POLITECNICO DI TORINO

Master Degree in Data Science and Engineering

Master Thesis

Hybrid Movie Recommender System

Using NLP techniques for items' features generation

Supervisor
Prof.ssa Maria Elena Baralis

Co-supervisors:
Dott.sa Ilaria Gianoli
Dott. Lorenzo Vita

Candidate
Giovanni Cioffi

April 2022

Hybrid Movie Recommender System using NLP techniques for items' features generation
Master Thesis. Politecnico di Torino, Turin.

April 2022.

Abstract

Recommender systems address the information overload problem in the Internet by estimating users' preferences and recommending items they might like and interact with. For online entities and companies, these tools have become key components of their websites or applications in order to boost activities, enhance customer experience and facilitate users' decision-making through personalization. The growing availability of online information and the advancements in the field of Deep Neural Networks have determined the transition from traditional methods such as purely content-based or collaborative filtering to hybrid models: capable of improving the recommendation quality, capturing more complex user-item relationships and better in tackling user-item cold start problem. The objective of this Master Thesis, developed during an Internship in Data Reply, is to build a hybrid recommender engine capable of exploiting both user-item interactions and their related metadata, working on a practical use-case. We create a movie recommender system trained on "The Movies Dataset" from Kaggle, an open-source ensemble of thousands movies' metadata such as genre, language, plot, and millions of user-item ratings collected from grouplens.org, the Social Computing research lab at the University of Minnesota and The Movies DataBase (TMDB), a community built online movie and TV database. We explore the state-of-the art of recommender system algorithms, together with their related similarity and evaluation metrics. We train, evaluate and compare several recommendation models: a pure content-based model whose topK ranking is solely based on similarities between users' profiling vectors and movies' vectors, a collaborative-filtering model built using matrix-factorization over interactions matrix and, finally, hybrid models exploiting both ratings and user-item metadata. Main experiments were carried out with LightFM, a hybrid recommendation engine library provided by the fashion company Lyst. Especially, in this thesis we investigate how natural language information can contribute to better suggest items tailored to users' preferences. We explore the state-of-the-art of Natural Language Processing (NLP) techniques for textual feature engineering starting from context-free models like TF-IDF and word2vec until transformer-based models such as BERT and GPT-2. We investigate and implement techniques for documents' similarity and topic modelling in order to train a hybrid recommendation model able to accurately suggest movies based on users' past interactions but also taking

advantage of items' metadata. Movies' textual synopsis are engineered as points in a multi-dimensional embedding space using BERT and categorized in topics with HDBScan clustering in order to be digested by the recommendation engine. Moreover, we make experiments for validating the capability of those hybrid models to properly tackle user cold-start and item cold-start issues: recommendation settings in which new users or items without past interactions are introduced, also when trained in production.

Contents

List of Tables

List of Figures

0.1 Introduction

This thesis was developed during a 9 month Internship at Data Reply S.r.l. with the supervision of Dott.ssa Ilaria Gianoli and Dott. Lorenzo Vita. Our objective was to build a hybrid recommendation engine capable of making user tailored suggestions by exploiting both user-item interactions and their related metadata. Particularly, we focused on how to exploit the latest Natural Language Processing (NLP), a recent field of Machine Learning applied to text, for incorporating textual metadata into the engines and improve their performances. This work is divided in two main parts:

1. **First part: Theory Background**. In this part we cover the preliminary theoretical aspects related to recommender systems and the main NLP techniques for processing sentences. In Chapter 1 we define recommender systems and their terminology. We describe the main families of engines such as content-based filtering, collaborative filtering and hybrid models. For each recommendation method, we discuss its advantages and limitations depending on the context it is used (e.g. cold-start problem). We dive into their general architecture, similarity metrics and functioning. In particular, we focus on LightFM, an open-source recommender system provided by Lyst fashion company, which is capable of exploting both content and collaborative filtering for better tackling the cold-start problem. Since we want to incorporate textual metadata into recommendation systems, in Chapter 2 we investigate the state-of-the-art of Natural Language Processing (NLP) techniques for sentence encoding, similarity and categorization. The following methods were explored:

   - Traditional statistical methods such as Term Frequency – Inverse Document Frequency (TF-IDF);
   - Context-free pre-trained models such as word2vec;
   - Pre-trained models such as BERT and GPT-2 with a brief introduction on Transformers.

   For each of those NLP techniques, we describe their architectures and functioning, pointing our their advantages and drawbacks.

2. **Second part: Use-case and Experiments**. From Chapter 3 to 8, we explain the details of our use-case, which is the implementation of a hybrid movie recommender system trained on "The Movies Dataset", an

open-source dataset containing thousands of movies' metadata (genre, cast, director, plot etc.) and millions of user-item ratings from 0.5 to 5. The dataset is deeply explored and analyzed with proper visualization in Chapter 4. We particularly focus on the analysis of the main textual features of the dataset that are incorporated into the recommender models: overviews of the movies and details about cast and directors. Before describing our experimental models, data had to be appropriately pre-processed. Data cleaning, feature selection and engineering are described in Chapter 5. Particularly, we show how to obtain sentence embeddings from BERT and exploit them for recommendation. After having pre-processed data, we show how our recommendation models were implemented, trained and evaluated (in cold-start setting too) with the proper metrics. Hyperparameter tuning and experiments' results are showed and compared in Chapter 7. We finally comment our outcomes and discuss about further improvements of this work in Chapter 8.

Part I

First part: Theory Background

Chapter 1

Recommendation Systems

According to McKinsey[5], 35 percent of what consumers purchase on Amazon and 75 percent of what they watch on Netflix come from product recommendations based on recommendation algorithms. Recommendation systems are a type of intelligent information filtering that aims to extract value by finding similarities between users and/or catalog items. They are used to generate an ordered list of suggestions tailored to the preferences of the end user. Nowadays, recommendation systems are widely exploited by online entities and companies, whether the item to be recommended is a movie, a video, music, news articles, fashion products or anything we can search online. We firstly dive into the terminology of recommendation systems, their general architecture and similarity measures. Then, we introduce the main families of filtering algorithms:

- Content-based filtering;
- Collaborative filtering;
- Hybrid filtering.

1.1 Terminology

Before describing the background of recommender systems, we look at the terminology that we are going to use throughout this thesis when referring to those algorithms. Recommender systems recommend **items** given a **query** relying on an appropriate **embedding** representation of items and users, let's briefly explain those terms:

- **Item**: items are entities that the recommender system tries to recommend to users, whether they are a product or a service. For instance, for Youtube recommendation engines items are videos, for Amazon marketplace they are every object you can buy on the marketplace. The whole set of items is called *corpus*, the size of the corpus is the number of unique items of the dataset;
- **Query**: the information given as input to the recommendation engine and that it used to provide a topK ranking. Queries can be a combination of user information (for instance the id of the users), they can be items that users previously interacted with together with additional context (time of day, the user's device) or preferences;
- **Embedding**: a mapping from a discrete set, which is represented by the set of queries, or the set of items to recommend, to a vector space called the embedding space $E = \mathbf{R}^d$, where d represents its dimensionality. Many recommendation systems, such as matrix factorization-based recommenders, rely on learning an appropriate embedding representation of queries and items. Both content-based and collaborative filtering map each item and each query to an embedding vector in a common multi-dimensional embedding space. Typically, the embedding space has lower dimensions compared to the size of the interaction matrix. In other words, d is much smaller than the size of the corpus. This space captures some latent structure of the item or query set. Similar things in the embedding space, such as YouTube videos watched by the same user, end up close together. A similarity measure, such as the cosine similarity score, defines the concept of "closeness," which we will explore in further in the following paragraphs.

1.2 General architecture

Although there exist several typologies of recommender systems in literature, they all share similar architecture and established procedures for the generation of candidates from the corpus of items, the selection of the topK and the re-ranking adjustments.

- **Candidate Generation**: the system builds a collection of probable relevant candidates from all conceivable objects based on a query. In this first phase, the algorithm starts with a potentially vast corpus and develops a much smaller subset of options. The candidate generator on

YouTube, for example, reduces a corpus of billions of videos to hundreds or thousands of candidates. The model must evaluate queries quickly due to the enormous size of the corpus. A single model may produce many candidate generators, each of which nominates a different subset of candidates.

- **Scoring**: following that, the recommendation system rates and ranks the candidates in order to choose a collection of K items (on the order of ten) to show the user. The system can utilize a more accurate model based on additional queries because this model only analyzes a limited selection of items.
- **Re-ranking**: finally, for the user's final ranking, the recommender must take into account additional constraints. For example, the system removes items that the user has already seen or has expressed an explicit dislike for. In user-item cold start settings, it can also improve the score of fresher content by boosting it in some way. Re-ranking can also help ensure diversity, freshness, and fairness.

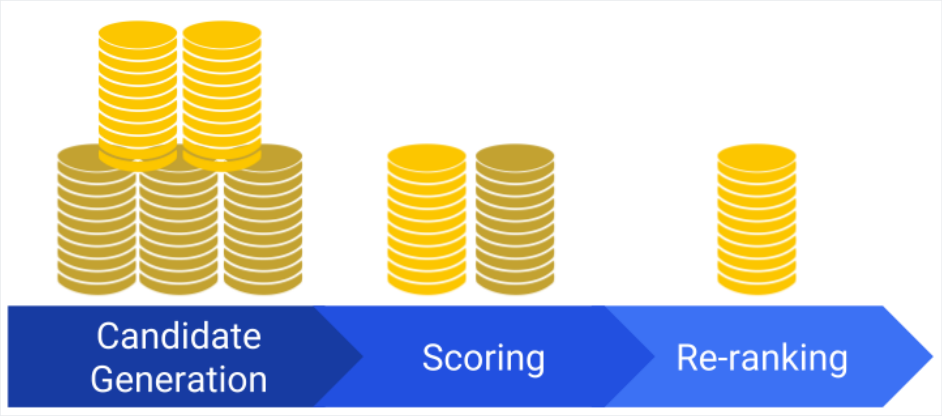

Figure 1.1. Through candidate generation, scoring and re-ranking phases the size of the corpus reduces to K [1].

1.3 Similarity measures

In the embedding space exploited by recommendation systems similar items end up close together. The notion of "closeness" is defined by a similarity

measure [6] such as cosine similarity and euclidean distance.

1.3.1 Cosine similarity

In the realm of information retrieval, cosine similarity is one of the most commonly used methods for rating the similarity of documents in a vector space model. It calculates the cosine of the angle formed by two vectors projected in a multi-dimensional space. The inner product of the vectors divided by the product of their norms is the cosine similarity formula. The Euclidean Dot Product formula can be used to obtain:

$$\langle \mathbf{a}, \mathbf{b} \rangle = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta \tag{1.1}$$

Given two vectors of attributes, $\mathbf{a}$ and $\mathbf{b}$, the cosine similarity, $\cos\theta$, is represented using a dot product and magnitude as

$$\cos\theta = \frac{\langle \mathbf{a}, \mathbf{b} \rangle}{\|\mathbf{a}\| \, \|\mathbf{b}\|} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} {a_i}^2} \sqrt{\sum_{i=1}^{n} {b_i}^2}} \tag{1.2}$$

For instance, the cosine similarity of two vectors with the same orientation is 1, the similarity of two vectors oriented at right angles to each other is 0, and the similarity of two vectors diametrically opposed is -1. In positive space, where the result is nicely limited in [0,1], the cosine similarity is very useful. The name comes from the phrase "direction cosine": unit vectors are maximum "similar" if they are parallel, and maximally "dissimilar" if they are orthogonal in this case (perpendicular). The cosine, which is unity (highest value) when the segments subtend a zero angle and zero (uncorrelated) when the segments are perpendicular, is equivalent to this. The cosine similarity is most typically utilized in high-dimensional positive spaces, and these constraints apply for any number of dimensions. In information retrieval and text mining, for example, each term is given a separate dimension, and a document is represented by a vector, with the value in each dimension corresponding to the number of times the term appears in the document. Cosine similarity is a useful metric for determining how similar two documents are in terms of topic matter.

1.3.2 Euclidean distance

Euclidean distance is another possible metric of similarity for comparing users or items in recommendation systems. Given $\mathbf{a}$ and $\mathbf{b}$, two n-dimensional

vectors, their euclidean distance is computed with the following formula:

$$d(\mathbf{a}, \mathbf{b}) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \cdots + (a_i - b_i)^2 + \cdots + (a_n - b_n)^2} \tag{1.3}$$

A smaller distance between two vectors means higher similarity. We not much interested in the distance itself, but in comparing distances. It has to be noticed that when the embeddings are normalized, the squared Euclidean distance coincides with dot-product (and cosine) up to a constant.

1.4 Content-based filtering

Content-based filtering is the most traditional recommendation system method. It uses similarity between items to recommend items similar to what the user likes, without taking into account information about other users. For instance, if user A watches two cartoon movies related to animals, then the system can recommend cartoon-movies with dogs and cats to that user. Content-based systems suggest goods that are the most similar to those with which the user has already interacted. The basic premise behind these recommendation engines is that if a person enjoyed something in the past, he or she will like something similar in the future. This system exploits item metadata features, such as genre, director, plot description, cast, etc. for movies, to make these recommendations. Since those items' attributes are hand-engineered, a content-based system requires some sort of domain knowledge to properly extract them. For a given user, a content-based system computes its *user profiling vector*, representing the feature values of the items it has interacted with in the past. This vector is compared with all the other items of the corpus with a similarity measure in order to provide a topK recommendation.

1.4.1 Limitations of content based filtering

Since the suggestions are exclusive to a person, a content-based approach does not require any data about other users' interactions with any item. This allows scaling the recommendation system to a greater number of users easily. Furthermore, the model may be able to recognize a user's individual preferences and make recommendations for niche items that only a few other users are interested in. However, due to a number of drawbacks, the usage of content-based filtering has been limited in real applications. This approach

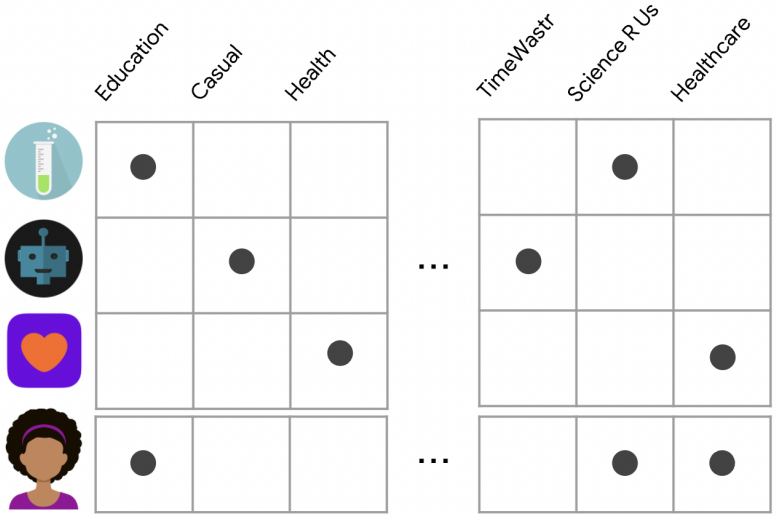

Figure 1.2. Example of content-based feature matrix for apps on a store [1]. Each row of the feature matrix represents an item (i.e. an app on a store) and each column is a hand-engineered feature. The user vector (i.e. last row) is represented in the same feature space.

necessitates a great deal of domain expertise because the feature representation of the objects is hand-engineered to some extent. As a result, the model can only be as good as the characteristics that were hand-engineered. It can only provide recommendations based on the user's previous interests, or "user bobble." In other words, the model's potential to expand on users' existing interests is limited, and it will be difficult to achieve proper performances.

1.5 Collaborative filtering

Collaborative filtering is based on the idea that people who have previously agreed on preferences will continue to agree in the future, and that they will enjoy comparable items. This type of recommender system makes recommendations based on similarities between queries and items at the same time. The collaborative filtering strategy has the advantage of not relying on machine analyzable content, which allows it to accurately propose items such as movies without requiring "knowledge" of the item. If user A is similar to user B, and user B likes movie 1, then the system can recommend movie 1 to user A (even if user A has not seen any movie similar to video

1). Using the user interaction data, explicit or implicit, we do not have to generate the features. The features are extracted autonomously from patterns in the data, which are user-item interactions. To address some of the limitations of content-based filtering, collaborative filtering uses similarities between users and items simultaneously to provide recommendations. This allows for serendipitous recommendations, which means unplanned discoveries for users. In other words, collaborative filtering models can recommend an item to user A based on the interests of a similar user B. Thus, the embeddings can be learned automatically, without relying on hand-engineering of features. The prediction of the model for a given (user, item) pair is the dot product of the corresponding embeddings.

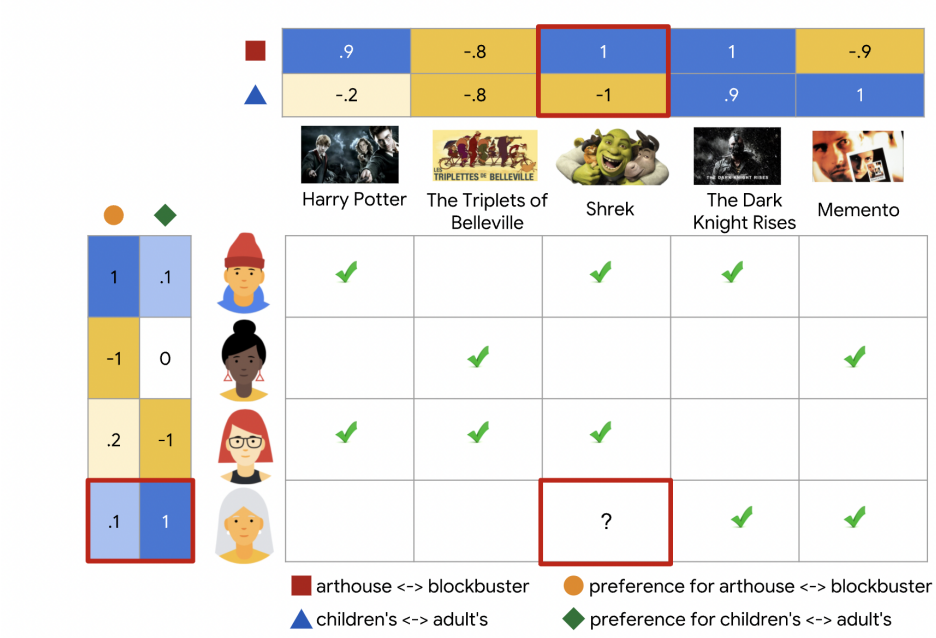

Figure 1.3. Toy example of collaborative filtering approach with matrix-factorization [1]. Embeddings are matrices on left-side and upper-side of the picture with 2 dimensions.

We represent both items and users in the same embedding space. This may seem surprising since users and items are two different entities. However, we can think of the embedding space as an abstract representation common to both items and users, in which we can measure similarity or relevance using a similarity metric. Consequently, embeddings of users with similar preferences will be close together and embeddings of movies liked by similar

users will be close in the embedding space.

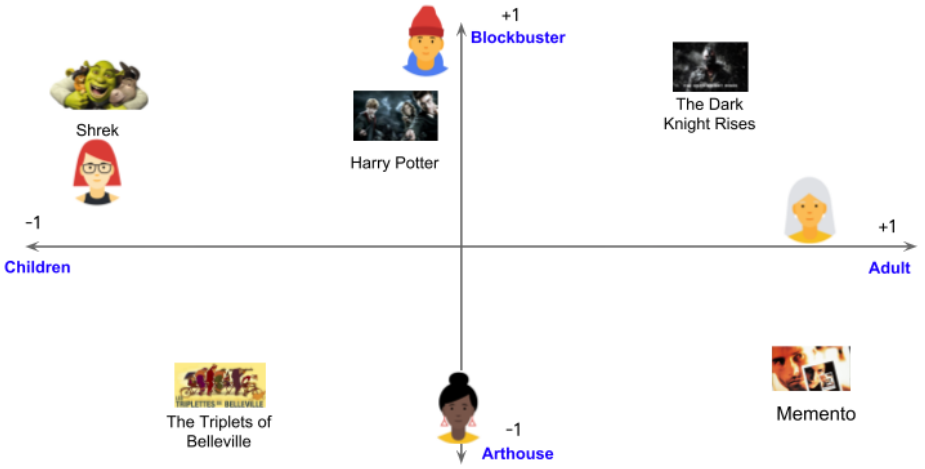

Figure 1.4. Toy example of items' representation in a 2-dimensional embedding space [1]. For the sake of simplicity, in this picture embeddings are hand-engineered. In practice, they are be learned automatically, which is the power of collaborative filtering models.

1.5.1 Matrix factorization

Matrix-factorization [7] is an embedding model that decomposes the interaction matrix into two matrices (i.e. user and item matrices) with lower dimensions, such that when you multiply them you retain the original interaction matrix. We call A the interaction matrix $A \in R^{mxn}$, where m is the number of users and n is the number of items. The model learns:

- A **user embedding** matrix $U \in R^{mxd}$ where row i is the embedding for user i.
- An **item embedding matrix** $V \in R^{nxd}$, where row j is the embedding for item j.

d is the number of dimensions. The embeddings are learned such that the product UV^T is a good approximation of the feedback matrix A. The (i, j) entry of $U.V^T$ is simply the dot product $\langle U_i, V_j \rangle$ of the embeddings of user i and item j which we want to be close to A_{ij}. One advantage of using this approach is that instead of having a high dimensional matrix containing

abundant number of missing values we will be dealing with a much smaller matrix in lower-dimensional space.

1.5.2 Advantages of collaborative filtering

There are several advantages with this approach. It handles the sparsity of the original matrix better than memory-based ones and comparing similarity on the resulting matrix is much more scalable especially in dealing with large sparse datasets. Matrix factorization produces a more compact representation than memorizing the entire matrix. The complete matrix has $O(nm)$ entries, whereas the embedding matrices U,V have $O((n+m))d$ entries, with d often being much less than m and n. As a result, assuming that observations are near to a low-dimensional subspace, matrix factorization uncovers latent structure in the data. It can be much more compact in real-world recommendation systems than learning the entire matrix. No domain knowledge is necessary with this approach because the embeddings are automatically learned, simply using the interaction matrix. In practise, the system doesn't need contextual features. Another characteristic of these systems is serendipity of recommendation: collaborative filtering solves the problem of "only safe recommendations" and "user bubbles", since it can help users discover new interests. In isolation, the system may not know the user is interested in a given item, but the model might still recommend it because similar users are interested in that item.

1.5.3 The cold-start problem

Even without items or users metadata available, collaborative filtering models tend to be really powerful, granted that we have many data about their interactions with users. But in the case of new items or new users in which there are only a few transactions, those models tend to perform poorly. Those typical settings in real application of recommendation system algorithms are known as *user-cold start* and *item-cold start*. Let's describe a situation of item cold-start. For recommender systems using matrix factorization, the prediction of the model for a given (user, item) pair is the dot product of the corresponding embeddings. Thus, when an item is not seen during training, the system cannot create an embedding for it and cannot query the model with this item. The recommender is required to make recommendations from a pool of items for which no collaborative information has been gathered, and only content metadata are available. Hybrid recommender models come in

help in those situations.

1.6 Hybrid Recommender Systems

Hybrid recommender systems combine content-based and collaborative filtering strategies in different ways to benefit from their complementary advantages. Netflix is an excellent example of how hybrid recommender systems may be used. The website provides recommendations by comparing comparable users' viewing and searching behaviors (collaborative filtering), as well as by suggesting films that share qualities with films that a user has rated highly (content-based filtering)[8]. These engines can improve prediction performances and are capable of overcoming the limitations of native approaches such as sparsity and loss of information. Hybrid recommender systems exploit both interactions data between users and items (e.g. ratings from users in movie recommendation systems), item features and user features, which we call items metadata and users metadata. Furthermore, hybrid models are better suited to tackle cold-start problem. When no interaction data is available, we can rely on the intrinsic information provided by metadata. For the purpose of this work we exploited LightFM models to investigate performances of a hybrid recommender system on our use-case dataset.

1.6.1 LightFM

LightFM is a recent open-source recommendation model provided by the fashion company Lyst. It is a hybrid recommendation system method, a special typology of recommender algorithm that uses both collaborative and content-based filtering for making recommendations. For this reason, LightFM was proven not to suffer from the cold-start problem [9], both for items and users if high-quality metadata is provided. Users and items are represented as latent vectors or embeddings in a multi-dimensional space in LightFM, within a collaborative filtering paradigm. These are wholly described by functions (in this case, linear combinations) of embeddings of the content attributes that describe each item or user, exactly as they are in a content-based model. If the movie 'Wizard of Oz' is described by the features 'musical fantasy, 'Judy Garland,' and 'Wizard of Oz,' then the total of these features' latent representations will be its latent representation. Instead of immediately identifying latent representations of movies and users (as in normal

collaborative filtering models), we receive the latent representation of each feature for each movie and user in the case of LightFM. According to this theory, a movie's latent representation is simply the sum of the latent representations of its features. Similarly, to obtain a latent representation for a customer, we simply add the latent representations of the customer's features. The cosine similarity of the latent representations of the movie and the user determines the score for a movie-user pair. The sum of a user's features' latent vectors is the user's latent representation:

$$\mathbf{q_u} = \sum_{j \in f_u} e_j^U \tag{1.4}$$

The same holds for item i:

$$\mathbf{p_i} = \sum_{j \in f_i} e_j^I \tag{1.5}$$

The bias term for user u is given by the sum of the features' biases:

$$\mathbf{b_u} = \sum_{j \in f_u} b_j^U \tag{1.6}$$

The same holds for item i:

$$\mathbf{b_i} = \sum_{j \in f_i} b_j^I \tag{1.7}$$

The model's prediction $\hat{r}_{ui}$ for user u and item i is then given by the dot product of user and item representations, adjusted by user and item feature biases:

$$\hat{r}_{ui} = f(\langle \mathbf{q}_u, \mathbf{p}_i \rangle + b_u + b_i) \tag{1.8}$$

In order to express user preferences over items, the model learns embeddings (or latent representations in a high-dimensional space) for users and items. These representations are multiplied together to provide scores for each item and for each user; items with high scores are more likely to be attractive to the user and contribute to its topK suggestions. LightFM embeddings are trained using stochastic gradient descent algorithms that minimize a loss, such as WARP or BPR, which we will look at in the following chapter.

Chapter 2

NLP Techniques for sentence embedding

In this Chapter we are going to describe the Natural Language Processing (NLP) techniques for sentence embedding taken into account in this thesis. The objective is to vectorize (i.e. encode) our movies' plots and make them comparable and digestible by our recommendation systems. We will start from TF-IDF, a numerical statistic that is intended to reflect how relevant a word is to a document in a collection or corpus. Then, we will discuss context-free language models such as word2vec and doc2vec. Finally, after a brief introduction to transformers, we will deeply dive into the state of the art of NLP sentence embedding, transformers-based language models, particularly BERT and GPT-2.

2.1 TF-IDF

TF-IDF is one of the most popular weighting schemes that stands for "Term Frequency - Inverse Document Frequency", widely used in recommender systems employing textual features [10]. Given a term t, a document d and the corpus the document belongs to D, the TF-IDF is computed as the product of term frequency (TF) and inverse document frequency (IDF) as follows:

$$\text{tfidf}(t, d, D) = tf(t, d) * idf(d, D) \tag{2.1}$$

$tf(t, d)$ simply represents the number of times a word appears within a document, i.e its frequency,

$$\mathrm{tf}(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}} \tag{2.2}$$

where the numerator indicates the raw count of terms in a document and the denominator simply represents the overall number of terms in document d (counting each occurrence of the same term separately). $idf(d, D)$ is the inverse fraction of documents that contain the word, which is calculated by dividing the total number of documents by the number of documents that contains the term and then calculating the logarithm of that quotient.

$$\mathrm{idf}(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|} \tag{2.3}$$

idf diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. The tf–idf value rises in proportion to the number of times a word appears in a document and is offset by the number of documents in the corpus that contain the term, which helps to compensate for the fact that some words appear more frequently than others. A phrase with a high frequency in a document but a low frequency in the corpus has a high TF-IDF score. Although TF-IDF can provide a good comprehension of the importance of words, it has the disadvantage of not providing linguistic information about the words, such as semantic meaning, linguistic relationship and similarity with other words, and so on. For the purpose of this thesis, TF-IDF will be used in the context of BERT Topic Modelling in order to extract meaning from several clusters. After BERT embeddings are extracted from our sentences and clusterized (clusterization process is described in detain in the next section), we will concatenate together documents (i.e. movie plots) belonging to the same cluster with the aim of understanding their semantic meaning and evaluating the quality of the clustering. For this purpose, we apply TF-IDF on each of the concatenated clusters' documents and extract their most relevant words. Some of those experiments with topic modelling using TF-IDF are showed in the Data Preprocessing Chapter.

2.2 Context-free language models, word2vec and doc2vec

Context-free models such as word2vec generate a single word embedding representation for each word in the vocabulary. They represent words in a way that captures meaning-related relationships. The term context-free indicates that words embeddings do not take into account the context in which they are inserted within sentences. For instance, with word2vec the word “bank” would have the same context-free representation in “bank account” and “bank of the river"[11]. In word2vec, different senses of the word, if they exist are combined into one single vector.

2.2.1 word2vec

Word2vec [12] is an artificial neural network for natural language processing. Using this model we get to properly represent words in a way that captures semantic relationships: the ability to tell if words are similar, or opposites, or that a pair of words like “Stockholm” and “Sweden” have the same relationship between them as “Cairo” and “Egypt” have between them, as well as syntactic, or grammar-based, relationships (e.g. the relationship between “had” and “has” is the same as that between “was” and “is”) [13]. The method takes a corpus as input and outputs a set of vectors that indicate the semantic distribution of words in a text. A vector is built for each word in the corpus to represent it as a point in the newly created multidimensional space. Words will be closer together in this region if they are recognized as semantically similar. Understanding the CBOW and Skip-Gram architectures is required to comprehend how word2vec can produce word embedding.

Current token	**Context window**
nel	(nel,mezzo) (nel,del)
mezzo	(nel, mezzo) (mezzo,del) (mezzo,cammin)
cammin	(mezzo, cammin)(del,cammin)(cammin,di)(cammin,nostra)
di	(del,di)(cammin,di)(di,nostra)(di,vita)
nostra	(cammin,nostra)(di,nostra)(nostra,vita)
vita	(di,vita)(nostra,vita)

Table 2.1: Context window with C=2 for the sentence "Nel mezzo del cammin di nostra vita".

CBOW and Skip-Gram

Word2vec uses one of two architecture models to generate word embedding: continuous bag-of-words (CBOW) or Skip-Gram. To train the models, a large corpus of textual content is necessary. From this corpus, we will be able to extract a vocabulary of independent terms (i.e. tokens). Each distinct token is then represented using one-hot encoding. By examining other tokens in the vicinity, the two models analyze the token context. The following are the main differences between the CBOW and Skip-Gram designs: the former design attempts to predict the current (output) token from a set of context (input) words, whereas the latter aims to predict context words (output) from the current token (input). C parameter represents the size of the context window: included in the context are C tokens immediately preceding/next to the current token are included in the context. Both models are implemented using a three-layer artificial neural network: an input layer, a hidden layer, and an output layer. The input varies depending on the model's design and the type of projected output. In fact, the current token and context window will be the model's input or output, depending on the architecture. The word embedding representations of the V vocabulary tokens will be contained in the weight matrix W following model training via backpropagation, where N is the size of the word embedding; in the example above, the size of V is 7. In other words, the row $[w_{(i,1)}, w_{(i,2)}, ..., w_{(i,N-1)}, w_{(i,N)}]$ of the matrix W will contain the word embedding representation of the token V_i. The softmax function is applied in the last layer of the neural network.

Below we describe the characteristics of the two models:

- The **CBOW** [14] architecture aims to predict the current token (output, e.g., "nel") from a window of context words (input, es. "mezzo" e "del"). Thus, the input consists of the one-hot encoding representation of $C * 2$ context tokens. The matrix W is the same as described in the general architecture, but, in this case, the hidden layer is the average of the vectors W_i corresponding to the input context words. By doing so, the hidden layer loses the information of the position in the text of the tokens that are part of the context window as well as in the bag of words representation.

- The **Skip-Gram** architecture aims to predict context words (output, e.g., "nel", "del", and "cammin") from the current token (input, e.g., "mezzo"). The input thus consists of the one-hot encoding representation of the current token. The hidden layer then corresponds to the word

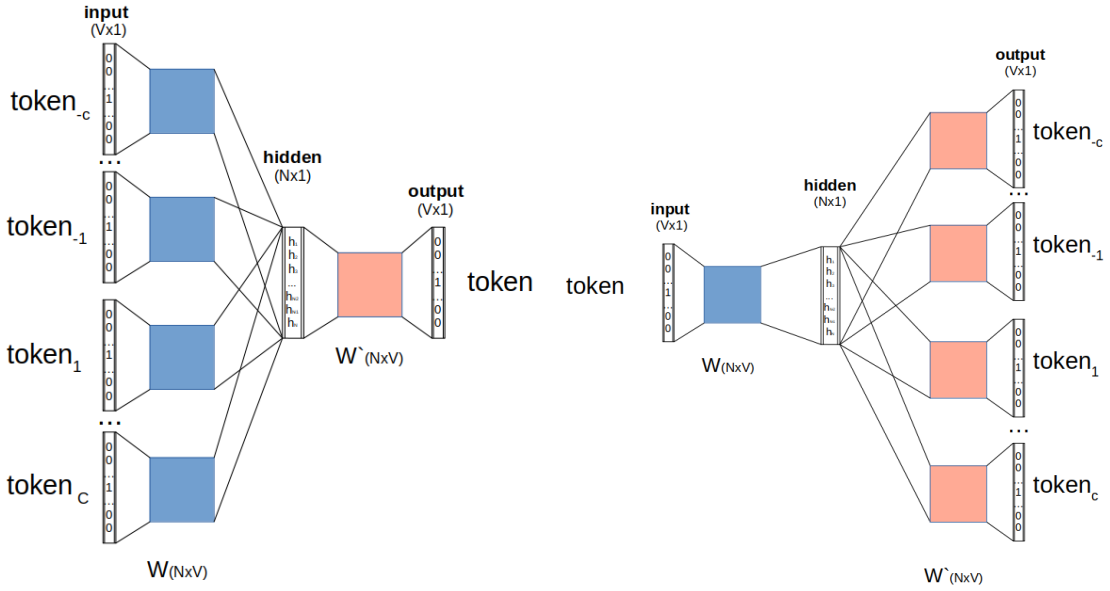

Figure 2.1. CBOW (left) and Skip-gram (right) architecture

embedding representation of the current token. The model will generate an output vector for each context token. Consequently, there will be as many error vectors of size Vx1. In order to start the backpropagation, $C*2$ error vectors are added together to obtain a single Vx1 vector. The weights of the hidden layer will then be updated based on this cumulative error vector. Also in the skip-gram architecture the position in the text of the context tokens is therefore not taken into account.

The skip-gram architecture prioritizes close context words over far away context words. CBOW is faster, while Skip-Gram is better for terms that occur infrequently. Two key hyperparameters in the word2vec training process are the size of the context window C and the number of negative samples.

2.2.2 doc2vec

Doc2vec[15] is a model that represents each document as a vector, it can be considered as a generalization of word2vec for sentences. It usually outperforms simple-averaging of word2Vec vectors within a sentence. There are two implementations of doc2vec:

- **Paragraph Vector - Distributed Memory (PV-DM)** is a word2vec CBOW-like algorithm. The doc-vectors are obtained by training a neural

network on the synthetic task of predicting a center word based on an average of both context word-vectors and the doc-vector of the entire document.

- **Paragraph Vector - Distributed Bag of Words (PV-DBOW)** is similar to Word2Vec Skip-Gram. The doc-vectors are obtained by training a neural network on a synthetic task of predicting a target word only based on the doc-vector of the entire document. It's also typical to combine this with skip-gram testing, which predicts a single target word using both the doc-vector and neighboring word-vectors, but only one at a time.

Limitations of context-free models

Word2vec generates distributed representations of words, or word types, which means that each word in a vocabulary has its own unique word vector **independently of the context** in which the token is inserted. A further drawback of this method is that the size of the vector representing the corpus is determined by the vocabulary size, which may be rather enormous. This huge data usage could manifest itself in the curse of dimensionality. The model would also have to be re-trained from beginning if tokens were added or deleted from the word embeddings. Furthermore, this method of word representation ignores *polysemy*, or the presence of several alternative interpretations for a single word or phrase. The context of a word type determines its meaning, which is reflected by the words that surround it. We can apply the same arguments to doc2vec because it is a generalized extension of word2vec.

2.3 Transformers-based language models

In this section we analyze transformers-based models, the state-of-the art techniques for language modelling in Natural Language Processing. Particularly, we are going to focus on BERT and GPT-2. Transformers were introduced in 2017, they employ an encoder-decoder architecture that allows for passing sequences of data in parallel. A transformer is a deep learning model that adopts the mechanism of self-attention [2], differentially weighting the significance of each part of the input data. Transformers were introduced for solving language translation tasks. They have largely replaced LSTM neural networks for sequence to vector, sequence to sequence and vector to sequence

tasks. Their encoder-decoder architecture has the advantage of being faster to train, since it allows for computation in parallel with textual content: words can be processed simultaneously. Moreover, transformers are better at capturing contextual meaning throughout sentences, since they can learn it from both directions simultaneously, i.e. they are deeply bi-directional. If we imagine to train a transformer for English to French translation, it would be composed of two main blocks:

- An **encoder** that learns the context, it takes words of a sentence simultaneously, e.g. the English words, and generates embeddings for every word. These embeddings are vectors that encapsulate the meaning of each word and are generated simultaneously: there is no concept of time step for the input. Similar words have similar embeddings;

- A **decoder** that learns how to map input to output. e.g. English words to French words. It takes the output embeddings from the encoder and the outputs of the previously generated output, e.g. French words, and it uses them to generate the next output.

Both the encoder and the decoder, even separately, have some underlying understanding of language. Therefore, we can pick apart this transformer architecture and build systems that understand language. For instance, we can extract the decoders and get the GPT architecture. Conversely, if we stack just the encoders we get BERT.

2.3.1 Representing the meaning, input embeddings

Machines do not get words, they get numbers, vectors and matrices. Thus, we want to map every word to a point in space where similar words with **similar meanings** are physically close to each other. The space in which they are present is called an embedding space. Embedding spaces are usually used pre-trained to save time (e.g. GloVe [16]), they map a word to a vector, but the same word in different sentences may have different meaning (e.g. the word dog in “AJ’s dog is a cutie” and “AJ looks like a dog”). For this reason positional encoders are used.

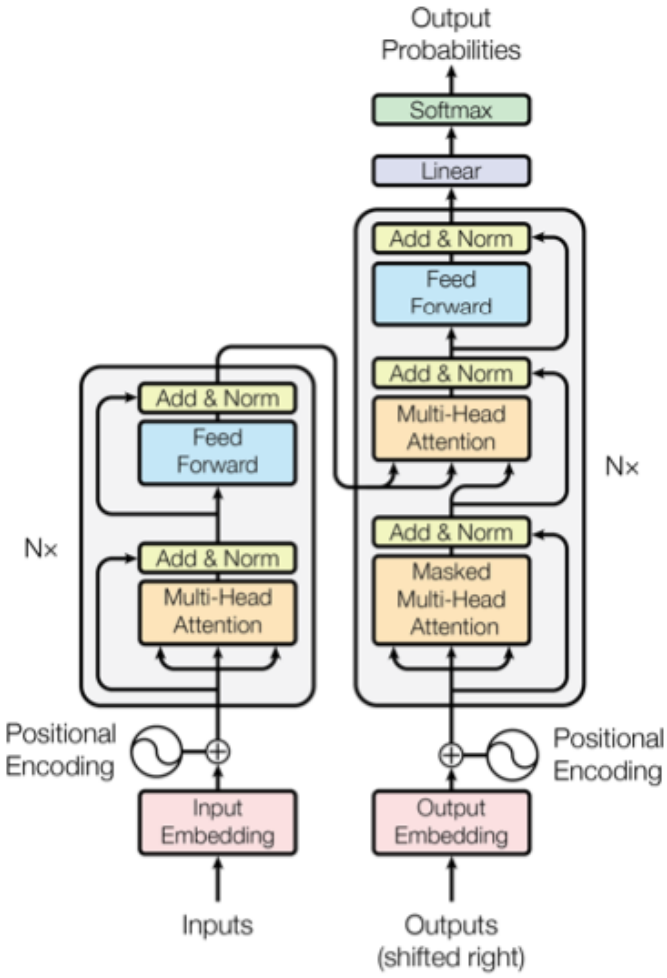

Figure 2.2. Architecture of a transformer illustrated [2]

2.3.2 Getting the context of the word in the sentence, positional encoders

Positional encoders are vectors that give **context** and information about the position of words in a sentence.

$$PE_{(pos,2i)} = sin(\frac{pos}{10000^{\frac{2i}{d_{model}}}}) \tag{2.4}$$

$$PE_{(pos,2i+1)} = cos(\frac{pos}{10000^{\frac{2i}{d_{model}}}}) \tag{2.5}$$

where "pos" refers to the position of the word in the sequence and i refers to each individual dimension of the embedding. In the original paper [2], a sine and cosine function is exploited to generate this vector, but it could be any other reasonable function. Let's consider an English to French machine translation. After passing the English sentence through the input embedding and applying the positional encoding we get word a vector that has positional information that is the context.

2.3.3 Encoder block

The output of the positional encoders is passed through the encoder block. It is composed of a Multi-Head attention layer and a feed-forward layer.

- **Attention mechanism** tries to answer the following question: "What part of the input should we focus on?". Self-attention means attention with respect to one-self. For instance, given an English input sequence to be translated in French, for each word we have an attention vector that answers the question: "How relevant is the i-th word in the English sentence with respect to other words in the same English sentence?". Therefore, i-th attention vectors are computed in the attention block. For every word we can have an attention vector generated which captures contextual relationships between words in the input sentence. The attention vector for each word may not be too strong. For every word, its attention vector will tend to weight higher the related word with respect to the other, focusing too much on itself. This reasonable but quite useless since we are usually interested in the interactions and relationships with different words in the sentence. For this reason we may want to compute several attention vectors per word. Attention mechanism is called **Multi-Head attention** when multiple attention vectors are computed per each word and then those attention vectors are averaged in a weighted manner in order to compute the final attention vector for every word.

- **Feed-forward net**: it is a simple feed-forward neural net that is applied to any of the attention vectors previously described, one vector at a time. Attention vectors are passed to the feed-forward net in order to be transformed in a form that is more digestible by the next encoder or decoder block. Each one of the attention nets are independent of each other, so we can use *parallelization*. This allows to pass all the words

of the sentence at the same time into the encoder block and the output would be a set of encoded vectors for every word.

2.3.4 Decoder block

During the training phase from English to French, we feed the output French sentence to the decoder, which is processed using input embeddings to get the vector form of each word of the sentence, as already explained above, for getting the meaning of each word. A positional vector is computed here too, in order to get the notion of context of the word in the sentence. This vector is finally passed into the decoder block that has the following main components.

- The **Self-Attention Block** generates attention vectors for every word in the output French sentence to represent how much each word is related to every word in the same sentence. The original paper calls this **Masked Multi-Head Attention** because generating the next French word, we can use all the words from the English sentence, but only the previous words of the French sentence. If we are going to use all the words from the French sentence then there would be no learning, it would just speed out the next word. Thus, while performing parallelization with matrices operations, we make sure that the matrix will mask the words appearing later by transforming them into zeros, so the attention network can't use them.

- These attention vectors and the vectors from the encoder, which contain one vector for every word from the English and French sentences, are passed to another attention block, the **Encoder-Decoder Attention block**. This block will determine how related each word vector is with respect to each other. This is where the main English-French word mapping happens. The output of this block is attention vectors for every word in the English and French sentence, each vector representing the relationships with other words in both the languages.

- Each attention vector is then passed to a **Feed-Forward Unit** for making the output vector more digestible by either the next decoder block or a linear layer.

- The **Linear Layer** is another feed-forward fully connected linear layer used to expand the dimensions in the number of words existent in the

French language, thus the number of neurons of the feed-forward layer would be equal to the number of words in French language.

- The **Softmax Layer** is used to transform the output of the linear layer into a probability distribution which is human interpretable. The final word (the predicted next French in the sentence) word would be the one with the highest probability in the softmax output. Overall, the decoder has predicted the next word and this process is executed over multiple timesteps until the end of the sentence tokens is generated.

2.3.5 The Multi-Head Attention networks

We now describe how Multi-Head Attention blocks look. Each attention block looks like in the figure below (right-side). We have:

- **Q**, the Query is a representation of the current word used to score against all the other words (using their keys). We only care about the query of the token we're currently processing.
- **K**, Key vectors are like labels for all the words in the segment. They're what we match against in our search for relevant words.
- **V**, Value vectors are actual word representations, once we have scored how relevant each word is, these are the values we add up to represent the current word.

vectors for every single word. Those abstract vectors extract different components of an input word. We use those vectors for computing the attention vector for each word with the following formula [17]:

$$Z = softmax_k(\frac{\mathrm{Q\ K^T}}{\sqrt{\mathrm{d_k}}})\ \mathrm{V} \tag{2.6}$$

where Z represents the attention given Q, K, V.

For Multi-Head Attention we have multiple weight matrices related to V, K and Q: W^Q, W^K and W^V. Thus, we will have multiple attention vectors Z for every word. However, the neural network is only expecting one attention vector per word. Thus, we use another weighted matrix W^Z to make sure that the output is still an attention vector per input word making the following operation:

$$Z = [Z_1, Z_2, Z_3].W^Z \tag{2.7}$$

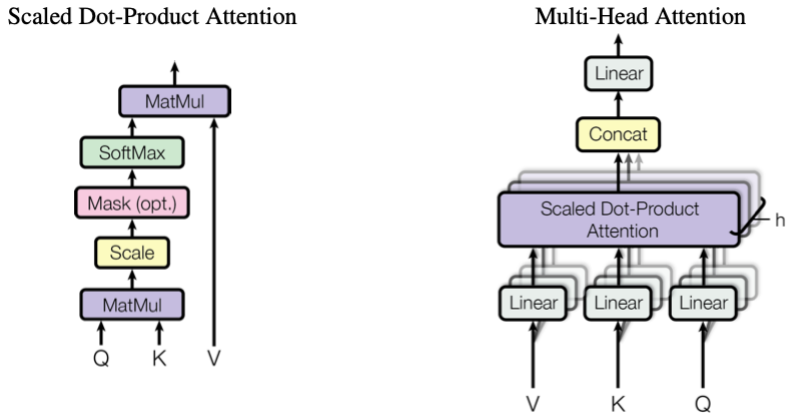

Figure 2.3. Scaled Dot-Product Attention (left), Multi-Head Attention(right).

2.3.6 Add Normalization layers

Additionally, after every layer some form of normalization is applied. Typically, we would apply a Batch Normalization, this smoothes out the loss surface making it easier to optimize while using larger learning rates. Another form of normalisation is the Layer Normalization, making the normalization across each feature instead of each sample, it is better for stabilization.

2.4 BERT

As we have just discussed above, the encoder block of a transformer takes the input words and generates embeddings for every world simultaneously. These embeddings are vectors that encapsulate the meaning of the word: similar words have closer numbers in their vectors. Therefore, the encoder of a transformer learns the input words and what is context from them. Because of this underlying understanding of knowledge, we can pick apart the trasformers' architecture and build systems capable of understanding language. In the case of BERT, we extract the encoders and get a **Bideractional Encoder Representation from Transformers** indeed.

BERT [18] was introduced in 2018 by researchers at Google AI Language.

It reached state-of-the-art results for various tasks in Natural Language Processing, including Question Answering (SQuAD v1.1), Natural Language Inference (MNLI). Unlike previous models, BERT is a deeply bidirectional, unsupervised language representation, pre-trained using only a plain text corpus. Context-free models such as word2vec or GloVe generate a single word embedding representation for each word in the vocabulary, whereas BERT takes into account the context for each occurrence of a given word. For instance, whereas the vector for "running" will have the same word2vec vector representation for both of its occurrences in the sentences

"He is running a company" and "He is running a marathon"

BERT provides contextualized embeddings for each word that will be different according to the sentence. As opposed to directional models, which read the text input sequentially (left-to-right or right-to-left), the Transformer encoder reads the entire sequence of words simultaneously. Therefore, it is considered bi-directional [19], though it would be more accurate to say that it's non-directional. This characteristic allows the model to learn the context of a word based on all of its surroundings (left and right of the word).

BERT can be used in many tasks that require the understanding of language such as neural machine translation, question answering, sentiment analysis, text summarization. It is **pre-trained** in order to get an understanding of language and then **fine-tuned** depending on the task to be solved.

2.4.1 How BERT works

Here is how BERT language model works in summary. We pre-train BERT with Masked Language Modelling (MLM) and Next Sentence Prediction (NSP). For every word in input (in red), we get the Token Embedding (in yellow) from the pre-trained WordPiece embeddings. Then we add the Position Embeddings (in grey) and Segment Embeddings (in green) to account for the ordering of the inputs. These are then passed into BERT, which is a stack of transformer encoders and it outputs several word vectors for MLM and a binary value for NSP. The word vectors are then converted into a distribution to train using cross-entropy loss. Once training is complete, the BERT model has some knowledge about the language. The next step is the fine-tuning phase, where we perform a supervised training depending on the task we want to solve. Performance of course depend on how big we

want BERT to be. The BERT large model, for instance, has 340M parameters can achieve higher performances than the BERT base model with 110M parameters.

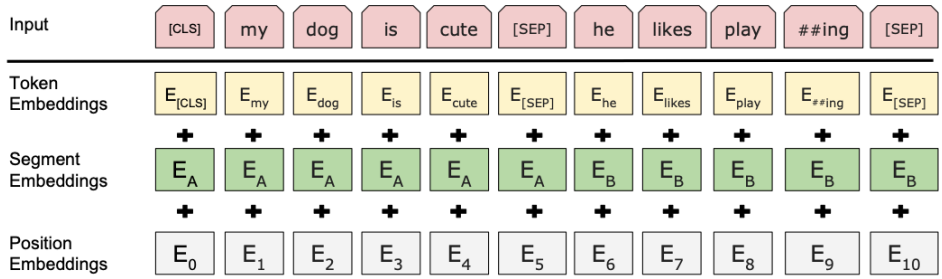

Figure 2.4. How BERT generates embeddings from input words.

BERT can be trained to understand language and context (what we are going to do for this thesis), and then fine-tuned depending on the task. This process can be divided in two steps: pre-training phase and fine-tuning phase.

2.4.2 BERT pre-training phase

The goal of the pre-training phase is to make BERT learn what is language and what is context. BERT learns language by training on two unsupervised tasks simultaneously:

- **Masked Language Modelling (MLM)**. BERT takes in a sentence with random words filled with "[MASK1]" tokens such as

  "The [MASK1] brown fox [MASK2] over the lazy dog".

  where [MASK1] = "quick" and [MASK2] = "jumped".
  The goal is to output these mask tokens in a process that tries to fill in the blanks. This helps BERT understand bidirectional context within a sentence.

- **Next Sentence Prediction (NSP)**. BERT takes in two sentences and it determines if the second sentence actually follows the first, in a binary classification problem setting. Let's say we have

  sentence A: "Ajay is a cool dude."

sentence B: "He lives in Ohio"

We train the language model to understand that sentence B follows sentence A. This helps BERT understand context across different sentences themselves.

The input of the pre-training phase is a set of two unlabeled sentences, sentence A and sentence B with some of the words being masked. Each token Tok_i is a word that is converted to embeddings using pre-trained embeddings. This provides a good starting point for BERT to work with. On the output side we have:

- **C** is the binary output for the Next Sentence Prediction. Therefore, it would output 1 if sentence B follows sentence A in context, 0 otherwise.
- **T__n** are word vectors that correspond to the outputs for the Masked Language Modelling problem.
- **E__i** are word embeddings corresponding to our input tokens.

The number of word vectors in input is the same as the number of words in output. Using both MLM and NSP together, BERT gets a good understanding of language.

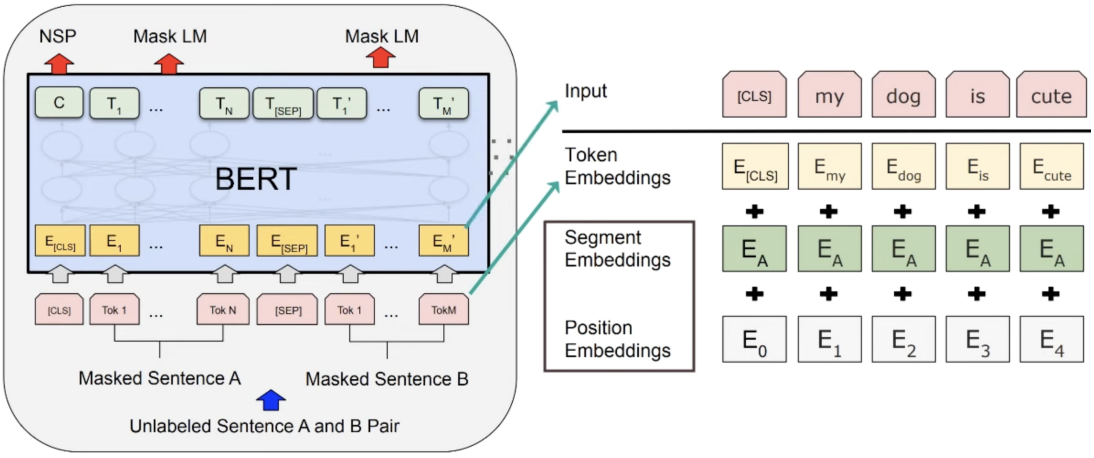

Figure 2.5. BERT Pre-Training phase illustration [3]

2.4.3 BERT Fine-tuning phase

In this phase BERT is further trained on a specific NLP task. In a Question Answering task, for instance, this is usually done by replacing the fully connected output layers of the network with a new set of output layers that can basically output the answer to the question we want. Then, supervised training is performed with a specific dataset on Question Answering. Only the output layers parameters are learned from scratch, the rest of the model parameters are just slightly fine-tuned. As a result, training would be fast. This process of replacing the output layers and then train with a specific dataset can be repeated for several NLP tasks. In a Question Answer task, the model would be trained by modifying the inputs and the output layer. Sentence A and Sentence B would be substituted with a Question and Answer pair. The question is passed followed by a passage containing the answer as inputs. In the output layer we would output the start and the end words that encapsulate the answer, assuming that the answer is within the same span of text.

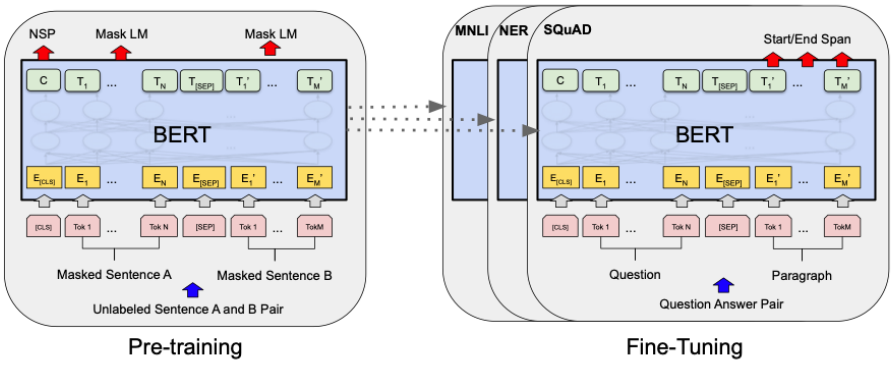

Figure 2.6. BERT Pre-Training and Fine-Tuning phases.

Now let's discuss the input side: how BERT generates initial embeddings from input words. Embeddings E_n are generated from the word token inputs from three vectors:

1. **Token Embeddings E_token**: they are the pre-trained embeddings. The main paper uses "WordPieces" embeddings, a 30K vocabulary as Token Emdeddings;

2. **Segments Embeddings E_sentence**: the sentence number that is

encoded into a vector;

3. **Position Embeddings E_n**: is the position of a word within that sentence that is encoded into a vector.

Addings this three vectors together we get an embedding vector that is exploited as input to BERT. The Segment + Position embeddings are required for temporal ordering since all these vectors are fed in simultaneously into BERT and language models need this ordering preserved.

Now, we analyze the output side of BERT. The output is composed of a binary output C (NSP) and of several word vectors T_i (MLM). During training a *Cross-Entropy Loss* is minimized. All T_i have the same size and they are generated simultaneously. Each word vector T_i is passed into a fully connected layered output with the number of neurons equal to the number of tokens of the vocabulary. Therefore, in the case of the Question Answering Task, it would be an output layer corresponding to 30.000 neurons, i.e. the size of "WordPieces" vocabulary. A softmax activation would be applied for converting a word vector into a distribution and the label of this distribution would be a one-hot encoded vector for the actual target word. We would compare two distributions and train the network using Cross-Entropy loss beween:

- The predicted word, represented by a softmax layer with 30.000 neurons;
- The one-hot encoded vector of 30.000 elements corresponding to the actual word.

Since BERT is giving as output all the words even though inputs were not masked, we can point our that the loss receives contribution only from the masked words, ignoring all the other words that are output by the network. This is done to ensure that more focus is given to predicting this masked values correctly and increasing context awareness.

2.4.4 Topic modelling with BERT embeddings

For the purpose of exploiting also item-metadata for hybrid recommentation, we are going to represent and clusterize textual metadata as points in a multidimensional embedding space using Topic modelling with BERT [20] technique. We want to make sure that movies with comparable plots are grouped together so that the topics inside these clusters may be found. We

use classes and methods of *SentenceTransformers*, a Python framework for state-of-the-art sentence, text and image embeddings [21] based on SBert work. Sentence-BERT [22] (SBert) is a slight modification of the pre-trained original BERT network that uses siamese and triplet network structures to faster derive semantically meaningful sentence embeddings comparable using cosine-similarity. We extract sentence embeddings and contextual knowledge from movies' plots, cast and directors. Once embeddings are obtained from sentences, the objective is to make the textual feature, i.e. the plot of the movie, digestible to LightFM models that require categorical features. Categorization is carried out with HDBScan clustering on top of BERT contextualized embeddings after applying UMAP dimensionality reduction. Clusterization of BERT embeddings is equivalent to a topic modelling operation, where we want similar plots to be inserted in the same cluster. Textual features are embedded in the same embedding space using BERT and topic modelling is applied in order to assign each movie a cluster for each of those features. Topics within each cluster are derived with TF-IDF.

2.4.5 UMAP Dimensionality Reduction

Since clustering algorithms handle high dimensionality poorly, we need to lower it. Uniform Manifold Approximation and Projection for Dimension Reduction [23] UMAP is a non-linear dimensionality reduction technique that is particularly useful for displaying clusters or groupings of data points as well as their relative distances. It can give a balance between focusing on local versus global structure. We reduce the dimensionality of the embeddings before using HDBScan clustering on BERT embeddings derived from textual movie characteristics, as clustering algorithms struggle with high dimensionality. We use UMAP dimensionality reduction on document embeddings for this purpose since it maintains a large percentage of the high-dimensional local structure in reduced dimensionality. UMAP feature reduction most important parameters are:

- **n_neighbors**, low values of this parameter force UMAP to concentrate on very local structure, while large values will push UMAP to look at larger neighborhoods of each point when estimating the manifold structure of the data, losing fine detail structure for the sake of getting the broader of the data.

- **n_components**, the dimensionality of the reduced dimension space we will be embedding the data into.

- **metric**, it controls how distance is computed in the space of the input data.
- **min_dist**, provides the minimum distance apart that points are allowed to be in the low dimensional representation. It controls how tightly UMAP is allowed to pack points together. Lower values mean the points will be clustered closely and vice versa. Better set low for clustering.

2.4.6 HDBScan Clustering

After reducing the dimensionality of the sentence embeddings, we can categorize the documents using HDBScan clustering. HDBScan [24] is a density-based algorithm that works quite well with UMAP since it maintains local structure even in lower-dimensional space. Moreover, HDBScan does not force data points to clusters as it can consider some of them as outliers, i.e. it could not assign points to any cluster as it considers them outliers. We cluster similar sentences together. Each cluster should be expression of a certain topic. Most important HDBScan clustering parameters [25] are:

- **min_samples**, the higher the number of min_samples, the more conservative the clustering — more points will be designated as noise, and clusters will be limited to increasingly dense areas. The grouping will get increasingly conservative when min_samples is increased.
- **min_cluster_size**, the smallest size grouping you want to count as a cluster.
- **cluster_selection_epsilon**, We may want to set a minimal min_cluster_ size in some circumstances since even little groups of points may be of interest to us. This parameter choice, however, can result in a huge number of micro-clusters if our data set also contains partitions with high concentrations of objects. Clusters in these locations can be merged by setting cluster_selection_epsilon to a value. In other words, it ensures that clusters that fall below a certain threshold are not further divided. For example, if we don't want to separate clusters that are fewer than 0.5 units apart, we can set the value to 0.5.
- **metric**, distance metric such as euclidean.
- **cluster_selection_method**, establish how the cluster tree structure is used to select flat clusters 'eom' stands for Excess of Mass and is the default approach.

2.4.7 Topic interpretation using TF-IDF

Once the clusters of sentences are created, each movie will be assigned an "overview cluster". We can find interpretable topic of each cluster by finding the most relevant tokens with the highest TF-IDF values. In other words, documents belonging to the same clusters are concatenated in the same string, TF-IDF is applied for each cluster and tokens with the highest TF-IDF should give an idea of the topic of that cluster.

2.5 GPT-2

Generative Pre-trained Transformer 2 [26] (GPT-2) is general purpose language model created by OpenAI in February 2019. It was trained on a massive 40GB dataset of textual records called WebText crawled from the Internet. Its architecture implements an unsupervised and undirectional transformer model, which uses attention in place of previous recurrence and convolution-based architectures. Attention mechanisms allow the model to selectively focus on segments of input text it predicts to be the most relevant. This model allows for greatly increased parallelization, and outperforms previous benchmarks for RNN, CNN and LSTM-based models. GPT-2 is trained with a simple objective: predict the next word, given all of the previous words within some text. The diversity of the dataset causes this simple goal to contain naturally occurring demonstrations of many tasks across diverse domains. GPT-2 is a direct scale-up of GPT, with more than 10X the parameters and trained on more than 10X the amount of data.

2.5.1 GPT-2 and BERT

GPT-2 and BERT are both pre-trained transformers, thus they allow to extract text embeddings in similar ways. However, they have profound differences in terms of nature, architecture, objectives. First of all, the GPT-2 built using transformer decoder blocks whereas BERT uses transformer encoder blocks. BERT is trained as an Auto-Encoder, it uses the entire surrounding context all-at-once. It is self-attention, where each token in an input sentence looks at the bidirectional context, i.e. other tokens on left and right of the considered token. On contrast, GPT is trained for next-sentence prediction as Auto-regressive model, because it outputs one token at a time and that token is added to the sequence of inputs. Thus, GTP-2 generative model is

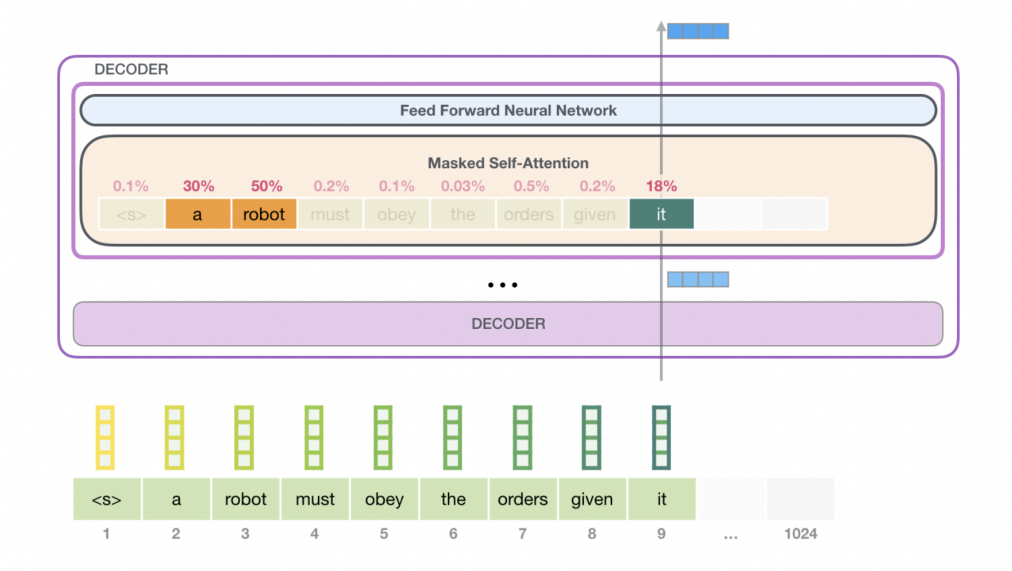

Figure 2.7. A GPT-2 decoder in-dept illustration [4] during the processing of a sentence.

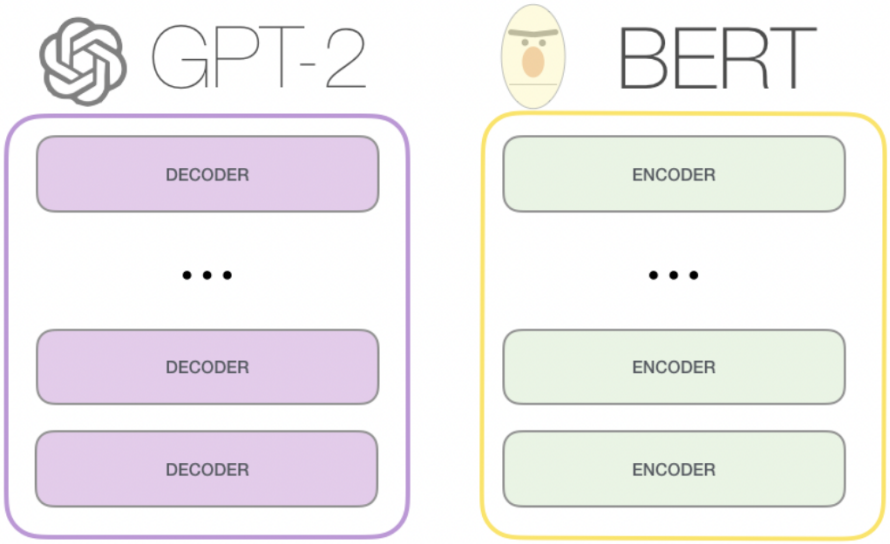

Figure 2.8. Overview of GPT-2 and BERT architectures [4]

looking at only the past or left side context. The decoder is only permitted to extract information from the earlier words in the sentence through obfuscation masking of the remaining word positions.

Part II

Second part: Use-case and Experiments

Chapter 3

Implementation details

Our experiments where carried out in cloud using the free version of Google Colaboratory notebooks in Python language. Google Colab provides virtual machines with 12GB of RAM and Tesla K80 Graphical Processing Unit (GPU) or Tensor Processing Units (TPU) as hardware accelerators depending on the availability of those machines.

Chapter 4

Dataset

In this section we go through a general exploration of The Movies Dataset from Kaggle [27], a dataset containing movies' metadata and user-item interactions. It was used in this thesis as basement for carrying out experiments on various families of recommendation systems and Natural Language Processing tecnhiques for sentence encoding, similarty and topic-modelling.

4.1 The Movies Dataset

Movielens is a web-based recommender system from GroupLens [28], a research lab at the University of Minnesota in the United States. It recommends its users movies to watch based on their film preferences, using collaborative filtering of members' movie ratings and movie reviews. "The Movies Dataset" from Kaggle [27], the dataset we are using for this work, contains metadata for all 45,000 movies listed in the Full MovieLens Dataset and 26 million ratings on a scale of 0.5-5 from 270.000 users to around 45.000 movies gathered. This dataset is an ensemble of data collected from GroupLens and TMDB [29], acronym of The Movie Database, which is a community built movie and TV database:

- the movie **Details, Credits and Keywords** are gathered from the TMDB Open API,
- the movie **Links and Ratings** are obtained from the Official GroupLens website.

In this dataset one can also gather IMDb movies ids: IMDb [30] is an abbreviation of Internet Movie Database, an online database owned by Amazon that

contains information related to films and other multimedia content. "The Movies Dataset" from Kaggle is composed of the following comma separated values (csv) files:

- **movies_metadata.csv**, the file where movies' metadata such as genre, language, budget and revenue, release dates, production companies and countries are stored;
- **keywords.csv**, which contains textual movie plot keywords for MovieLens movies.
- **credits.csv**, which consists of cast and crew information for all movies;
- **links.csv**, which contains links between the TMDB and IMDb ids of all the movies featured in the Full MovieLens dataset;
- **links_small.csv**, contains links between the TMDB and IMDb ids of a small subset of around 9.000 movies of the Full Dataset;
- **ratings.csv**, which contains around 26 million ratings from over 270.000 users;
- **rating_small.csv**, a subset of around 100.000 ratings from 671 users on 9.066 movies.

4.1.1 Movies metadata

The Movies Dataset contains metadata information about 45,466 movies featured in the Full MovieLens dataset in movies_metadata.csv file. Each movie has got 24 attributes that are listed in the table below together with their object type, number of non-null values and some deeper description.

Attribute	**Object Type**	**Non-null values**	**Description**
adult	stringified JSON Object	45466	whether the movie is for adults or not
belongs_to _collection	stringified JSON Object	4494	movie collection info, nan if the movie does not belong to any collection

Attribute	Object Type	Non-null values	Description
budget	string	45466	budget for the movie production in USD
genres	stringified JSON Object	45466	id and name of the movies' genres
homepage	string	7782	URL of the movie's site homepage
id	string	45466	tmdbId from The Movie DataBase
imdb_id	string	45449	imdbId from the Internet Movie Database
original_ language	string	45455	original language of the movie
original_ title	string	45466	title of the movie in its original language
overview	string	44512	textual plot of the movie
popularity	float	45461	metric of popularity of the movie from TMDB
poster_path	string	45080	path of the movie's poster
production_ companies	stringified JSON Object	45463	name and id of the movie's production company
production_ countries	stringified JSON Object	45463	ISO name and id of the movie's production countries

Attribute	Object Type	Non-null values	Description
release_date	string	45379	release date of the movie in yyyy-mm-dd format
revenue	float	45460	revenue generated by the movie in USD
runtime	float	45203	duration of the movie in minutes
spoken_ languages	stringified JSON Object	45460	info about all languages spoken in the movie
status	string	45379	whether the movie is released, planned, rumored etc.
tagline	string	20412	short textual slogan of the movie
title	string	45460	title of the movie in English language
video	string	45460	whether the movie has video available
vote_average	float	45460	metric from TMDB dataset
vote_count	float	45460	metric from TMDB dataset

Table 4.1: Attribute, object type, number of non-null values and information about movies_metadata.csv attributes.

Looking at the attributes, each movie has got two different ids: “id” and “imdb_id”, corresponding to tmdbId from The Movie DataBase (TMDB) and imdbId from the Internet Movie Database (IMDb) respectively. Movies can belong to the following 20 genres:

Animation, Comedy, Family, Adventure, Fantasy, Romance, Drama, Action, Crime, Thriller, Horror, History, Science Fiction, Mystery, War, Foreign, Music, Documentary, Western, TV Movie.

Each movie can take up more than one genre. In fact, most of the movies belong to 1 to 3 genres. Not all genres are "equally represented" in the dataset: Drama and Comedy movies are the most present above with 12.500 records against less than 1.250 Western and TV Movies films. One can also notice that summing up all the frequencies, we do not get the total number of movies in the dataset, since each movie can belong to more than 1 genre. In the plot below we show the number of movies per genre.

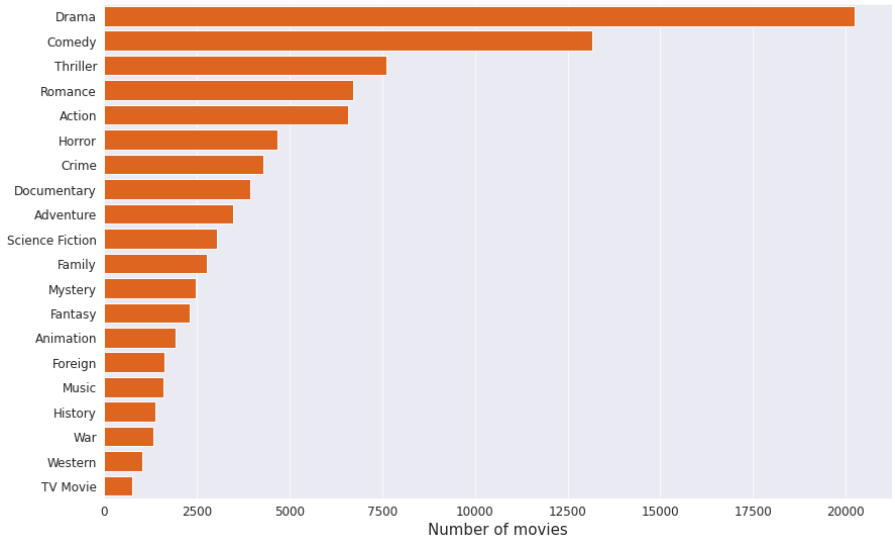

Figure 4.1. Number of movies per genre.

Production Company and Country

Each movie has its own Production Company and Country, there are more than 23.000 unique companies and 160 distinct Production Countries. Most common Production Companies are *Warner Bros, Metro-Goldwyn-Mayer* and *Paramount Pictures* whereas most common countries where movies were produced are the United States, United Kingdom and France.

Language and release date

Every movie has its original spoken language, most common are English, French, Italian and Japanese. Each movie has a release date, from year 1874 to 2020. Every release date is a string in the format "yyyy-mm-dd". As depicted in the barchart below, the dataset contains records from last decades of 1800 century until 2020, but most of the films have been released between 2000 and 2010. In the plot below we can see the distribution of release years of the movies of out dataset.

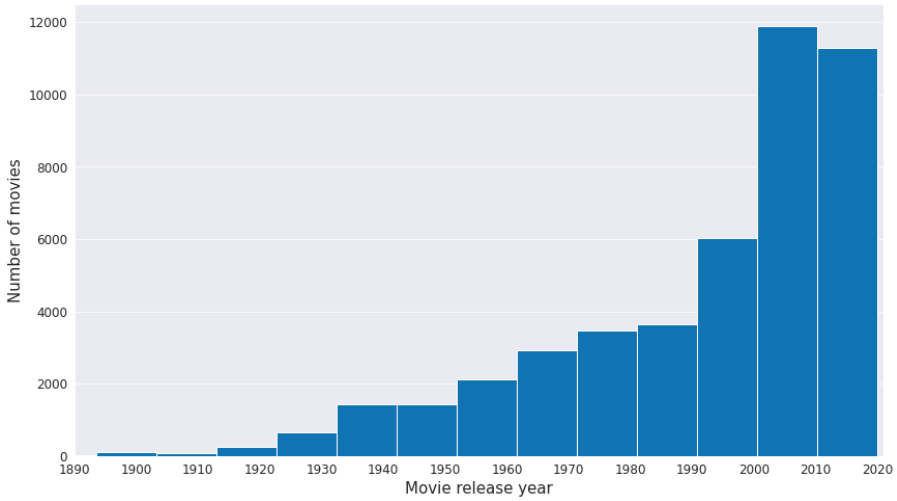

Figure 4.2. Movie release year distribution.

Movie duration

The attribute runtime refers to the duration of the movie in minutes. The average duration for movies is about 94 minutes. By inspecting the average duration by movie genre, we find out that History and War films are the longest, while the shortest are Animation, Family and Documentaries.

Textual plot or "overview"

Overview is a textual attribute (i.e. a string) representing the plot of the movie, written in English language. All plots have an average length of around 317 characters, including spaces. Here is an example of textual overview for the Italian movie "Life Is Beautiful":

"A touching story of an Italian book seller of Jewish ancestry who lives in his own little fairy tale. His creative and happy life would come to an abrupt halt when his entire family is deported to a concentration camp during World War II. While locked up he tries to convince his son that the whole thing is just a game.".

For the sake of visualization, we depict wordclouds for movies belonging to Science Fiction and Music genres based on the magnitude of the Term Frequency Inverse Document Frequency (TFIDF). Tokens with the highest values of TFIDF such as "earth", "world", "planet", "scientist" for Science Fiction and "music", "band", "rock", "concert" for Music are representative of the respective genre. Furthermore, in the plot below we show the Kernel Density Estimate (KDE) distribution of the overview length in terms of number of characters.

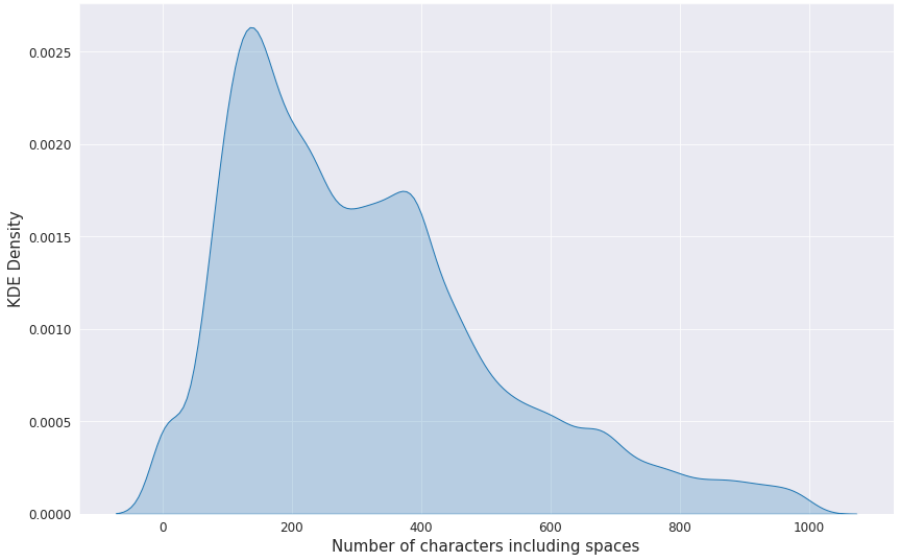

Figure 4.3. Distribution of the number of characters of "overview" attribute, spaces included.

Popularity, Vote Average, Vote Count from TMDB

The vote_average and vote_count attributes are movie-related scores referring to TMDB dataset and they should not be confused with actual vote average and count that will be obtained grouping movie reviews from our ratings.csv or ratings_small.csv from GroupLens. Popularity [31] is a daily

Figure 4.4. Wordclouds for genres Science Fiction (up) and Music (down) based on TFIDF.

metric that ranges between 0,00 and 547,49, computed by TMBD with respect to several factors such as number views, favourites, whatchlist, votes for the day, release date, number of total votes and previous days score. The vote_average is another score in the interval from 0,0 to 10,0 that records the average rating for that movie. vote_count takes values from 0,0 to 14.070,0 and records the total number of ratings received by that movie. In the table below we show the 10 movies with the highest product between vote average and vote count (in order to not highlight those movies having few high ratings), together with their average rating and rating count.

TMDB Ranking	Movie Title	TMDB vote_count	TMDB vote_average
1	Inception	14.075	8,1
2	The Dark Night	12.269	8,3
3	Interstellar	11.187	8,1
4	The Avengers	12.000	7,4
5	Avatar	12.114	7,2
6	Deadpool	11.444	7,4
7	Fight Club	9.678	7,2
8	Django Unchained	10.297	7,4
9	Guardian of the Galaxy	10.014	8,3
10	Pulp Fiction	8.670	7,8

Table 4.2: Top 10 movies per TMDB vote_count and vote_average product.

Directors and actors

In credits.csv file we are given Cast and Crew information for all our movies. These records are available in the form of stringified JSON Object. Each record of this dataset represents a movie, linked with other metadata through its id, which is the same id of the movies_metadata.csv dataset. Cast contains information about the actors that acted in the movie in order of decreasing importance: the main characters are the first ones to be written in the JSON strings with their character's name, real name, gender id and profile path. Most popular actors with number of performed movies are John Wayne 106, Jackie Chan 89, Robert De Niro 86, Michael Caine 86, Gérard Depardieu 84, Christopher Lee 75. Crew contains records about the director of each movie but also people from other departments related to the movie such as Production, Writing, Photography, Sound, Editing Visual Effect, Art. The dataset contains movies directed by 17.573 unique directors, most popular ones are John Ford 66, Michael Curtiz 65, Werner Herzog 54, Alfred Hitchcock 53, Georges Méliès 51 movies directed.

4.1.2 Interactions and ratings

ratings.csv and ratings_small.csv files contain records about interactions gathered from Grouplens between users and movies included in the movies metadata file. Each row of those datasets tells that a certain user, described

by its userId, has watched a certain movie identified by its movieId at a certain timestamp. This user has reviewed it with a certain rating from 0,5 to 5 with steps of 0,5 or. In other terms, $r \in [0.5, 1, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0]$. ratings.csv contains 26.024.289 rows with 45.115 unique movie ids and 270.896 unique users. An example of rating's record is shown below.

userId	**movieId**	**rating**	**timestamp**
1	147	4,5	1425941529

Table 4.3: Example of user review record for a movie from rating.csv or ratings_small.csv

Due to our limited computational resources available on Google Colab we made use of ratings_small.csv dataset, which is a smaller subset of the greater ratings dataset. This choice can be justified by assuming that ratings_small.csv reviews distribution can be a good approximation of ratings.csv. In order to make this statement reasonable and meaningful, we have to ensure that the ratings' distributions of the two dataset are similar.

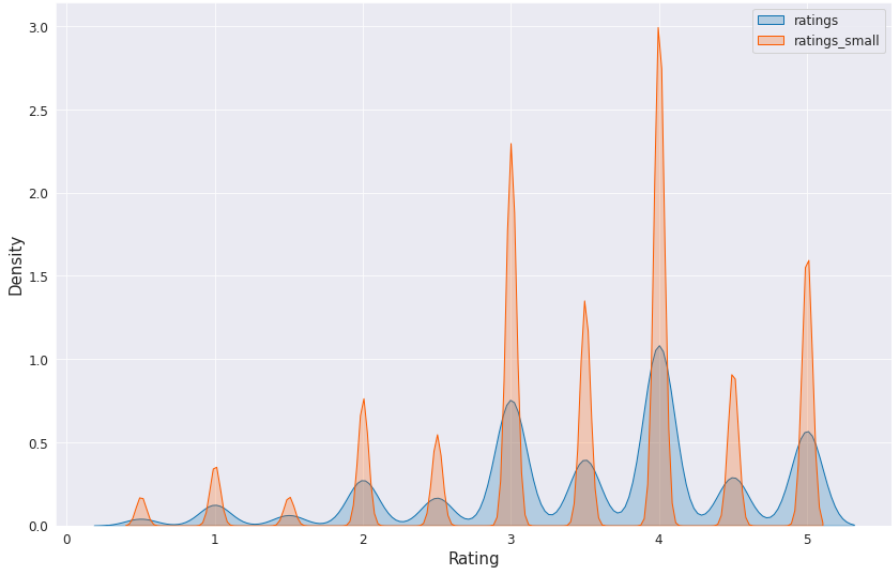

Figure 4.5. KDE plot of ratings and ratings_small distributions.

In order to inspect this similarity, we applied Kernel Density Estimate (KDE) method for visualizing the distribution of observations. KDE represents the data using a continuous probability density curve in one or more

dimensions. Looking at the plot, we can see that reviews have very similar distributions, thus the records in the smaller dataset can be a good approximation of the larger one. Thus, from now on we will always refer to ratings_small.csv dataset which contains 100.004 ratings of 671 unique userId that reviewed 9.066 unique movies. Users have seen and rated 140 movies on average.

The purpose of link.csv file is to link all the files we have mentioned above. It contains the linkage between TMDB and IMDd ids of all the movies featured in the Full MovieLens dataset and ratings.csv or ratings_small.csv. In fact, ids in movies_metadata.csv are not directly associated with ids in the reviews files ratings.csv or ratings_small.csv. Using links.csv or links_small.csv we can obtain the right correspondences. The "movieId" in ratings.csv or ratings_small.csv corresponds to the "movieId" in links.csv or links_small.csv, in which the "imdbId" and "tmdb_id" correspond to the "imdb_id" and "id" respectively in movies_metadata.csv. In short:

- **tmdbId** in links.csv coincides with **id** in movies_metadata.csv
- **imdbId** in links.csv coincides with **id** imdb_id in movies_metadata.csv
- **userId** in links.csv coincides with **id** userId in ratings.csv/ratings_small.csv

We merge movies' ratings dataset with movies metadata using the id of the movies as key of the join in order to extract useful information. On average, each user has reviewed around 148 movies whereas each movie has got on average 11 reviews. The distribution of the number of movies reviewed per user and per movies is showed below with boxplots.

The user with the lowest number of reviews has rated 19 movies. This aspect will be interesting when splitting the dataset for training. There exist some outlier user that has seen more than 300 movies, they are represented as individual points beyond the whiskers on the box-plot. Another important information is that 9.013 out of the more than 45.000 movies from movies_metadata have been reviewed, by considering the subset of rating records in ratings_small.csv. The 10 movies in terms of higher product (sorted decreasingly) between Grouplens vote average and vote count are shown in the table below together with their average rating and rating count.

Grouplens Ranking	**Movie Title**	**# reviews**	**Avg. Rating**
1	The Shawshank Redemption	311	4.49

Grouplens Ranking	Movie Title	# reviews	Avg. Rating
2	Forrest Gump	341	4.05
2	Pulp Fiction	324	4.26
4	The Silence of the Lambs	304	4.14
5	Star Wars	291	4.22
6	The Matrix	259	4.18
7	Schindler's List	244	4.30
8	Jurassic Park	274	3.70
9	The Empire Strikes Back	234	4,23
10	Toy Story	247	3.87

Table 4.4: Most popular movies according to Grouplens number of reviews and average rating product.

We can consider popular those movies with a high number of reviews but also a high average rating. By comparing this ranking with the top10 movies with the highest TMDB "popularity" (explored in the previous table), we can confirm that Grouplens records and TMDB scores are different, either using the full reviews' dataset or the smaller one. As showed in the following table, the top 5 genres reviewed by users, in terms of number of review's records, are Drama, Comedy, Thriller, Romance, Action in decreasing order of ratings.

Ranking	Movie genre	# reviews
1	Drama	47021
2	Comedy	35864
3	Action	25650
4	Thriller	25556
5	Adventure	21882
6	Romance	18834
7	Crime	17472
8	Science Fiction	15150
9	Fantasy	12515
10	Family	11865
11	Mystery	8803
12	Horror	6139
13	Animation	6119

Ranking	Movie genre	# reviews
14	History	3968
15	Music	3914
16	War	3776
17	Western	1610
18	Documentary	1453
19	Foreign	199
20	TV Movie	58

Table 4.5: Most popular genres in terms of GroupLens number of reviews.

For each genre, we can also show its average rating. It is to be noticed that the on average movies are always rated more than 3. This aspect will be mentioned when discussing about the metrics used for evaluating the performances of our models.

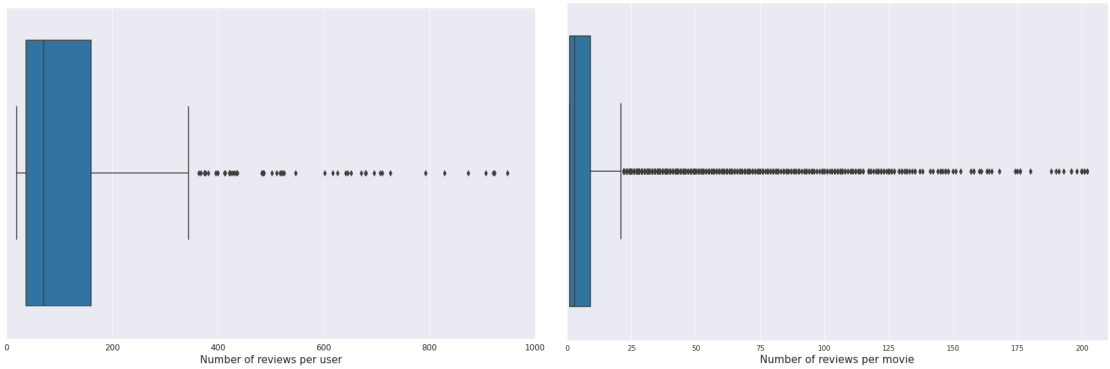

Figure 4.6. Boxplots representing the distribution of the number of reviews per user (left) and per movie (right).

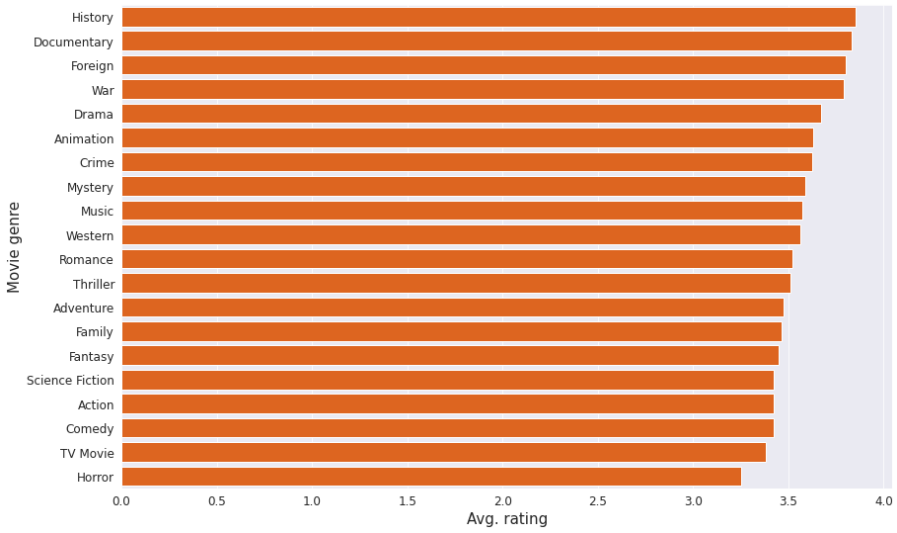

Figure 4.7. Average rating per movie genre.

Chapter 5

Data Preprocessing

In this section we dive into the main operations that were carried out on the original data before feeding them to recommendation models. We explore all the data processing steps from data cleaning to feature selection and engineering. The processing of some feature has been different depending of whether they were thought to be used by content-based and collaborative models. Thus, for each informative attribute retained from the native dataset, we explain which preprocessing was made depending on the recommendation model. We are going to particularly focus on textual-features encoding with NLP techniques. The main work on the data were conducted using *numpy* arrays and *Pandas* DataFrames data structures.

5.1 Data cleaning

Many values of the dataset attributes, such as genre, actors and directors were availabe in the form of strings representing JSON objects. Here an example for a movie genre:

"['id': 16, 'name': 'Animation', 'id': 35, 'name': 'Comedy', 'id': 10751, 'name': 'Family']".

Stringfied JSON objects were treated using *literal_eval* function from *ast* Python module in order to easily extract key and values of the stringfied JSON dictionary. As showed in previous Dataset chapter, some movies metadata attributes present some null or nan value. Outlier rows in movies_metadata.csv having dates instead of movie ids, numbers instead of language of the movie or other irregularities were removed. Moreover, some duplicated rows or

containing too many nan values were found and deleted. In the previous chapter we also discussed about the presence of two different movie ids for a given film record: TMDB and IMDb ids. Since imdbId attribute has some outliers, tmdbId was preferred and used to link the several csv files of the dataset. After applying those data cleaning operations, from the original 45.466 movie records, we end up having 44.405 of them. When merging then ratings_small.csv with movies_metadata the resulting number of movies reduces to 9.013, since those file is a subset of the original reviews' dataset. Moreover, movie ids where transformed in order to become consecutive from 0 to the total 9013. Rating datasets are instead discretely clean: there are neither empty or nan values or duplicated rows, no user has reviewed a movie more than once. By removing some outlier movies, the total number of reviews in the dataset passed from 100.004 to 99.796. Timestamp attribute values were converted into *datetime* objects.

5.2 Feature selection

Not all the attributes from The Movies Dataset were preserved. Some features were removed for not being informative or for having too many outliers or missing values, others were deleted because of pairwise high correlation. After merging movies metadata records with rating ones, we conducted a correlation analysis in order to inspect pairwise correlation between couples of numerical attributes, since they may have negatively affected recommendation models. For this purpose, we exloited *Pearson correlation coefficient* (or linear correlation) between two statistical variable X and Y, which is the covariance of the two variables cov(X,Y) divided by the product of their standard deviations σ_X and σ_Y. The formula for a sample, obtained by substituting estimates of the covariances and variances, is shown below:

$$r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}} \tag{5.1}$$

We built correlation heatmaps between the following attributes: rating, release_year, revenue, vote_average, vote_count, runtime, popularity, budget.

The attribute vote_count resulted to be highly and positively correlated with budget and popularity. Revenue's attribute was highly and positively

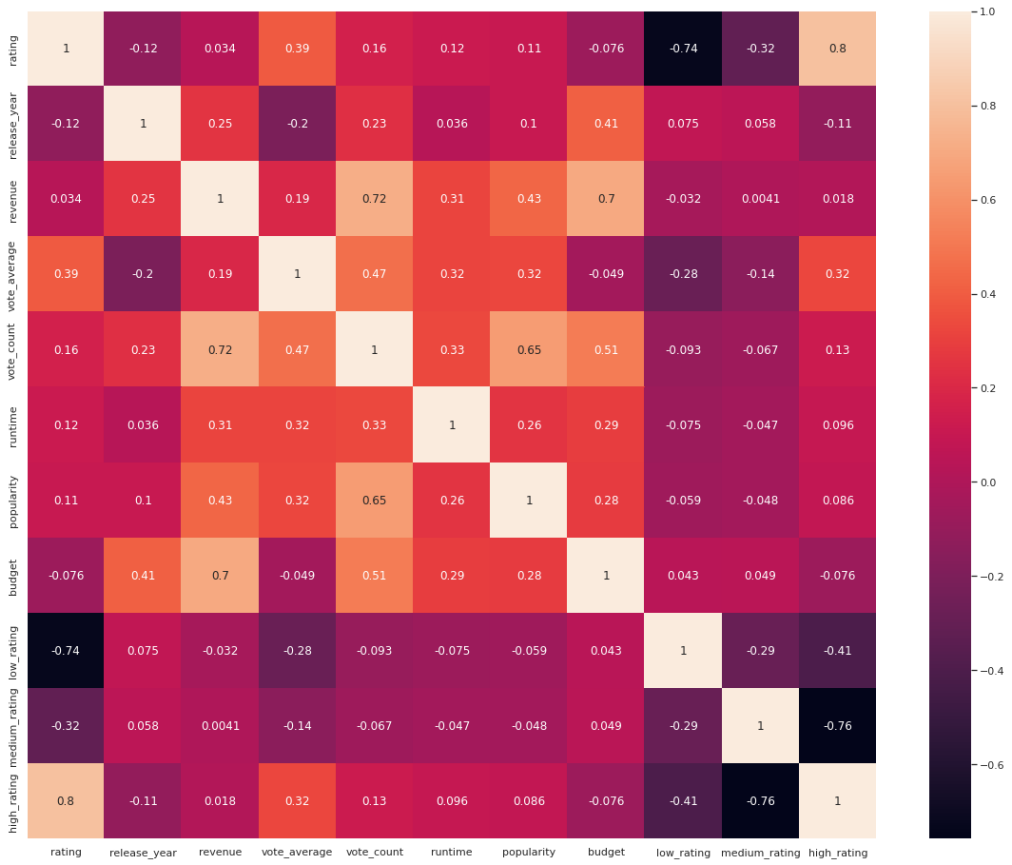

Figure 5.1. Pearson Correlation heatmap for numerical attributes and rating.

correlated with budget, popularity and vote_count. Moreover, the rating is positively correlated with vote_average, showing that the TMDB score is probably coherent with our dataset reviews. For the sake of visualization, we also show positive and negative correlation using pairwise scatter plots between couples of the following attributes: budget, vote_count, popularity, revenue. On the diagonal we can see can also see the KDE distributions for those attributes divided by rating_type, a custom categorical attribute which is assigned to each record. Given the rating $r \in [0.5, 1, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0]$, the attribute is generated as follows:

- **Low rating**, when $r \leq 2$
- **Medium rating**, when $2 < r < 4$
- **High rating**, when $r \geq 4$

More positively correlated features tend to have scatters that follow the direction of the bisector, which means direct proportionality: the higher one attribute the higher the other. This is the case for attributes budget, revenue, vote_count, popularity, since many of them show high positive correlation. The diagonal KDE plot do not show attributes' strong capability of distinguish between rating types, given that KDE distributions are almost overlapping.

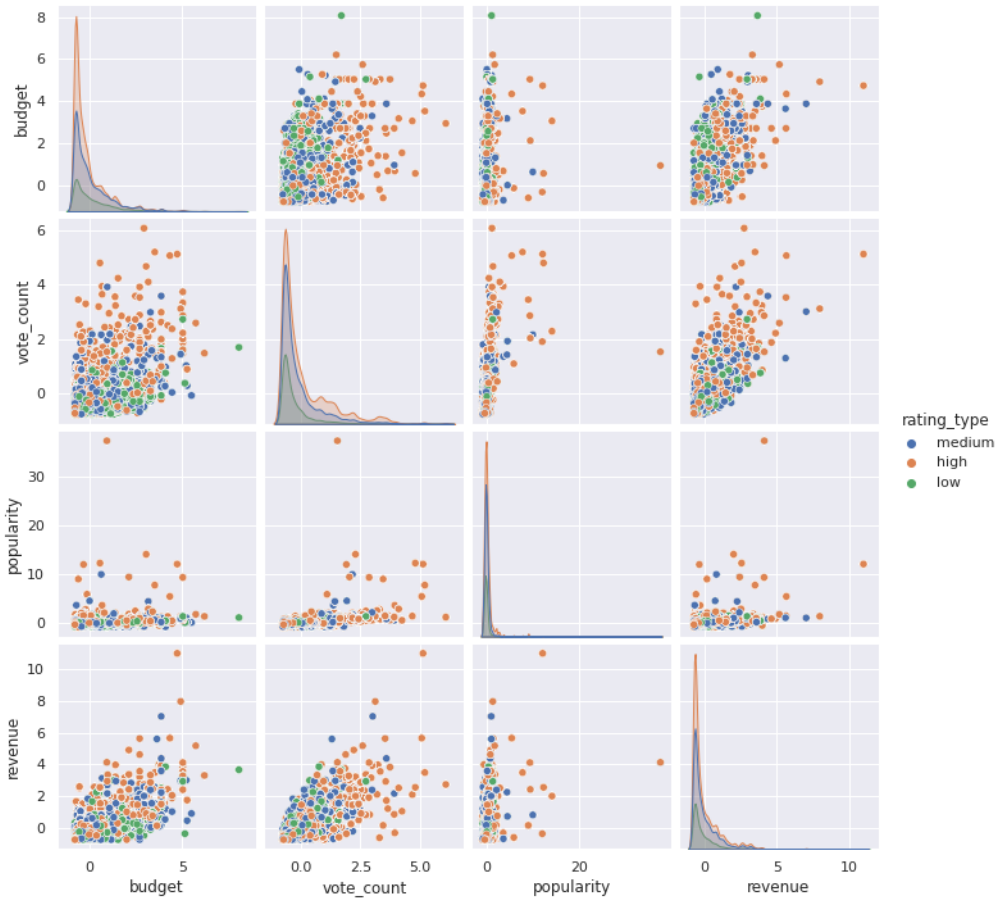

Figure 5.2. Pairwise scatter plot between couples of attributes.

Therefore, given those high positive correlations, we decided to keep in the dataset only the following features: popularity, runtime, release_year, vote_average. After applying one-hot encoding to the list of genres per movie, we also plotted correlation heatmap for genres.

Most of the genres show *Pearson correlation* score close to zero, but it is interesting to notice how Family and Animation, Action and Adventure,

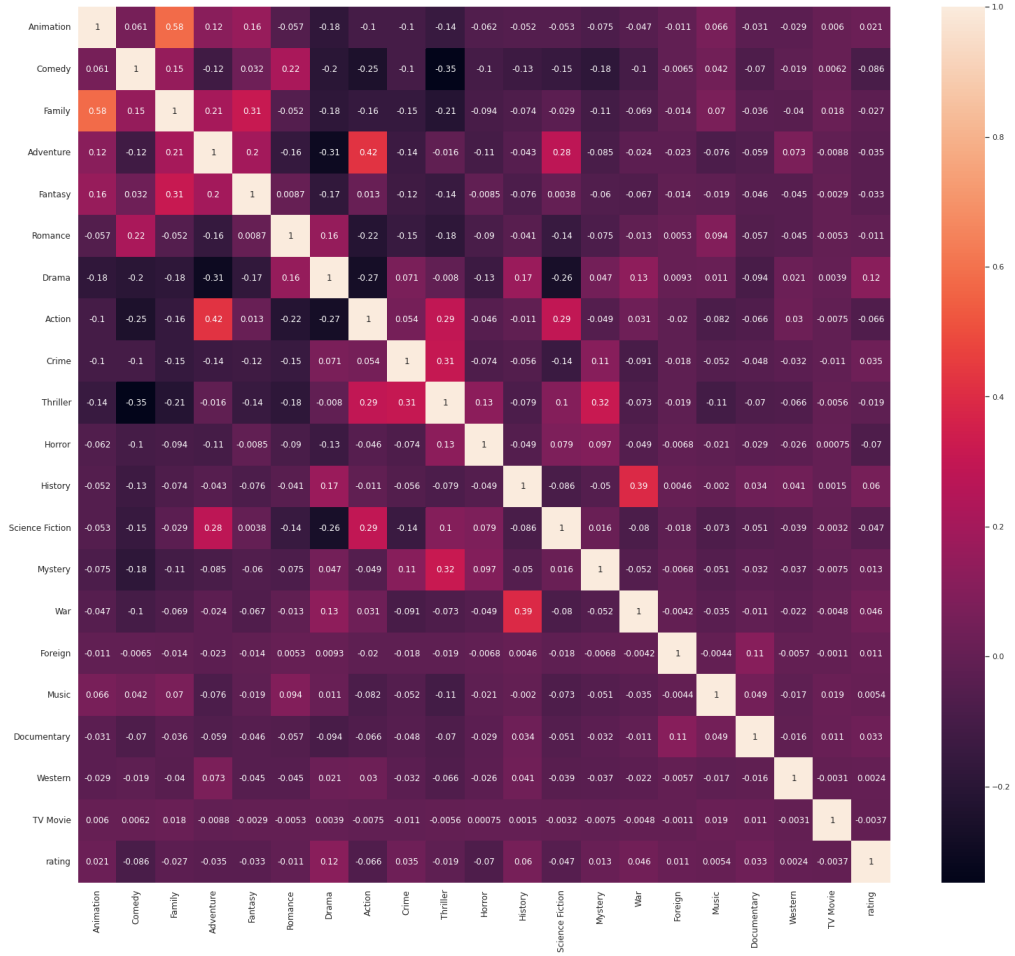

Figure 5.3. Pearson Correlation heatmap for genres and rating.

History and War, Comedy and Thriller are correlated, the last couple with negative correlation coefficient. This means that a Family movie tends to be also an Animation movie or, by contrary, a Thriller movie will unlikely be a Thriller. Finally, those shown in the table below were the attributes kept out of the original dataset and the reason for exclusion.

Reason for exclusion	**Attribute**
Not intrinsically informative	video, poster_path, spoken_languages, adult, status
Too many nan values	homepage, belongs_to_collection, tagline, keywords

Reason for exclusion	Attribute
Too many unique values	production_companies, production_countries
High correlation with other attributes	budget, revenue, vote_count, popularity
Redundant	imdbId, original_title

Table 5.1: Justification for the exclusion of several features from the original dataset.

5.3 Data normalization

For content-based models, numerical features were standardized, since they were differently scaled. In fact, popularity is floating score from 0.00 to 547.49, runtime an integer expressing the duration of a movie in minutes, release_year is a year, vote_average is a number from 0,00 to 10.00, vote_count takes values from 0,0 to 14.070,0. StandardScaler class methods from *sklearn* library were used. It standardizes features by removing the mean of the attribute and scaling to unit variance as follows:

$$\mathbf{z} = \frac{\mathbf{x} - \mu}{\sigma} \tag{5.2}$$

where μ is the mean and σ the standard deviation of all the occurrences x_i of the attribute-vector $\mathbf{x}$. Regarding BERT embeddings obtained with *SentenceTransformer*, each of them is already zero-mean-centered with standard deviation 0.05.

5.4 Feature Engineering

5.4.1 Movie genres

Movie genres were treated differently between content-based and collaborative filtering models using LightFM. Given that each movie can take up several genres, in content-based models genres were encoded using a One-Hot Encoding. In other words, we obtain for each genre a vector of boolean values from a list of genres. In practise, each of the 20 genres becomes an attribute of the dataset and every movie record has its own genres represented by a vector of shape 20: $\mathbf{x^m}$ the one-hot vector of the movie record m. Given a

genre g such as "Comedy" or "Science Fiction", the one-hot vector $\mathbf{x^m}$ can take values:

$$x_g^m = \begin{cases} 1 & \text{if the movie m belongs to genre g} \\ 0 & \text{otherwise} \end{cases}$$

with $g \in [$*Animation, Comedy, Family, Adventure, Fantasy, Romance, Drama, Action, Crime, Thriller, Horror, History, Science Fiction, Mystery, War, Foreign, Music, Documentary, Western, TV Movie*$]$ and $m \in [0, 9013]$

Given that each movie can take up more than 1 genre, each one-hot vector can have multiple ones. In all collaborative models where item features were used, instead, genres were provided as strings were all the genre were concatenated, since LightFM requires categorical attributes. For instance, for our collaborative filtering models, the movie ToyStory has got genre "Animation, Comedy, Family". Different combinations (in terms of position of the genres in the string) of the same set of genres are all ordered in the same way, in order to make them appear equal to LightFM models.

5.4.2 Release date splitting

Only the year was kept from the release date of each movie, since it was considered a sufficiently informative attribute within the data. For instance, we kept only "2020" from a string "2020-12-16" by simply splitting it.

5.4.3 Movie original language encoding

The original language of the movie was also one-hot encoded. Among the 89 unique languages, only the 7 most frequent were kept. All remaining languages having lower number of records were represented as a generic feature "other". The following are the languages that were kept: English (en), Spanish (es), French (fr), Italian (it), Japanese (ja), German (de), Russian (du) and "other", representing all the other languages.

5.4.4 Vote average, popularity and runtime

As described in the Dataset section, vote_average, popularity and runtime are numerical features having totally different scales. For content-based models, each one of those features was standardized removing mean and dividing by the standard deviation. For collaborative filtering models using

item metadata, they needed to be categorized in order to be digestible to LightFM. Categorization was carried out exploiting *qcut* quantile labels with a quantile-based discretization function from *Pandas*, which discretize variable into equal-sized buckets based on rank or based on sample quantiles. Used quantiles are [0, 0.25, 0.75, 1]. Therefore, each movie has its vote average, popularity and runtime quartile in the range [0-25Q, 25-75Q, 50-75Q, 75-100Q].

5.4.5 Movie textual overviews embedding

Overviews of the movies were converted into vectors for all of the models that used item metadata. They were, in other words, stored as dense vector representations. In contrast to sparse vectors, the word "dense" alludes to the fact that every member inside the vector contains a value. As a result, movie textual plots were embedded in a vector space with comparable embeddings close together. These embeddings may then be compared to locate phrases with comparable meanings, for example using cosine-similarity. BERT (Bidirectional Encoder Representations from Transformers) is the state-of-the-art language model for making dense vectors with context. Each of its encoder layers produces a set of dense vectors, the size of which is determined by the BERT model utilized, which is commonly 384 or 768. We utilized them as contextual word embeddings since they are numerical representations of a single token. Then, using clustering approaches, word embeddings from the same phrase were handled together in order to obtain a semantic representation of the text and identify important topics. Two main methods were explored for sentence embedding:

- Using **BERT pre-trained transformer models from *HuggingFace*** [32], by extracting the last hidden layers and making mean pooling operations. *bert-base-uncased* pre-trained model was used for this purpose, it is a model pretrained on BookCorpus, a dataset consisting of 11.038 unpublished books and English Wikipedia. Before passing sentences to the model, the *encode* method from BertTokenizer class is exploited for getting the list of input IDs with the appropriate special tokens of the sentence. Those IDs are token indices, numerical representations of tokens building the sequences that will be used as input by the model. Since there is one vector representing each token (output by each encoder), we are actually looking at a vector of size K by the number of tokens, where K is usually equal to 384 or 768. We can transform those vectors to create semantic representations of the input sequence. The

simplest and most commonly extracted tensor is the *last_hidden_state* tensor which is conveniently given in output by the BERT model and it represents the sequence of hidden-states at the output of the last layer of the model. We need to convert the *last_hidden_states* tensor to a vector of K dimensions using a mean pooling operation. Each of the tokens has respective K values. The pooling operation will take the mean of all token embeddings and compress them into a single 384/768 vector space creating a sentence vector.

- Using ***SentenceTransformer*** [21] library models, in particular *all-MiniLM-L6-v2* model, trained on a large and diverse dataset of over 1 billion training pairs. It maps sentences and paragraphs to a 384 dimensional dense vector space and can be used for tasks like clustering or semantic search. *SentenceTransformer* models take care of tokenization and encoding too. A list of sentences (our movie textual overviews) long as the number of movies of the dataset is passed to the model encoder, which then outputs an embedding matrix of dimensions (M, 384), where M is the number of movies rated in the dataset. Under the scenes, what this model does is to first pass the input through the transformer model, then to apply the right mean pooling-operation on-top of the contextualized word embeddings. We can generate a fixed-size representation for input phrases of various lengths using the pooling layer. The authors of SBert paper tested with several pooling algorithms, including MEAN and MAX pooling, as well as using the CLS token that BERT creates by default. Max length of sentences was increased from the default value of 256 characters to the maximum length overview in the dataset.

The second option, those with pretrained *SentenceTransformers* models, was preferred since it ended up to be the most easy to implement and efficient, given our available computational resources. In the scatter plot below, we show the average overview embedding per genre after PCA. Each 2-dimensional record representing a genre is obtained by averaging all the BERT embeddings of movies belonging to that genre. For each genre we obtained a vector of 384 entries. Principal Component Analyisis (PCA) was applied here with the purpose of reducing dimensions of the date for illustration purposes. This technique linearly transforms data by simultaneously mapping them into a new space whose dimensionality is smaller and tries to keep most of the information of the original data. What PCA does is finding the "directions" of the data that explain most of the information present in them.

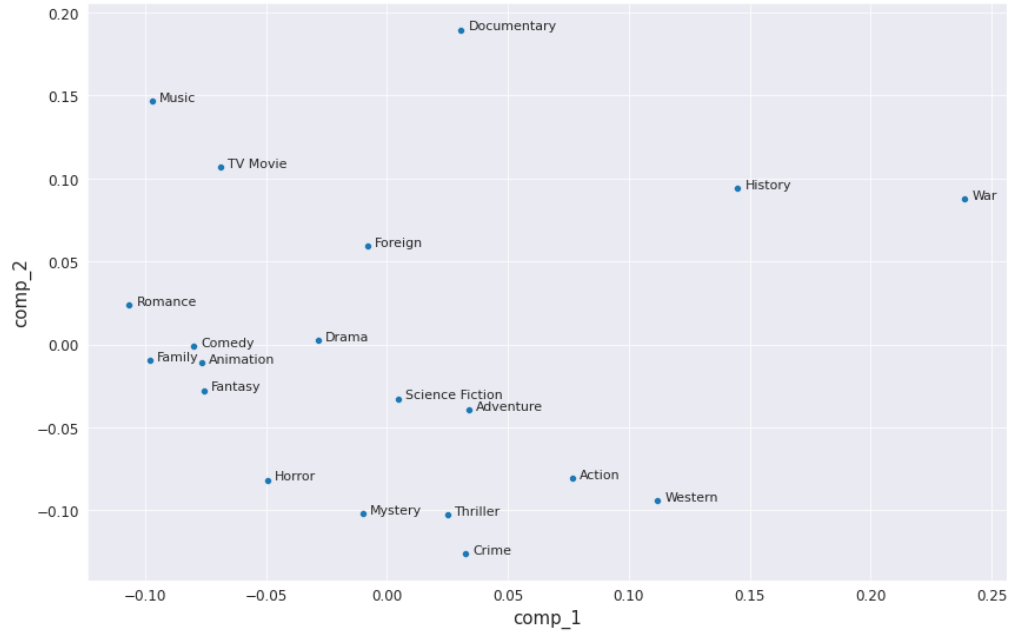

Figure 5.4. Movie overview BERT average embedding per genre after applying PCA with n_components=2

The plot clearly shows how BERT encoding is capable of representing semantic meaning and context from sentences. Although we are making an average of several embeddings and we are restricting the entire space from 384 to 2 dimensions, we can still see how overviews belonging to similar genres end up to be closer in the embedding space. For instance, History is reasonably close to War, Crime is close to Thriller and Mistery, Science Fiction is close to Adventure. Once movie overview embedding are obtained using BERT, we can compare them and find the most similar ones, that will be the closest ones in that embedding space. For instance, in the table below we show most similar overview to that of "Life Is Beautiful", an Italian movie about the story of a Jewish family during World War II.

Movie Title	**Cosine Similarity**	**Overview**
Life Is Beautiful	1.0000	A touching story of an Italian book seller of Jewish ancestry who lives in his own little fairy tale. His creative and happy life would come to an abrupt halt when his entire family is deported to a concentration camp during World War II. While locked up he tries to convince his son that the whole thing is just a game.
Europa Europa	0.6015	A Jewish boy separated from his family in the early days of WWII poses as a German orphan and is taken into the heart of the Nazi world as a 'war hero' and eventually becomes a Hitler Youth. Although improbabilities and happenstance are cornerstones of the film, it is based upon a true story.
August Rush	0.5523	A drama with fairy tale elements, where an orphaned musical prodigy uses his gift as a clue to finding his birth parents.
The Book Thief	0.5457	While subjected to the horrors of WWII Germany, young Liesel finds solace by stealing books and sharing them with others. Under the stairs in her home, a Jewish refuge is being sheltered by her adoptive parents.
Schindler's List	0.5379	The true story of how businessman Oskar Schindler saved over a thousand Jewish lives from the Nazis while they worked as slaves in his factory during World War II.

Movie Title	Cosine Similarity	Overview
Captain Corelli's Mandolin	0.5343	When a Greek fisherman leaves to fight with the Greek army during WWII, his fiancee falls in love with the local Italian commander. The film is based on a novel about an Italian soldier's experiences during the Italian occupation of the Greek island of Cephalonia (Kefalonia).

Table 5.2: Most similar movies to "Life Is Beautiful" in terms of cosine similarity between movie overview BERT embeddings.

We can see how the most similar overviews in terms of cosine similarity, using *sklearn*, are all movies in which the themes World War II, Jewish deportation, father-son relationship.

5.4.6 Overview clusters using HDBScan on BERT embeddings

When making recommendations with the use collaborative LightFM models, movie overview embeddings had to be categorized in order to let the model digest them. For this purpose, we exploit *Topic Modelling with BERT embedding* technique explained in Chapter 2 (which consists in UMAP dimensionality reduction plus HDBScan clustering), applied to movie overview embedding with the aim of extracting clusters of textual plots.

HDBScan clustering does not receive any number of clusters as input (as other clusters techniques such as KMeans), but it finds them autonomously depending on the hyperparameters set and leaving one cluster out for unassigned data points. Therefore, in order to derive a reasonable final number of clusters, we assume that each cluster we want to output from HDBScan should represent a plot theme within a set of movie genres. The hypothesis made here was to extract a number of clusters from movie overviews that should have been indicatively a reasonable multiple of the total number of genres. A way for inspecting the goodness of the clustering for a given set of hyperparameters is to depics a 2D illustration of the clusters, after applying UMAP dimensionality reduction.

Coloured points represent movies assigned to one of the 38 found clusters.

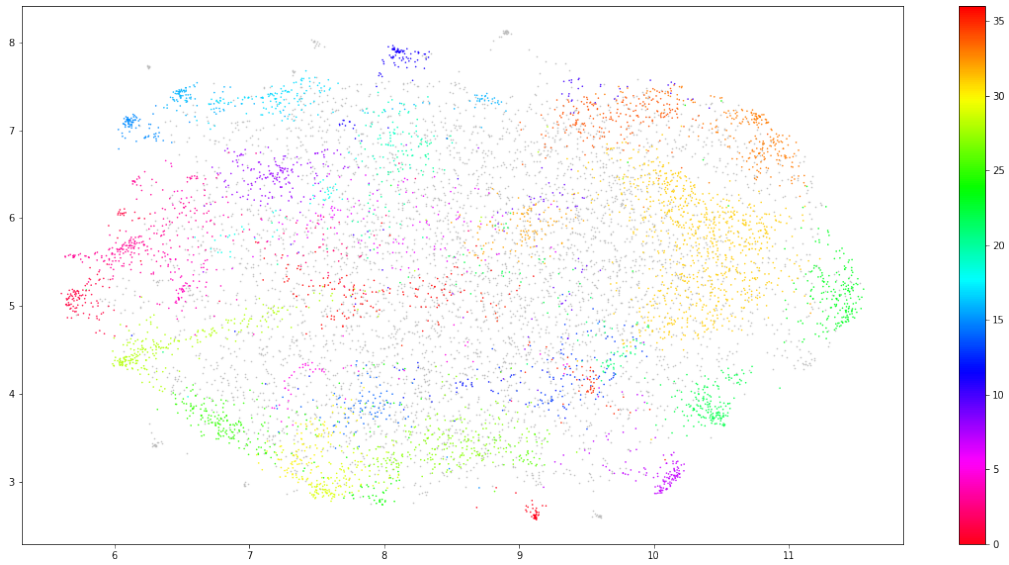

Figure 5.5. Movie overview clusters obtained with HDBScan Clustering on BERT embeddings.

4.158 grey points, the majority of the movies, are not assigned to any cluster, or better they are assigned to -1 cluster. All other clusters have an average size of 131 movies. Being a density base algorithm, HDBScan clustering is capable of detecting high density areas. Groups of closer points tend to be depicted with same colour, which means that they are clustered together. HDBScan clustering hyperparameters were set as follows:

- min_cluster_size = 40,
- metric = 'euclidean',
- min_samples = 1,
- cluster_selection_method = 'eom'

The most influencial parameter for this clustering was min_cluster_size, the smallest size grouping we wish to consider a cluster. It was coherently set close to the minimum number of movies belonging to a genre, explored in the previous chapter. Another method for ensuring the quality of the clusters found is by inspecting the most relevant tokens per clusters, which is done using TF-IDF topic extraction after overview concatenation, as described in Chapter 2. Below, we show two examples of movie overview clusters. Wordcloud on the left shows that main topic of that cluster could be martial

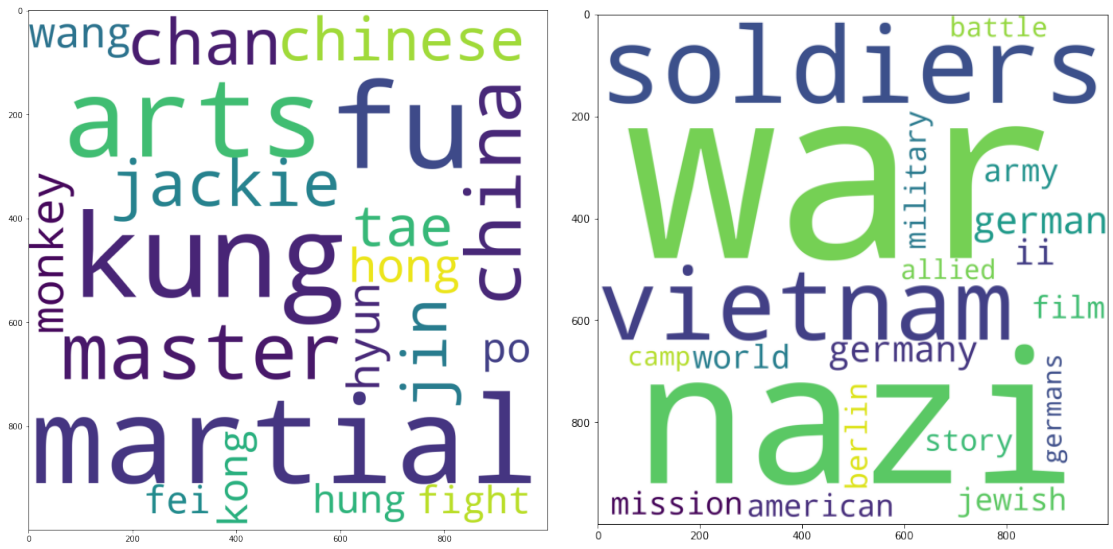

Figure 5.6. Wordclouds depicting tokens with highest TFIDF score within two overview clusters.

arts and oriental culture, given the words kung, fu, china, martial, arts, fight. In fact, some of movies belonging to this clusters are Kung Fu Panda, The Karate Kid, Street Fighter. Wordcloud on the right side contains words like war, nazi, vietnam, soldiers, germany, army, jewish let us think about a clusters in which main topics have to do II World War or Vietnam War related stories and battles. In fact, in this cluster we found movies like: Anne Frank Remembered, Blind Spot: Hitler's Secretary, Nuremberg or Shinder's List.

5.4.7 Movie director and actors embedding with BERT

The director and main actors of the movie were considered informative for the recommendation: a user could be enthusiast of films made by the same director or be a fan of some actors and like their movies. One movie has only one director whereas more than one actors acted for it. Since actors are recorded in order of importance, as explained in the Dataset chapter, we decided to keep 3 most relevant actors for each movie. The encoding of the movie director attribute and main actors was made using BERT in the following way: for each movie the director surname was extracted from the crew attribute and the actors' surnames from the attribute cast of credits.csv. Each movie will be assigned a string with director, first main actor, second

main actor, third main actor concatenated using spaces. For instance, the movie "Edward Scissorhands" directed by Tim Burton and acted by Johnny Depp, Winona Ryder and Dianne Wiest will be assigned this string: "Burton Depp Ryder Wiest". Each of those strings is passed through BERT language model using *all-MiniLM-L6-v2* from *SentenceTransformer*, embeddings are obtained and used for finding movies with similar director and cast using cosine similariy. We can see an example of the most similar movies to "Edward Scissorhands" for director and cast feature in the table below.

Movie Title	**Cosine Similarity**	**Director & Cast**
Edward Scissorhands	1.0000	Tim Burton, Johnny Depp Winona Ryder, Dianne Wiest
Beetlejuice	0.7235	Tim Burton, Geena Davis, Alec Baldwin, Winona Ryder
Corpse Bride	0.6702	Tim Burton, Johnny Depp, Freddie Highmore, David Kelly
Ed Wood	0.6642	Tim Burton, Johnny Depp, Helena Bonham, Carter Emily Watson
Dark Shadows	0.6533	Tim Burton, Johnny Depp, Helena Bonham, Carter Alan Rickman
Sweeney Todd: The Demon Barber of Fleet Street	0.6481	Tim Burton, Johnny Depp, Michelle Pfeiffer, Helena Bonham Carter

Table 5.3: Most similar movies to "Life Is Beautiful" in terms of cosine similarity between director and actors BERT embeddings.

Movies in which Tim Burton worked together with other actors, particularly Johnny Depp, tend to have cosine similar director and cast embeddings.

Director and actor clusters using HDBScan on BERT embeddings

For LightFM collaborative filtering models, director and actors embedding need too be categorized too. The same procedure described for overview BERT embedding is applied here by adapting UMAP dimensionality reduction and HDBScan clustering hyperparameters. The resulting clusters, after a dimensionality reduction, are shown in the 2D plot below.

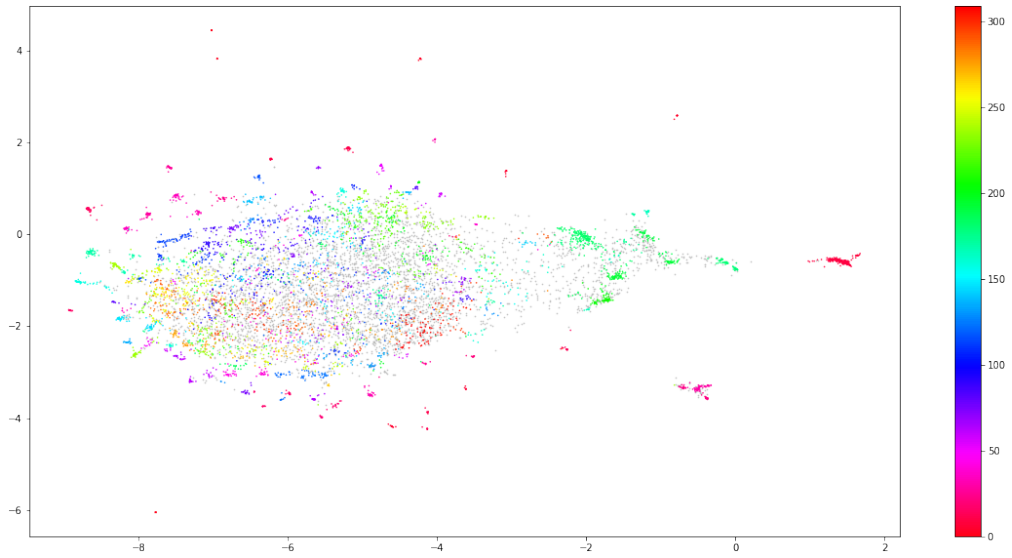

Figure 5.7. Director and actor clusters obtained with HDBScan Clustering on BERT embeddings.

Director and actors clusters tend to arrange in a more dense way with respect to those of overviews. Looking at the 2D plot, we can see many small groups of movie data points. Final resulting number of clusters is 313, with 4.220 movies not assigned to any cluster. By inspecting most relevant tokens in those clusters, we can see how HDBScan over BERT embeddings is capable of put together movies with similar directors or actors as we show in the following wordclouds for some of the largest clusters found. In those cases, clusters group together directors and actors that worked frequently together and consequently belonging to the same country: Italy on the left and France on the right.

5.4.8 User synthetic features

The Movies Dataset does not provide user intrinsic metadata such as sex, job, income of the people watching and reviewing their movies. Thus, for the collaborative model exploiting user features, the only way to get users' metadata was to infer them from the movies metadata the user has seen and rated. This procedure introduces an issue when it comes to user cold start, we will discuss it in the dedicated paragraph. Apart from the user id, the user features extracted by movies metadata were:

- **Preferred genres**, top3 genres seen by that user;

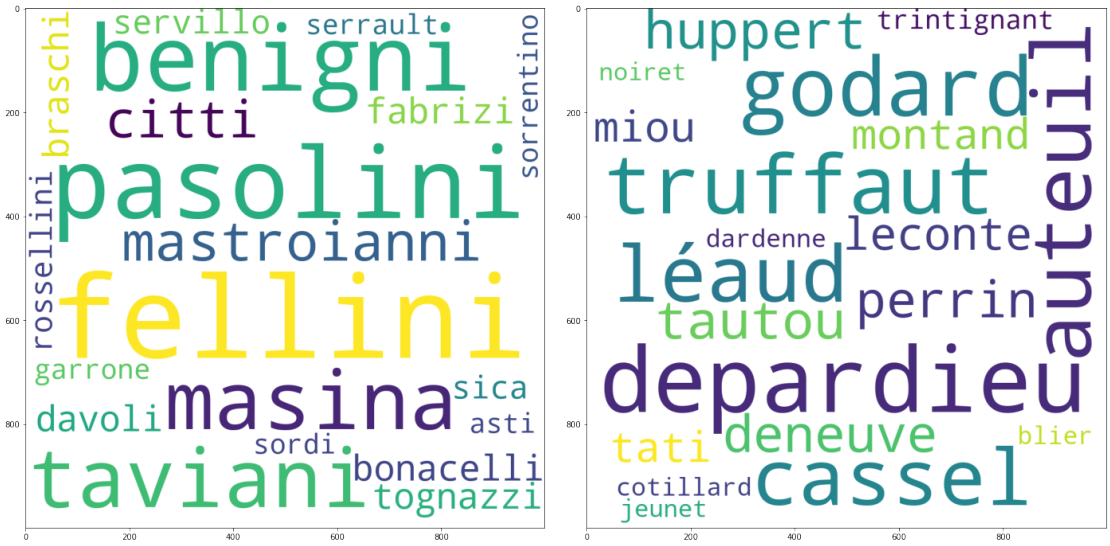

Figure 5.8. Wordclouds depicting most relevant tokens within two director and cast clusters.

- **Top cast string**, top3 actors performing in movies seen by that user, always ordered in the same way in order to be categorical;
- **Top directors string**, top3 directors making movies seen by that user, always ordered in the same way in order to be categorical;
- **Top overview clusters**, top3 overview clusters whom the movies that user saw belonged to;
- **Top directors and cast clusters**, top3 director and cast clusters whom the movies that user saw belonged to;
- **User average rating**, average rating assigned by that user to the movies it has seen;
- **User rating count**, number of reviews of that user.

Chapter 6

Recommendation Models

In this section we describe the rationale and the functioning of each method that was used for our experiments on recommendation systems on The Movies Dataset. We mainly built and tested four typologies of models:

- **Custom pure content-based model**, it uses only the item metadata of movies the users have interacted with in order to provide them topK recommendations;

- **Pure LightFM collaborative filtering model**, a simple matrix factorization model which exploits only users-items interactions or ratings in order to make topK recommendations;

- **LightFM Hybrid model with item metadata**, a matrix factorization collaborative model that exploits users-items interactions and movies' features too;

- **LightFM Hybrid model using both item and user metadata**, a matrix factorization collaborative model that exploits users-items interactions but also users and movies features.

We will also describe how user cold-start and item cold-start problems were simulated and evaluated. In Chapter 7 related to the Experiments, we will discuss more in details which were the metrics used for evaluation and we will show the results obtained. Those will be commented in the Chapter 8.

6.1 Pure Content-based model

As introduced above, we primarly built a pure content-based filtering model which for each users exploited only movies' metadata he has seen for providing him a recommendation. User's suggestions are made by only looking at its past tastes and preferences. The main steps of this model are for each user:

1. The generation of the user profiling vector with weighted average by rating of the its watched movies' vectors,

2. The comparison between this vector and other movies in the corpus,

3. Finally, the scoring of the topK recommended movies.

Among all attributes of "The Movies Dataset", discussed in the Pre-processing chapter, the followings are the attributes that were used by the.pure content-based model:

- One-hot encoded vector of the movies genres;
- One-hot encoded vector of Top 8 languages;
- Movie Overview BERT embeddings;
- Director and Cast BERT embeddings;
- Movie Runtime standardized;
- Movie Release year standardized;
- TMDB Movie Popularity standardized;
- TMDB Movie Vote average standardized.

Elements of those attributes' vectors are stacked together in the same vector for obtaining movie vectors. Every movie is represented by a vector and every user's representation depends on the movies' representations of items he has seen and rated in the past. This user representation will be called user profiling vector.

6.1.1 User Profiling Vector

The user profiling vector is the vector that represents a certain user's preferences, based on its previous interactions with items in the dataset. For the purpose of this thesis, we are going to use **weighted average over ratings** in order to compute the user profiling vectors from movies' vectors. For each couple of user-movie, we are interested in the magnitude of the interaction, which is the rating from 0.5 to 5 that the user has given to that movie. The rationale behind the usage of the weighted average instead of a simple average is to discriminate between positive and negative interactions based on a weight: we give more importance to movies that have been rated higher with respect to those that the user has reviewed with a low rating. For each user we compute its user profiling vector, which will be then compared with all the movies of the corpus to find the topK most similar items with the highest cosine similarity.

6.1.2 Content-based recommendation

For each user, identified by its user id, we compute its user profiling vector $\mathbf{p}$ with the only contribution of metadata of the movies it has interacted with (those within the training set). Thus, with content-based model we completely do not take into account ratings of other users of the dataset. The user profiling vector has shape number of stacked features and it should represent a combination of all pre-processed metadata related to the movies watched by that user, as we discussed in the Data Preprocessing chapter. Given a user, it is represented by movie vectors of each movie he has interacted with or, in other words by a matrix U of shape (m, f), where m represents the number of movies the user has seen and rated and f is the total number of stacked features. The user's ratings for those movies are stored in vector $\mathbf{r}$, where r_i is the rating for movie m. For each user, we compute a weighted average for each column vector $\mathbf{u}$ of the matrix U using ratings r_i as weights. We obtain user profiling vector $\mathbf{p}$, its f elements are computed as follows:

$$p_i = \frac{\sum_{i=1}^{m} r_i u_i}{\sum_{i=1}^{m} r_i} \tag{6.1}$$

In this manner, movie metadata of the items the user liked most will give a proper contribution to the average. Each movie of the corpus, except for those the user has already seen and rated is then compared with the user

profiling vector. This way we compute similarities between movies' vector and **p**, using cosine similarity between vectors of the same shape. Movies are then sorted with decreasing values of the cosine similarity score. The topK movies are selected as recommendation for that user.

6.1.3 Pure weighted content-based model

We also explored a possible variation of the content-based model described above, which can provide more customised recommendations. Let's say the owner of a recommendation system wants to emphasize some movie metadata features with respect to the others when computing similarities between user profiling vector and the corpus. For instance, one could want to give more emphasis to the director and cast feature and recommend movies to some user mainly because it has previously watched movies performed by certain actors and directors. Or, in the same way, one could want to recommend movies to a certain user only because they treat same topic (with similar synopsis). This can be applied to all attributes of the final dataset and it is achievable by providing the model a set of weights, one weight for each feature. Thus, instead of concatenating all features together in the same one dimensional vector before computing **p**, we keep each attribute separated and we will obtain several cosine similarities scores (times the number of chosen movies' attributes) between user profiling vector and movies' vectors: one plot-related similarty, another genre-related similarity and so on. The resulting similarity would be a weighted average of all scores, with weights set arbitrarily by the recommendation system owner for giving emphasis to some features with respect to others.

6.2 Pure collaborative filtering model

The second model experimented in this thesis is a LightFM simple collaborative filtering model without the use of item or user feature, which is equivalent to a matrix factorization model that learns users and items embedding by minimizing a loss through Stochastic Gradient Descent, as described in the Recommendation System Chapter. Such model's training only needs to be fed with records containing minimal information about user and item interactions:

- User id;
- Movie id;

- Rating given from the user to the movie.

When the LightFM model object is created, some relevant hyperparameters have to be set:

- **no_components**, the dimensionality of the feature latent embeddings learnt by matrix factorization;
- **learning_rate**, initial learning rate for the *adagrad* learning schedule;
- **learning_schedule**, *adagrad* [33] (adaptive gradient algorithm) or *adadelta*[34], an extension of that adapts learning rates based on a moving window of gradient updates;
- **loss**, the typology of loss function to be minimized during training.

The LightFM model is trained on the training set by calling the *fit* method on user-item interactions, which is a COOrdinate format sparse matrix of dimensions (number of users, number of items) where each entry is the rating of a user to a movie. This matrix is extracted as attribute of a previously created instance of *DatasetHelper*. Moreover, the model is provided the same matrix as sample_weight, expressing weights of individual interactions from the interactions matrix. In fact, our interactions data does not only provide either positive or negative interactions, but we are given a magnitude of every interaction, which is the user rating for a movie. The *LightFM.fit()* method takes as most relevant inputs:

- **interactions**, the user-item interaction matrix;
- **user_features**, only the user ids for this model;
- **item_features**, only the item ids for this model;
- **sample_weight**, the user-item interaction matrix;
- **epochs**, the number of training epochs.

What the model does is learning embeddings for users and items of size *no_components* that should approximate the original feedback matrix through Stochastic Gradient Descent, is an iterative method for optimizing an objective function, the loss function. Embeddings are latent representations in a high-dimensional space such that they encode user preferences over items. Those users' and items' latent representations are expressed in terms of their

features' representations. For this model, no feature matrices are provided to the *LightFM.fit()* or *LightFM.predict()* methods. Thus, they are implicitly assumed to be identity matrices: that is, each user and item are characterised by one feature that is unique to that user (or item). These indicator features are simply the users and items ids. In this case, LightFM reduces to a traditional collaborative filtering matrix-factorization method. When multiplied together, users' and items' embeddings produce scores for every item for a given user. The evaluation of these model works as follows: for each K, that is the number of items to be recommended to users and for each user id we call *LightFM.predict()* for obtaining recommendation scores for each movie in the corpus, with the exception of those movies already seen by that user. This method takes as input:

- **user_ids**, user ids for the user-item pairs for which a prediction has to be computed;
- **item_ids**, item ids for the user-item pairs for which a prediction is to be computed;
- **user_features**, not set for this model;
- **item_features** not set for this model.

Movie ids are associated with their own scores and sorted afterwards with decreasing order. The top K movies in the ranking are selected, those represent the recommendation for that user. At this point, we can compute recall and precision for that user with respect to the test set, that is the set of movies that he has seen and rated for real. At the end of the K loop, mean average precision and mean average recall are computed.

6.2.1 BPR and WARP losses

LightFM can use several types of losses, such as logistic, BPR and WARP. In this thesis we have taken into account Weighted Approximate-Rank Pairwise (WARP) [35] loss, which maximises the rank of positive examples by repeatedly sampling negative examples until rank violating one is found. Similarly to the the BPR loss [36], WARP deals with (user, positive item, negative item) triplets. Unlike BPR, the negative items in the triplet are not chosen by random sampling: they are taken among those negative items which would violate the desired item ranking given the state of the model. This approximates a form of active learning where the model selects those triplets

that it cannot currently rank correctly. This procedure yields roughly the following algorithm [37]:

1. Pick a negative item at random from all the available items for a given (user, positive item combination). Calculate predictions for both items; if the negative item's prediction is greater than the positive item's plus a margin, apply a gradient update to raise the positive item and lower the negative item. If there isn't a rank violation, keep sampling negative item until one is discovered.

2. If you find a violating negative example on the first try, do a big gradient update: this suggests that, given the present state of the model, a huge number of negative items are rated higher than positive ones, and the model has to be updated by a substantial amount. If finding a violating example requires a lot of sampling, do a minor update: the model is probably near to the optimum and should be updated at a low pace.

6.3 Hybrid model using item features

LighFM greatest power is the possibility to allow item features to be incorporated into classic matrix factorization algorithms [9]. The rationale behind this is to feed the model movie metadata acquired from our use-case dataset in order to improve recommendation accuracy and quality. Each item in LightFM is represented as the sum of the latent representations of its features, allowing recommendations to be generalized to new movies based on item features. The presented feature matrix has got rectangular shape (number of items, number of features). Afterwards, an embedding will be estimated for each feature, resulting in a large number of feature embeddings. The model will look up the i-th row of the feature matrix to locate the features with non-zero weights in that row, then add the embeddings for these features to produce the item representation. For example, if a particular item has weight 1 in the 5th column of the item feature matrix and weight 3 in the 20th column, the item's representation will be determined by adding the embedding for the 5th and 20th features and multiplying the latter by 3. After feature' selection, the chosen item-features to be fed to LightFM were the following:

- Movie Release year;
- Movie Runtime quantile label;

- Movie Popularity quantile label;
- Movie Overview cluster;
- Movie Director and Cast cluster.

Item features are provided to the model through *DatasetHelper*[38] class methods by passing them a *Pandas* dataframe whose records are item metadata, those just listed above, for each movie id. For this model, as users' metadata we simply provide user ids.

6.4 Hybrid model using both item and user features

After having extracted user features, as described in the Pre-processing chapter, we can exploit them for enriching the quality of our embeddings. This second hybrid model is based on the same reasonings of the last one we have just discussed. In addiction to item features, here we provide the model users' features too. Again, to obtain the representation for user i, the model will look up the i-th row of the feature matrix to find the features with non-zero weights in that row; the embeddings for these features will then be added together to arrive at the user representation. Selected item features after features' selection were the following:

- Preferred movie genres;
- Top Cast string;
- Top Director string;
- Top Overview clusters;

6.5 Modelling user-item cold start

User and item cold start are common situations in recommender systems that can be described as follows:

- **User cold start**, we want to make a recommendation to **user** that has no past interactions with any movie, e.g. a new user that has not seen and rated any movie within the dataset;

- **Item cold start**, we want to recommend **items** that have no related interactions with users such as newly released movies that no user has seen.

Modelling the user cold-start means simulating a setting in which a selected portion of users' interactions is excluded from the training set. A model without those users' interactions is trained with LightFM, in order to simulate cold-start. Those users will be called called "cold users". Afterwards, we want to test what how our model is capable of recommending items to those users in two cases: we can have no information at all about that user, i.e. we only have its user id, or we have some user metadata collected for instance during sign-up process with a survey. In fact, one can think of asking users to fill out a form and to provide some preference during the registration process, such as preferred genres, actors, directors. In order to simulate item cold start we used a similar approach: cold movies' interactions where kept out of the training dataset. A model without those movies' interactions (users' ratings for those movies) was trained and tested on users that had reviewed those cold movies. Moreover, we inspected whether the model was capable, in the item cold-start setting, to recommend those cold items that is how much capable was to score them properly in the topK.

Chapter 7

Experiments and results

In this chapter we are going to dive into the details of our experiments with recommendation models described in Chapter 6: content-based, collaborative filtering and, finally, hybrid recommendation with user and item metadata. We will describe how The Movies Dataset was split in training and testing set, how feature selection and cross-validation were made and which metrics we took into account. For each recommendation system we provide results in terms of Mean Average Precision and Recall. Also AUC will be considered. Moreover, we will compare LightFM models with each other and examine how they are able to tackle the problems of user and item cold start.

7.1 Training and testing splitting

After the pre-processing steps, final dataset containing user-item interactions and metadata is split in training and testing sets with proportion 7:3. The splitting is made with shuffling and stratification per user_id using *train_test_split* method from *sklearn*, that ensures a better distribution of the users ratings between training and testing sets. Stratification is intended to avoid that some users, those with fewer ratings discussed in the Dataset section, could end up not being represented in the test set. On average, each user has got around 100 movies in the training set and around 49 in the test set. In order to inspect the goodness of the splitting, we looked at the average distance, in terms of cosine similarity, between users' feature vectors in the training and in the testing set after the splitting. This analysis showed that the splitting is discretely able to preserve the preferences of each user, given that the average cosine similarities are always above 0.6. Another consideration to be done is that, as shown in the scatter plot below, users that

reviewed a larger set of movies reasonably show higher similarities between training and testing feature vectors.

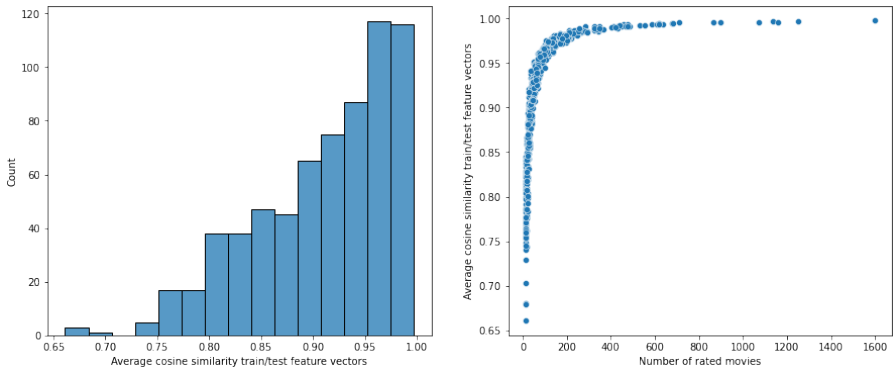

Figure 7.1. Dataset splitting analysis. Frequency barchart of cosine similarities between movies' feature vector in train-test set (left). Relationship between similarities between movies' feature vector and number of movies rated by users (right).

7.2 Metrics

In order describe the details of the metrics for evaluating and comparing results of the experiments with our models, we need to explain the concept of "relevance" of a movie for a user. A movie is considered relevant for a user if the user has seen and reviewed it with a rating over a certain threshold. This threshold can be obtained with two different methods:

1. set to an arbitrary fixed value,
2. be derived dynamically from our data.

In the latter case, we can set a custom dynamic threshold for each movie, which is the average rating for that movie in the whole dataset. As already discussed in Chapter 2, the average rating for movies in our dataset is always above 3. Since it is hard to set a reasonable fixed treshold, we opted for the dynamic threshold. Thus, each movie is mapped with its average rating. If a user rates that movie with a rating higher or equal to its average rating, we can consider that movie relevant for him. Moreover, when making a recommendation to a user, movies that it has already seen are not taken into account since they would negatively affect the results. Each user will be

recommended K items, those will be compared with items the user has seen and rated for real, movies in the testing set.

7.2.1 Precision@K and MAP

In information theory, precision is defined as the fraction of relevant items among the retrieved items and it is also called Positive Predictive value. For the purpose of this thesis, we define precision as follows:

$$p = \frac{\#\text{ relevant recommendations}}{\#\text{ recommended items}} \tag{7.1}$$

where the denominator will be simply called K. Precision tells us how the model is good at recommending relevant items among its topK recommendations. As such, Precision@K is very focused on the ranking quality at the top of the list: it doesn't matter how good or bad the rest of the ranking is as long as the first K items are mostly positive. This is an appropriate metric when showing users the very top of the list. The Mean Average Precision (MAP), which we will use to evaluate our models, is simply the average precision across all users of the dataset for a given value of K.

7.2.2 Recall@K and MAR

Recall, also known as sensitivity, is generally defined as the fraction of the relevant items that are successfully retrieved. For movies' recommendation systems, it is derived as follows:

$$r = \frac{\#\text{ relevant recommendation}}{\#\text{ relevant items}} \tag{7.2}$$

It can be viewed as the probability that a relevant document is retrieved by the query. In our setting, the denominator of the equation below represents the number of items inside the testing set for that user. The Mean Average Recall (MAR) is simply the average recall across all users of the dataset for a given value of K.

7.2.3 F1-Score

Recall and precision alone are not completely informative, they need to be evaluated in pair. Alos, when evaluating cross-validated models for hyperparameters' tuning, we needed a metric that could take into account both recall

and precision. F1-Score ensures equal contribution of those two metrics. It can be interpreted as a harmonic mean of the precision and recall. This metrics reaches its best value at 1 and worst score at 0, whereas precision and recall maximum values depend on the size of the testing set.

$$F1 = 2\,\frac{p\,r}{p+r} \tag{7.3}$$

7.2.4 AUC

AUC is the probability that a randomly chosen positive example has a higher score than a randomly chosen negative example. By "positive" example, we mean a film that has been viewed and rated by a user. The maximum possible AUC value is 1.0, which means that no negative item is ranked higher than any positive one. The AUC is used to assess the overall ranking's quality. An AUC near to 1.0 indicates that, on the whole, the ranking is right, even if none of the first K items are positives. In truth, it does not take into account the expected order-rank of movies. A high AUC score indicates that the ranking is of excellent quality all the way through. However, given that AUC considers either positive or negative items, we have to consider that AUC ignores information on the magnitude of positiveness of items.

7.3 Hyperparameter tuning and feature selection

Every model was cross-validated using *StratifiedKFold* from sklearn, a class which provides train and test indices to split data in train and test sets. Cross-validation was intended to choose the best hyperparameters and to evaluate their robustness with respect to randomness of the splits. In other words, we wanted ensure that performances on the test set would not be affected by the type of split. With LightFM models, for each fold, we applied random search over sets of hyperparameters. The best ones were chosen in terms of better F1-Score in order to give contribution to both precision and recall metrics. For hybrid models, there was the need of choosing the most informative items' and users' metadata. Given the limited number of possible handcrafted attributes, in those cases we applied an exhaustive search over possible combinations, always choosing those one proving the best F1-Score.

7.4 Results

In this section we are going to show and compare results of our experiments with recommendation systems models described in the previous chapter. Using several values of $K \in (3,5,7,9,11,13,15,17,19,21)$, with K being the number of recommendations for users, each model is evaluated on the training and testing set, using the same split. Most relevant metrics will be Mean Average Precision and Recall with a custom threshold for each movie as described above, F1-Score and AUC.

7.4.1 Pure content-based

As explained in the previous chapter, pure content recommendation model bases recommendations on item metadata and uses users' ratings for computing user profiling vectors. For each K and user id, we compute the user profiling vector using weighted average with user ratings as weights. This vector is then compared with all movies in the corpus: the total number of movies except for those ones already seen by the user, i.e. movies in the training set for that user. Then, from the whole ranking the model provides a topK recommendation. Given this set of recommended movies we compute MAP and MAR across users.

K	**MAP**	**MAR**	**F1-Score**
3	0.2960	0.0491	0.0362
5	0.2718	0.0734	0.0522
7	0.2522	0.0942	0.0585
9	0.2415	0.1116	0.0647
11	0.2327	0.1305	0.0671
13	0.2224	0.1442	0.0704
15	0.2136	0.1566	0.0728
17	0.2076	0.1704	0.0740
19	0.2003	0.1828	0.0765
21	0.1946	0.1941	0.0774

Experimenting content-based model we observed a reasonable behaviour of MAP and MAR: they tend to get closer to each other as K increases, as showed in the results reported in the table and displayed in the lineplot figure blow. Precision@K is inversely proportional to K: as the value of K gets higher, Precision@K tends to decrease because it becomes harder to

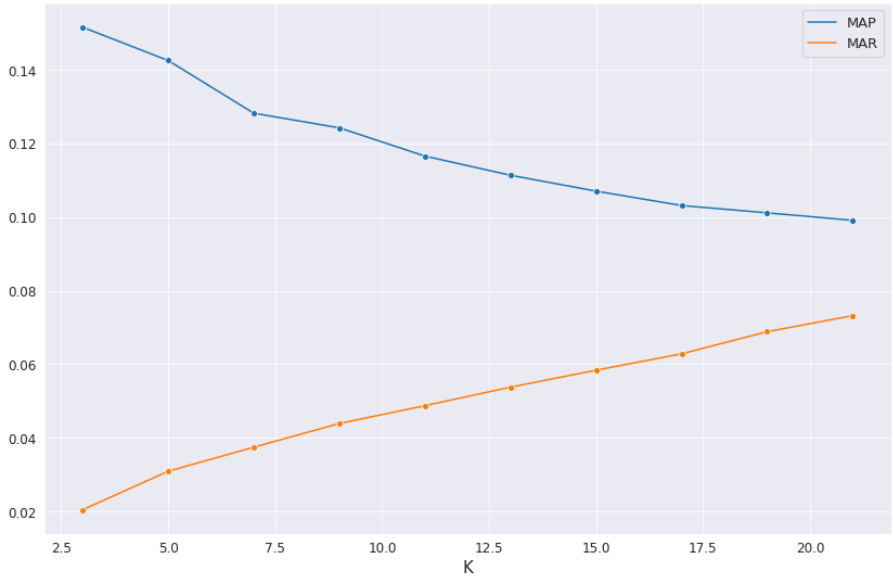

Figure 7.2. Content-based model Mean Average Precision (MAP) and Mean Average Recall (MAR).

successfully recommend more and more items to users, i.e. the denominator increases more rapidly than the numerator, which is the number of recommended movies that the user has seen and rated as relevant. Recall@K, instead, tends to be directly proportional to K. In fact, as the number of recommended movies increases, we have more chances to recall relevant items within all the movies that user has viewed and rated as relevant. This behaviour of MAP and MAR will be recurrent in all our experiments: as K increases Precision@K decreases and Recall@K increases. Thus, the maximum value of MAP will be always reached with the lowest K, the maximum value of MAR will be reached with the highest K, and viceversa. By analyzing the results, we can see how item metadata alone are not sufficient to provide a reliable recommendation that should be "precise" and "sensible". On average, around 1 over 10 movies recommended where really seen by the users and the model is not able to recall more that 7% of the movies belonging to the testing sets. The same model was also evaluated on the training set, i.e. the user profiling vector was also compared with those movies that were taken into account for its computation. From the lineplots we can see how the model is discretely fitting the training set, expecially in terms of MAP, but it hardly replicates those results on the testing set.

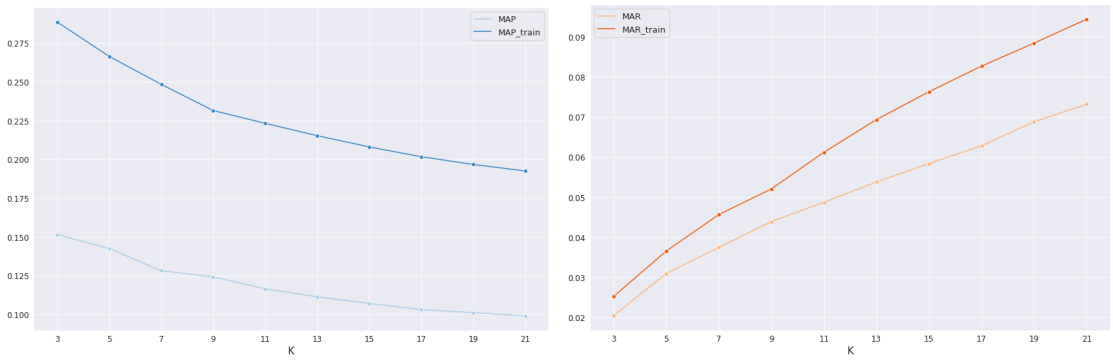

Figure 7.3. MAP (left) and MAR (right) on training and testing sets.

7.4.2 Weighted content-based

As described in the previous section, this additional model is intended to provide a customized recommendation, by giving more importance to some features with respect to others. We experimented the performances in terms of average F1-score using *StratifiedKFold* cross-validation with several sets of weighs summing up to 1, one weight for each of the features:

- overview embeddings,
- director and cast embeddings,
- genre and language one-hot vectors,
- numerical features such as vote average, release year, popularity, duration.

Resulting performances are worse than the pure-content based model: precision never overcome 7% and recall is always below 3%. However, this recommendation method is more able to focus the topK recommendation on certain attributes: when a higher weight is given to "genre" attribute, it tend to recommend movies of same genres or when a higher weight is given to popularity, it tends to recommend more popular movies. The same reasoning is valid for all attributes when their cosine similarity scores are "boosted" by arbitrary weights. This system can be useful when the recommendation system owner wants to provide users recommendations tailored on their preferences related to specific hand-crafted features. Best results give higher weights to popularity, overview embeddings and genre.

7.4.3 Pure Collaborative Filtering

As explained in the previous Chapter, collaborative filtering system exploits matrix factorization algorithm for movie recommendation. It uses the power of the interactions' data: records reporting user id, movie id and rating. User and item representations are learned from the training interaction matrix and exploited to provide topK recommendation. Those recommendations are then evaluated in terms of Precision@K and MAP, Recall@K and MAR. User embeddings have size (671, no_components), the item representation is a matrix (9013, no_components). LightFM also takes into account the magnitude of the interactions through sample_weight parameter, which is the matrix with entries expressing weights of individual interactions from the interactions matrix, i.e. the users-movies interactions matrix. Its row and columns arrays are the same as those of the interactions matrix. LightFM hyperparameters were tuned with a random search over several splits of the original dataset. For each dataset splitting, a random search is performed sampling hyperameters in this manner:

- **no_components** was randomly drawn from interval (15,100) without replacement;
- **learning_schedule** was randomly chosen between "adagrad" and "adadelta";
- **loss** was randomly chosen between "warp" and "warp-kos";
- **learning_rate** was drawn from an exponential distribution with $\beta = 0.05$ scale parameter, inverse of the rate parameter;
- **num_epochs** was randomly sampled from interval (10,50) without replacement.

The best found hyperparameters, shown in terms of highest average F1-Score, are those ones showed in the table below.

K	MAP	MAR	F1-Score
3	0.2960	0.0491	0.0842
5	0.2718	0.0734	0.1155
7	0.2522	0.0942	0.1371
9	0.2415	0.1116	0.1526
11	0.2327	0.1305	0.1672
13	0.2224	0.1442	0.1749

K	MAP	MAR	F1-Score
15	0.2136	0.1566	0.1807
17	0.2076	0.1704	0.1871
19	0.2003	0.1828	0.1911
21	0.1946	0.1941	0.1943

Table 7.2: Pure LightFM collaborative filtering model Mean Average Precision (MAP) and Mean Average Recall (MAR).

LightFM parameter	Value
no_components	88
loss	'warp'
learning_schedule	'adagrad'
learning_rate	0.03
epochs	24

Table 7.3: Best LightFM hyperparameters in terms of F1-Score after cross-validation.

LightFM collaborative model outperformed content-based model, reaching almost 1 out of 3 correct recommendations on average (30% of Mean Average Precision) and 20% Mean Average Recall, meaning that 1 out of 5 movies are correctly recalled from the testing set.

In order to analyze whether the model is correctly fitting training records, we show and compare F1-Scores on training and testing set versus K and AUC score versus the number of epochs. In LightFM, the AUC routine returns an array of metric scores: one for every user in your test data. We averaged AUC metrics for each user and depicted them versus the number of epochs for training and testing sets. As the size of the topK ranking increases, the range between training and testing F1-Score enlarges, meaning that the model is loses capability of generalization as K increases. Similar outcome comes out from AUC versus epoch plot: as training goes on the model starts overfitting on training interactions.

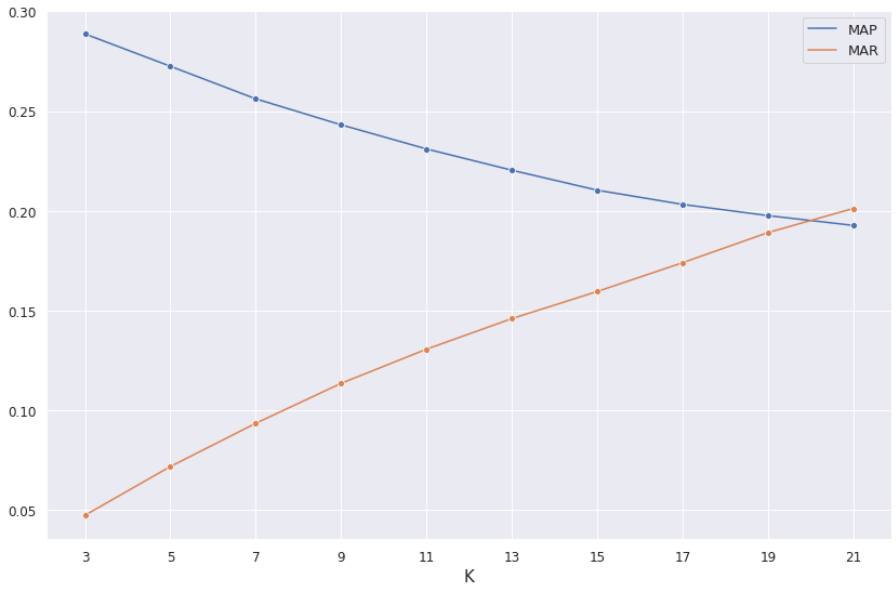

Figure 7.4. Pure collaborative filtering model MAP and MAR versus K.

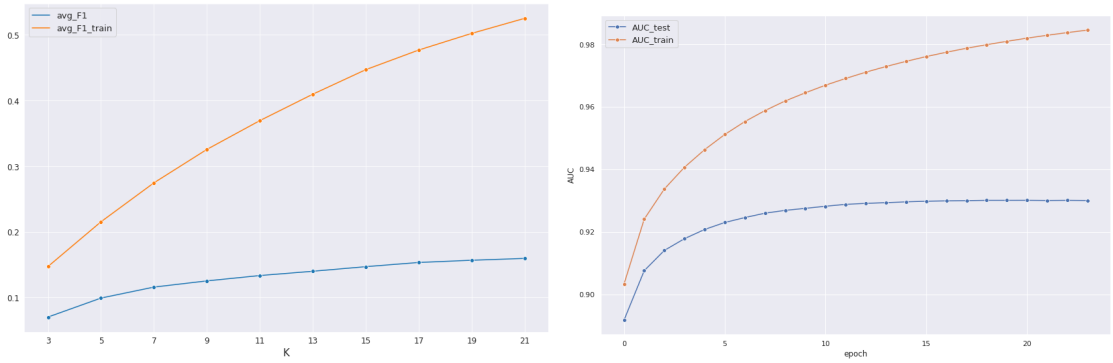

Figure 7.5. F1-score vs K (left) and AUC vs training epochs (right) on training and testing set for LightFM pure collaborative model.

7.5 Hybrid Collaborative Filtering

LightFM library makes possible to incorporate item and user metadata into the collaborative matrix factorization models. In this section we show the result of those experiments with movies' attributes and users' handcrafted metadata as discussed in the Data Preprocessing chapter.

7.5.1 Hybrid model with item metadata

The Movies Dataset, deeply explored in Chapter 4, provides a rich set of items metadata that can be useful for improving quality of recommendation. A *Pandas* dataframe size (9013, num_item_features) is provided to LightFM model through the use of *DatasetHelper* library. Chosen movies' features after feature selection were: genres, original language, release year, runtime, popularity, overview, director and cast. LightFM requires categorical values for the items's attributes, the related feature engineering was already discussed in the Data Preprocessing chapter. LightFM *fit()* method is provided a matrix where each row contains that item's weights over features one-hot encoded. Those weights, or in other words each row, should sum up to 1. The best hyperparameters for LightFM model, in terms of average F1-Score after random search are those described in the following table.

LightFM parameter	**Value**
no_components	72
loss	'warp'
learning_schedule	'adagrad'
learning_rate	0.05
epochs	25

K	**MAP**	**MAR**
3	0.3341	0.0541
5	0.2925	0.0801
7	0.2827	0.9889
9	0.2653	0.1332
11	0.2501	0.1451
13	0.2483	0.1579
15	0.2344	0.1678
17	0.2289	0.1802
19	0.2209	0.2001
21	0.2175	0.2198

The pure collaborative model was outperformed by LightFM Hybrid model with item metadata, especially in terms of precision. We show this evidence with the lineplot that compares their Mean Average Precision and Recall versus K. Again, we analyzed the capability of generalization of the model.

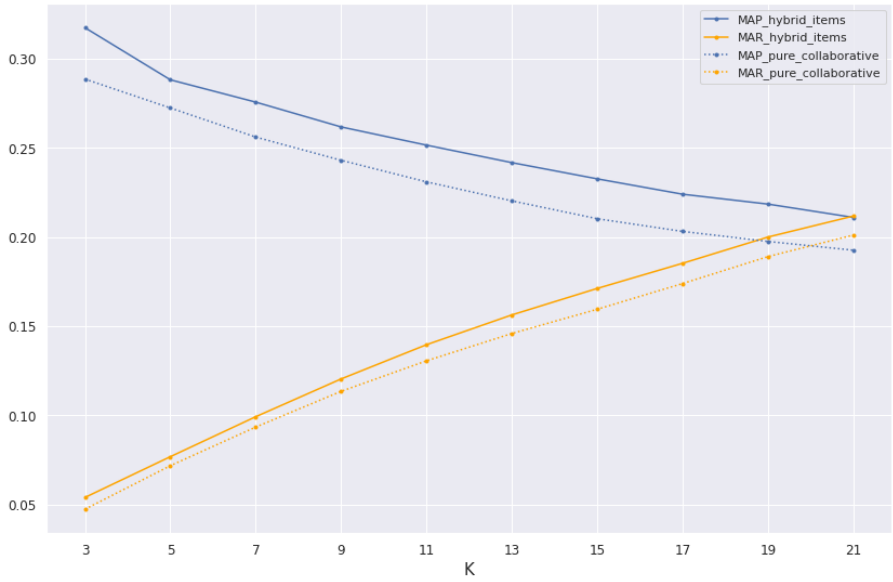

Figure 7.6. Pure LightFM collaborative filtering model vs Hybrid model MAP and MAR.

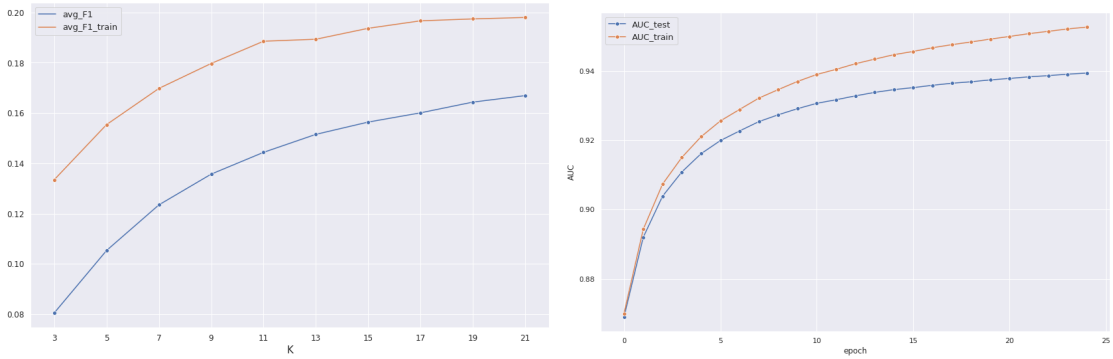

Figure 7.7. F1-score (left) and AUC (right) on training and testing set for LightFM collaborative model with movies' metadata.

The hybrid model turned out to be less prone to overfit on the training data: both F1-Score and AUC improved with respect to LightFM pure collaborative and the range between training and testing performances lower. The only drawback of incorporating item metadata into LightFM model was the increase of training and predicting time.

We also inspected how performances on the test set of our model were influenced by the size of the training set, the number of movies rated by our

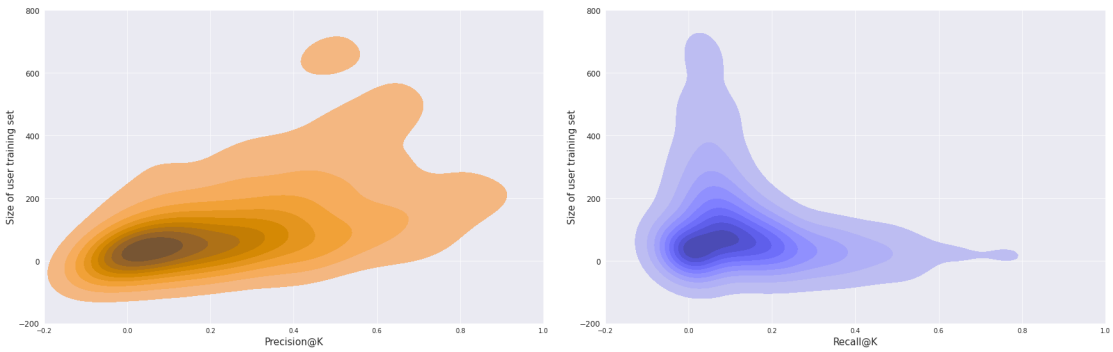

Figure 7.8. Bivariate KDE distribution of Precision@K and Recall@K versus the size of the user training set.

users. Precision@K tends to show direct proportionality with the size of the training set: the more movies a user has seen and rated, the better the model is capable of providing an accurate recommendation. The same statement cannot be totally confirmed with recall, the size of the training set seemed not directly affect the sensitivity of the model, i.e. its capability to recall relevant items from the testing set.

7.5.2 Hybrid model with user and item metadata

The Movies Dataset does not provide intrinsic user metadata. However, we experimented a Hybrid movie recommender system feeding LightFM also users' handcrafted metadata, pre-processed and engineered as explain in Chapter 5. Therefore, this time LightFM *fit()* method is also provided a matrix where each row contains that users's weights over features one-hot encoded item features'. Feeding the model with also user handcrafted metadata did not bring to any significant improvement in terms of Mean Average Precision and Recall. Performances of this model remain better than Pure Collaborative, but did not overcome the Hybrid model with only items' metadata.

7.6 Cold-start

7.6.1 Item cold-start

Item cold-start is the setting in which new movies that no users have seen are integrated into our model. We will refer to items that were not seen by any user as "cold items". We can think to item cold-start as a situation in which the recommender system owner would like to add new users to the corpus that no one has seen and rated. We imagine that those movies are added to the corpus with not only their id information, but also some related features that are able to describe them: genre, language, plot etc. Our experiments with item-cold start using LightFM Hybrid model followed the steps below:

1. Randomly splitting training e testing set with user_id stratification;
2. Removing interactions from 25% of movies randomly chosen with no replacement from the training set in order to simulate item cold-start;
3. Training the Hybrid LightFM model on all users of the training set but excluding interactions (i.e. ratings) related to cold movies;
4. Evaluating the model in terms of F1-Score on the testing set for users which have had interactions with cold movies.

The same procedure was applied using a Pure Collaborative LightFM models and results were compared with Hybrid LightFM in terms of F1-Score on the testing set for those users that interacted with cold movies. Results show that Hybrid LightFM clearly outperform Pure Collaborative LightFM in a item cold-start setting. However, cold movies hardly appear in the topK recommendations for out set of K. From our experiments we observed that, on average, cold movies are recommended only with values of K above 100, and the hybrid model is more capable of scoring them higher thanks to items features. Thus, although the Hybrid model is more precise and capable of recalling right movies to users, it needs some external boosting to recommend cold items.

7.6.2 User cold-start

LightFM, like any other recommender algorithm, cannot make predictions about entirely new users if it is not given additional information about those users. When trying to make recommendations for new users we want to

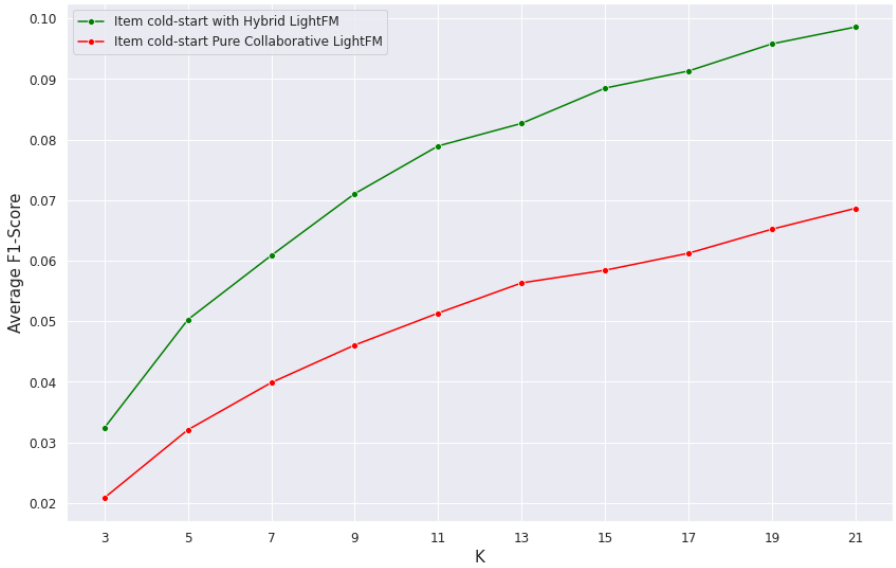

Figure 7.9. F1-Scores for LightFM Hybrid model (green) and Pure Collaborative LightFM (red) trained without cold items interactions and tested on those users that rated cold items.

describe them in terms of the features that the algorithm has seen during training. An overview of the modelling of user-cold start was given in Chapter 6. We will refer to users that have had no interactions with any movies "cold users". Those are the details of the main steps of our experiments:

1. Randomly splitting training e testing set with user_id stratification;
2. Removing interactions from 25% of users randomly chosen with no replacement from the training set in order to simulate user cold-start;
3. Training a Hybrid LightFM model on the training set using only interactions from not cold users. Both users' and items' metadata from all users, cold and not cold, were fed to the model;
4. Evaluating the model on the testing set with average F1-Score only for those cold users interactions;

The results were again compared to those ones of a Pure Collaborative Model in the same user cold-start setting. This time we cannot notice any relevant improvement due to Hybrid Model and its users' features. Except for the lowest values of K, F1-Score are very similar. This performances let us think that users' features are not enough for improving performances on

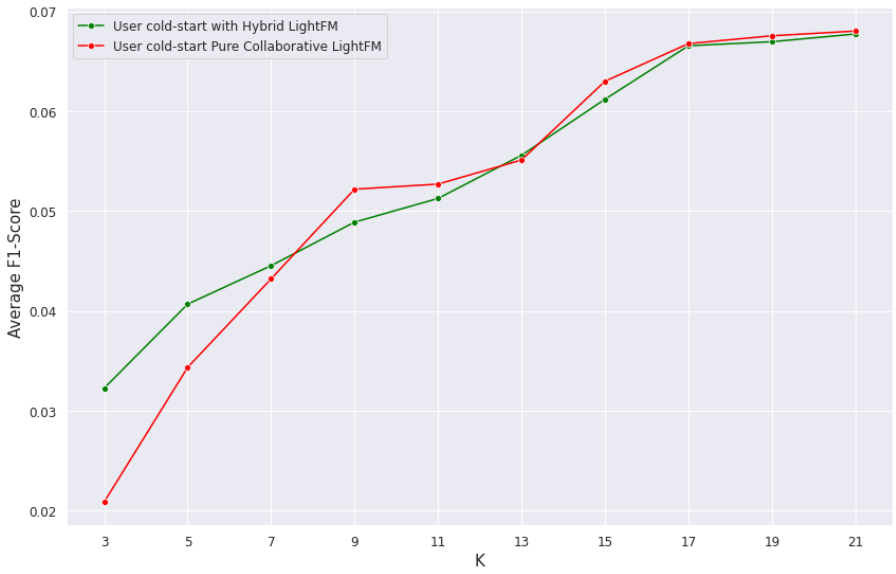

Figure 7.10. F1-Scores for LightFM Hybrid model (green) and Pure Collaborative LightFM (red) trained without cold users interactions and tested on cold users.

new users, since they do not provide any additional information with respect to the user-movie interaction matrix. Inspecting the typology of recommendations, we can notice that both models tend to recommend popular items, i.e. items seen and rated frequently. Thus, the Hybrid Model in this case is not really capable to provide cold users a topK recommendation, tailored on their features.

7.6.3 LightFM in production

In real use-cases, we generally do not have any cold users' and items' information available in advance. Thus, the model has to be trained several times in production. When new users and items are available with or without metedata, the model has to be re-fitted on those new data. We made experiments to make it possible to train LightFM in a real item-user cold-start setting. In order to train LightFM in production the *fit_partial* method was used. As explained in the Pre-preprocessing Chapter, we used *DatasetHelper* for easily feeding the models our *Pandas* dataframes. In order to continue using the *DatasetHelper* class methods and make them compatible with training in production, we needed to make a little modification to the source code. In fact, when creating a *DatasetHelper* object, the *Dataset* class from LightFM is

called. user_identity_features and item_identity_features parameters make the model create a unique feature for every user/item in addition to other features. The default setting of those parameter of *Dataset* class has to be manually set to *False* in order to preserve the dimensionality of the features' space when re-training the model on cold users-items.

Chapter 8

Conclusion and further improvements

The aim of this thesis was to investigate families of Recommender Systems and to build a Hybrid Movie Recommender engine capable of exploiting both user-item interactions and items' metadata gathered from "The Movies Dataset". We also aimed to inspect how textual natural language information could be exploited to improve the quality of the suggestions and better tackle the cold-start problem.

First of all, we compared pure content-based models with LightFM simple collaborative filtering. Results pointed out the importance of the use of interactions' data between users and items for high quality recommendations. In addition to being faster to train and evaluate, all models based on matrix-factorization outperformed our content-based models. The pure LightFM model doubled the performances of content-based in terms of average F1-Score, which reflected its capability of both accurately recommend items within topK ranking and recalling real interactions. In Chapter 5 we deeply investigated our item-metadata, by focusing on the usage of NLP for embedding textual information such as movie synopsis, cast and crew as points in a multidimensional embedding space where similar items lie close to each other. Surely, having high quality item attributes makes it easier for content-based recommenders to scale to larger number of users, without the need of many data about other users' interaction. When those attributes properly interpret similarity between items, content-based models can capture specific interests of a user based on its preferred genres, plot, actors, directors etc. and can recommend niche items that very few users are interested in. However, even

the model the best quality hand-engineered features will never be able to consider the diversity of users' tastes, since recommendations would be limited to the existing interests, i.e. the user-bobble. Those findings pointed out the limitations of content-based approach: need of human domain knowledge for feature engineering, poor training and evaluation efficiency, lack of serendipity. The pure LightFM collaborative approach was more straightforward and accurate, since only the feedback matrix was needed for the embeddings to be automatically learned by matrix-factorization. Thus, this model needed poor human domain knowledge and resulted to provide diversity in recommendations. It might recommend movies different from users' history, helping them to discover new interests.

LightFM library gave us the opportunity to experience the advantages of hybrid recommendation system models. Our experiments showed that feeding LightFM matrix-factorization model with items' metadata, in addition to ratings, can improve the quality of recommendations, especially in terms of Precision@K. Despite a longer training and testing time, hybrid models were less prone to overfitting and more capable of generalize. Those improvements can be also interpreted as proof of the quality of our sentence-embeddings, which demonstrated to provide information to the models. Differently from context-free models, BERT confirmed to supply the best contextualized embeddings for sentence similarity and topic modelling. word2vec provided context independent embeddings, GTP-2 was able to only handle past or left side context. BERT, instead, was better in learning the context based on tokens surroundings, thanks to self-attention mechanism. We surely believe that even better results could be achieved with heavier versions of the transformer model.

"The Movies Dataset" does not come with users' intrinsic metadata such as users' demographics. Thus, we manually extracted them from their interactions with movies (e.g. users preferred genre). Feeding also synthetic users' metadata to hybrid models did not improve performances. We believe that the availability of users' metadata in the dataset such as sex, age, income etc. could help further improving the quality of recommendations.

The low amount of information provided by users' metadata was pointed out by the results in user cold-start setting. When new users are introduced in the engine without past interactions but personal preferences (e.g. provided

within a registration online form) the hybrid engine is not capable of better interpreting their input preferences. In fact, the simple LightFM model performances compared to the hybrid ones are similar: both models tend to recommended cold users popular items within the dataset, without being able to interpret their input preferences. We would expect improvements of performances in user cold-start with proper users' demographic metadata.

In the item cold-start setting, our hybrid models brought slight improvements with respect to pure collaborative filtering. When new items, without past interactions, are added into the system together with their genre, textual plot, cast, etc., our hybrid engine is better than a simple matrix-factorization model in recommending them. In fact, the hybrid system is capable of describing new items in terms of the features that the algorithm has seen during training. However, as explained in Chapter 6, even the hybrid model needs a lot of attempts before recommending cold items, since they are suggested only with high values of K. This is a common situation in item-cold start setting: new items are hard to recommend, therefore they need to be boosted with respect to those items already having past interaction history.

Bibliography

[1] "Recommendation systems - google developer." https://developers.google.com/machine-learning/recommendation/. Accessed: 2022-03-01.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[3] CodeEmporium, "How to get meaning from text with language model bert | ai explained." https://www.youtube.com/watch?v=xI0HHN5XKDo. Accessed: 2022-03-01.

[4] J. Alammar, "Illustrated gpt-2." https://jalammar.github.io/illustrated-gpt2/. Accessed: 2022-03-01.

[5] "How retailers can keep up with consumers." https://www.mckinsey.com/industries/retail/our-insights/how-retailers-can-keep-up-with-consumers. Accessed: 2022-03-01.

[6] F. Fkih, "Similarity measures for collaborative filtering-based recommender systems: Review and experimental comparison," *Journal of King Saud University - Computer and Information Sciences*, 2021.

[7] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, p. 30–37, aug 2009.

[8] C. A. Gomez-Uribe and N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Trans. Manage. Inf. Syst.*, vol. 6, dec 2016.

[9] M. Kula, "Metadata embeddings for user and item cold-start recommendations," in *Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015.* (T. Bogers and M. Koolen, eds.), vol. 1448 of *CEUR Workshop Proceedings*, pp. 14–21, CEUR-WS.org, 2015.

[10] J. Beel, B. Gipp, S. Langer, and C. Breitinger, "Research-paper recommender systems: A literature survey," *International Journal on Digital Libraries*, pp. 1–34, 07 2015.

[11] Mohdsanadzakirizvi@gmail.com, "Demystifying bert: A comprehensive guide to the groundbreaking nlp framework." https://www.analyticsvidhya.com/blog/2019/09/demystifying-bert-groundbreaking-nlp-framework/, 2019. Accessed: 2022-03-01.

[12] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013.

[13] J. Alammar, "The illustrated bert, elmo, and co. (how nlp cracked transfer learning)." https://jalammar.github.io/illustrated-word2vec/, 2021. Accessed: 2022-03-01.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *Proceedings of Workshop at ICLR*, vol. 2013, 01 2013.

[15] A. M. Dai, C. Olah, and Q. V. Le, "Document embedding with paragraph vectors," 2015.

[16] C. D. M. Jeffrey Pennington, Richard Socher, "Glove: Global vectors for word representation." https://nlp.stanford.edu/projects/glove/. Accessed: 2022-03-01.

[17] TensorFlow, "Transformer model for language understanding." https://www.tensorflow.org/text/tutorials/transformer. Accessed: 2022-03-01.

[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[19] J. Devlin and G. A. L. Ming-Wei Chang, Research Scientists, "Open sourcing bert: State-of-the-art pre-training for natural language processing." https://ai.googleblog.com/2018/11/open-sourcing-bert-state-of-art-pre.html, 2018. Accessed: 2022-03-01.

[20] "Topic modelling with bert." https://towardsdatascience.com/topic-modeling-with-bert-779f7db187e6. Accessed: 2022-03-01.

[21] "Sentencetransformers documentation." https://www.sbert.net/docs. Accessed: 2022-03-01.

[22] N. Reimers and I. Gurevych, "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, 11 2019.

[23] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," 2020.

[24] R. J. G. B. Campello, D. Moulavi, and J. Sander, "Density-based clustering based on hierarchical density estimates.," in *PAKDD (2)* (J. Pei, V. S. Tseng, L. Cao, H. Motoda, and G. Xu, eds.), vol. 7819 of *Lecture Notes in Computer Science*, pp. 160–172, Springer, 2013.

[25] "Hdbscan documentation." `https://hdbscan.readthedocs.io/en/latest/`. Accessed: 2022-03-01.

[26] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[27] R. Banik, "The movies dataset - kaggle." `https://www.kaggle.com/rounakbanik/the-movies-dataset`. Accessed: 2022-03-01.

[28] "Grouplens, social computing research at the university of minnesota." `https://grouplens.org`. Accessed: 2022-03-01.

[29] "The movie database api." `https://developers.themoviedb.org/3/getting-started/popularity`. Accessed: 2022-03-01.

[30] "Imdb." `https://www.imdb.com`. Accessed: 2022-03-01.

[31] "The movie database api - popularity." `https://developers.themoviedb.org/3/getting-started/popularity`. Accessed: 2022-03-01.

[32] "Bert documentation from huggingface." `https://huggingface.co/docs/transformers/model_doc/bert`. Accessed: 2022-03-01.

[33] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, p. 2121–2159, jul 2011.

[34] M. D. Zeiler, "Adadelta: An adaptive learning rate method," 2012.

[35] J. Weston, H. Yee, and R. J. Weiss, "Learning to rank recommendations with the k-order statistic loss," in *Proceedings of the 7th ACM Conference on Recommender Systems*, RecSys '13, (New York, NY, USA), p. 245–248, Association for Computing Machinery, 2013.

[36] S. Rendle and C. Freudenthaler, "Improving pairwise learning for item recommendation from implicit feedback," in *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, (New York, NY, USA), p. 273–282, Association for Computing Machinery, 2014.

[37] P. Rais, "Learning-to-rank using the warp loss." `https://notebook.community/paoloRais/lightfm/examples/movielens/warp_loss`. Accessed: 2022-03-01.

[38] M. E. Omari, "Lighfm dataset helper." `https://`

lightfm-dataset-helper.readthedocs.io/. Accessed: 2022-03-01.