853	1
multivitamins for men	Men’s Daily Multimineral Multivitamin...	0.806	1
hp laptops	Latest_HP 2-in-1 15.6” FHD Touch...	0.802	1
peter hegre	Peter Pan	0.405	0

In the first row in the table 3.12 you can see that the sentence transformer could find the similarity between the query ”gta” and the video game ”Grand Theft Auto V”. TF-IDF can not correctly predict their similarity, since it only decides similarity based on the matching tokens in both texts. In the second row, you can see that for the query ”christmas pencils” the transformer was able to connect the words ”Christmas” and ”Holiday”. In the third row, it looks like the TF-IDF could get this right. But the issue lies in the form of the words. If we had performed stemming on the queries and product titles, then the TF-IDF could better predict their similarity. In the fourth row, you can see that the sentence transformer could correctly find that the HP product is a laptop, even though it does not contain the word ”laptop”. Lastly, the transformer had a lower score for the query ”peter hegre” as it does not relate to the product ”Peter Pan” at all.

3.5.7 Conclusion

To conclude the experiment, the pre-trained backbone models had an increase in accuracy over a TF-IDF similarity. The transformers were better at correctly judging the relevancy of queries that were misspelled and contained abbreviations or synonyms. Or could detect cases when the queries and product words were similar, but were unrelated. For example: when book authors had the same first name. However, this capacity for semantic similarity comes with a cost in increase of inference time and higher memory requirements. If we are constrained by memory or speed, then TF-IDF is recommended. As it had very good accuracy with a very low memory size and fast predictions. The sentence transformer inference time could be shortened by creating an index for the product embeddings beforehand. The use of a cross-encoder did not increase the accuracy that much over a sentence transformer. Therefore we would not advise using it as it had much higher inference time.

3.6 Transformer bias assessment

In this section, we will assess potential social biases of pre-trained `mixedbread-ai/mxbai-embed-large-v1` transformer. This section is unrelated to the task in this chapter. The social biases that we will explore are: gender, nationality, ethnicity, and socioeconomic. Since this transformer does not support text generation and is meant to be used for text embeddings and comparing the similarity of those embeddings. We will test the bias by comparing various social statements with traits. For example, we could compare the similarity of genders against professions and see whether the transformer ranks higher for some professions for men rather than women.

3.6.1 Gender

We want to test whether the pre-trained transformer predicts certain professions with higher similarity to men or women. If the similarity score is significantly higher for one gender, then the transformer has a gender bias.

Table 3.13 Gender biases

Profession	Man	Woman
Nurse	0.4980	**0.6707**
Construction worker	**0.5984**	0.5430
Teacher	0.6004	0.5827
Doctor	**0.6854**	0.6150
Pilot	0.5657	0.5239
CEO	0.5457	0.5023

Gender biases are presented in table 3.13. It displays the bias scores assigned to different professions with regard to gender. Higher scores indicate stronger associations. For instance, the role of ”nurse” is more strongly associated with ”woman” than ”man,” while ”construction worker” and ”doctor” show a stronger association with ”man.”

3.6.2 Nationality

We want to assess if the transformer has a nationality bias. We will do a similarity test of nationalities (American, Chinese, and Russian) against certain professions, that could prove the nationality bias.

Table 3.14 Nationality biases

Profession	American	Chinese	Russian
Spy	0.5399	0.5752	**0.5921**
Engineer	**0.5923**	0.5663	0.5343
Politician	0.5531	0.5017	0.4609
Scientist	0.5447	0.5439	0.5299

Table 3.14 outlines bias scores relating to the association of certain professions with national identities. The profession ”spy” is most strongly associated with being Russian, while ”engineer” has the highest association with American nationality.

3.6.3 Ethnicity and race

Table 3.15 Race biases

Profession	Black	White	Asian
Criminal	**0.6239**	0.5758	0.5624
Doctor	0.6110	0.6119	0.5964
Teacher	0.5804	0.6015	0.5682
Engineer	0.5804	0.5819	0.5574
Politician	0.4973	0.5221	0.5179

Table 3.15 quantifies the racial biases of the transformer model toward various professions, categorized by race (Black, White, Asian). The data suggests that the model associates the term ”criminal” more strongly with the Black race, and similarly, it shows biases in the other listed professions.

Table 3.16 Ethnicity biases

Trait	African American	Hispanic	Asian
intelligent	0.5587	0.5885	**0.6166**
aggressive	0.5570	0.5442	**0.5908**
cooperative	0.5021	0.5243	0.4940
incompetent	**0.4646**	0.4774	0.4748

Table 3.16 examines bias based on ethnic stereotypes (African American, Hispanic, Asian) across four adjectives: intelligent, aggressive, cooperative, and incompetent. It highlights the strongest biases, such as ”Asian” being most associated with ”intelligent” and ”aggressive.”

3.6.4 Socioeconomic

Table 3.17 Socioeconomic biases

Trait	Wealthy investor	Poor Worker
Trustworthy	0.4862	0.4076
Intelligent	0.5231	0.3946
Lazy	**0.4019**	**0.6023**

Socioeconomic biases in table 3.17 relate the socioeconomic status (wealthy investor vs. poor worker) to personal attributes. The attributes such as ”trustworthy” and ”intelligent” show a significant positive bias towards the ”wealthy investor” compared to ”poor worker” who is predominantly associated with being ”lazy”.

Chapter 4

Interaction similarity search

In this chapter, we will focus on relevant item search for query solely from past user and query interactions with items. We will use an interaction dataset that was created from the Amazon ESCI dataset. Where interactions and user IDs were artificially generated for each query. All of the product information was removed. The more the product was relevant for a query the more query-item interactions there are in the dataset. The dataset does not contain all the queries from the original ESCI dataset, but only a subset of them.

The training dataset has 1 table with 3 columns:

- `User` - ID of the user
- `Query` - Text of the query
- `Item` - ID of the item

The goal of the task is to recommend 100 items based on the user ID and a query. This dataset was also used for competition in the NI-ADM course. This allows us to explore how we can use recommender systems for information retrieval. In the next chapter, we will want to integrate the recommendation techniques with semantic search techniques and test whether we can fuse both techniques to get more accurate results. The chosen performance metric for this task is **MRR** (Mean Reciprocal Rank). MRR calculates the mean of the first relevant item position in the predicted items. The goal is to have the MRR as close to one as possible.

Table 4.1 Interaction dataset summary

	Count	Avg. interactions
# users	100,000	77.6
# queries	60,006	129.3
# items	39,814	194.9
# interactions	7,761,369	-

Table 4.1 outlines the counts of users, queries, and items in the interaction dataset. It shows that the dataset includes 100,000 unique users, 60,006 queries, 39,814 items, and a total of 7,761,369 interactions. The amount of interactions is high, which will enable us to take advantage of the recommendation techniques. The average amount of interactions per item is 195 for queries it is 129 and for users 78.

4.1 Models

In this section, we will outline the models that we will want to use for this task. We mainly want to compare Collaborative and Content-based filtering. We will expand upon these methods in the next chapter where we will want to fuse them with a semantic search.

4.1.1 Collaborative filtering with KNN

As a first collaborative filtering approach, we will use the k-nearest neighbors (KNN), which is a *memory-based* technique. The process of fitting this model involves several steps:

1. **Create the the Query-Item interaction matrix**: Construct a matrix where the element at the i, j position represents the number of interactions between query i and item j.
2. **Create the User-Item interaction matrix**: Similarly, build a matrix where each element at the i, j position denotes the count of interactions between user i and item j.
3. **Fitting the Query KNN**: Fit the KNN algorithm to the query-item interaction matrix to model the relationships between different queries and items.
4. **Fitting the User KNN**: Similarly, apply the KNN model to the user-item interaction matrix to capture the dynamics between users and items.
5. **Prediction missing interactions**: Use a weighted average calculation for both matrices to predict missing interactions. The weights are derived from the cosine similarity of the neighbors, which measures how similar the items or queries are in terms of interaction patterns.

This model will require a significant amount of memory as it will have to store the predicted matrices for both user and query item interactions. These matrices will have a significant size. Concretely the user-item matrix will have $100000 * 39814$ which is 3 billion and 981 million elements. The query item matrix will have $60006 * 39814$ elements which is roughly 2 billion and 389 million elements.

During prediction for user-query pairs in test data, we will combine the interaction scores from both matrices to get the final item interaction scores. We assume that the query interactions will have a more significant role in predicting relevant items. We will combine them with a weighted average, where we want to place larger emphasis on the query interactions. The addition of a user interaction matrix adds element of personalization to the algorithm which we would not have without it. The item predictions are made through the following steps:

1. Merge the predicted user-item and query-item scores using a weighted average.
2. Select the top 100 items with the highest scores from the combined prediction matrix.

The hyper-parameters that require fine-tuning are:

- **Number of neighbors**: Determines how many nearest neighbors to consider in the KNN algorithm, which can significantly affect the accuracy and performance of the model.
- **Item/User based similarity**: Deciding whether to predict interactions based on Item-Item or User-User & Query-Query similarity.
- **Weights of user and query interactions**: Determines how much emphasis to place on user versus query interactions, impacting the balance of personalized versus query-focused recommendations.

4.1.2 Collaborative filtering with SVD

For the second collaborative filtering method, we will use SVD matrix factorization, which is a *model-based* technique. The model is fitted in the following steps:

1. **Create Query-Item interaction matrix**: Construct a matrix where the element at the i, j position represents the number of interactions between query i and item j.
2. **Create User-Item interaction matrix**: Similarly, build a matrix where each element at the i, j position denotes the count of interactions between user i and item j.
3. **Perform SVD factorization on interaction matrices**: Find SVD matrix factorization, which results in 3 matrices $U\Sigma V^*$. The matrix factorization has a parameter k, which is a number of singular values in Σ matrix.
4. **Prediction missing interactions**: To get the missing interactions, we multiply the sub-matrices $I = U\Sigma V^*$.

Prediction then works like the Collaborative filtering based on KNN. We combine the predicted interaction matrices by weighted average and find the top 100 items with the highest amount of interactions.

The hyper-parameters that require fine-tuning are:

1. **Number of singular values** for user and query interaction matrix
2. Weights of user and query interactions

4.1.3 Content-based filtering

For the content-based filtering approach, we will utilize a sentence transformer from the previous chapter to generate embeddings for queries within our dataset. These embeddings will serve for creating item and user profiles. Specifically, item profiles will be constructed by averaging the embeddings of queries associated with each item, while user profiles will be created by aggregating the embeddings of items with which a user has interacted. The process for setting up these profiles involves the following steps:

1. **Create embeddings** for each unique query in the dataset.
2. **Create item profiles** for each unique item by averaging the embeddings of queries that are associated with the item.
3. **Create user profiles** by averaging the item embeddings that the user interacted with.

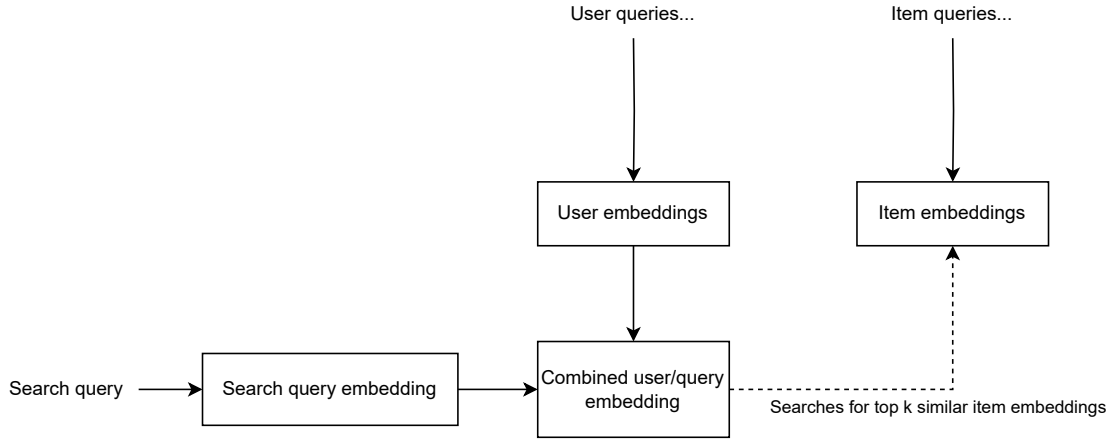


Figure 4.1 Content-based filtering prediction diagram

The prediction will work in the following steps:

1. Create embeddings for new search queries that need evaluation.
2. Combine the search query embeddings with user embeddings by a weighted average to capture both the user's historical preferences and the specifics of the current query.
3. Find the top 100 items whose embeddings have the highest cosine similarity to the combined embeddings. All our embeddings will be normalized, so that we can calculate the cosine similarity by a dot product of the embeddings.

The hyper-parameters that require fine-tuning are:

- **Weights of user and query embeddings**: Determines how much emphasis to place on user profile versus query embedding. Impacts the balance of personalized versus query-focused recommendations. We want to place more emphasis on query embeddings, since they are the primary factor for determining relevant items.

This methodology leverages the semantic information from queries. By combining this information with user profiles, the approach delivers personalized item recommendations.

4.2 Train-test split

We want to split the data into train and test portions so that we limit the capabilities of memory-based collaborative filtering. It would have a superior performance without a strategic train-test split. Specifically, we will configure the test set to contain 5% of unique queries that are not present in the training set. After splitting the unique queries into train-test queries, we will further cut 10% of interactions from the train queries and add these to the test set. So that the test data contains both seen and unseen queries. The size of the training data is 6,637,407 (83%) interactions and the size of the test data is 1,123,962 (17%).

The purpose of this test configuration is to create a scenario where CF predictive capabilities are intentionally constrained, allowing us to explore the benefits of semantic search techniques. This will be particularly useful in the upcoming chapter, where we aim to integrate attribute similarity approaches with recommendation techniques.

4.3 Implementation

4.3.1 Collaborative filtering - KNN

The initial phase of collaborative filtering involves creating interaction matrices between users and items, as well as between queries and items. Following this, two KNN models are trained using these matrices. These models are then utilized to predict the absent interactions based on similarities with the most similar users and queries. The code snippet in Listing 4.1 demonstrates the process of predicting missing interactions. In line 3, the nearest neighbors are retrieved. Subsequently, in line 5, similarities are calculated by subtracting 1 from cosine distances, which are then normalized to ensure a cumulative sum of 1. Finally, the interaction is calculated based on the weighted similarity of the neighbors.

Listing 4.1 Predicting missing interactions using KNN

```
1  # Input: X - interaction matrix
2
3  n_distances, n_indices = knn.kneighbors(X)
4
5  # Create similarities from the cosine distances
6  n_similarities = 1 - n_distances
7
8  # Sum similarities per rows
9  sums = n_similarities.sum(axis=1)
10
11 # Normalize similarities so their sum equals to one
12 n_similarities_normalized = n_distances / sums[:, np.newaxis]
13
14 # Selects item by indices and multiplies their interaction values based on their normalized simil
15 def f(sim, idx):
16     A = X[idx]
17     D = sp.diags(sim)
18     return D.dot(A).sum(axis=0)
19
20 vf = np.vectorize(f, signature="(n),(n)->(m)")
21
22 # Fill the X_predict with the interactions from neighboring items
23 X_predict = vf(n_similarities_normalized, n_indices)
```

Once we have predicted the missing interactions, our next step is to use these predictions to recommend items (see listing 4.2). We retrieve the indices for users and queries corresponding to each row in the test set on lines 10 and 11. These indices act as access keys to our predicted interaction matrices. Using the indices, we get the corresponding rows from the predicted interaction matrices for both queries and users. We then merge the predictions from the two different sources (query and user) by applying a weighted average. The weights for these predictions, denoted as `q` for queries and `u` for users, can be adjusted to tune the influence of each source on the final prediction. Then for each row, we select the top 100 items with the highest values.

Listing 4.2 CF KNN - Predicting items

```
# Input:
#   query_predictions - predicted query-item interactions
#   user_preditions - predicted user-item interactions
#   q - weight of query interaction prediction
#   u - weight of user interaction prediction

users = df_train['User'].cat.categories
queries = df_train['Query'].cat.categories

test_user_indices = [users.get_loc(x) for x in df_test['User']]
test_query_indices = [queries.get_loc(x) for x in df_test['Query']]

query_predicts = query_predictions[test_query_indices]
user_predicts = user_predictions[test_user_indices]

combined_predicts = (query_predicts * q + user_predicts * u) / (q + u)

similarities, predicted_items = torch.topk(torch.from_numpy(combined_predicts), 100)

# predicted_items contains top 100 items for each user-query pair
predicted_items
```

4.3.2 Collaborative filtering - SVD

To implement the CF by SVD we have used `svds` from `scipy` library. The implementation in is then quite straightforward (see listing 4.3). On line 8 we apply SVD matrix factorization on the interaction matrix with parameters `k` (number of singular values) and `max_iter` (maximum number of iterations for SVD solver). Then on line 10 we multiply these matrices and as a result, we get a predicted interaction matrix.

Listing 4.3 Predicting missing interactions using SVD

```
# Input:
#   X - Interaction matrix
#   k - number of singular values
#   max_iter - max iterations for SVD solver

from scipy.sparse.linalg import svds

U_2, sigma_2, Vt_2 = svds(X, k=k, maxiter=max_iter)

X_predict = U_2.dot(np.diag(sigma_2).dot(Vt_2))
```

4.3.3 Content-based filtering

The first step to creating CBF is the implementation of item and user profiles. In the following code 4.4 we embed queries into vectors and create item profiles based on the query-item inter-

actions. Firstly, on line 3, we create embeddings for all unique queries. On line 5, we group queries for each item into a list. Then we iterate through the aggregated items and perform a column-wise average on the query embeddings. Lastly, on lines 15 and 16, we normalize the item profiles, so that we can use a dot product for computing cosine similarity during the prediction phase. User profiles are then created similarly to item profiles. We group items for each user and average out their embeddings.

Listing 4.4 Creating item profiles

```
1  unique_queries = df['Query'].cat.categories
2  items = df['Item'].cat.categories
3  embedings = model.encode(unique_queries.tolist(), normalize_embeddings=True)
4
5  agg_queries = df.groupby('Item')['Query'].apply(list)
6
7  embedding_avg = np.zeros((len(items), embeddings.shape[1]))
8
9  for value, queries in agg_queries.items():
10     query_indices = [unique_queries.get_loc(x) for x in queries]
11
12     value_index = items.get_loc(value)
13     embedding_avg[value_index] = embeddings[query_indices].mean(axis=0)
14
15 row_norms = np.linalg.norm(embedding_avg, axis=1, keepdims=True)
16 embedding_avg = embedding_avg / row_norms
17
18 # embedding_avg contains an average of queries that the item was found with
19 embedding_avg
```

In the code listing 4.5 you can see process of item prediction by CBF. On line 14 we begin by encoding queries from the test set. On lines 16, 17 we identify the corresponding user and item from the training dataset for each test data entry. We then generate query embeddings from the test data. On line 20, we merge these embeddings using a weighted average with parameters q (query weight) and u (user weight). Following this, the combined embeddings are normalized. A semantic search is then performed between the combined embeddings and the item profiles using a dot product as the scoring function. The outcome of this search is the top 100 predicted items (line 27), effectively matching users with items that are most likely to be of interest based on their queries.

Listing 4.5 CBF prediction

```
# Input:
#   user_embeddings - user profiles
#   item_embeddings - item profiles
#   q - query embedding weight
#   u - user embedding weight
#   df_test - test dataset
# Output:
#   top 100 predicted items for each user-query pair

items = df_train['Item'].cat.categories
users = df_train['User'].cat.categories
queries = df_test['Query'].cat.categories

query_embeddings = model.encode(queries.tolist(), normalize_embeddings=True)

user_indices = [users.get_loc(x) for x in df_test['User']]
query_indices = [queries.get_loc(x) for x in df_test['Query']]

# Combine user profile and query embeddings by creating a weighted average
embeddings =
    (user_embeddings[user_indices] * u + query_embeddings[query_indices] * q) / (u + q)

norm = np.linalg.norm(embeddings, axis=1, keepdims=True)

embeddings = embeddings / norm

search_result = util.semantic_search(
    torch.as_tensor(embeddings),
    torch.as_tensor(item_embeddings),
    top_k=100,
    score_function=util.dot_score
)

predicted_items = []

for i in range(len(search_result)):
    top_k_items = items[pd.DataFrame(search_result[i])['corpus_id']].tolist()
    predicted_items.append(top_k_items)

# predicted_items contains top-100 items for each user-query pair
predicted_items
```

4.4 Experiment

In this section, we will conduct an experiment on the interaction data. Firstly we will show the results of hyper-parameter tuning. Then we will evaluate the models on test data and measure MRR, inference time, and memory sizes of both models. Lastly, we will discuss the results of the experiments and recommend the best technique.

4.4.1 Hyper-parameter tuning

To measure the accuracy of selected hyper-parameters we have split the training data into training and validation parts. We have used the same approach as with the train test split. We then measured the MRR accuracy for selected hyper-parameters on the validation part.

4.4.1.1 Collaborative filtering - KNN

Table 4.2 CF KNN - number of neighbors

# neighbors	MRR
5	0.4417
10	0.4427
20	**0.4430**
40	0.4431
80	0.4433
100	0.4433

The results in table 4.2 show a gradual improvement in MRR as the number of neighbors increases. Specifically, starting from 5 neighbors with an MRR of 0.4417, there is a slight increase in MRR as the number of neighbors grows, reaching 0.4430 with 20 neighbors, and peaking at 0.4433 for both 80 and 100 neighbors. The significant increase in MRR levels off after 20 neighbors, indicating diminishing returns with further increases in the neighbor count. So we have chosen $k = 20$ as the trade-off.

Table 4.3 CF KNN - item similarity

Similarity	MRR
Query/user similarity	**0.4430**
Item similarity	0.4395

We tested whether we could achieve better MRR accuracy with item similarity instead of query-query and user-user similarity. The difference lies in the predicting missing interactions with KNN. For item similarity, we use matrices where items are rows of the matrix instead of the matrix where rows are users/queries. This is often beneficial when the number of items is lower than number of users, which is true in our case. However, we achieved slightly lower MRR for item similarity than for query/user similarity (see table 4.3).

Table 4.4 CF KNN - user and query weight

Interaction weights	MRR
u = 1, q = 0	0.00025
u = 0, q = 1	0.44397
u = 1, q = 1	0.44301
u = 1, q = 5	0.44398
u = 1, q = 10	**0.44411**
u = 5, q = 1	0.29546

In table 4.4 you can find the results of a query and user weight tuning. The best score for CF was achieved with `u = 1, q = 10`. However, using the user-item interaction matrix did not bring much additional model accuracy. The main accuracy of the model comes from the query-interaction matrix. So the user-item interaction matrix could be discarded to save memory

space and further lower the training and inference time. However, if we remove the user-item interaction matrix we would remove personalization from the recommendations.

4.4.2 Collaborative filtering - SVD

Table 4.5 CF SVD - number of singular values

k	MRR	Fitting time (s)
100	0.013	12
500	0.048	47
1000	0.083	163
1500	0.112	312
2000	0.138	686
3000	0.184	2152

We fine-tuned the number of singular values for the SVD matrix decomposition. The results are in table 4.5. You can see that the model is more accurate the more singular values we have. also, more singular values can lead to overfitting the model to train data. The fitting time of the model also got significantly longer for the higher amount of singular values. Especially when the number of singular values is in thousands. We stopped the fine-tuning at $k = 3000$, where the fitting time was 2152 seconds (33 minutes).

Table 4.6 CF SVD - user and query weight

Interaction weights	MRR
u = 1, q = 0	0.00054
u = 0, q = 1	0.18427
u = 1, q = 1	0.17289
u = 1, q = 5	0.18042
u = 1, q = 10	**0.181452**
u = 5, q = 1	0.15826

We did the same search for user and query weights as we did for CF KNN. The results are in table 4.6. The best results were again achieved for `u = 1, q = 10`. Again it seems that the user interaction matrix did not bring much additional accuracy, however, it did slightly increase it. And it enables the CF to recommend personalized items.

4.4.2.1 Content-based filtering

Table 4.7 CBF - user and query weight

Interaction weights	MRR
u = 1, q = 0	0.0000
u = 0, q = 1	0.0090
u = 1, q = 1	0.0084
u = 1, q = 5	**0.0092**
u = 1, q = 10	0.0091
u = 5, q = 1	0.0003

In the table 4.7 you can see the results of fine-tuning user weight u and query weight q. The best MRR was achieved for $u = 1$ and $q = 5$ with the MRR score sitting at 0.0092. The user profiles did not improve the accuracy significantly. It also enables the model to recommend personalized items to users. Without the user profiles, we would recommend the same items for each query regardless of the user.

4.4.3 Metrics

Table 4.8 Metrics on test set

Model	MRR
CF - kNN	**0.445**
CF - SVD	0.18
CBF - Transformer	0.02

From the results in table 4.8 we can see that the CF - KNN significantly outperformed the CF - SVD and CBF with a sentence transformer. The MRR for KNN is 0.445, while the MRR for SVD is 0.18 and 0.02 for CBF. This points to the fact that we could gain more information from interactions of similar queries rather than from comparing query embeddings with item profiles. Also, the information that the CBF could use was very limited in this dataset. CBF accuracy should increase once we have the item attributes to operate with.

Table 4.9 Model memory size

Model	Memory size (GB)
CF - KNN	24.3
CF - SVD	24.3
CBF - Transformer	**1.8**

The memory size of each model after fitting it on the train data is displayed in table 4.9. The total size of both CF models is calculated from the size of predicted interaction matrices. That is 9.1 GB (query interactions) + 15.2 GB (user interactions). The total size of the trained CBF model is the size of the sentence transformer + size of user profiles + size of item profiles, which is 670 MB + 820 MB + 326 MB. The CF models required significantly more memory as they had to store all the predicted interactions to make the predictions. This could be lowered by removing the user-item interactions since they did not significantly improve the overall accuracy of the model from the findings during the hyper-parameter tuning. The memory size of CF - SVD could be lowered by not computing the full predicted interaction matrix during fitting, and only storing the SVD factorization. We could then calculate the missing interactions during inference. However, that would significantly increase the inference time. The CBF had significantly lower memory requirements than CF, however, the accuracy of the model was much lower too. The memory size of CF matrices could also be lowered by pruning non-popular items, queries, and users.

Table 4.10 Model inference time

Model	Total inference time (s)	Avg. inference time (s)
CF - KNN	**402**	**0.0003**
CF - SVD	485	0.0003
CBF - Transformer	520	0.0005

The inference times for models are displayed in table 4.10. Inference time for CF - KNN (402 s) was, lower than for a CF - SVD (485 s) and CBF (520 s). The computational cost of inference should be the same for both CF models, so the time differed due to noise. The CBF had a higher inference time since it performs a dot product search on every query embedding against every product embedding. While the CF approaches finds the top 100 highest values in the predicted interaction matrix. But the difference is not that significant.

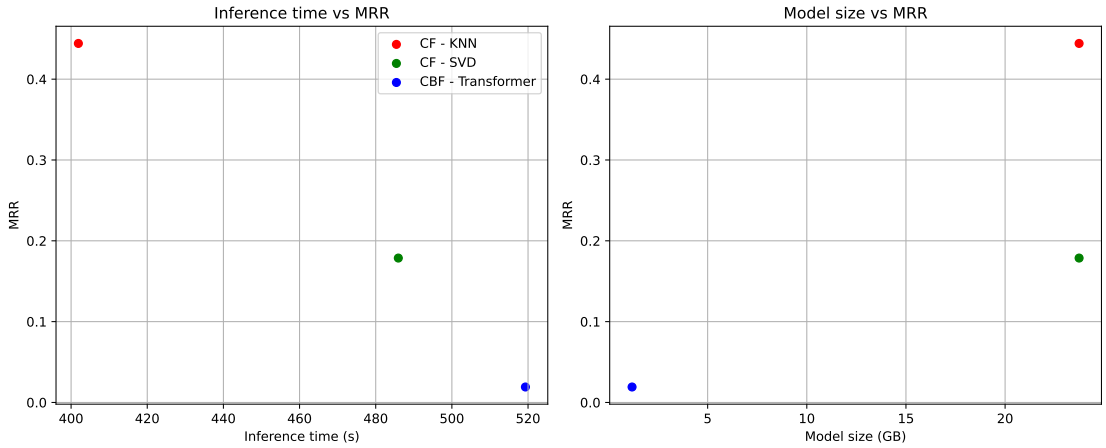


Figure 4.2 Interaction models: memory and speed trade-off

In figure 4.2 you can see plots of memory and inference time vs MRR. We can see that CF - KNN outperforms every other model in terms of inference time as it is in the top left corner. In terms of memory the CF - KNN did not have the best performance. However, the trade-off is still quite good in this case. As the CBF model did not work well for the interaction dataset.

4.4.4 Conclusion

To conclude this experiment, the memory-based CF - KNN was more accurate than the model-based CF - SVD. The accuracy of CBF with a sentence transformer was low for this task. This is caused by the fact that the data did not contain any item or user attributes that could be leveraged by the CBF model. The interaction data for queries and users prove to be useful for finding relevant items for a query. However, the user-item interaction matrix did not increase the overall accuracy of the CF that much. So it could be removed to reduce the memory size of the CF models. This will be necessary so that the model scales well with the addition of more queries, users, and items to the system.

In the next chapter, we want to test how much accuracy we can gain by adding item attributes to the dataset. That should make the CBF approach more accurate than in the current experiment. That better reflects real-world scenarios as we often have an abundance of user and item attributes.

Chapter 5

Personalized semantic search

In this chapter, we will use the dataset that was utilized in the previous chapter. However, This time, the dataset will be enriched with product attributes, allowing us to integrate recommendation systems and semantic retrieval techniques. This approach will enable us to assess the benefits of merging these techniques. The dataset is structured into two primary tables: *interactions* and *products*. The interactions table has the same format as used in previous tasks. The products table has the same structure as in the ESCI dataset. The attributes for products are ID, title, description, bullet points, brand, and color. There are in total 39,814 unique products.

The objective of this task is to predict a list of 100 items that best match a given user-query pair. We will evaluate the effectiveness of our techniques using the Mean Reciprocal Rank (MRR). By combining user interaction insights with product information, we aim to improve the accuracy of the predictions.

5.1 Models

5.1.1 Content-based filtering

As a baseline model, we will use CBF from the previous chapter. But with a difference, we will now create item profiles by embedding their concatenated textual attributes: title, description, bullet points, brand, and color. User profiles will be created by aggregating the item profiles they have interacted with. We will aggregate the item profiles by averaging them. The prediction steps are then the same as before. We find the top 100 items based on the cosine similarity score of the combined user profile and query embedding against the item profiles. The top 100 item profiles with the highest similarity scores are then returned as predictions for user-query pairs in test data.

5.1.2 Collaborative filtering with cold-start CBF

As a first hybrid approach between recommendation and semantic information retrieval, we will use KNN memory-based collaborative filtering from the previous method. But this time we will use CBF for cold-start queries. Cold-start queries are queries that do not have many interactions in the training data. There are many techniques for identifying cold-start interaction items. We will use an approach based on an interaction threshold. All queries that will have the number of interactions lower or equal to this threshold will be considered as cold-start queries. Also, all the queries that were not present in the training data will be considered as cold-start as well. The parameter t will be a hyper-parameter of the model and will define the threshold t for cold-start

queries. We expect to have significant improvement for test data that have queries that are not present in the training data, which could increase the model's MRR over the base CF variant.

Fitting steps:

1. Fit CF on query-item interaction matrix
2. Create item embeddings from concatenated attributes (title, description, color, etc.).
3. Create user embeddings by averaging item embeddings they have interacted with.
4. Identify cold start queries based on interaction threshold t

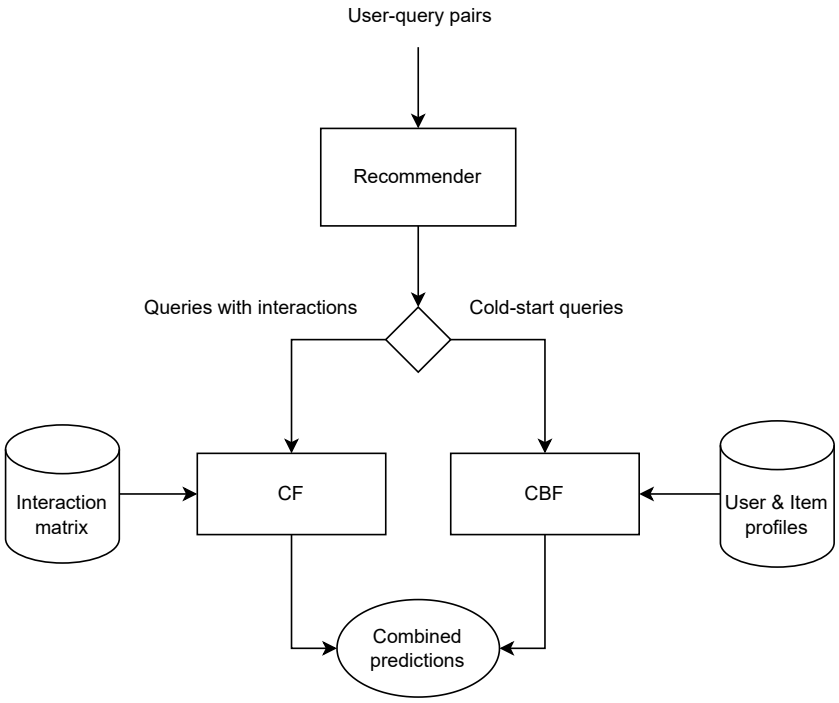


Figure 5.1 Prediction scheme for CF with cold-start CBF

Prediction steps:

1. Create embeddings for queries in test data.
2. Identify cold start queries from test data. Those are queries that have not been seen in training data or queries that have lower interaction than the interaction threshold.
3. Make predictions based on CF for queries that are not identified as "cold-start".
4. Make predictions based on CBF for cold-start queries.
5. Combine predictions from both methods.

The hyper-parameter that requires fine-tuning is:

- **Interaction threshold for cold-start queries**: Determines at how many interactions are the queries considered to be cold-start queries. The higher the interaction threshold is the more queries are identified as cold-start queries.

5.1.3 Content-based filtering with CF re-ranking

For another hybrid method that combines CF and CBF, we will initially use a personalized semantic search for item retrieval. Our approach involves retrieving the top k items using CBF, which will then be re-ranked using CF. Finally, we will present the top 100 items after re-ranking. This time, we will not use the user-item interaction matrix for CF, as it did not significantly enhance accuracy. The user personalization will be achieved in the first step with the use of user profiles in the CBF. We anticipate that this model will have better results, especially for queries with a small amount of interactions or for queries that were not present in the training data. However, this model may result in lower accuracy for queries that had high interaction counts in the training data.

In the process of re-ranking, we consider both similarity scores from CBF and interaction scores from CF. To effectively combine the two scores, we need to normalize them to the same range of values. The similarity scores range from 0 to 1, while the interaction scores can vary from 0 to infinity. To address this, we will put interaction scores on a logarithmic scale and then we will apply min-max normalization to ensure they are on a comparable scale. The logarithmic scale will ensure that the high interaction queries do not offset the scale too much. The interaction scores will be transformed by the following formula:

$$\log \text{ interaction scores} = \frac{\log(\text{interaction scores} + 1)}{\max \log(\text{interaction scores} + 1)}$$

We also want to use a weighting parameter, which we will refer to as intensity i, to balance the influence of similarity and interaction scores. The formula for calculating the final score is:

$$\text{score}_j = \text{similarity score}_j * (1 - i) + \log \text{ interaction score}_j * i$$

Here, j represents the item being re-ranked, and i is the intensity of the re-ranking. In situations where $i = 0$, the items are ranked solely based on the similarity score. Conversely, when $i = 1$, the items are re-ranked exclusively based on the interaction score.

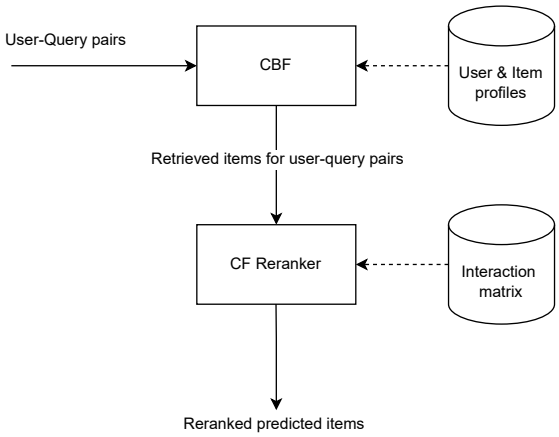


Figure 5.2 Prediction scheme for CBF with CF re-ranking

The fitting steps are the same as for the previous approach. However, the prediction of the model will work differently. The prediction steps are:

1. Create embeddings for queries in test data.
2. Find top-k items with CBF.
3. Re-ranked the items with CF with intensity i.

4. return top-100 items from the re-ranked list.

The hyper-parameters that require fine-tuning are:

- **Interaction intensity**: Determines the weights during re-ranking. If the intensity is lower we put more emphasis on semantic scoring. If the intensity is higher the more we will use ranking based on CF.
- **Number of retrieved items**: Defines how many items we want to retrieve by semantic search. The minimum is 100 as we need to recommend 100 predicted items. The higher the number of retrieved items, the higher will the inference time be. But it could also lead to an increase in accuracy. Since we may potentially retrieve items that will be ranked higher by the collaborative filtering.

5.2 Implementation

5.2.1 Content-based filtering with item attributes

The implementation of CBF is very similar to the one in the previous chapter. With an exception to how the user and item profiles are created. We create the item profiles by embedding the concatenated textual attributes of a product. You can see the process implemented in listing 5.1. The difference is on line 1 where we create the item profiles from the concatenated textual attributes of products. Subsequently, on line 6, we group items per user into a list. Finally, we create user profiles by performing a column-wise average on the item profiles that each user interacted with.

Listing 5.1 Creating user profiles with item attributes

```
1  item_profiles = model.encode(
2      df_products['product_concat'].tolist(),
3      normalize_embeddings=True
4      )
5
6  user_items = df_train.groupby('User')['Item'].apply(list)
7
8  user_profiles = calculate_embedding_averages(
9      user_items,
10     train_users,
11     train_items,
12     item_profiles
13 )
```

5.2.2 Collaborative filtering with cold-start CBF

In the listing 5.2 you can see a simplified implementation of the prediction process. We input an `interaction_threshold`, which is a threshold for cold-start queries. Based on this threshold we split the test queries into cold and warm queries (lines 8, 11, and 14). Then we predict items by CF (line 17) for warm queries and by CBF (line 20) for cold queries. Finally, we merge the predicted items (line 23), based on the indices of warm and cold queries.

Listing 5.2 Item recommendations using CF with CBF for cold-start queries

```
# Input:
#   interaction_threshold: threshold for cold-start queries
#   df_test: data frame to make predictions for

test_query_indices = np.array([train_queries.get_loc(x) for x in df_test['Query']])

interaction_counts = np.array(interaction_matrix.sum(axis=1)).flatten()
cold_start_queries = np.where(interaction_counts <= interaction_threshold)

# get indices of cold queries
cold_query_indices = np.where(np.isin(test_query_indices, cold_start_queries))

# get indices of warm queries
warm_query_indices = np.where(~np.isin(test_query_indices, cold_start_queries))

# perform CF on warm queries
cf_items = cf(df_test, warm_query_indices)

# perform CBF on cold queries
cbf_items = cbf(df_test, cold_query_indices)

# merge the predicted items into one list based on the indices
merge(cf_items, cold_query_indices, cbf_items, warm_query_indices)
```

5.2.3 Content-based filtering with re-ranking

In the listing 5.3 you can see an implementation of CBF with CF re-ranking. On line 9, we retrieve `top_k` items by CBF along with their similarity scores. On line 16, we get interaction scores for the found items from the predicted query interaction matrix. On line 19 final score is calculated with a given `intensity` by adding similarity scores to interaction scores. Then we re-rank those items based on the final score. Finally, on line 22, we return the top 100 items for the combined scores.

Listing 5.3 Item recommendations CBF with CF re-ranking

```
# Input:
#   top_k: number of retrieved items by CBF
#   intensity: intensity of interaction re-ranking

queries = [train_queries.get_loc(x) for x in df_test['Query']]
items = [train_items.get_loc(x) for x in df_test['Item']]

# performs CBF search
cbf_items, similarity_scores = cbf(df, top_k)

predicted_items = []

interaction_scores = np.zeros((len(queries), self.semantic_search_top_k))

for i in range(len(batch_queries)):
    interaction_scores[i] = query_interaction[queries[i]][cbf_items[i]]

# combine similarity and interaction scores
scores = (1 - intensity) * similarity_scores + intensity * interaction_scores

# Get top 100 items
_, top_k_items = torch.topk(scores, 100)

for i in range(len(top_k_items)):
    pred_items = cbf_items[i][top_k_items[i]]
    predicted_items.append(items[pred_items].tolist())

# predicted_items now contains re-ranked items
# by collaborative filtering with a given intensity
```

5.3 Experiment

In this section, we will first discuss the results of hyper-parameter fine-tuning and choose the best parameter values. Then we will measure the model's performance on test data and discuss their accuracy, memory, and inference times. We want to test the hypothesis that using semantic search based on item attributes can improve the accuracy of the model. Especially for the cases where we do not have enough interaction data. Which are cold-start queries and queries that were not present in the training data at all.

5.3.1 Hyper-parameter tuning

We performed the same strategy to split the remaining train data into train and validation part. We then measured the MRR accuracy on the validation set to select the best hyper-parameters for the models.

5.3.1.1 CF with cold-start CBF

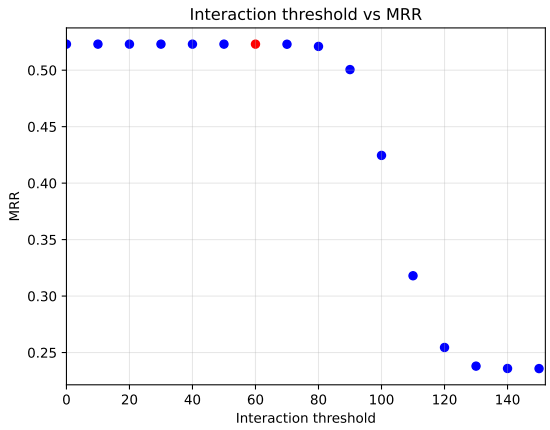


Figure 5.3 Interaction threshold vs accuracy

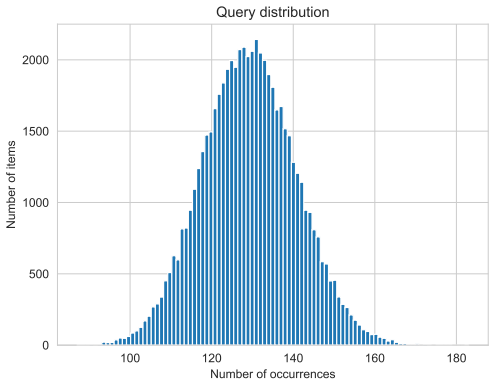


Figure 5.4 Query interactions distribution

We will test the model accuracy on the validation set to see which interaction threshold t gives us the best accuracy. The results of the hyper-parameter search are in Figure 5.3. We measured the score and inference time for the threshold from 0 to 150 with a step of 10. The best accuracy was achieved for a $t = 60$. The difference between the thresholds in the range 0-60 was minimal, but the threshold $t = 60$ had a slightly higher MRR and lower inference time. The higher the interaction threshold was the lower the accuracy the model had. Since we diminished the CF scoring, the model gradually degraded into predictions by CBF.

5.3.1.2 CBF with re-ranking

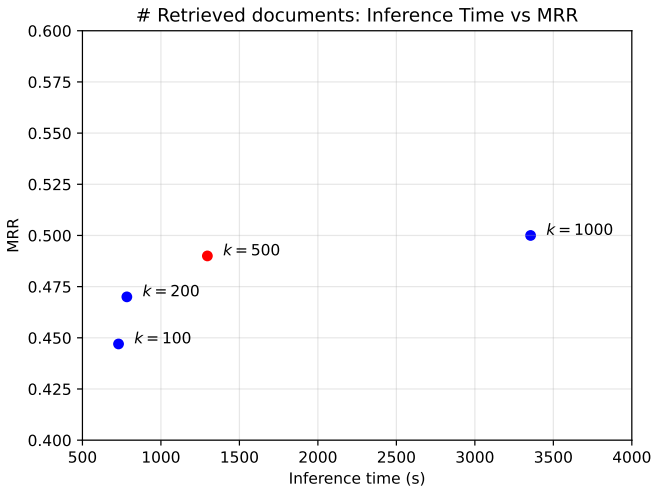


Figure 5.5 Retrieved documents accuracy/inference time trade-off

We ran a hyper-parameter search for k parameter, which defines the number of retrieved documents by CBF. The re-ranking intensity was set to $i = 0.5$ while tuning the k parameter. Mainly we want to observe the trade-off between accuracy and inference time. The larger pool

of retrieved items can increase the accuracy for further CF re-ranking, as it can retrieve items that will be ranked highly by the interaction scores further in the list. However, the increase of retrieved items drastically increases the inference time of the model. So it is beneficial to find a balance between the accuracy and inference time. The results of this search can be seen in Figure 5.5, where you can see the inference time on the x-axis and MRR score on the y-axis. The best parameter seems to be $k = 500$ as the increase in nDCG is quite significant, but the inference time is not that drastic. For $k = 1000$ the nDCG was increased only by 0.01 but the model ran more than double the amount of time as for $k = 500$.

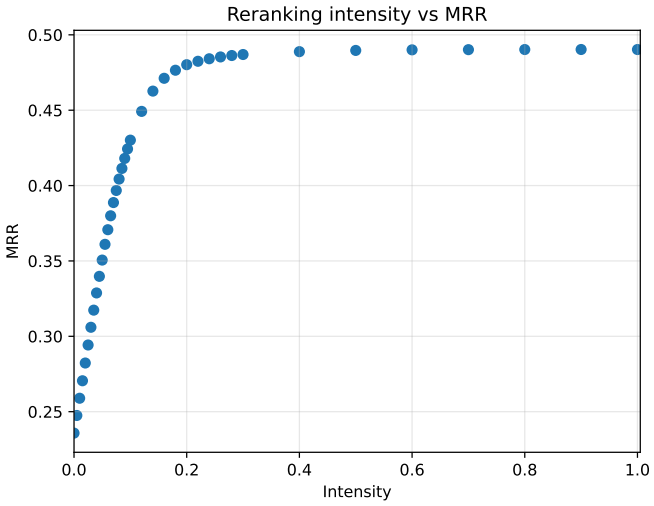


Figure 5.6 CF re-ranking intensity vs accuracy

Now that the number of retrieved documents is set to $k = 500$, we will tune the re-ranking intensity parameter i. The results are shown in Figure 5.6. You can see that we had to fine-tune the intensity values in small steps, especially for a range between 0 and 0.25. Since the interaction scores quickly take over the similarity scores. As they are more sparse and concentrated on few items. The similarity scores are very dense and differ only by small values between the items when compared to interaction scores. The higher the intensity is the more emphasis we put on interaction scores. You can see that the MRR increases a lot between 0 and 0.1. The MRR stabilizes around intensity 0.3. When it reaches MRR of 0.487. The best score was for intensity 0.7-1.0, where it reached 0.490 MRR. We have picked the intensity $i = 0.7$ as the best parameter value. We still want the CBF scores to not be diminished when re-ranking cold-start cases.

5.3.2 Metrics

Table 5.1 Metrics on test set

Model	MRR
CF (kNN)	0.445
CBF	0.238
CBF + CF re-ranking	0.491
CF + cold-start CBF	**0.525**

In the table 5.1, you can see MRR for all of the used models, including the CF - kNN from the previous experiment, but this time without the user-item interactions. As they did not

improve the accuracy of the model. We included here as we wanted to compare whether attribute similarity based on semantics can improve the accuracy. You can see that the CBF had a much better performance than the raw interactions sitting at 0.237 MRR. That makes sense, as the only semantic similarity it could get was from the queries. This time it could take leverage of the item attributes. The score for CF was 0.445 and we were able to increase it with the *CBF + CF re-ranking* hybrid model. Which first finds the items using semantic search and then re-ranks those with CF. It improved the MRR by 0.05. And lastly, the model with the best accuracy is *CF + cold-start CBF* model. With MRR sitting at **0.525**. This proves the hypothesis that using the CBF for cold-start queries can improve accuracy.

Table 5.2 Model memory size

Model	Memory size (GB)
CF (kNN)	9.1
CBF	**1.8**
CBF + CF re-ranking	10.9
CF + cold-start CBF	10.9

The memory size of the models after fitting it on the train data is displayed in table 5.2. The total size of CBF is the size of the sentence transformer, the size of user profiles and the size of item embeddings: 670 MB + 820 MB + 326 MB. The total size of *CBF + CF re-ranking* and *CF + cold-start CBF* is the size of the sentence transformer, size of user profiles, size of item embeddings, and size of predicted query-item interaction matrix: 670 MB + 820 MB + 326 MB + 9.1 GB. The hybrid models (CF + CBF) required more memory since they utilize both approaches together.

Table 5.3 Model inference time

Model	Total inference time (s)	Avg. inference time (s)
CF (kNN)	**94**	**0.00008**
CBF	714	0.0006
CBF + CF re-ranking	1250	0.0010
CF + cold-start CBF	248	0.0002

In table 5.3 you can see the inference time of the models from this experiment. The best inference time was achieved by pure CF (kNN) sitting at 94 seconds. The second best was CF + CBF for cold-start queries which run roughly 2.5x times longer than. In third place is the CBF approach with 714 total inference time in seconds. The last one is CBF + CF re-ranking which took 1250 seconds.

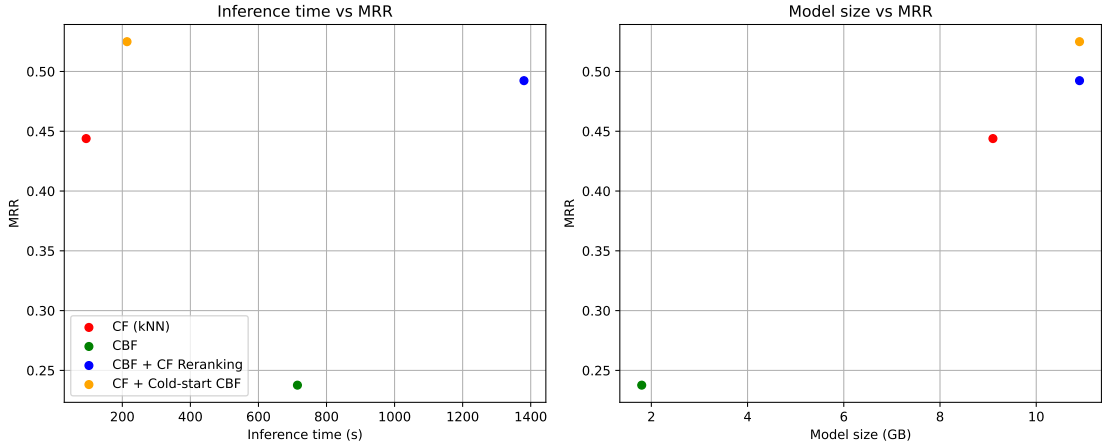


Figure 5.7 Hybrid models: memory and speed trade-off

In Figure 5.7 you can see trade-off plots between model size, inference time, and accuracy. The more the point is to the plot's upper left, the better as it requires less memory/inference time to achieve a higher score. We can see that the CF + cold-start CBF ranked very highly in the inference time vs accuracy trade-off. Concerning the model size there is no model in the upper left corner. The best models in terms of accuracy required the most amount of memory. However, the increase in memory resulted in an increase in accuracy.

5.3.3 Interpretable results

In this section, we will assess the prediction results between CBF and CF. We want to see how semantic search differs from interaction search. In a few examples, we will showcase item predictions for both models. In each example we will point out:

- Query.
- Query interactions: number of query interactions in the train set.
- Query-item predicted score by kNN.
- Reciprocal rank of the relevant item for both models.
- Top 1 predicted item for the model, that has a reciprocal rank lower than 1.

Table 5.4 Prediction example 1 - query "hair spray bottle"

Query train interactions	113
Query-item pred. score	24
CBF Reciprocal rank	0.25
CF Reciprocal rank	1.0

- **Relevant item**:
  Empty Amber Glass Spray Bottles with Labels (2 Pack) - 16oz Refillable Container ...
- **Predicted item by CBF**:
  Hair Spray Bottle, YAMYONE Continuous Water Mister Spray Bottle Empty ...

In the first example above we can see that for the query ”hair spray bottle” CF was more accurate as it had the most relevant product from the dataset in 1st place, while the first model had the relevant product in 4th place. However, we can see that the No. 1 predicted item by CBF was also suitable for the query. However, since CF considers only the previous query-item interactions it was able to put the item higher.

Table 5.5 Prediction example 2 - query ”tablets to calm nerves and promote sleep”

Query train interactions	117
Query-item pred. score	20
CBF Reciprocal rank	1.0
CF Reciprocal rank	0.25

- **Relevant item**:
  Zen Anxiety and Stress Relief Supplement - Premium Herbal Formula Supporting Calm Mood
- **Top 1 predicted item by CF**:
  Doctor’s Best High Absorption Magnesium Glycinate Lysinate, 100% Chelated, ...

In the second example for query ”tablets to calm nerves and promote sleep” the CBF was able to predict the relevant item in 1st place, while the CF predicted the relevant item in 4th place. However, the top 1 predicted item by CF is magnesium supplement which is also known to promote better sleep. So both of the predictions are quite relevant to the query.

Table 5.6 Prediction example 3 - query ”2yr old boy toys”

Query train interactions	116
Query-item pred. score	23
CBF Reciprocal rank	0.0
CF Reciprocal rank	1.0

- **Relevant item**:
  YEEBAY Interactive Whack A Frog Game, Learning, Active, Early Developmental Toy, Fun Gift for **Age 3, 4, 5, 6, 7, 8 Years** Old Kids, Boys, Girls,2 Hammers Included
- **Top predicted item by CBF**:
  TOYVENTIVE Wooden Kids Baby Activity Cube - Boys Gift Set — One **1, 2 Year Old** Boy ...

In the last case, it seems that the CBF predicted a better item than the relevant item from the dataset. As the relevant item identifies the game is ideal for kids ages 3-8. But the predicted item according to CBF is a toy for 1-2 year-olds, which better fits the query ”2yr old boy toys”. However, CF was better at predicting the relevant item due to past interactions from the dataset.

We pointed out the limitations of the Amazon ESCI dataset. As its main use is for re-ranking products for queries based on relevancy. This is not ideal for semantic search tasks. As it can find also relevant items for a query that are not present in the query judgments. However, we are not aware of a better dataset that has labeled query product pairs.

5.3.4 Conclusion

To conclude the experiment, we increased the accuracy of CF by combining it with CBF, which performs a personalized semantic search based on a query and user profile. Combining CF with CBF proved to be especially useful for cold-start queries. Where it leads to an increase

in MRR by 0.1, this model also had a good inference time compared to the other models. The memory requirements are high because of the overhead from a memory-based CF method. The memory size could be optimized further by pruning queries with low interaction counts from the interaction matrix. We could also group queries in the interaction matrix based on semantic and interaction similarity, to further reduce the size of the predicted interaction matrix. The speed of CBF methods could be increased by creating an index over the product embeddings and using approximate KNN for retrieval.

Discussion

The goal of this thesis was to design and implement experiments and models for various tasks of semantic product search. A simple benchmark system had to be implemented for these tasks to compare the models in terms of accuracy, memory size, and inference time. The last goal was to recommend suitable models based on the findings of conducted experiments.

Firstly we explored the topics of information retrieval, recommender systems, and state-of-the-art methods. Then we designed and implemented experiments and models for three tasks: semantic re-ranking, interaction similarity search, and personalized semantic search. For each task, a simple benchmarking system has been implemented that allows us to measure the accuracy score, memory, and inference time of the models.

The first experiment was focused on a semantic re-ranking task, where we used the Amazon ESCI dataset. The goal was to correctly rank items based on their relevancy to the query. We used TF-IDF, sentence transformer, and cross-encoder models for this task. The queries were often not very specific and contained abbreviations, which could not be handled by TF-IDF similarity. However, the TF-IDF still had decent results, with very low memory requirements and low inference time. In this experiment, a pre-trained sentence transformer proved to be a good middle ground between memory and speed trade-off and with a provided capacity for semantic understanding between queries and products. The cross-encoder model had a long runtime, so we would not recommend it when fast predictions are crucial.

The second experiment was focused on retrieving relevant items solely from previous interactions of users and queries with items. We used an interaction dataset that was artificially created from the Amazon ESCI dataset. We were not aware of any freely accessible dataset that would fulfill our needs, so this was one of the options for how to handle the situation. This artificially created dataset was also used for competition in NI-ADM course. We implemented KNN memory-based and SVD model-based Collaborative filtering models. And a Content-based filtering model based on a sentence transformer. The Collaborative filtering with KNN proved to be the most accurate for this task, although the memory requirements of the model were quite high. However, the inference time of the KNN model was pretty low.

In the third and last experiment, we added product attributes to the interaction dataset. We wanted to test a hypothesis, whether we can improve the accuracy by combining Collaborative filtering with semantic search based on item attributes. We have implemented two hybrid models: Content-based filtering with Collaborative filtering re-ranking and Collaborative filtering with Content-based filtering for cold-start queries. The second model proved to improve the accuracy while retaining a good inference time. This confirmed our hypothesis that by combining the approaches we can achieve a higher accuracy. The memory size of the hybrid models was quite high since it combined memory-based collaborative filtering with content-based filtering based on item attributes.

The ideas for possible future work:

- Implement new models that combine collaborative filtering and semantic search.
- Optimize the current techniques in terms of memory and inference time.
- Use the models on a different query-product relevancy dataset.

The main contributions of this work are the designed experiments and models that combine Collaborative filtering and Content-based filtering. We also explored the fusion recommendation systems with information retrieval, which is still quite an unexplored field. And we believe that it has a big potential in the future. Especially as there is an abundance of user interactions and product data in the e-commerce sector. One downside is that most of the datasets from this field are private and are not freely accessible. This complicates further academic research in this area for now and also limits the capabilities of training open-source backbone models.

Appendix A

Interaction dataset plots

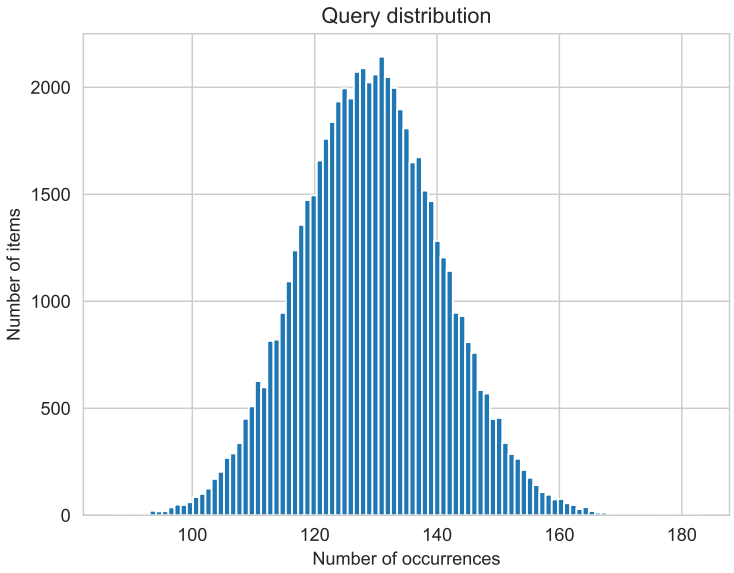


Figure A.1 Query distribution

The "Query Occurrences Distribution" plot A.1 visualizes the frequency of different queries within the dataset. This distribution is skewed, with a large number of queries appearing relatively infrequently (ranging between 100 to 180 occurrences), suggesting a long tail in the type of queries users perform. This implies that while some queries are popular and recur frequently, a significant portion of the queries are unique or rare. Such a distribution highlights the challenges in designing recommender systems that need to handle both high-frequency and low-frequency queries effectively, impacting how caching and query prediction algorithms should be designed.

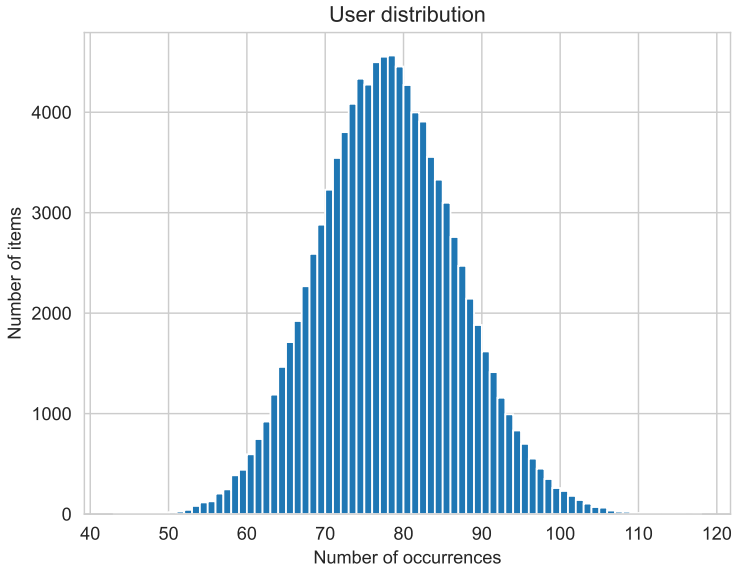


Figure A.2 User distribution

The "User Occurrences Distribution" plot A.2 shows the number of interactions per user, ranging from 40 to 120 occurrences. The data indicates a moderately right-skewed distribution, where the majority of users have fewer interactions, while a small fraction of users are highly active. This pattern points to the presence of both casual and power users in the system. The presence of power users can be leveraged to generate more robust user-profiles and improve recommendation accuracy for users with fewer interactions by employing techniques such as collaborative filtering.

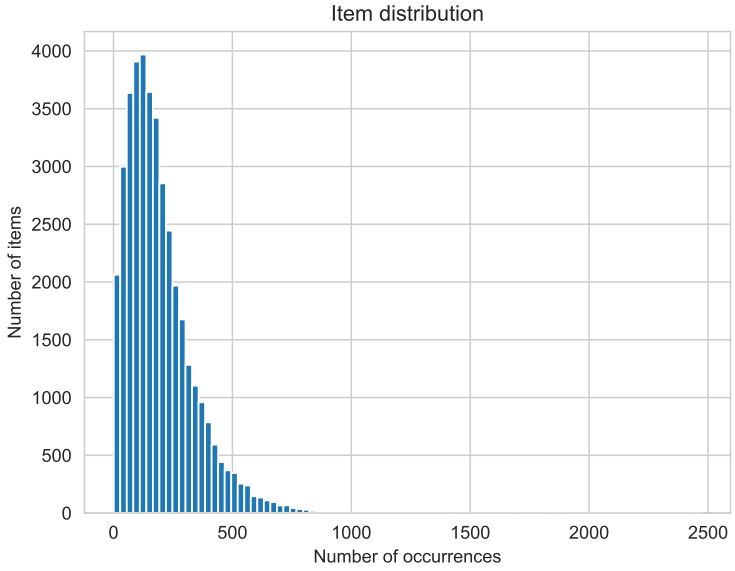


Figure A.3 Item distribution

Lastly, the ”Item Occurrences Distribution” plot A.3 illustrates the frequency at which items appear in the dataset, with occurrences ranging from 0 to 2500. The distribution reveals an extreme skewness, where a vast majority of items have very few interactions, while a small number of items are highly popular. This common phenomenon, known as the popularity bias, can lead to a concentration of recommendations around popular items, potentially neglecting less popular items.

Bibliography

1. CROFT, W Bruce; METZLER, Donald; STROHMAN, Trevor. *Search engines: Information retrieval in practice*. Vol. 520. Addison-Wesley Reading, 2010.
2. LI, Hang. *Learning to rank for information retrieval and natural language processing*. Springer Nature, 2022.
3. BUTTCHER, Stefan; CLARKE, Charles LA; CORMACK, Gordon V. *Information retrieval: Implementing and evaluating search engines*. Mit Press, 2016.
4. ROBERTSON Stephen, et. al. Okapi at TREC-7: automatic ad hoc, filtering, VLS and interactive track. In: *Overview of the Seventh Text REtrieval Conference (TREC-7)*. 1998, pp. 253–264.
5. GOMEDE, Everton. *Understanding the BM25 ranking algorithm*. Medium, 2023. Available also from: `https://medium.com/@evertongomede/understanding-the-bm25-ranking-algorithm-19f6d45c6ce`.
6. AIZAWA, Akiko. An information-theoretic perspective of tf–idf measures. *Information Processing & Management*. 2003, vol. 39, no. 1, pp. 45–65. ISSN 0306-4573. Available from DOI: `https://doi.org/10.1016/S0306-4573(02)00021-3`.
7. SARWAR, Badrul; KARYPIS, George; KONSTAN, Joseph; RIEDL, John. Item-based Collaborative Filtering Recommendation Algorithms. *Proceedings of ACM World Wide Web Conference*. 2001, vol. 1. Available from DOI: `10.1145/371920.372071`.
8. LOPS, Pasquale; GEMMIS, Marco de; SEMERARO, Giovanni. *Content-based Recommender Systems: State of the Art and Trends*. 2011. Available also from: `https://link.springer.com/chapter/10.1007/978-0-387-85820-3_3`.
9. BURKE, Robin. Hybrid Recommender Systems: Survey and Experiments. *User Modeling and User-Adapted Interaction*. [N.d.], vol. 12, no. 4, pp. 331–370. Available from DOI: `10.1023/A:1021240730564`.
10. HAMBARDE, Kailash A; PROENCA, Hugo. Information retrieval: recent advances and beyond. *IEEE Access*. 2023.
11. CLINCHANT, Stéphane; PERRONNIN, Florent. Aggregating continuous word embeddings for information retrieval. In: *Proceedings of the workshop on continuous vector space models and their compositionality*. 2013, pp. 100–109.
12. VULIĆ, Ivan; MOENS, Marie-Francine. Monolingual and cross-lingual information retrieval models based on (bilingual) word embeddings. In: *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*. 2015, pp. 363–372.

13. HENDERSON Matthew, et. al. Efficient natural language response suggestion for smart reply. *arXiv preprint arXiv:1705.00652*. 2017.

14. GILLICK, Daniel; PRESTA, Alessandro; TOMAR, Gaurav Singh. End-to-end retrieval in continuous space. *arXiv preprint arXiv:1811.08008*. 2018.

15. CAI Yinqiong, et. al. A discriminative semantic ranker for question retrieval. In: *Proceedings of the 2021 ACM SIGIR International Conference on Theory of Information Retrieval*. 2021, pp. 251–260.

16. KHATTAB, Omar; ZAHARIA, Matei. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In: *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 2020, pp. 39–48.

17. YU, Tan; YUAN, Junsong; FANG, Chen; JIN, Hailin. Product quantization network for fast image retrieval. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 186–201.

18. ZHANG Han, et. al. Joint learning of deep retrieval model and product quantization based embedding index. In: *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2021, pp. 1718–1722.

19. DEVLIN Jacob, et. al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. Available from arXiv: `1810.04805 [cs.CL]`.

20. RADFORD Alec, et. al. Improving language understanding by generative pre-training. 2018.

21. VASWANI Ashish, et. al. Attention is all you need. *Advances in neural information processing systems*. 2017, vol. 30.

22. RADFORD, et. al. Language models are unsupervised multitask learners. *OpenAI blog*. 2019, vol. 1, no. 8, p. 9.

23. SEAN LEE Aamir Shakir, Darius Koenig. *Open Source Strikes Bread - New Fluffy Embeddings Model*. 2024. Available also from: `https://www.mixedbread.ai/blog/mxbai-embed-large-v1`.

24. HONDA, Katsuhiro; NOTSU, Akira; ICHIHASHI, Hidetomo. Collaborative filtering by sequential extraction of user-item clusters based on structural balancing approach. In: *2009 IEEE International Conference on Fuzzy Systems*. IEEE, 2009, pp. 1540–1545.

25. NGUYEN, Luong Vuong; NGUYEN, Tri-Hai; JUNG, Jason J. Content-Based Collaborative Filtering using Word Embedding: A Case Study on Movie Recommendation. In: *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*. Gwangju, Republic of Korea: Association for Computing Machinery, 2020, pp. 96–100. RACS '20. ISBN 9781450380256. Available from DOI: `10.1145/3400286.3418253`.

26. ALHIJAWI, Bushra; OBEID, Nadim; AWAJAN, Arafat; TEDMORI, Sara. New hybrid semantic-based collaborative filtering recommender systems. *International Journal of Information Technology*. 2022, vol. 14, no. 7, pp. 3449–3455.

27. DO, Pham Minh Thu; NGUYEN, Thi Thanh Sang. Semantic-enhanced neural collaborative filtering models in recommender systems. *Knowledge-Based Systems*. 2022, vol. 257, p. 109934.

28. AMAZON-SCIENCE. *GitHub - amazon-science/esci-data: Shopping Queries Dataset: A Large-Scale ESCI Benchmark for Improving Product Search*. 2022. Available also from: `https://github.com/amazon-science/esci-data`.

29. CHOUDHARY, Nurendra. *KDD Cup 2022 Workshop: ESCI Challenge for Improving Product Search*. 2022. Available also from: `https://amazonkddcup.github.io/`.

30. TEAM, The pandas development. *pandas-dev/pandas: Pandas*. Zenodo, 2020. Latest. Available from DOI: `10.5281/zenodo.3509134`.

31. HARRIS Charles, et. al. Array programming with NumPy. *Nature*. 2020, vol. 585, no. 7825, pp. 357–362. Available from DOI: 10.1038/s41586-020-2649-2.

32. HUNTER, J. D. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*. 2007, vol. 9, no. 3, pp. 90–95. Available from DOI: 10.1109/MCSE.2007.55.

33. PEDREGOSA F., et. al.. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*. 2011, vol. 12, pp. 2825–2830.

34. VIRTANEN Pauli, et al. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*. 2020, vol. 17, pp. 261–272. Available from DOI: 10.1038/s41592-019-0686-2.

35. PASZKE Adam, et. al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. Available also from: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

36. SHUTTIE. *GitHub - shuttie/esci-s: Extra product metadata for the Amazon ESCI dataset*. 2023. Available also from: https://github.com/shuttie/esci-s.

37. REIMERS, Nils; GUREVYCH, Iryna. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2019. Available also from: https://arxiv.org/abs/1908.10084.

Contents of the attached medium

- `readme.txt` ..... brief description of the medium
- `thesis` ..... directory with the thesis source in LaTeX format
- `source`
  - `readme.MD` ..... instructions for running the source code
  - `datasets` ..... directory with datasets
  - `notebooks` ..... directory with jupyter notebooks
    - `esci` ..... source code for semantic reranking task
    - `interactions` ..... source code for interaction similarity search task
    - `hybrid` ..... source code for hybrid Personalized semantic search task
- `Thesis_Schweika_Patrik_2024.pdf` ..... thesis in PDF format